FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Entwurf eines Systems für die experimentelle Analyse und Visualisierung von Dynamischer Programmierung auf Baumzerlegungen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Thomas Ambroz BSc
Matrikelnummer 0925772

Andreas Jusits BSc
Matrikelnummer 0928977

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Mitwirkung: Projektass.(FWF) Dipl.-Ing. Günther Charwat

Wien, 21. Jänner 2016

Thomas Ambroz

Stefan Woltran

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Designing a System for Experimental Analysis and Visualization of Dynamic Programming on Tree Decompositions

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Thomas Ambroz BSc
Registration Number 0925772

Andreas Jusits BSc
Registration Number 0928977

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Assistance: Projektass.(FWF) Dipl.-Ing. Günther Charwat

Vienna, 21st January, 2016

Thomas Ambroz

Stefan Woltran

Erklärung zur Verfassung der Arbeit

Thomas Ambroz BSc
1120 Wien, Österreich

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 21. Jänner 2016

Thomas Ambroz

Danksagung

Wir möchten an dieser Stelle unserem Betreuer Stefan Woltran danken, der uns die Möglichkeit gegeben hat, dieses interessante Thema zu behandeln und im Zuge dessen in einer Diplomarbeit zu verarbeiten. Er hat uns immer voll und ganz unterstützt. Außerdem hat er uns zahlreiche großartige Ideen und Vorschläge für unsere Diplomarbeit geliefert.

Des Weiteren möchten wir Günter Charwat danken, der uns bei der gesamten Diplomarbeit ebenfalls immer zur vollsten Zufriedenheit unterstützt hat und uns bei Fragen und Problemen immer zur Seite gestanden ist. Ohne ihn wäre die Diplomarbeit in dieser Art und Weise nicht möglich gewesen.

Daher nochmals ein richtig großes Dankeschön für eure großartige Betreuung und Mitwirkung!

Acknowledgements

We would like to take the chance and thank our supervisor Stefan Woltran who gave us the opportunity to work on this interesting topic and cover it in our master thesis. He always supported and assisted us entirely. Furthermore, he supplied us with numerous awesome ideas and suggestions for our master thesis.

Moreover, we would also like to thank Günter Charwat who always assisted us during the whole master thesis to our full satisfaction and supported us when we had questions or problems. Without his assistance our master thesis would not have been possible.

Therefore, once again a really big thank you for your great support and involvement!

Kurzfassung

Um schwere Probleme lösen zu können, gewann das Konzept der dynamischen Programmierung (DP) auf Baumzerlegungen in den letzten Jahren im Forschungsbereich der Informatik immer mehr an Bedeutung.

Da das Verständnis und die Verbesserungen von Zerlegungsansätzen und DP Konzepten aufgrund der Komplexität der Algorithmen, der großen Datenmengen und der Schwere der Probleme eine enorme Herausforderung darstellen, wird im Rahmen dieser Diplomarbeit ein Tool entwickelt, um Systeme, die DP auf Baumzerlegungen anwenden, analysieren zu können. Bestehende Systeme bieten keine Möglichkeit um den kompletten DP-Prozess - von der Instanz, über die Baumzerlegung bis hin zur Berechnung, welche durch die Anwendung spezieller Algorithmen auf jeden Knoten der Baumzerlegung entsteht - zu analysieren. Im Speziellen wird der Ansatz des BEB-basierten DP auf Baumzerlegungen betrachtet, welcher das Konzept von DP mit dem Abbilden der Informationen anhand von Binären Entscheidungsbäumen (BEB) verbindet. Um komplexe Probleme lösen zu können, liegt unser Bestreben zusammenfassend im Design und in der Entwicklung eines Tools um Systeme, welche BEB-basierte DP auf Baumzerlegungen anwenden, zu analysieren.

Mithilfe der Untersuchung vorhandener Visualisierungskonzepte und einer Analyse der technischen Anforderungen entwickeln wir *DecoVis*, um - neben der Visualisierungsmöglichkeit von Instanzen, Baumzerlegungen und Berechnungen - mehrere Analysemöglichkeiten für DP Systeme anzubieten. Die Analysemöglichkeiten umfassen unter anderem das Erstellen eines Statistik-Reports, das Hervorheben von Metadaten-Informationen oder die Animation der Ausführung von Algorithmen.

Um unterschiedliche Systeme, die DP auf Baumzerlegungen anwenden (*dynBDD*, *D-FLAT*, *D-FLAT^2* und *Sequoia*), bezüglich Leistungsfähigkeit und Stabilität vergleichen zu können, werden mehrere Benchmark-Tests erzeugt und deren Resultate analysiert. Die Probleme 3-Colorability, Stable Extension, Hamiltonian Cycle und Preferred Extensions werden durch die Benchmark-Tests abgedeckt.

Unterschiedlichste Verhalten von Tools, welche BEB-basierte DP auf Baumzerlegungen anwenden, entstehen durch die Ausführung der Systeme mittels verschiedenster Optionen und Einstellungen. Anhand der Analyse dieser unterschiedlichen Systemverhalten untersuchen wir die Anwendung von *DecoVis* im Praxiseinsatz. In weiterer Folge vergleichen wir unterschiedliche Konzepte von Algorithmen und Variabelordnungen in BEBs anhand speziell gewählter Instanzen aus den Resultaten der Benchmark Tests.

Abstract

The concept of dynamic programming (DP) on tree decompositions (TDs) has emerged in recent years as an important research field in computer science for solving hard problems efficiently.

Since the understanding and improvements of decomposition approaches and DP concepts represent an enormous challenge due to complex algorithms, large amount of data and the hardness of problems, we put the focus on the development of a tool used for the analysis of systems applying DP on TDs. Moreover, recent systems have failed to address the possibility for analyzing the entire dynamic programming process from the instance, over the tree decomposition to the computation that is resulting from the application of specific algorithms on each TD node. In particular, the approach of BDD-based DP on TDs is considered, that combines the concept of DP with the storage of information by means of Binary Decision Diagrams (BDDs). In summary, our attempt is to design and develop a tool for analyzing systems applying BDD-based DP on TDs for solving complex problems.

With the help of an examination of existing visualization concepts and a technical requirement analysis, we develop *DecoVis*, a system that provides - beside the visualization of instances, tree decompositions and computations - several possibilities for analyzing DP systems such as statistic report generation, emphasis of particular metadata information or algorithm execution animation.

In order to compare various systems applying DP on TDs (*dynBDD*, *D-FLAT*, *D-FLAT^2* and *Sequoia*) regarding performance and stability, several benchmark tests are generated and the results are analyzed. The benchmark tests cover the problems 3-Colorability, Stable Extension, Hamiltonian Cycle and Preferred Extensions.

Furthermore, we examine the usage of *DecoVis* in practice by analyzing different behaviors of various options and settings of systems applying BDD-based DP on TDs. Based on specific instances that are selected from results of benchmark tests, we compare different algorithm concepts and variable orders in the BDDs.

Contents

List of Figures

Listings

CHAPTER 1

Introduction

1.1 Motivation

Solving hard problems became one of the central challenges in computer science. Even if computers and their involved technologies are getting faster and better, it is still an important requirement that solving a problem is done in the most efficient way, in runtime as well as in memory usage. Nowadays there exist many approaches on how to solve a particular problem, but understanding which algorithm behaves well for a certain instance of the problem is often hard to retrace. One approach to gain such insights is a visualization or some kind of animation which represent what is going on exactly when applying an algorithm to a particular problem. Even more helpful is a visual comparison of different algorithms when applied to the same problem.

In this master thesis, we aim to find an appropriate way to meet these requirements by designing an easy to use tool which provides a visualization for solving complex problems together with the ability to compare algorithms. The concept of *Dynamic Programming on Tree Decompositions* is a recent approach to solve certain problems efficiently. Therefore we want to put our focus on this approach. In particular *BDD-based Dynamic Programming on Tree Decompositions* will be investigated. In this context, our goal is to visualize the whole process, beginning from the instances to the tree decompositions, up to the computations which are represented in BDDs. To the best of our knowledge, there is no other system which provides these advanced features yet.

1.2 Problem Statement

A concept for solving complex problems is splitting the problem into multiple components. Each component represents a new sub-problem. The whole problem is solved with the help of the partial results of these separately computed sub-problems. This concept is called dynamic programming (DP).

One particular approach is dynamic programming on tree decompositions. Here, sub-problems are arranged in form of a tree structure called tree decomposition (TD) [1]. The edges in the tree represent the relation between the sub-problems. To solve the overall problem the solutions of the sub-problems have to be computed from the leaf nodes to the root (i.e. in bottom-up order).

A special form of dynamic programming is called BDD-based dynamic programming where the solution of every sub-problem is represented by a Binary Decision Diagram (BDD) [2]. A BDD is a data structure for Boolean functions which is illustrated by a rooted directed acyclic graph (DAG). There are two types of leaf nodes: "0" (FALSE) and "1" (TRUE). Every path from the root to the leaf node "1" represents a solution and therefore a model of the Boolean function. BDDs can often be reduced in such a way that in practice they are exponentially smaller compared to the worst case. Hence, BDDs provide an efficient way to store the partial and also full solutions of a problem [3].

There exist many concepts and systems for dynamic programming on tree decompositions. Due to the high complexity of the used algorithms, the large amount of data and the hardness of problems, the understanding of dynamic programming is very difficult. Furthermore, the analysis of concepts of dynamic programming and the decomposition is an enormous challenge. Hence, it is hard to improve already existing algorithms or even to develop completely new concepts from scratch. Since small modifications can have a huge impact on the overall computation, a comprehensive debugging and visualization tool would be extremely useful.

1.3 Aim of Work

The aim of the work is to understand and improve systems using dynamic programming on tree decompositions for solving complex problems. On the one hand this can be achieved by obtaining a better understanding about the used concepts and algorithms and on the other hand by analyzing and comparing the results of such systems.

Our approach is to design and implement a system for the visualization of dynamic programming on tree decompositions. All tools that apply dynamic programming for problem solving could be used in our application if they provide an appropriate common file format. Moreover, the system should provide a way to visualize the entire dynamic programming process from the instances, over the tree decompositions to the application of specific algorithms for the computation in each tree decomposition node.

Hence, many further requirements have to be implemented. The main part of the visualization tool is the graphical representation of these three execution parts: instances, tree decompositions, and the computations of the sub-problems. Another requirement is a dynamically composed graphical interface that supports the execution of programs using dynamic programming algorithms on tree decompositions. Many further features are necessary to support the analysis of results, concepts and algorithms. For example, it should be possible to combine the three execution parts to get an overview of the problem. Another use case would be to compare a single execution part of multiple problems or configurations separately.

A particular focus is on BDD-based dynamic programming. To gain a better understanding of different approaches, solutions of the computations should be explained by means of visualization of the BDDs and also by animated BDD-based algorithm execution. The latter means that a fully animated BDD visualizes the generation of the BDDs and their contained solutions from bottom to top. Statistic diagrams should furthermore be used to illustrate and compare metadata, created during the computation of the problems and sub-problems, such as allocated memory, number of BDD variables, etc. Multiple metadata values of different configurations and problems should be comparable by an easy-to-use and clear interface.

On the basis of the realized requirements mentioned above, our implementation should contain the facility for experimental analysis of concepts and algorithms. By running benchmarks, we could get all the needed data for our tool. With the help of the results, algorithms can be developed and existing methods can be enhanced. By means of two different benchmarking types new insights and knowledge over dynamic programming algorithms regarding particular problems should be gained.

In order to get a comparison between currently available systems (i.e. *dynBDD* [3], *Sequoia* [4, 5], *D-FLAT* [6, 7] and *D-FLAT^2* [8]), the first benchmark type deals with the analysis of various dynamic programming tools. According to these results, statements about the suitability of the tools, algorithms and concepts for particular problems should be obtained.

The second benchmarking type should set the focus on a single BDD-based dynamic programming tool, called *dynBDD* [3]. Different configurations on problems should be processed to get conclusions about the used algorithms. In more detail, configuration settings should differ in options like the used ordering of variables in the BDDs, types of the tree decomposition and alternative algorithmic concepts.

In summary, the master thesis includes three main contributions:

- The development of a web application providing the visualization of dynamic programming on tree decompositions
- The comparison and experimental analysis of various dynamic programming tools by doing benchmark tests
- The analysis of a single BDD-based dynamic programming tool by means of benchmarks and different configurations

1.4 Methodological Approach

The following steps are necessary to receive the expected results described above:

1. **Background Research**
   By literature and online research, background information about concepts, terms and definitions are obtained. As a result, we will shortly describe several tools which have a main focus on dynamic programming or on visualization regarding problem solving.

2. **Requirements Analysis**
   Necessary requirements are gathered for the development of a graphical visualization tool that supports the analysis of systems, concepts and algorithms concerning dynamic programming on tree decompositions. On the one hand, our tool should be usable by non-experts for getting an overview of the problem and the concepts of dynamic programming. On the other hand, the tool should have expert features for a detailed analysis and comparison, including debugging features. More details about the required features were already described in the previous section.

   For a better overview of the requirements analysis, we use UML diagrams like the class or sequence diagrams which provide a convenient way for the specification of requirements.

3. **Implementation of Requirements**
   This part covers the development of a web-based tool that supports the analysis of dynamic programming algorithms. The focus lies on the visualization of instances, tree decompositions and the solutions. Furthermore, a dynamic graphical user interface should support the execution of systems based on tree decompositions. Animated BDDs based on the partial solutions, the highlighting of models represented by the BDD and the statistic diagrams of metadata assist the analysis part.

4. **Benchmarks**
   For the benchmark tests we will use two approaches: The first approach is to give an overview of the differences between dynamic programming systems on tree decompositions and their experimental analysis. The second part examines different aspects and configurations of a particular BDD-based tool, namely the *dynBDD* system [3]. Here, the influence of BDD-specific aspects on the computations is examined.

   Both approaches follow the same procedure. First, benchmark tests are executed, and then, the results are analyzed and compared in more detail.

1.5 Structure of the Work and Division of Master Thesis

The following enumeration as well as Figure 1.1 give an overview of the structure and the division of the work.

The parts addressed by both authors of the thesis consist of background, related work, technical requirements and implementation of the software (Chapter 1-4). In each of these chapters a further division and assignment of sections was done (see the list below), so that it is clear which part was written by which author. Later the master thesis is split into two parts, each having another main focus and using other features of the developed software. The visualization of the instance and tree decomposition including the dynamic highlighting of nodes is the common feature used by both students.

The first part is worked out by Thomas Ambroz and deals with the experimental analysis and comparison of tree-decomposition-based systems. In this part the used systems are seen as black box tools. The statistic diagram feature of our tool is used to compare and analyze the metadata generated during the computation. With the advanced statistic diagram feature the metadata of the results of multiple tools can be visualized.

The second part covers the configuration options of *dynBDD*. This component is created by Andreas Jusits and puts the focus on the emphasis of various *dynBDD* settings by means of BDD visualizations and animated BDDs. In fact, the growing, shrinking and merging of the BDDs representing the partial solutions up to the final solution can be viewed and played back by using the automated animation. Furthermore, the highlighting of paths in the BDD helps to understand the models of the solution in a better way.

Andreas Jusits ... AJ
Thomas Ambroz ... TA

1. Introduction TA AJ
    a) Motivation TA AJ
    b) Problem Statement TA AJ
    c) Aim of Work TA AJ
    d) Methodological Approach TA AJ
    e) Structure of the Work and Division of Master Thesis TA AJ
2. Background and Related Work TA AJ
    a) Dynamic Programming AJ
    b) Tree Decomposition AJ
    c) Dynamic Programming on Tree Decompositions AJ
    d) Binary Decision Diagrams TA
    e) Dynamic Programming Systems TA
    f) Visualization Concepts TA
3. Technical Requirements TA AJ
    a) High-Level Description TA
    b) Requirements Analysis AJ
    c) Workflow Design AJ
    d) System Architecture AJ

e) Graphical Representation Analysis TA

4. Implementation of the Technical Requirements in DecoVis TA AJ
   a) Technical Overview AJ
   b) Data Import/Export AJ
   c) DynBDD Execution AJ
   d) Window Management TA
   e) Interactive Visualization Elements TA
   f) User Modes TA
5. Comparison of TD-based DP Systems TA
   a) Challenges and Goals TA
   b) Requirements TA
   c) Benchmark Preparation TA
   d) Benchmark Runs TA
   e) Conclusion TA
6. DecoVis in Practice AJ
   a) Introduction AJ
   b) Different Approaches and Aspects of BDD-based Dynamic Programming AJ
   c) Instance Analysis AJ
   d) Tree Decomposition Analysis AJ
   e) Computation Analysis AJ
   f) Summary AJ
7. Summary and Future work TA AJ
   a) Summary TA AJ
   b) Future Work TA AJ

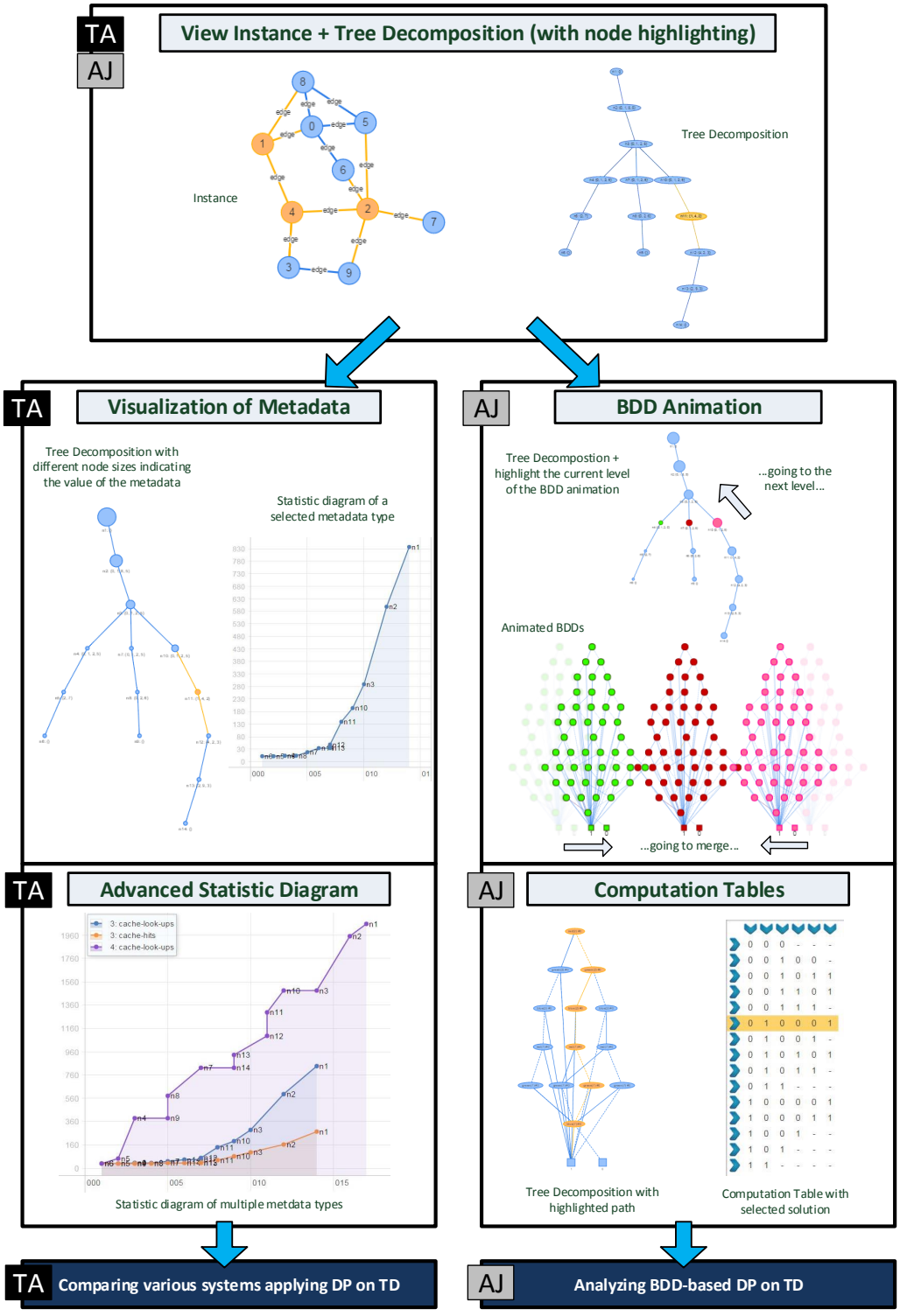


Figure 1.1: Topic overview and division of the master thesis

CHAPTER 2

Background and Related Work

This chapter puts the focus on background information about BDD-based dynamic programming on tree decompositions. Furthermore, similarities to systems using dynamic programming concepts are illustrated and related work regarding visualization of such systems are mentioned.

Since the understanding of dynamic programming is necessary for the further chapters and sections, Section 2.1 handles this concept. By means of an example, this method of breaking a problem down into simpler subproblems is illustrated.

Section 2.2 gives an overview over tree decompositions. The computation of complex graph problems can often be speed up by decomposing an instance graph into a tree. Furthermore, normalization types of tree decompositions (none, weak, semi and normalized) are explained in detail.

In Section 2.3 the connection between Section 2.1 and Section 2.2 is established. It shows the application of dynamic programming algorithms on tree decompositions. With the help of examples, two concepts for the computation on tree decompositions, early decision method and late decision method, are clarified.

The concepts and methods defined in Section 2.3 are extended in Section 2.4 by representing the solutions of each tree decomposition node as Binary Decision Diagram (BDD). Different BDD variants and BDD-based dynamic programming on tree decompositions are moreover explained in this section.

To get an overview over currently available dynamic programming systems for problem solving, Section 2.5 lists such systems and shortly illustrates the concepts behind.

Section 2.6 gives a summary on visualization concepts for windows, graphs, metadata, etc. Existing libraries and frameworks that provide solutions to these concepts are illustrated. Furthermore, existing visualizations in algorithmic design and logic programming are described.

2.1 Dynamic Programming

Dynamic programming defines a concept for mathematical optimization and a method in computer science for solving complex problems efficiently. The basic idea of dynamic programming is to break down a problem into subproblems. Firstly, the subproblems are solved and then the solutions of the subproblems are combined to reach an overall solution of the problem.

In contrast to the popular divide and conquer concept, where the subproblems are solved independently of each other, the dynamic programming method consists of overlapping subproblems. Since each subproblem should only be solved once in dynamic programming, the reusable solutions have to be stored.

By means of the Fibonacci numbers, the following example in Listing 2.1 illuminates the difference between a naive algorithm that follows a recursive definition and a dynamic programming algorithm.

```
-- Fibonacci numbers: F1 = F2 = 1 : Fn = Fn-1 + Fn+2
-- naive algorithm:
fib_naive(n){
  if n <= 2 then
    return 1
  else
    return fib_naive(n-1) + fib_naive(n-2)
}

-- dynamic programming approach:
fib_memory = {}
fib_dp(n){
  if fib_memory contains n then
    return fib_memory[n]
  else
    f = 1
    if n > 2 then
      f = fib_dp(n-1) + fib_dp(n-2)
    fib_memory[n] = f
    return f
}
```

Listing 2.1: Dynamic programming example - Fibonacci numbers

As one can see in Listing 2.1 the naive algorithm calls the function **fib_naive** with the same parameters several times, that means the time complexity of this algorithm is exponential in n. Since **fib_memory** saves each result in **fib_memory**, each computed value can be reused in the next recursion. As a consequence, the time complexity for **fib_memory** is linear in n.

The following sections use a special dynamic programming approach for solving complex problems. Without anticipating details of the following sections, this approach applies dynamic programming on tree decompositions.

2.2 Tree Decomposition

Many NP-hard graph problems can be solved efficiently when the input instance can be represented as tree. Hence, the concept of tree decompositions [1] in combination with the concept of treewidth [9, 10, 11] provides a powerful method for solving problems (representable as graphs) that are known to be intractable. The treewidth defines a measurement unit for the cyclicity of a graph. Furthermore, it gives a logical conclusion concerning the degree of problem solving efficiency (whether the problem can be solved efficiently or not).

Many problems lie in complexity class FPT (fixed-parameter tractable) with respect to treewidth. Problems in FPT [12] are computable in time $f(w) * n^{O(1)}$, where w is the parameter, n is the size of the input and $f(w)$ is some computable function only depending on parameter w. The complexity of the problem scales polynomially with the input size n, but arbitrary with the parameter w (even exponentially or worse). If w is fixed (i.e. $f(w)$ is a constant), the resulting complexity is polynomial. As a consequence, also many problems in NP and PSPACE can be solved in FPT. Courcelle's Theorem [4] states that problems that can be defined in monadic second-order logic (MSO) are fixed-parameter tractable with respect to a bounded treewidth.

The treewidth is defined on tree decompositions, that provide, roughly speaking, a mapping from the instance graph to a tree. As a consequence, each tree decomposition node includes multiple vertices from the instance graph. The challenge of building tree decomposition nodes lies in the distribution of the vertices in a way that cyclic components of the graph are broken up.

Figure 2.1 illustrates the transformation of an instance graph to several possible tree decompositions. As one can see, there are multiple normalization types that have an influence on the decomposition algorithms. Tree decompositions with the particular normalization types are defined as follows.

Definition 1. *A* tree decomposition *of a graph $G = (V, E)$ is a pair (T, χ) where $T = (N, F)$ is a (rooted) tree and $\chi : N \to 2^V$ assigns to each node a set of vertices (called the node's* bag*), such that the following conditions are met:*

1. *For every vertex $v \in V$, there exists a node $n \in N$ such that $v \in \chi(n)$.*
2. *For every edge $e \in E$, there exists a node $n \in N$ such that $e \subseteq \chi(n)$.*
3. *For every $v \in V$, the subtree of T induced by $\{n \in N \mid v \in \chi(n)\}$ is connected.*

We call $\max_{n \in N} |\chi(n)| - 1$ *the* width *of the decomposition. The* treewidth *of a graph is the minimum width over all its tree decompositions.*

Condition 1 and 2 ensure that each vertex and edge from the instance graph appear at least once in the bags of the tree decomposition nodes. Condition 3 guarantees that all tree decomposition nodes that contains a particular vertex are connected. Lets take **b** from each bag of Figure 2.1 and lets call these bags **b**-bags. As one can see no bag that contains **b** is disconnected from the subtree that only consists of **b**-bags.

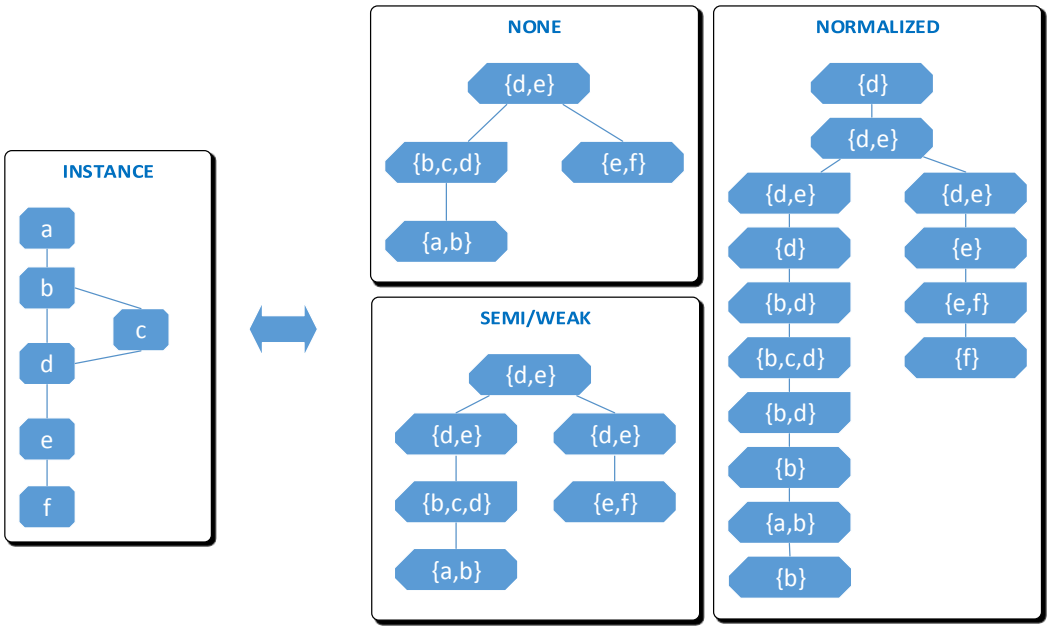


Figure 2.1: Creation of tree decomposition

Furthermore, lets determine the width of the tree decompositions and the treewidth of the graph from Figure 2.1. Since the maximum bag size of the **NONE**-decomposition nodes is 3 (**{b,c,d}**), the width of this tree decomposition is 2 (also the width of the **SEMI**, **WEAK** and **NORMALIZED** decomposition is 2). Since no further decompositions exist with width lower than 2, 2 is also the treewidth of the instance graph.

We distinguish between multiple tree decomposition types. This section introduces four tree decomposition approaches, also called normalization types. Figure 2.1 depicts examples for the normalization types **NONE**, **WEAK**, **SEMI** and **NORMALIZED** that are defined as follows.

In general, the normalization type of tree decompositions as given in Definition 1 is called **NONE**. Further tree decomposition normalization types fulfill the conditions from **NONE** and additional constraints.

Definition 2. *A tree decomposition $T = (N, F)$ is called* WEAK *if each $t \in N$ is of one of the following types:*

- *Leaf node: t has no child nodes.*
- *Exchange node: t has one child node.*
- *Join nodes: t has an arbitrary amount n of child nodes t_1 to t_n with $\chi(t) = \chi(t_1) = \ldots = \chi(t_n)$.*

Definition 2 specifies the conditions for the node types of **WEAK** tree decompositions. A join node defines a particular node type having more than one child nodes. Furthermore, each child bag contains the same vertices as the parent bag.

The **WEAK** tree decomposition in Figure 2.1 meets the conditions defined above. Furthermore, Figure 2.1 represents a sample **WEAK** tree decomposition that is equal to the **SEMI** tree decomposition (see below). Since **{a,b}** and **{e,f}** have no children, the condition for leaf nodes holds. Moreover, the exchange nodes **{d,e}**(two times) and **{b,c,d}** have only one child and the join node **{d,e}** has two children with the same bag(**{d,e}**). Since join nodes in **WEAK** tree decompositions allow more than one child node, **{d,e}** could also appear more than twice as child node of the root node.

Definition 3. *A tree decomposition* $T = (N, F)$ *is called* SEMI *if*

- *the tree decomposition is WEAK and*
- *Join nodes t have exactly two child nodes* t_1 *and* t_2 *with* $\chi(t) = \chi(t_1) = \chi(t_2)$.

Definition 3 states the conditions for the node types of **SEMI** tree decompositions and extends the constraints for **WEAK** tree decompositions. The difference to **WEAK** tree decompositions is, that a join node has exactly two child nodes instead of more than one child node.

Since **{d,e}** has exactly two child nodes, the **SEMI** tree decomposition in Figure 2.1 meets the conditions defined above. The explanations for the other conditions can be found above.

Definition 4. *A tree decomposition* $T = (N, F)$ *is called* NORMALIZED *if*

- *the tree decomposition is SEMI and*
- *each Exchange node is of one of the following types:*
  - *Introduction node: t has exactly one child node* t_1 *with* $\chi(t_1) \subset \chi(t)$ *and* $|\chi(t_1)| = |\chi(t)| - 1$.
  - *Removal node: t has exactly one child node* t_1 *with* $\chi(t) \subset \chi(t_1)$ *and* $|\chi(t_1)| = |\chi(t)| + 1$.

As Definition 4 shows, **NORMALIZED** tree decompositions follow the conditions from **SEMI**, **WEAK**, **NONE** and further constraints. The conditions for introduction node and removal node are equivalent to the requirement that the bag of the parent and the child node only differs in one vertex. The exception for this rule is the join node condition, where in case of two children the child bags have to be equivalent to the parent bag.

The **NORMALIZED** tree decomposition in Figure 2.1 meets the conditions defined above. If we check the tree from the root node **{d}** down to the leaf nodes **{b}** and **{f}**, no bags, except for the join node bag **{d,e}**, where the child bags contain the same vertices from the parent bag, differs in exactly one vertex to its parent bag.

Finding a tree decomposition with optimal width is an NP-hard problem. Several algorithms (see, e.g. [13]) exist for the generation of tree decompositions. These algorithms use heuristics based on good elimination ordering of graph vertices. Three possible heuristics for creating good elimination orderings are described in the following:

- The first heuristic picks the initial vertex in the ordering randomly from the graph. The vertex with the highest connectivity to the previously selected vertices is selected next. This last step is repeated until each vertex of the graph occurs in the ordering.
- The second heuristic selects the vertex with minimum edge count as initial vertex for the ordering. The vertex is removed from the graph and all its neighbors are connected (to form a clique). The following vertices in the ordering are picked in the same way until all vertices are eliminated from the graph.
- The last heuristic selects the vertex with the minimum degree as initial vertex for the ordering. Next, the vertex with the minimum degree (only edges to unselected neighbors are counted) is picked. This process is repeated until all vertices are selected.

2.3 Dynamic Programming on Tree Decompositions

This section describes the necessary process for solving NP-hard problems efficiently by means of dynamic programming on tree decompositions. The general four steps for DP on tree decompositions are defined with the help of an example. Furthermore, two different paradigms regarding the dynamic programming algorithm are described in this section.

As already mentioned above, the concept of solving a problem by means of dynamic programming on tree decompositions can be splitted into four steps:

1. Represent the input instance as graph: This is obsolete for problems such as 3-Colorability or Hamiltonian path, because their instances are already represented as graphs. Problems like Boolean Satisfiability have to be first mapped onto an appropriate graph. Figure 2.1 illustrates a possible instance used in the following steps.
2. Decompose the instance graph: As described in detail in Section 2.2, this step defines the generation of a tree with the help of the input instance. The left graph in Figure 2.2 depicts a sample tree decomposition created from the instance shown in Figure 2.1.
3. Traverse the tree decomposition: Figure 2.2 represents the computation of a 3-Colorability problem, where the tree is traversed in bottom-up order and the results of each tree decomposition node is stored in a computation table. Each row of the table stands for a possible coloring of the bag elements. Furthermore, each row

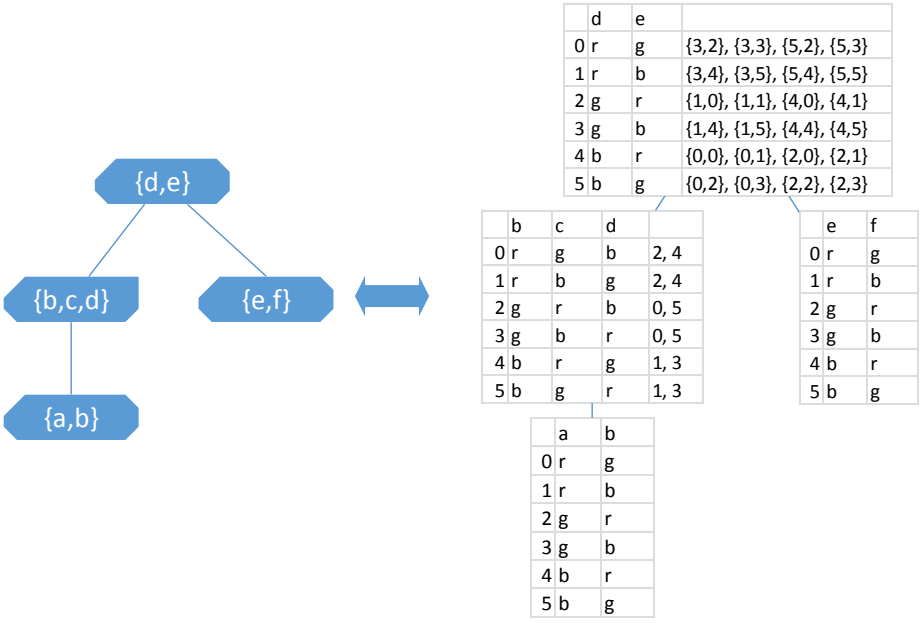


Figure 2.2: Dynamic programming on tree decompositions

contains an additional index used as reference for further processing and results of the tree decomposition nodes above. In general, the computation starts bottom-up, such that the leaf nodes (**{a,b}** and **{e,f}**) consist of all possible coloring variants for their bag elements. First, the possible colorings for **{a,b}** are generated. Next, the solutions for **{b,c,d}** are computed. As one can see, the solutions depend on the coloring of the child node **{a,b}**. The last column contains the possible references to the colorings (rows) of the child node(s). For example, the assignments from the first row (**b=r, c=g, d=b**) can be combined either with **a=g** or **a=b**. The next node to be resolved is **{e,f}**. The join node (and root node) **{d,e}** consists of the possible colorings for **{d}** and **{e}** depending on the colorings from **b=r, c=g, d=b** and **{d,e}**.

4. Obtain the solutions: At the end, the root node contains all information to obtain the overall solution to the problem. In Figure 2.2 **{d,e}** consists of the possible colorings for the instance graph of the 3-Colorability problem. Each solution has to be resolved by means of the coloring(s) inside the table(s) and the reference(s) to the child table(s). As one can see, each row consists of multiple possible colorings. The first possible coloring can be defined as follows: **d=r, e=g, b=g, c=b, a=r, f=r**.

These four steps provide the basis for applying dynamic programming on tree decom-

positions. We distinguish between two paradigms regarding the dynamic programming algorithm that differ in the time of "fixing information" during traversal of the tree [3]:

1. ***Early Decision Method (EDM)***:
   When the tree is traversed bottom-up, all information gathered during this process is saved immediately, i.e. every information is incorporated as early as possible. *EDM* is comparable with the strategy used for the example in Figure 2.2 (and in the four steps defined above). If the instance is unsatisfiable, then the *EDM* paradigm may detect conflicts at an early stage.

2. ***Late Decision Method (LDM)***:
   In contrast to the previous paradigm, every information is incorporated as late as possible, i.e. a vertex is not considered for solutions of tree decomposition nodes when traversing the tree bottom-up, until the vertex is going to be removed. Hence, computation operations are delayed as long as possible. This provides advantages when using algorithms which are more involved. But nevertheless, for unsatisfiable instances this may lead to delayed detection of conflicts.

One possible approach for solving problems is BDD-based dynamic programming on tree decompositions. Instead of using computation tables or other structures for the resolution of the tree decomposition nodes, BDDs (Binary Decision Diagrams) are generated as solutions for tree decomposition nodes and as basis for further BDD generation in the parent nodes. More details about BDDs can be found in the following Section 2.4.

The *LDM* approach is generally more difficult to implement on table implementations than when using BDDs. Figure 2.3 represents the instance and tree decomposition from Figure 2.2 solved with the two BDD-based dynamic programming paradigms *EDM* and *LDM*. Since *EDM* incorporates all informations received when traversing the tree decomposition bottom-up, the resulting BDDs are mostly large and complex. As one can see, each corresponding BDD to its tree decomposition node consists of all its bag-vertices with the possible coloring variants. In contrast to the *EDM* solution, the BDDs generated by *LDM* are quite small, because BDDs are only manipulated when vertices are removed from the bag. For example, the BDD, that corresponds to node **{b,c,d}**, only consists of BDD nodes including vertex **b**.

2.4 Binary Decision Diagrams

A Binary Decision Diagram (BDD) [14, 15, 16] is illustrated by a rooted directed acyclic graph consisting of decision nodes and two leaf nodes (see Figure 2.4). A BDD is used as data structure representing Boolean functions.

Definition 5. *A BDD (Binary Decision Diagram)* $B = (V_B, A_B)$ *is a rooted directed acyclic graph where* $V_B = V_T \cup V_N$ *and* $A_B = A_\top \cup A_\bot$. *The following conditions have to be satisfied:*

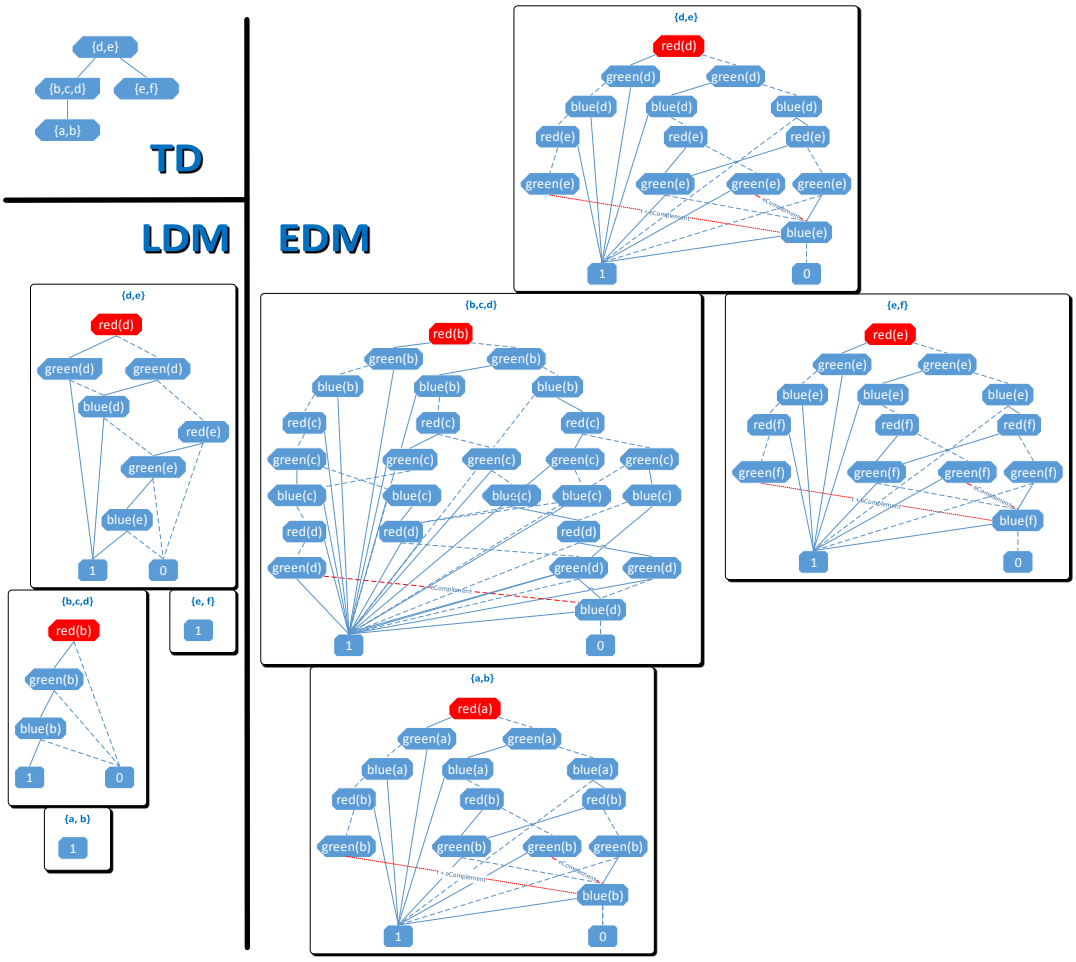


Figure 2.3: Computations as BDDs

- *V_T consists of exactly two terminal nodes $v_\top$ and $v_\bot$.*
- *V_N contains the decision nodes, where each $v \in V_N$ represents a variable v.*
- *Each $v \in V_N$ has exactly one outgoing arc in $A_\top$ and one in $A_\bot$.*

Each decision node represents a variable and the edges to the two children, called THEN-child and ELSE-child, represent the variable assignment. Hence, the edge to the THEN-child represents a variable assignment of 1 (TRUE), the edge to the ELSE-child stands for a variable assignment of 0 (FALSE). To make these two types of edges visually distinguishable, the edge to the THEN-child is usually drawn by a solid line and the edge to the ELSE-child with a dashed line. The two leaf nodes, also called terminal nodes, represent the TRUE (1-terminal) and the FALSE (0-terminal) value of the Boolean function. In general, all paths (assignments) from the root node to the 1-terminal

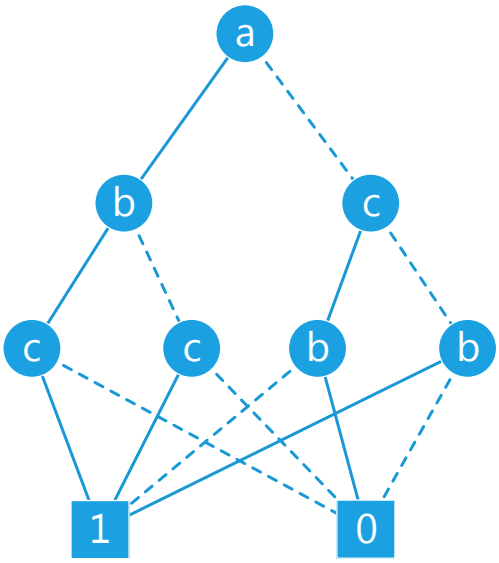


Figure 2.4: Binary Decision Diagram (BDD)

represent models of the formula represented by the BDD. A BDD has multiple levels. The root node has always level 1. With each assignment, we reach a new level, thus incremented by 1. Hence, the leaf nodes have always the highest level because there are no other new assignments below them. There exist various types of BDDs as defined in the following.

An Ordered Binary Decision Diagram (OBDD) [17, 18] is a special form of a BDD, where the nodes and edges are reordered in such a way that the paths match a predefined ordering of the variables (see Figure 2.5).

Definition 6. *An OBDD (Ordered Binary Decision Diagram)* $B = (V_B, A_B)$ *extends a BDD with the following condition:*

- *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).*

As we see, an OBDD can simplify the BDD because the assignments of all existing path from the root to terminal have the same order. Thus every assignment is done at a particular level independent of the path.

Now we want to describe the term *variable order*. Each level has a particular variable assignment. The variable order determines the order in which these variables are or can be assigned. Hence, it implicitly determines the order and the hierarchy of the levels.

The OBDD can now be further processed to a so-called Reduced Ordered Binary Decision Diagram (ROBDD) [16, 19, 20], to get a smaller BDD, but still containing the same information as the Boolean function (see Figure 2.6).

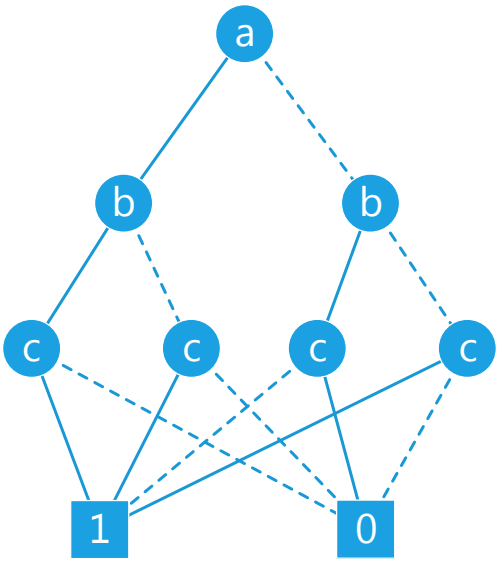


Figure 2.5: Ordered Binary Decision Diagram (OBDD)

Definition 7. *An ROBDD (Reduced Ordered Binary Decision Diagram) extends the OBDD with the following conditions:*

- *Isomorphic nodes are merged into a single node with several incoming edges.*
- *Nodes $v \in V_N$ where both outgoing arcs reach the same node $v' \in V_N$, are removed.*

As we can see from the definition of a ROBDD, the OBDD can be usually reduced by combining all the nodes which are redundant regarding the assignment information. Hence, isomorphic nodes are merged and thus the BDD can be very compact. By doing this reduction, there is also another possibility to shrink the BDD. If the THEN-edge and the ELSE-edge point to the same node, then these edges inclusive their start node can be omitted, since in this case the assignment can have both values, TRUE or FALSE, without having any effect on the represented models. Of course, this can occur over multiple levels too. Then the edge can directly point to the first node, where such assignments do not occur any longer. This means that edges can skip one or multiple levels. Even though the BDD does not explicitly show this assignment, it is still done in the background. In detail each skipped level doubles the results, one result which has a variable assignment of TRUE, and another one which has an assignment of FALSE.

The BDD can be further compressed by using the concept of *complement nodes* and *complement edges* as described below:

- ***Complement nodes***:
  As already mentioned above, generally all models are represented by the assignments

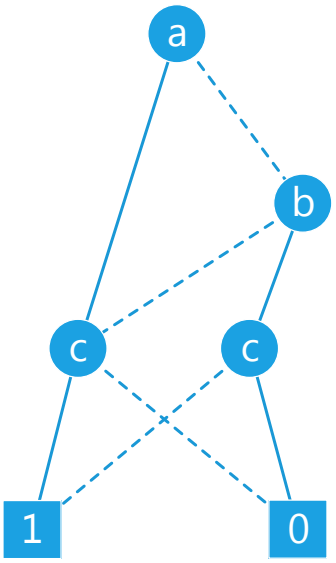


Figure 2.6: Reduced Ordered Binary Decision Diagram (ROBDD)

across the edges to the 1-terminal. If we use so-called complement nodes in the BDD, then the result of the Boolean function is flipped every time when such a complement node is traversed. This means, that even if the path ends in the 0-terminal, the real result of the Boolean function can be true if an odd number of complement nodes are included in the path. To make a complement visually distinguishable from all the other nodes, the node has a red background. As we see in Figure 2.7, the variable assignment with $a = false, b = true, c = false$ results in the 0-terminal. But since we come across a complement node, the result has to be flipped and in fact it is a model. In our example, the introduction of the complement node does not compress the BDD.

- ***Complement edges***:
  Similar to complement nodes, complement edges also flip the result of the Boolean function if they are contained in a path. But in contrast to a complement node, which would always have an effect on both outgoing edges, a complement edge only affects the edge itself and the further path. This is useful when a node has many incoming edges and the complement meaning should only be applied to a particular incoming edge. The edge is drawn using a red color. As we see in Figure 2.8, the use of a complement node reduces the size of the BDD because an already existing part of the BDD can be used again since it represents exactly the complementary structure. Hence, this complementary structure in combination with the contrary meaning of the complement edge results in the original Boolean function. Therefore a node together with its edges can be removed.

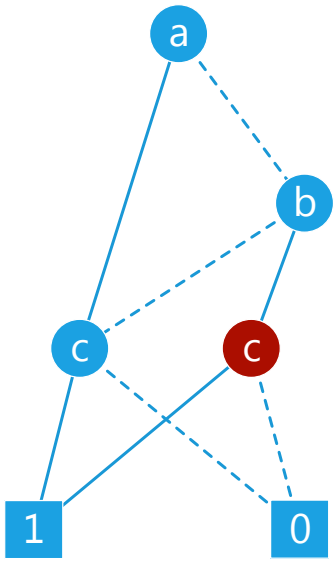


Figure 2.7: ROBDD with complement node

2.5 Dynamic Programming Systems

In this section we present some systems for dynamic programming on tree decompositions in more detail. They will be used later on when doing the benchmark tests, hence it is necessary to understand the basics of these tools. Each tool uses different methods to implement dynamic programming. We describe the present systems and give some theoretical background. We discuss the differences, the advantages and also the disadvantages. It is important to say that all mentioned information about the tools will be quite theoretical for now because it is one of the main contributions of the master thesis to verify the stated characteristics by executing benchmark tests and analyzing the results in Chapter 5 and Chapter 6. In this chapter we refer to already existing benchmark comparisons. We will then see later on whether we can verify the existing comparisons and the statements made in the related papers of the dynamic programming systems.

2.5.1 D-FLAT

D-FLAT [6, 21, 7] is a system for dynamic programming on tree decomposition in combination with Answer Set Programming (ASP) [22], meaning that all the algorithms are defined in the declarative language ASP. The advantage of this approach is that all specified algorithms are solid and clear. The *D-FLAT* software uses libraries like *Gringo* [23] and *Clasp* [24] for answer set solving and *htdecomp* [25] to generate tree decompositions of the instances. It supports all problems which can be defined in monadic second-order logic (MSO) to be solved in FPT, therefore it is well-suited for the most

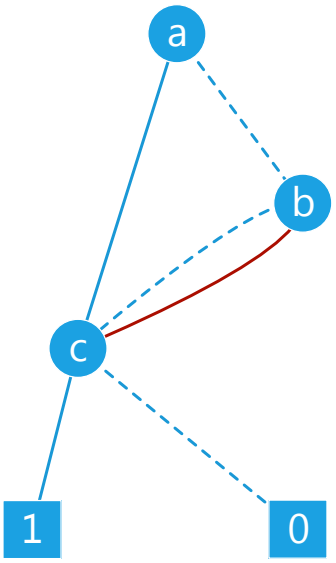


Figure 2.8: ROBDD with complement edge

use cases. Other problems are still supported but may not be solvable in FPT. In detail, *D-FLAT* uses the following approach:

1. First the problem instance is parsed which is defined by a set of facts in ASP. Then an internally constructed hypergraph is generated.

2. Now a tree decomposition is constructed. To get a small tree width of the tree decompositions, which is necessary for an efficient solving of the problems, different heuristics (see Section 2.2) are provided by the external library *htdecomp*. Furthermore, once the tree decomposition is generated, there are different ways how to normalize the tree, but this process is only optional. As already mentioned in Section 2.2 normalization is often important to make the specification of algorithms easier because less cases have to be taken into account. It supports the normalization types *WEAK*, *SEMI*, *NORMALIZED* and *NONE*.

3. The next step is to calculate the partial solutions by using a data structure called *item tree*. Every node of the tree decomposition is associated with an *item tree*. This calculation is done by means of the ASP algorithm defined by the user (an example can be seen in Listing 2.2 and Listing 2.3). The *item tree* is a tree which contains for each node a set of ground ASP terms, called the *item set*. Each node has a particular type (*or*, *and*, *accept*, *reject*) and it also includes the information how the complete solution is built. Each branch in this *item tree* represents a computation sequence. The leaf nodes have either *accept* or *reject* as node type, whereas all non-leaf nodes have either *and* or *or*. In addition, the user is able to

switch to a table mode, where the tree depth is just 1. Then the root node is defined as *or* node with an empty *item set* and all children represent a row in the table. They all have the node type *accept*. It is important to say that in most cases the table mode is used since it is usually sufficient for problems in NP.

4. The last step is to merge all these partial solutions and then output the results, meaning the complete solutions are calculated (see Section 2.3). In this way *D-FLAT* is able to solve all the problems in the solving time FPT (fixed-parameter tractable, see Section 2.2).

2.5.2 D-FLAT^2

A variant of *D-FLAT* is called *D-FLAT^2* [8]. This tool eases the development of dynamic programming algorithms where subset minimization is required. Usually the user has to define recurring patterns in each dynamic programming algorithm to execute the subset minimization or maximization. In *D-FLAT^2* this is no longer required since the tasks like subset minimization are done automatically here. Therefore, the dynamic programming algorithms defined as ASP code by the user are much simpler and easier to obtain. Furthermore this leads to better maintainability and an increased readability since much less code has to be written. There is even another advantage by letting the system run the subset minimization task automatically. Since the code is now directly implemented in the software, this leads to an increased performance regarding runtime and memory. Nevertheless, it is important to say that the user still has to define some extra code in ASP to let the software know what should be minimized exactly. But these statements are very simple, in contrast to the definition of a completely own algorithm for the subset minimization task from scratch.

The *D-FLAT^2* tool executes the dynamic programming algorithm in two stages. In the first stage, the tool calculates all the solution candidates without any optimization and minimization procedures. Only in the next step the required optimization regarding subset minimization is done by searching for counterexamples. This leads to faster computations of the final solutions and also requires less memory in contrast to the naive algorithms where the user-defined dynamic programming code including optimizations is executed in one single stage.

The procedure of computing the overall solution is quite similar to *D-FLAT*. But it uses so-called *reduced item trees* as data structure instead of *item trees*. *Reduced item trees* have always a depth of 1 and store additional information for the minimization, called *minimization item set*. This item set contains all the items where minimization should be applied.

There are already some benchmark tests, where the tool is compared with two other tools: *D-FLAT* and *ASPARTIX* [26]. The latter is a program to generate extensions of argumentation frameworks directly with ASP [27]. The used instances are grid-based, meaning that the nodes are arranged in a matrix and the nodes are connected to all the neighbours, and contain 40 to 65 nodes. The used problem is the enumeration of

all preferred extensions, a problem from the area of abstract argumentation, with a treewidth of 4. As seen in the comparison in Figure 2.9, *D-FLAT^2* has the best results regarding execution time. *D-FLAT* is a little bit slower and *ASPARTIX* takes much more time for the computation. When we compare the used memory, then we see that *D-FLAT^2* is still the leader and requires much less memory than the other two tools. But *ASPARTIX* has better results and requires less memory than *D-FLAT* in this case.

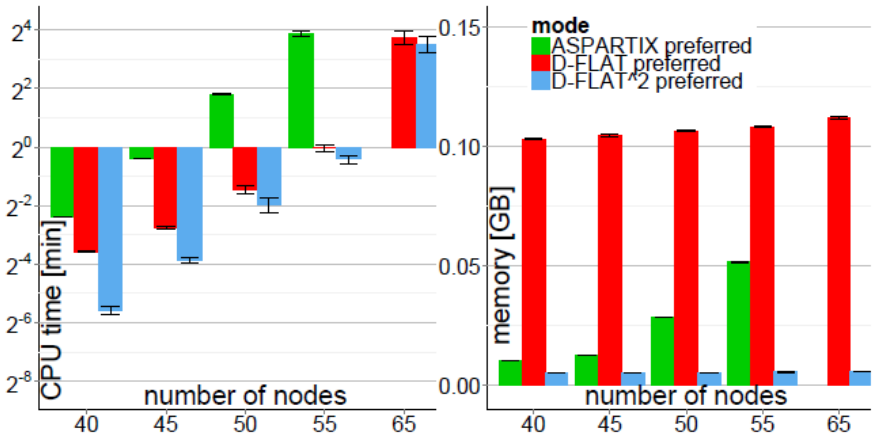


Figure 2.9: System comparison with *D-FLAT^2* [8]

When comparing different problems between *D-FLAT^2* and *D-FLAT*, for example admissible, preferred and semi-stable sets (see [8] for details), the results are varying. For admissible sets *D-FLAT^2* is a little bit slower and requires also slightly more space because of the overhead of using *reduced item trees*. However, for this problem, minimization is not required. Nevertheless, the overhead can be neglected in fact. For the other problems *D-FLAT^2* always shows the best results in time and space. Therefore, the two-stage algorithm seems to be very effective.

2.5.3 Sequoia

Like *D-FLAT*, *Sequoia* [4, 5, 28] is also a system which solves problems on graphs having a small treewidth. The main difference to the other approach is that the user does not have to specify any dynamic programming algorithm. Instead only the problem has to be provided as encoded MSO formula (an example can be seen in Listing 2.4). As a result, the system produces a tree decomposition and the MSO formula is automatically evaluated on basis of the tree decomposition.

The *Sequoia* tool is based on the game-theoretic proof of Courcelle's Theorem (discussed in Section 2.2). The only requirement is that a data structure with bounded tree-width is used. This can be done by generating and using a tree automaton of constant size which accepts or rejects the particular tree decompositions. The only problem is that the required constants may take much memory space because a power set construction

```
length(1). or(0).
1 { item(1,map(X,red;X,grn;X,blu)) } 1 ← current(X).
reject ← edge(X,Y), item(1,map(X,C;Y,C)).
extend(0,S) ← root(S).
1 { extend(1,S) : sub(R,S) } 1 ← root(R).
← item(1,map(X,C1)), childItem(S,map(X,C2)), extend(_,S), C1 ≠ C2.
reject ← childReject(S), extend(_,S).
accept ← final, not reject.
```

Listing 2.2: 3-Colorability algorithm in *D-FLAT*

```
length(2). or(0). and(1).
1 { item(L,map(X,red;X,grn;X,blu)) } 1 ← current(X), L=1..2.
← edge(X,Y), item(L,map(X,C;Y,C)).
← item(2,map(V,red)), not item(1,map(V,red)).
extend(0,S) ← root(S).
1 { extend(L+1,S) : sub(R,S) } 1 ← extend(L,R), L = 0..1.
← item(L,map(X,C1)), childItem(S,map(X,C2)), extend(L,S), C1 ≠ C2.
item(2,fail) ← item(1,map(V,red)), not item(2,map(V,red)).
item(2,fail) ← extend(2,S), childItem(S,fail).
reject ← final, item(2,fail).
accept ← final, not reject.
```

Listing 2.3: Subset-minimization variant of 3-Colorability in *D-FLAT*

```
ThreeCol(R, B) :=
        All x (
                (x notin R or x notin B)
            and
                All y (
                    ~adj(x,y) or (
                        (x notin R or y notin R) and
                        (x notin B or y notin B) and
                        ((x in R) or (x in B)
                                                or
                         (y in R) or (y in B))
                    )
                )
        )
```

Listing 2.4: 3-Colorability algorithm in *Sequoia*

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

Figure 2.10: 3-Colorability EDM algorithm in *dynBDD*

is needed for every quantifier which results in problems in real world applications. The number of these constants depends on the MSO formula and the treewidth.

Sequoia wants to eliminate this memory explosion of the power set construction and introduced new approaches, for example by computing the transition of the automaton states on-the-fly instead of constructing the whole tree automation explicitly. Another approach is to consider the input structure and store only the data information which the algorithm needs for this particular instance. This way also the transitions in the tree decomposition would only be constructed when they are really needed.

By applying these approaches the *Sequoia* software is able to solve problems like the *3-Colorability* or *Minimum Vertex Cover* more efficiently and also preliminary experiments with similar problems seem very promising.

2.5.4 DynBDD

As mentioned before, the *D-FLAT* tool usually uses tables as data structure implemented by simplified *item trees* with depth 1. But the disadvantage of simple tables is that they require much memory in practice. By implementing heuristics or by using a reduced number of concurrent tables, the required memory usage can be reduced. But nevertheless, the dynamic programming algorithms have to be adjusted and implemented for a particular problem.

The *dynBDD* software [3] uses BDDs as data structure instead of relying on simple tables. In general, the software is trying to use the memory more efficiently and to reduce the overall calculation time. The concept of BDDs is well-suited to many different areas and is already used in the software verification or model-checking, for example.

The process of the computation is similar to *D-FLAT*. The *dynBDD* tool also provides the ability to specify the algorithms on a logical level, but here an algorithm is defined by Boolean formulas (see Figure 2.10 and Figure 2.11). Another difference is that the algorithms modify the models concurrently in the BDD when they are executed instead of only a single model like in table approach where only a single tuple is modified at once.

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

Figure 2.11: 3-Colorability LDM algorithm in *dynBDD*

The *dynBDD* tool relies on two libraries: *CUDD* [29] is used for an efficient BDD management and *htdecomp* [25] is required for generating the tree decompositions.

There are already done some tests and experiments regarding the effectiveness of the *dynBDD* tool in contrast to other tools in this area like *D-FLAT*, *SEQUOIA* and *dynPARTIX* [30]. Three different types of problems were tested: 3-Colorability, Stable Extension and Hamiltonian Cycle.

As the results of this experiments in Figure 2.12 show, *dynBDD* is the leader in all the tests done so far, both in memory usage and time. It is important to say that only decision problems are considered.

2.6 Visualization Concepts

To visualize some type of data such as graphs, metadata or other information, a visualization framework or library is necessary since a development of visualization algorithms and elements from scratch is not recommended and also not a focus of this master thesis. Depending on the software and the use case, there are many libraries and frameworks available for client-based tools (e.g. in Java and C++) or also server-client architectures for web-based applications (usually available for JavaScript). Furthermore, many window manager concepts and window management systems are available. It is recommended to provide an efficient user interface and to handle many concurrently opened windows fully automatically in such a way that manual actions from the user side are not really necessary or at least only with a minimum of effort.

2.6.1 A Window System

The terms *window system* and *window manager* usually describe a part of a system which is responsible for the appearance, the layout and placement of windows in a graphical user interface [31, 32, 33, 34]. It provides graphical elements like to open, close, minimize and maximize the window. Furthermore the ability to move a window or to change its size is also a main feature of a window system. Thereby, multiple windows can overlap each other or can protrude outside the visible screen. A window usually also has a title

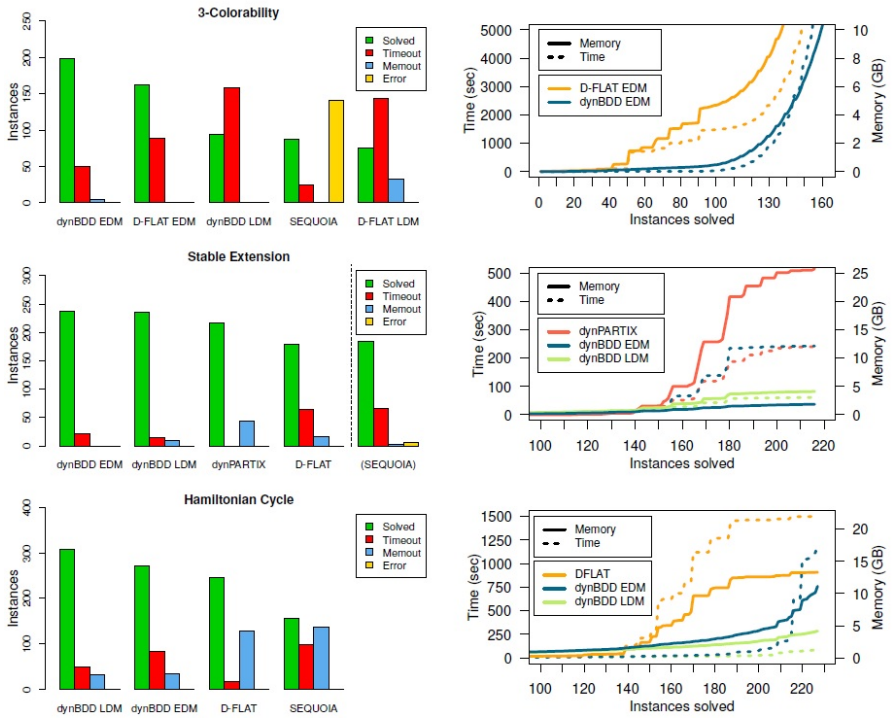


Figure 2.12: System comparison with *dynBDD* [3]

bar and an icon at some certain position like at the top of the window. At the bottom there is often a status bar located. Furthermore the overall window has some certain style with or without a border and different colors. To get an overview of all running windows and to provide some additional features, a task bar or a window bar is required which contains all currently running windows as a small icon or text or a combination of both. Some window systems also offer a docking feature to dock a window at a particular position. In order to simplify the placement of the opened windows, a feature for an automatic reordering can be available so that the user does not have to spent time on organizing, resizing and reordering windows.

In addition to the usual window manager and the user interface elements provided by the operating system, there are also already many window manager and libraries available for web-based systems, such as:

1. ***jQuery UI*** [35]: This is a JavaScript library to implement user interface elements, effects, widgets and interactions. It is based on jQuery [36] and also provides some window elements which can be combined with resizeable and dragable features.

2. ***gridster.js*** [37]: A JavaScript library which provides an interactive grid-based layout for elements.

3. ***OS.js*** [38]: An implementation of a completely web-based desktop in JavaScript.

4. ***Venuts*** [39]: Another implementation of a web-based desktop in JavaScript. In addition a mode is available to see all windows at a glance.

2.6.2 Graph Libraries and Frameworks

A small overview and a part of available libraries, frameworks and concepts regarding graph visualization provides the enumeration below. To provide a solution for visualizations independent of the underlying operating system, only libraries and frameworks for Java (standalone applications) and JavaScript (web-based applications) are considered since they can be run on nearly every operating system.

1. ***JUNG*** [40]
   JUNG stands for *Java Universal Network/Graph Framework* and is a pure JAVA library providing the modeling, visualization and analyzing of graphs and networks. Different types of graphs can be used such as hypergraphs, directed/undirected graphs and multi-modal graphs. Furthermore metadata information can be added to the graph elements.

2. ***jGraphT / jGraph*** [41, 42]
   The Java library *jGraphT* in combination with *jGraph* supports quite the same variety of graph types as the previous one. The *jGraphT* library has a focus on algorithms and the data structure, whereas *jGraph* is responsible for the visualization, the rendering and the GUI. But in contrast to *JUNG*, it is more suitable for creating and editing static graphs and diagrams instead of interactive and dynamic visualizations.

3. ***GraphStream*** [43]
   GraphStream is also a Java library but in contrast to previous libraries, it provides a more dynamic graph representation. It is possible to import/export graphs and to create an evolution of the graph regarding time. This means, nodes and edges can be added or removed dynamically using many effects and layout styles suitable for an animation but without an interaction by the user.

4. ***arbor.js*** [44]
   The graph visualization library called *arbor.js* can be used to draw and handle graphs in JavaScript using different available layout algorithms. Hence, it is designed for web-based tools running directly in the browser. The library provides also some type of graph organization and uses jQuery and web workers to provide all its features.

5. ***sigmajs*** [45]
   The visualization library *sigmajs* is also designed for JavaScript and offers drawing of graphs similar to the previous library. But it is not as interactive as the *arborjs* because the nodes and edges are static and cannot be adjusted by the user. But nevertheless the graph can handle mouse events and multiple settings are available

to customize how the graph or network should be drawn. The library also provides a good documentation, tutorials and also a well-looking visualization in total.

6. ***Cytoscape.js*** [46]
   Cytoscape.js is another JavaScript-based library but with many more features. It provides a lot of different layout algorithms, graph visualizations and interactive networks. The drawn graphs and networks are very interactive. There are quite a lot of demos available which give a good overview of all the features provided by the library. In addition, the graphs can be visualized in many different ways, not only depending on the layout algorithm but also on the ground visualization concept by providing various visualization elements, styles and structures like in Figure 2.13. Also a tree structure is supported.

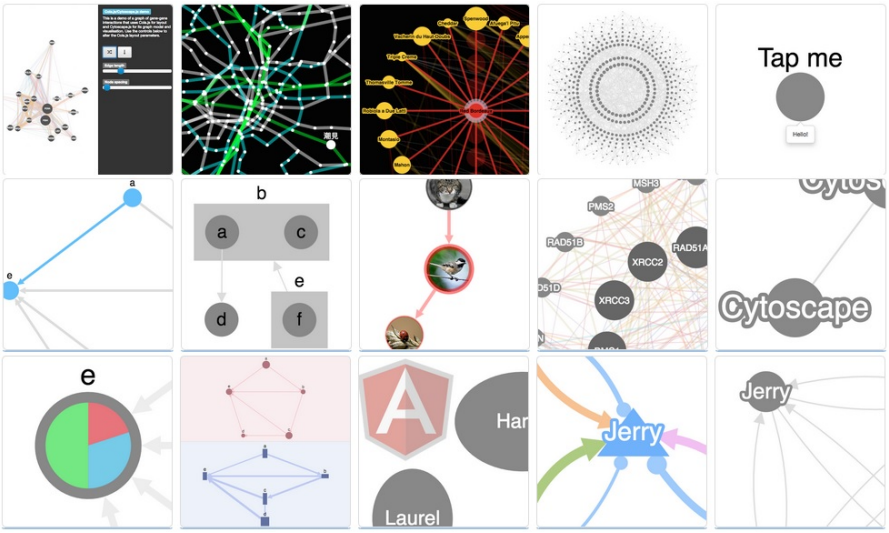


Figure 2.13: *Cytoscape.js* examples [46]

7. ***vis.js*** [47]
   Another JavaScript graph libraray is *vis.js* which provides also many features and settings. Probably it has slightly less visualization styles for networks compared to *Cytoscape.js* but it additionally provides 2D-based graphs like bar charts on an interactive timeline or even 3D-based graphs (see Figure 2.14). Hence, it provides additional visualization concepts beyond the usual network paradigm. Tree structures for the graph visualization are also supported. All the graphs and networks are created in a completely dynamical way and are very interactive in total. This means, the user can interact with the graph by moving the nodes and edges, by zooming in/out, by selecting nodes and edges, by triggering key/mouse events and much more.

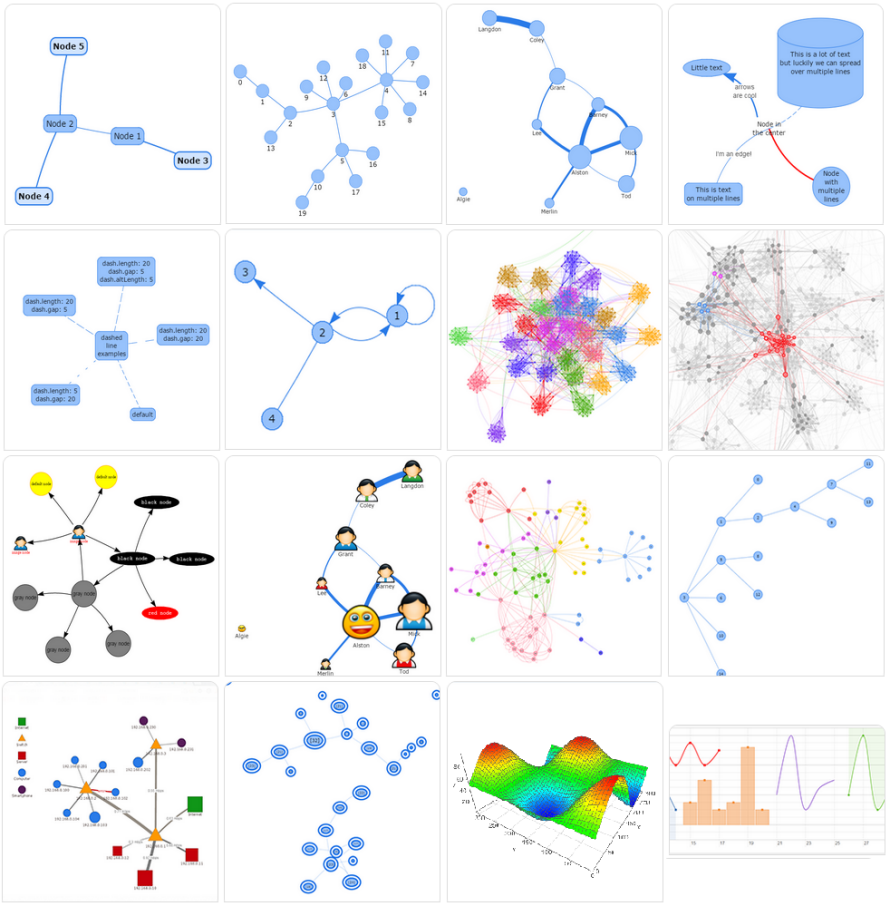


Figure 2.14: *Vis.js* examples [48, 49, 50]

2.6.3 Existing Visualizations in Algorithmic Design and Logic Programming

The following software applications give a small overview of visualization tools and concepts in the area of algorithmic design and logic programming:

1. ***D-FLAT Debugger*** [51]
   The tool *D-FLAT Debugger* has a focus on assisting and implementing new algorithms for dynamic programming in *D-FLAT* by visualizing and debugging the dynamic programming algorithms and their impact on the calculation of the solutions (see Figure 2.15 for an example).

2. ***clavis*** [52, 53]
   The software *clavis* is a visualization tool for the ASP solver *clasp*. With the help of a modified version of *clasp* a log file is created during the solving process.

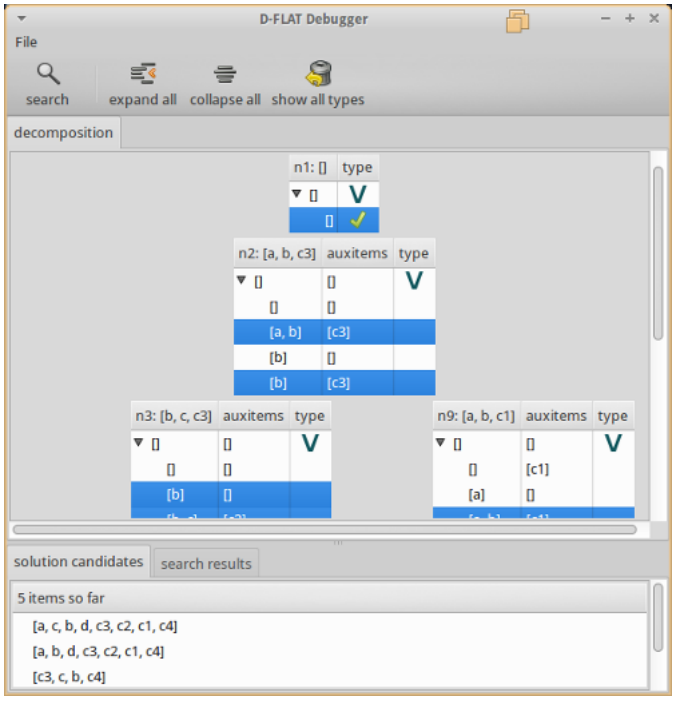


Figure 2.15: *D-FLAT Debugger* [51]

Afterwards the log files are analyzed and visualized by using structural and temporal views which give a detail insight into how the problem was solved (see Figure 2.16).

3. ***nomore*$^<$** [54, 55]
   Another system for profiling and visualizing answer set programming is called *nomore*$^<$. It reads an ordered logic program, generates an ordered rule dependence graph and then turns it into a colored graph. The colors depend on the preferred answer sets of ordered logic programs.

4. ***DeLP-Viewer*** [56]
   DeLP Viewer is a software to visualize another knowledge representation formalism called *Defeasible Logic Programming* (DeLP). In contrast to logic programming languages like *Prolog*, *DeLP* offers constructors for potentially contradictory and incomplete information. The *DeLP Viewer* explores the whole reasoning process and visualizes the inference and the relationships of the involved arguments by using a so-called *dialectical tree*, which is a tree-shaped structure (see Figure 2.17).

General visualization tools and universal applications which provide a debugging feature for dynamic programming are currently not available except for specific applications which support only a single tool like the *D-FLAT Debugger* which offers a debugging feature for *D-FLAT* only.

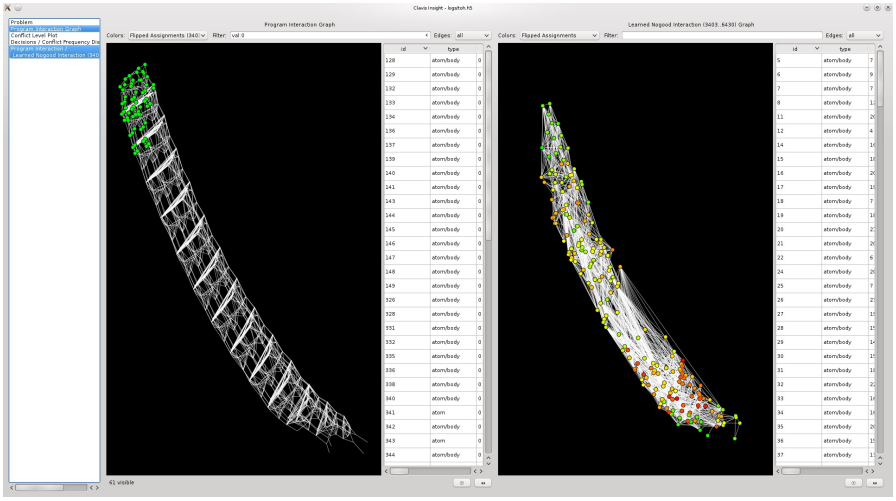


Figure 2.16: *Clavis* [52]

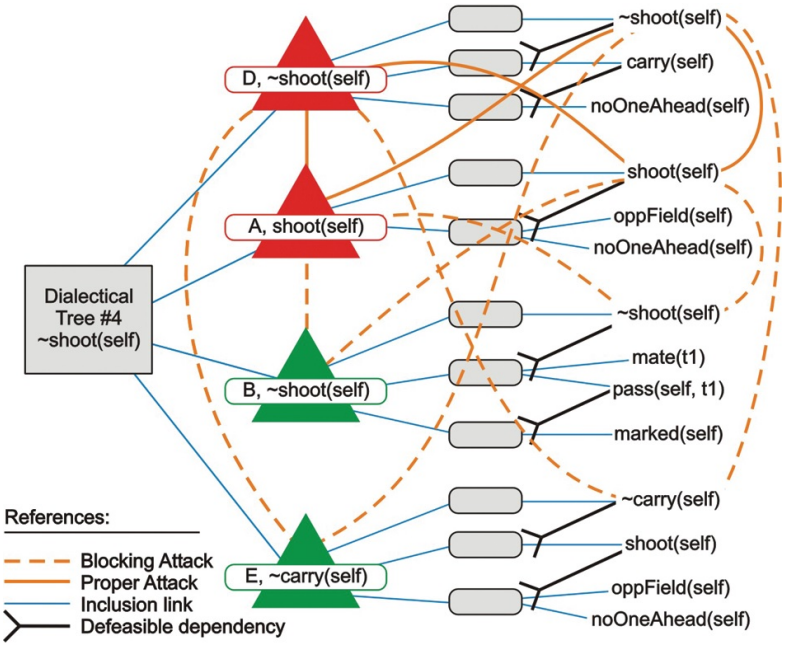


Figure 2.17: *DeLP-Viewer* [56]

CHAPTER 3

Technical Requirements

This chapter specifies the technical requirements for the development of a graphical visualization tool for dynamic programming on tree decompositions. Section 3.1 gives a high-level description and a small overview of the requirements. Section 3.2 deals with the enumeration of the essential technical requirements by means of a UML Use Case Diagram. Especially the following two modes have to be considered in the analysis. The expert mode provides the required functionalities for project generation and publication, while the user mode can be seen as read-only mode.

With the help of UML Activity Diagrams the workflow within the application is specified in Section 3.3. In particular this section emphasizes the possible work sequences for expert and user mode. Furthermore, the three main execution parts of the entire dynamic programming process should be clearly visible and pointed out: instances, tree decompositions and the solutions of the sub-problems, also called computations.

Section 3.4 deals with the definition of the structure of the three execution parts. On the basis of a UML Class Diagram necessary entities for instances, tree decompositions and computations are specified in detail. Moreover, the relations between these entities are defined.

The graphical analysis in Section 3.5 consists of the design of the user interface for the visualization tool. Beside the design of the three main components (instances, tree decompositions and computations) further interface plans for the main page, the window management or the statistic diagrams are developed.

3.1 High-Level Description

The goal of this chapter is to design and develop a tool that considers the following requirements:

- Visualize the instances of particular problems in a completely interactive and dynamical way.

- Compare different tree decompositions which were generated by various normalization modes and tree decomposition algorithms.
- Compare computations and visualize BDDs, supported by interactive visualization tools and different view modes.
- Understand the dynamic programming algorithms on tree decompositions with the help of interactive and graphical elements and visualization effects.
- Improve already existing algorithms on the basis of the previous requirement.
- Develop completely new algorithms by taking the results and understandings of the visualization of similar algorithms into account.
- All these features should be handled by a well-thought window management system to keep the overview of all the windows which are required during the process of visualizing, understanding and developing dynamic programming algorithms.

In summary, we have three layers: the instance, tree decomposition and computation layer. Each instance can have multiple tree decompositions (but at least one). Each tree decomposition can be used to run multiple computations.

3.2 Requirements Analysis

In software engineering, the requirements analysis forms the basis for developing professional and high-quality systems. The main goal consists of the identification, structuring and verification of customer requirements. The content of this section elaborates the abstraction of all specified requirements into a list of use cases. One use case defines one goal of an actor which consists of several actions and events.

In general two actors are identified: Expert and User. In opposition to the User actor, that possesses facilities for project visualization and analysis, the Expert actor owns (in addition to the User actor's facilities) capabilities for project generation and publication.

The UML Use Case Diagram in Figure 3.1 illustrates these two actors for the interaction with the system's use cases. In the following the use cases are described in detail.

3.2.1 Expert Use Cases

Generate tool output The expert user has to choose a suitable command line tool that uses dynamic programming algorithms on tree decompositions. Furthermore, this tool should have the capability for generating an output in form of files for the three categories instances, tree decompositions and computations. A configuration file offers the possibility of setting the path to that program.

In order to provide in the graphical user interface arbitrary options from the command line tool, the form fields displayed in the user interface representing these options have

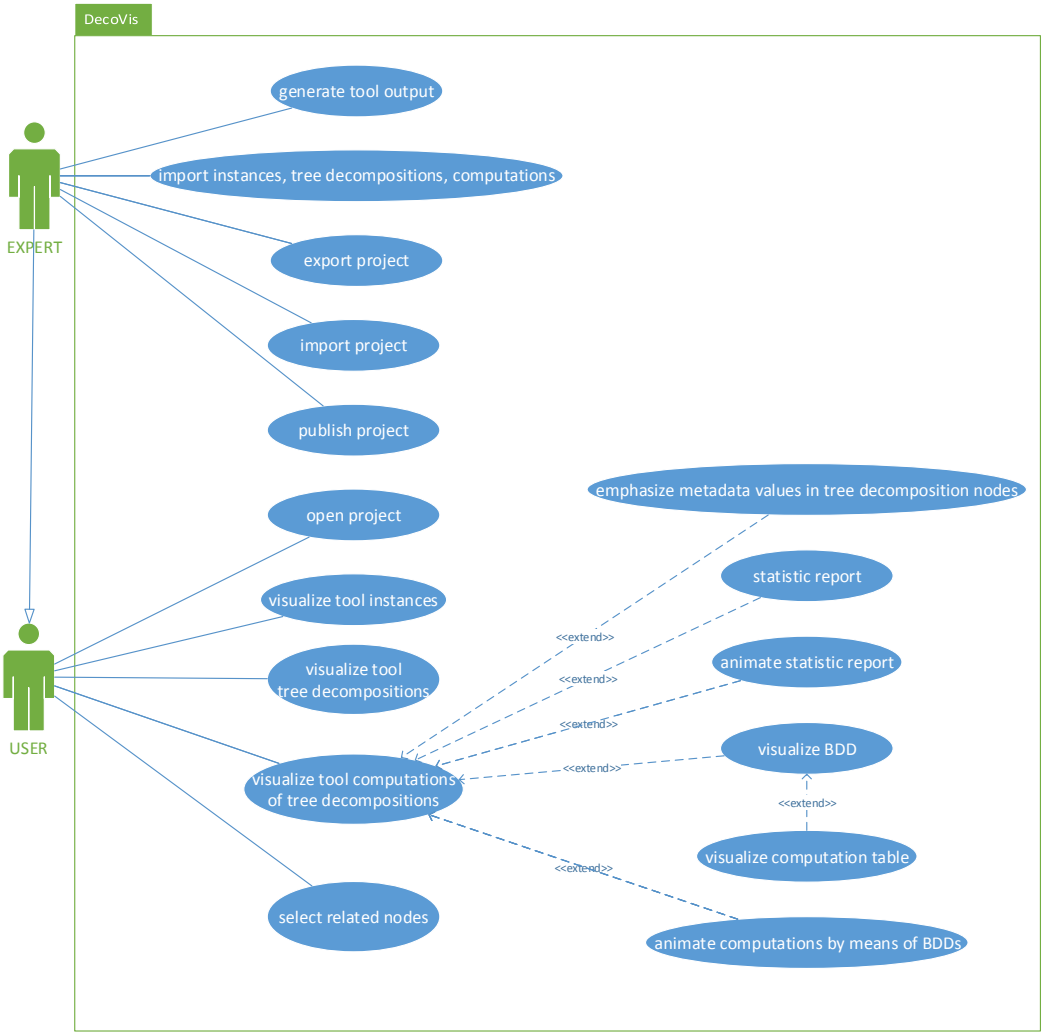


Figure 3.1: UML use case diagram

to be defined in the configuration file. The preparation of the three categories in the visualization tool as well as the definition of their options in the configuration file should be treated separately for each category. Option name, the possible parameter values, the default parameter value, a label and the input type define the appearance of each form field in the user interface. The execution command is built up by appending the options and parameters to the command string. The name of the generated output file is composed of the name of the input file and the used options. An additional property in the configuration file implies the filtered options for the output file name. After the nonrecurring set-up the expert user has unrestricted use of the graphical user interface for the output generation.

In addition to the three separated graphical user interfaces for the generation of instances, tree decompositions and computations, a fourth interface for an overview over all generated files has to be shown. Furthermore this overview gives an information about the relations between the generated tree decompositions and the corresponding computations.

Import instances, tree decompositions, computations The expert user has two opportunities for importing files: Either the output is imported that was created with the visual output generator described above, or previously generated files from the local hard disk are uploaded. If the latter is the case, several instances and tree decompositions can be imported at once. Each composition is associated with a particular tree decomposition that has to be loaded beforehand. For the loading of computations the user has to choose the corresponding tree decomposition.

Export project The graphical user interface should contain a possibility for exporting the generated or imported files. All available components should be linked up to one project file. Additional information such as the association between tree decompositions and computations or the original file names have to be appended. As a result, the expert user downloads the project file to his computer.

Import project By means of a file selection dialog, the exported project files can be uploaded again from the local hard disk to the system at a later time. In the course of a project import, the state of the visualization tool from the time of the project export has to be restored. This means, that all present instances, tree decompositions and computations from the time of project export appear in the visualization tool.

Publish project The stored project files from a particular folder have to be itemized in the graphical user interface and loaded after a selection. A further setting in the configuration file contains the file path to the public folder that is reachable from the application.

3.2.2 User Use Cases

Open project One or more published projects that are listed in the user interface of the visualization tool can be chosen and loaded. The project is loaded and visualized as in the import project use case.

Visualize tool instances The system visualizes the instance input file. The prerequisite is that the file is already available in the system. Hence, the instance was obtained by building from the visual output generator, the import via file selection or loading from an open/imported project. The instance has to be represented in the form of an undirected hypergraph. The connection of an edge with more than two vertices should be

emphasized. Furthermore, vertex name and vertex type should be clearly depicted. To gain a clear view of the instance graph, the system should take care of arranging vertices.

Visualize tool tree decompositions In contrast to the instance that is represented by a graph, a tree decomposition should be displayed as a tree. One tree node represents a bag, i.e. a list of the corresponding vertices from the instrance graph. The normalization type of the tree decomposition should be explicitly depicted.

Visualize tool computations of tree decompositions The visualization of a computation looks the same as the tree from its corresponding tree decomposition. In addition to the tree decomposition data, the computation consists of computation data such as metadata and solutions of tree decomposition nodes in form of BDDs. Hence, computation data enable further use cases like visualizing each tree decomposition result as a BDD, statistic reports, animations, etc.

Emphasize metadata values in tree decomposition nodes Each tree decomposition node has assigned multiple metadata values from the computation. A tree decomposition view has to be displayed where the metadata values are emphasized. As starting point the user has to select one metadata type. The assigned metadata values of each tree node should be emphasized in any way, that the relative difference between the tree nodes and its assigned metadata values are visually recognizable. An additional feature is the information of the concrete metadata values.

Statistic report There are two types of statistic reports: the single metadata statistic report and the advanced statistic report that stretches over multiple metadata types and computations. The statistic report is prepared with the help of a line chart where the x-axis describes the elapsed time and the y-axis the metadata value. One line in the chart shows the metadata values per metadata type and per computation. In contrast to the single metadata statistic report that consists of just one line, the advanced statistic report consists of several user-defined metadata types over multiple computations.

Animate statistic report The animation of statistic report represents an extension of the statistic report. Throughout the elapsed time on the x-axis the chart is continuously built up from left to right.

Visualize BDD The computation result of each tree decomposition node should be visualized as Binary Decision Diagram. After the selection of a specific tree decomposition node, the BDD should appear. Basically the BDD is depicated as a rooted directed acyclic graph where the only leaf nodes are TRUE and FALSE. Appropriate shapes and colors for nodes, leaf nodes, edges (different for ELSE-edge and THEN-edge), complement nodes and complement edges have to be used. The BDD is built up from the root node on the top to the leaf nodes on the bottom where the nodes should be clearly arranged.

Visualize computation table Given a BDD, another possibility for visualizing the models of the solution is to list the models with their variable assignment. A table has to be generated where each row represents a path from the root node to the leaf node and therefore a model and each column shows a level in the BDD. After the selection of a specific row in the computation table, the corresponding path in the BDD has to be highlighted. Furthermore the selection of a column in the computation table results in the highlighting of the BDD nodes that are located on the appropriate level.

Animate computations by means of BDDs Given a tree decomposition window and an animation window, computations are visualized in form of animations. By means of the visualization of the BDDs from each level of the tree decomposition, the computation should be animated from bottom to top. Especially the merging of branches has to be emphasized. Simultaneously to the depiction of the BDDs, the corresponding tree decomposition nodes should be highlighted. The tool should offer the possibility to control the animation step at any time with the options: play, stop, forward, backward, begin and close. There are three animation modes: The first mode calcutates the overall animation view height by means of the highest BDD from all levels. The second mode defines for each level an individual height according to the highest BDD in this level. The third mode uses the predefined variable order from the input computation file. The height of the view conforms to the number of the variables in the computation file. Details about animation and the three animation modes can be found in Section 3.5.

Select related nodes Three selection modes (description can be found in detail in Section 3.5) should support the highlighting of related nodes within instances, tree decompositions and BDDs. The first mode marks nodes with exactly the same information content from the selected node(s). The second mode marks nodes containing the full information content from the selected node(s). And the last selection mode marks nodes consisting of any part of the information content from the selected node(s).

3.3 Workflow Design

The control flow of the system from one activity to another is described with the help of activity diagrams. Moreover, this control flow is illustrated by means of a flow chart where each activity stands for an operation of the system. With the help of four activity diagrams, the workflows for project generation and analysis (use published projects) are illustrated. The project generation workflow (shown in Figure 3.2) is performed by experts and the analysis workflows (Figure 3.3, Figure 3.4 and Figure 3.5) are used by the users.

The following two subsections give an insight into the activities of the project generation workflow and the analysis workflow by means of scenarios covering most of the available operations in the system.

3.3.1 Project Generation Workflow

The workflow *project generation* is performed by experts and leads to the generation of project files composing instances, tree decompositions and computations. In addition to that, further information such as the relations between tree decompositions and computations are packed into the project file. Finally, generated projects can be published for the users of the visualization tool.

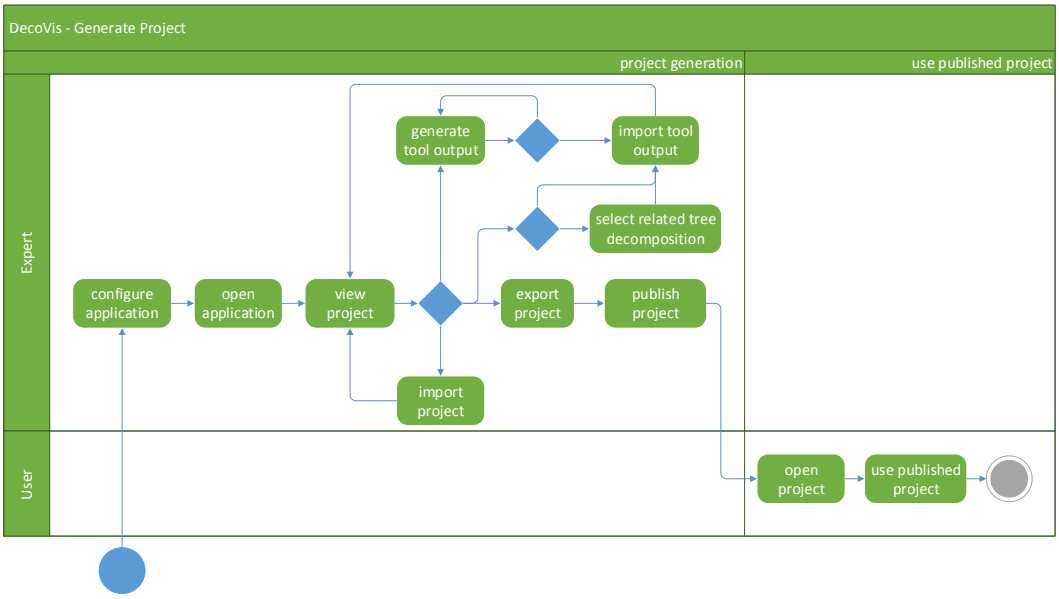


Figure 3.2: Workflow: Project generation

The scenario in Figure 3.2 illustrates the workflow with its different variants for creating an user-defined project containing an arbitrary number of instances, tree decompositions and computations.

As prerequisite for the usage of the expert functions, the expert user has to deploy and configure the application. Settings for the used user mode, the output generation tool, the temporary file path, the publishment folder, etc. have to be set before the application can be opened. After the application is opened, a graphical user interface provides multiple possibilities to view project components (instances, tree decompositions and computations). Moreover, the several ways for project creation and project editing are explained in the following.

Generating tool output provides one way for creating instances, tree decompositions and computations in form of files. As one can see, this activity can be repeated as often as desired. Moreover, already generated instances or tree decompositions could be used as parameters for further processing.

There are two variants for including input data into the project file, which can be

repeated arbitrarily often:

1. *import tool output*
   On the one hand, generated tool output data can be imported after their creation and, on the other hand, already generated files from the computer's hard disk can be uploaded. In contrast to the first approach, the upload needs in case of computations the information about the corresponding tree decomposition.

2. *import project*
   Another possibility for importing data is to import already existing projects via file upload. As a consequence the current project can either be set up on the basis of the uploaded project, or extended with its content. Furthermore, additional information like the original file names and the relationships between included tree decompositions and computations are also taken over.

The subsequent activity after the import of the data represents the generation and export of a project file. This project file containing the output files, the original file names and the relationship between included tree decompositions and computations is downloaded to a user-defined folder on the computer.

After these steps for the generation and export of the project file, the expert user has to copy the project to the publication folder defined in the configuration file. After a restart of the application, the project has to appear in the list of published projects in the user interface. The released projects can be opened and used in further consequence by each user having access to the application. How these published projects can be used in the user mode is described in the next subsection.

3.3.2 Analysis Workflows

Starting point for the analysis workflows are the generated and published projects that are used by the users of the visualization tool. Beside the visualization of the three main components (instances, tree decompositions and computations) several workflows are provided for the analysis of them such as BDD animation, statistic diagram, etc.

The activity diagrams in Figure 3.3, Figure 3.4 and Figure 3.5 represent the usages of a published project. In all activity diagrams the basic activities, represented by the first three activities (*open project*, *select tree decomposition* and *choose computation(s)*), assume the existence of an instance, a tree decomposition and computation(s). Hence, the first three activities only indicate the selection of these elements in the graphical user interface.

Figure 3.3 illustrates the workflow for finding and highlighting valid paths in particular BDDs. After the basic activities, the user has to choose a particular BDD for further analysis of the computation by selecting the corresponding tree decomposition node of the BDD. As a result, the BDD in form of a tree with the TRUE and FALSE leaf node gets displayed.

The BDD window furthermore contains a possibility for visualizing the appropriate computation table. The computation table is eventually another representation of the

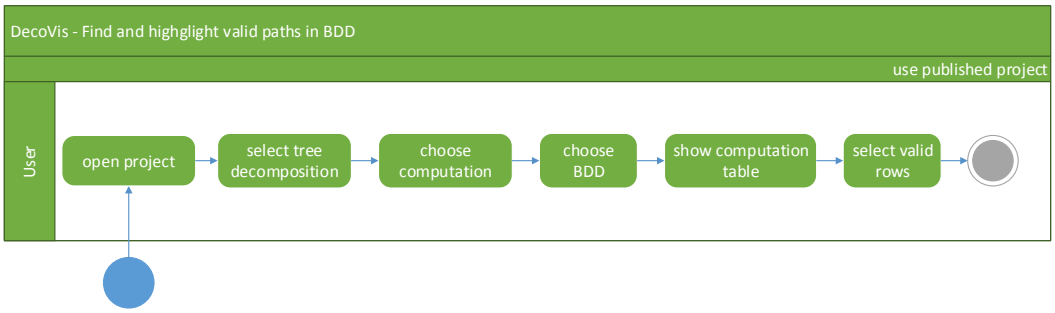


Figure 3.3: Workflow: Path highlighting in BDD

BDD. Because a row contains variable assignments, each row is equivalent to a path in the BDD passed through by the variable assignments from the root node to the TRUE leaf node. A more detailed explanation can be found in Section 3.5.

After the selection of the rows, the valid models are found and the corresponding paths are highlighted in the BDD. Since this activity was the last step in the scenario, the terminal node in the activity diagram is reached.

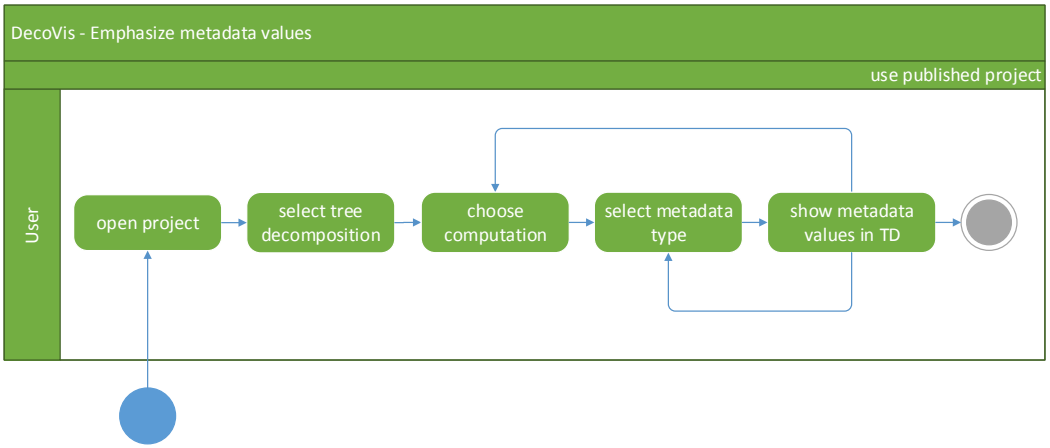


Figure 3.4: Emphasize metadata values workflow

Figure 3.4 represents the workflow for emphasizing metadata values by means of tree decomposition nodes. The first three activities in the activity diagram depict the basic activities described above.

After the selection of one computation, an arbitrary metadata type such as memory usage, elapsed time or number of BDD variables has to be chosen. The tree decomposition representation is used for metadata visualization by including the metadata value into the

corresponding tree decomposition node. The approaches for emphasizing metadata values in tree decomposition nodes (e.g. by computing the shape size of the node relatively to its metadata value) are described in Section 3.5.

The activities of selecting computations, selecting metadata types and the visualization of metadata values can be repeated as often as desired. As a consequence, several computations with their metadata types and values can be compared.

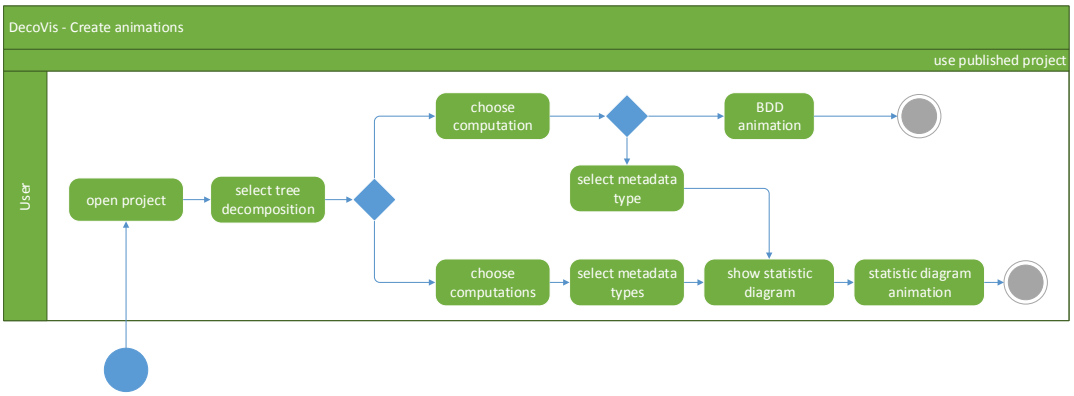


Figure 3.5: Create animation workflow

Figure 3.4 represents the workflow for creating statistic diagram animations and BDD animations. As shown in the activity diagram, the user has the choice between single and multiple computation and metadata type selection:

- Single selection
  After the selection of one computation, the user can pick either the visualization of the computation data in form of a BDD animation or the illustration of the metadata values of one metadata type in a statistic diagram. In contrast to the visualization of a statistic diagram depending on multiple computations and metadata types, the depiction after single selection contains just one line in the line chart representative for the selected metadata type.

- Multiple selection
  The selection of several computations and metadata types also results in the statistic diagram visualization. Each line in the visualized line chart represents a computation with its metadata values plotted on the y-axis.

In order to see the statistical progress of the selected metadata type(s) from its computation(s), the statistic diagram in form of a line chart provides an animation represented in the last activity before the terminal node. The animation continuously builds up the lines along the x-axis from left to right.

Since each tree decomposition node has a solution in form of a BDD stored in the computation, this activity starts the animation of the BDDs from the lowest level of

the tree decomposition to the root node. Beside the window for the animation of the BDDs, a second window contains the highlighted tree decomposition nodes associated to the visible BDDs. More details about the BDD animation and the statistic diagram animation can be found in Section 3.5.

3.4 System Architecture

A UML class diagram is a helpful instrument for designing business models. Since instance, tree decomposition and computation are often used terms in this master thesis, this section provides a brief insight into the abstractions of these three components.

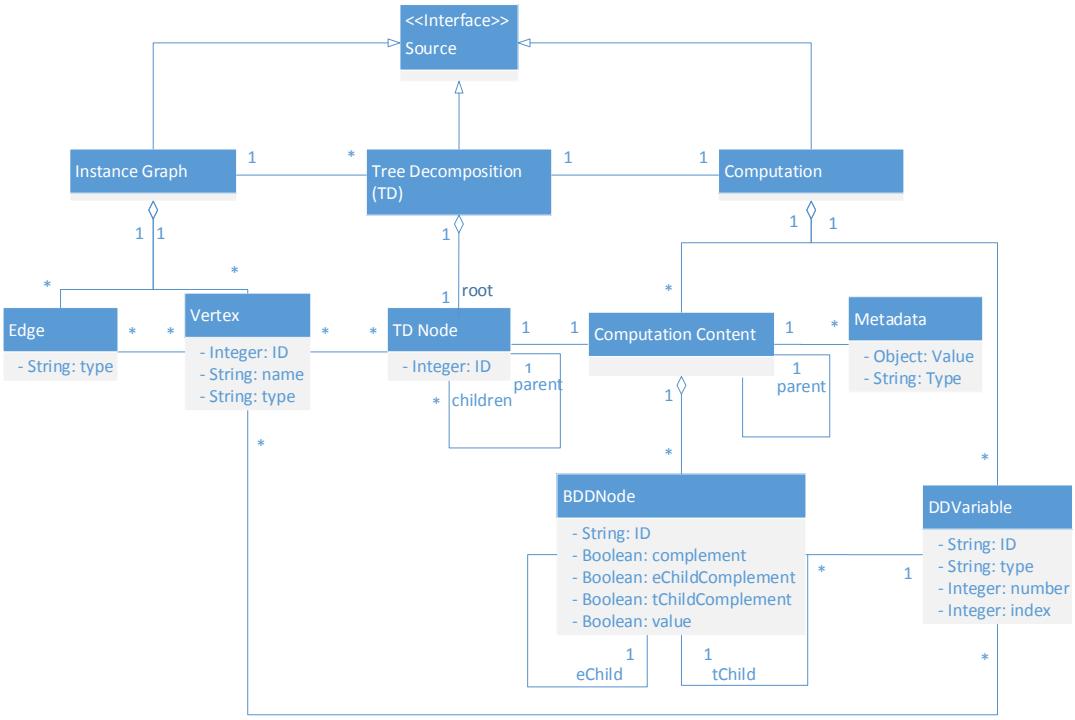


Figure 3.6: UML class diagram - instance, tree decomposition and computation

Figure 3.6 shows the entities and relations that are necessary to model instances, tree decompositions and computations. As one can see in the class diagram, each of these components implements the interface **Source**. This interface represents an abstract type for all output data generated by tools using dynamic programming on tree decompositions. Furthermore, the interface provides the basis for enhancements in the tool as well as in the UML class diagram. The next subsections give a detailed description over each component with a corresponding graph visualization.

3.4.1 Instance

The UML class diagram from Figure 3.7 is an extract from the diagram in Figure 3.6 and contains the essential entities and associations for instance information.

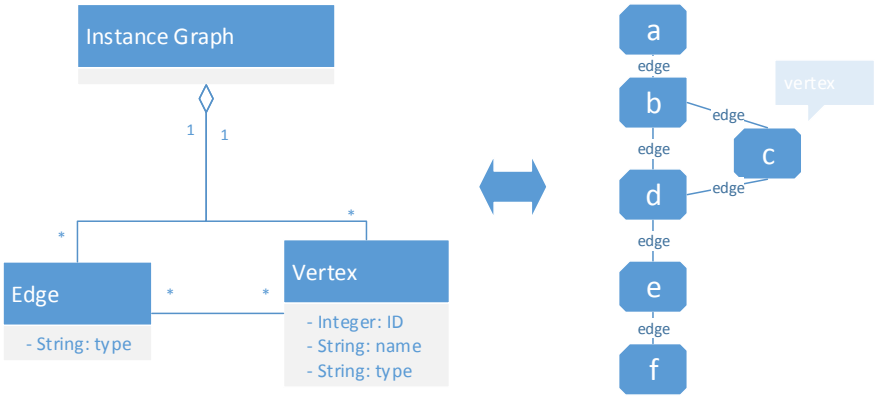


Figure 3.7: UML class diagram - instance

An **Instance Graph** consists of vertices and edges. Since we consider hypergraphs, an edge could have more than two endpoints. As one can see in Figure 3.7, the instance can be represented as undirected graph where vertices (e.g. **vertex**, represented as tooltip) as well as edges (e.g. **edge**) are characterized by a **type**. Furthermore, each vertex contains a unique identifier (**ID**, not visible in the graph) and a **name** (e.g. **a**).

As an example for explaining hypergraphs and the properties **name** and **type** of vertices and edges, family relationships can be used: A **Vertex** describes a person where the **ID**-property illustrates the social security number, the **name**-property contains the first name and surname and the **type**-property represents the sex of the person. *Married* and *child* define possible binary edge types and *team* represents a possible hyperedge type.

3.4.2 Tree Decomposition

As suggested by the name, a tree decomposition can be visualized as tree. As defined in the class diagram in Figure 3.8, (**TD**) is made up of a root tree decomposition node. Except for leaf nodes (having no children), each other node has at least one child node. More details about tree decompositions can be found in Section 2.2.

As the class diagram and the sample tree decomposition in Figure 3.8 show, a tree decomposition node (**TD**) is identified by a unique id (**ID**) and has an arbitrary amount of vertices from the instance assigned, that form the bag of the tree decomposition node. The sample tree decomposition nodes in Figure 3.8 are described by the **ID** (with prefix n) and a **name**-array of the assigned vertices.

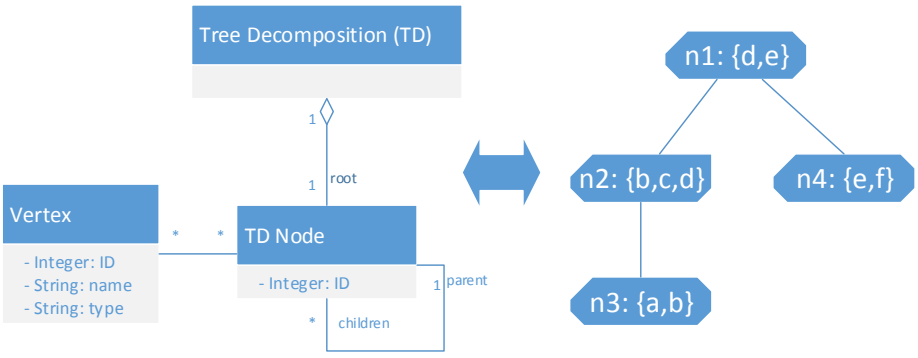


Figure 3.8: UML class diagram - tree decomposition

3.4.3 Computation

As one can see in the overview diagram in Figure 3.6, associations to the **Vertex** (from the instance) and to the **TD Node** (from the tree decomposition) are established.

As illustrated in the class diagram in Figure 3.9, **Computation** consists of multiple variables (**DDVariable**). Each variable is composed of a unique id (**ID**), a **type**, a **number** for unique identification, an **index** (node level) and the relation to several vertices from the instance. Details about the background of the properties can be found in Section 2.4.

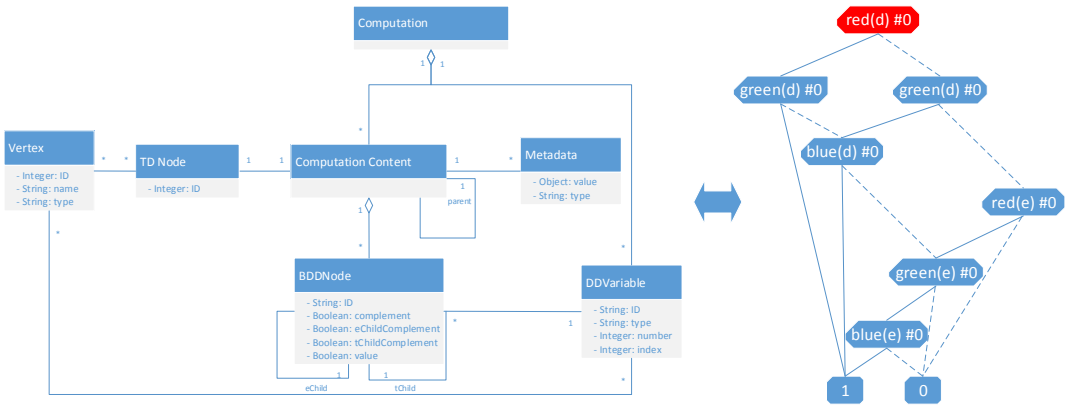


Figure 3.9: UML class diagram - computation

Furthermore, the **Computation** contains for each node in the tree decomposition an appropriate **Computation Content**, i.e. a Binary Decision Diagram with additional information. To support an optimized implementation of the tree traversion, each **Computation content** contains the information to its parent content, i.e. the parent

node from the tree decomposition. Each **Computation Content** is extended by any number of metadata information consisting of metadata type and the corresponding metadata value.

The following explanations focus on BDDs and its elements, as one can see by means of the class diagram and the sample BDD in Figure 3.9. Each **BDDNode** consists of a unique id (**ID**), the information if it represents a complement (**complement**, in Figure 3.9 the root node), the information if one of the child edges represents a complement (**eChildComplement** and **tChildComplement**), in the case of a leaf node the **value** FALSE (represented by **0**-node) or TRUE (represented by **1**-node) and the relation to a variable. Further information about BDDs and their representation can be found in Section 2.4 and Section 3.5.

The sample BDD in Figure 3.9 illustrates the solution to the root node **n1** from Figure 3.8. Several models for the tree decomposition node can be taken from the depicted BDD.

3.5 Graphical Representation Analysis

The *Graphical Representation Analysis* consists of three parts:

1. The first part is about the containers which display and visualize data to the user. In fact, they are simple windows embedded in the GUI of the web application.

2. The second part describes in detail how all the data is visualized to give the user the best and most convenient way to analyse and understand them. There are several types of windows, each having different use cases and user interface elements.

3. The last part is about the graphical data analysis. In this section the relationships between all the data elements and the interaction between them in the web user interface are examined carefully.

3.5.1 Windows

The main components in the web user interface form the windows. Each window has particular elements like buttons and dropdown fields, depending on the window type and the data which should be visualized. Figure 3.10 illustrates the conceptual design. The user can view multiple windows at once and is able to close or minimize each of them separately. Furthermore, there is a window management which allows to reorder the windows. As an additional feature, it supports the automatic adjusting of the window sizes to the screen and the browser window.

At the top of all the windows there is a tab bar, displaying all currently opened windows, including the visible and also the minimized ones. Additionally, the user is able to close and minimize them in both views, in the tab bar and also in the particular window for a more convenient user interface.

A further feature in this window management is the setting for defining how many windows should be displayed on the screen at once. Depending on the user choice the window elements are adjusted automatically, regarding the size and position. This user interface element can be implemented as shown in Figure 3.11.

Altogether, there are 7 window types, each having different menu elements:

- **Instance window**
  The instance window has only the two standard buttons for minimizing and closing the window.
- **Tree decomposition window without any computation**
  The tree decomposition window includes a button to add a computation file.
- **Tree decomposition window after loading the computation**
  This window type displays the metadata and solutions of the computations. Therefore it offers a dropdown menu to select one of the available metadata types, a play button for the animation of the BDD and a button to display the selected metadata in an own statistic diagram window.
- **BDD window**
  The BDD window displays a single BDD and therefore a partial or full solution of the tree decomposition. It only offers a single button to display the appropriate computation table.
- **Computation table window**
  This window type displays the computation table of a BDD.
- **BDD animation window**
  When clicking the "Play" button in the tree decomposition window (with a computation preloaded of course), the animation of the BDDs is started in the BDD animation window. Additionally, all windows are minimized, except the tree decomposition window and the currently used animation window. This provides a better overview and visualization since the animation is viewed in a larger scale. The BDD animation window includes buttons to control the animation, like playing or pausing the animation, going to the next or previous step and adjusting the animation speed.
- **Statistic diagram window**
  This window visualizes the metadata in a statistic diagram. To display each node automatically step by step, it provides a button to start this type of animation.

3.5.2 Graphical Representation

In this section we will describe the graphical representation in detail. It is necessary to design the visualization in such a way that it can handle large amounts of data since this

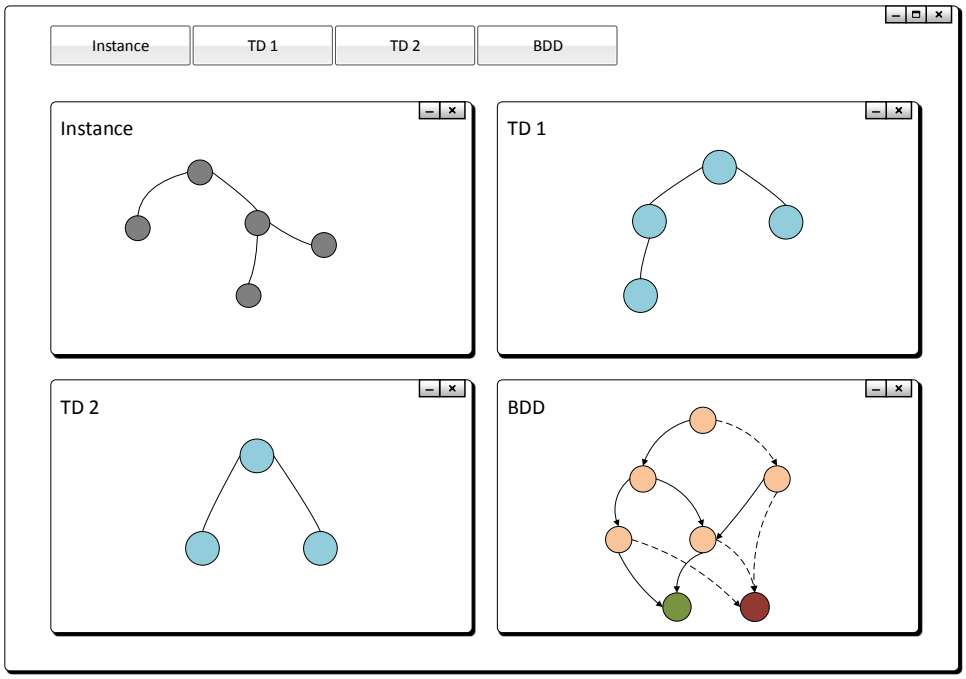


Figure 3.10: Window management

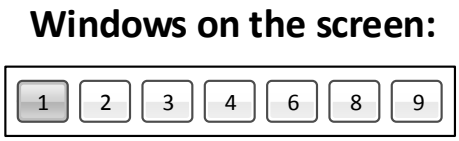


Figure 3.11: Change window number

will be required for a data analysis. Furthermore, hypergraphs are supported and should be visualized in an attractive way.

Large Amount of Data

For the benchmark tests and also for other purposes like analyzing the data, computations and algorithms, it is required that the underlying graph library can handle large amounts of data (like in Figure 3.12). This also includes a stable and smooth visualization where an interaction with the user is still possible, meaning selecting nodes, resizing the graph or moving nodes. But what are large amounts of data exactly? How many nodes and edges should be displayed without any further problems?

- We defined that for our purposes up to 100 nodes including the edges should be handled error-free for the instance graph (of course assumed when running on a conventional computer).
- With this decision for the instance graph, up to 300 nodes should be possible for a tree decomposition, assuming that a tree decomposition has approximately a three times higher node number in contrast to its instance.
- A BDD should handle up to 200 nodes without problems.
- Taking the previous three decisions into account, up to 1000 nodes should be possible for a BDD animation step when assuming that an average of 10 BDDs are displayed simultaneously and only an average of 100 nodes are used for a BDD.

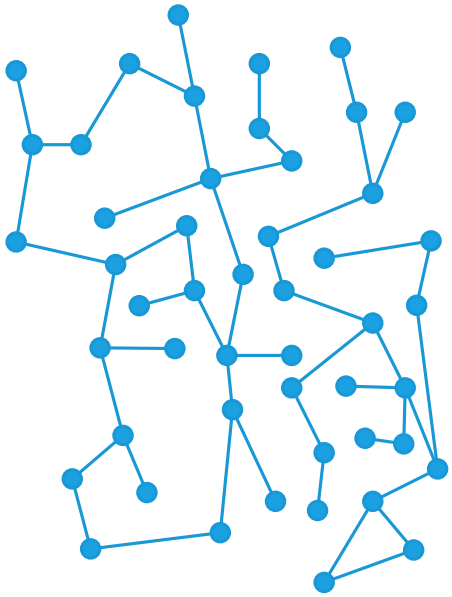


Figure 3.12: Large instance

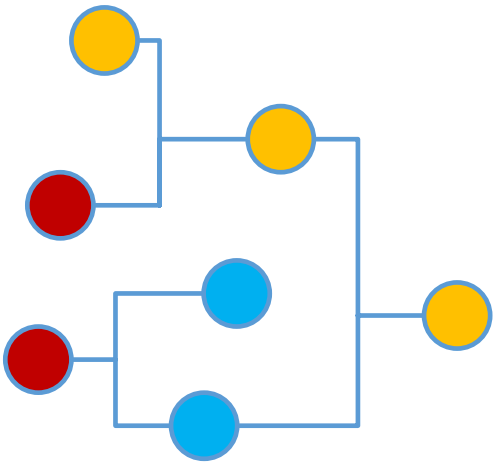


Figure 3.13: Hypergraph

Graphs & Hypergraphs

Hyergraphs are graphs which can have edges with more than two end points, meaning that an edge can connect also 3 nodes or more (see Figure 3.13).

Nodes and edges of a hypergraph or of a simple graph can have different types. To avoid that there are too many labels, different colors make the nodes and edges identifiable regarding their type. The concrete type should be visible by moving the mouse over the element.

There are many challenges regarding simple graphs and hypergraphs. How should the graph be visualized? Nodes should not be overlapping, the edges should be as short as possible and the graph should be displayed in such a way that the user can get the most out of them. There are already many algorithms for graph visualizations available, which have a focus on *force-directed graph drawing* [57, 58], meaning that the graph is drawn in a well looking way. But nevertheless, there are still many questions: which algorithm should be used exactly? And since they offer a lot of options to optimize the visualization, which options should be chosen?

In fact, the algorithms and options are always depending on the graph data and of course, on the amount of data. Both of them are very variable in our use cases. Therefore, it should be possible to display the graphs and hypergraphs in such a way that they are still providing optimal visualizations, independent of the data and the number of nodes. For our purposes we visualize many more data types than simple graphs and hypergraphs, such as tree decompositions, BDDs, computation tables and metadata as described in the following sections.

Tree Decomposition

For the visualization of the tree decompositions, a tree is used (see Figure 3.14). Of course, this is just a sub type of a simple graph, but the tree is displayed with some layout requirements: There is a root node at the very top and the branches are oriented to the bottom.

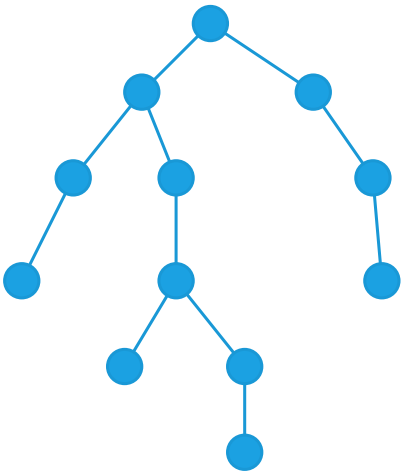


Figure 3.14: Tree decomposition

Binary Decision Diagram

For a graphical representation of the solutions, one approach is to use BDDs (see Figure 3.15). A BDD is visualized by a rooted directed acyclic graph as already described in Section 2.4. It looks like a tree structure, but it has only two leave nodes which should be visualized with "0" (representing False) and "1" (representing True). Furthermore, the branches can be merged together when walking from the root to the leaves, resulting in a more compact visualization. This should be considered when using graph algorithms. Since the BDD shows a Boolean function, each node has two outgoing edges. The THEN-edge should be drawn by a solid line, the ELSE-edge by a dashed line. To reduce the overall graph, complement nodes and edges are used which negate the result when the path is walking through. To make these nodes and edges distinguishable, appropriate colors should be used.

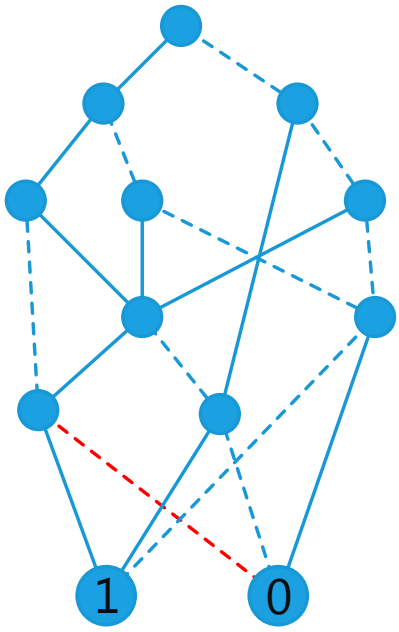


Figure 3.15: BDD

Table

Another way to display the solution is to use a table (see Figure 3.16). This should be implemented as an alternative to the BDD visualization. Actually it should extend the analysis of the solutions. Each row covers one or multiple solutions and therefore a path in the BDD, each column represents a horizontal level in the BDD. The table contains a horizontal header describing the variable assignments for the columns and a vertical header numbering the various solutions. The value of a cell in the table can be "0", "1"

or "-", whereas "0" represents the variable assignment *FALSE*, "1" the assignment *TRUE* and "-" means that both Boolean values can be used to get a solution. It is important to say that the table contains all real solutions and not necessarily the paths in the BDD which are ending in the "1" leave. That is because the BDD can contain complement edges and nodes, which negate the result. In fact, the table resolves the negations so that only valid solutions are shown. For example, if a path in the BDD ends in the "0" node and contains an odd number of complement nodes/edges, then the table is showing this path as valid solution since the result of the Boolean function in the BDD has to be negated.

	Variable 1	Variable 2	Variable 3	Variable 4	Variable 5
Model 1	1	1	1	1	1
Model 2	1	1	1	1	0
Model 3	1	1	0	-	1
Model 4	1	1	0	-	0
...	...	...	...	...	...
Model n	0	0	0	0	-

Figure 3.16: Computation table

Metadata

When the solutions are calculated and the computations are being done, we get metadata about the computation process, such as memory usage, elapsed time or number of BDD variables. The software should support the visualization of all these metadata types.

One approach to do that is to create a statistic diagram, which shows each metadata value as simple node on the y axis. The vertical x axis indicates the elapsed time since the beginning of the computation process (see Figure 3.17). Multiple metadata types can be added to this diagram. In addition, also a specific metadata type of multiple computations can be illustrated which provides an efficient way of comparing them. Another approach would be to include the metadata values already in the tree decomposition. For example, we can use different shapes, colors or sizes for the vertices in the graph. Since we deal with many different values, the best solution is to use different sizes. Smaller vertices

represents a low value, while bigger ones represents higher values (see Figure 3.18). When we apply some particular metadata type, such as the memory consumption, we perfectly see which nodes consume the most memory resources. Furthermore, we see the non-computed nodes by using a predefined color and small node size, for example if the partial solutions are unsatisfiable. Then, as we are going bottom-up in the tree decomposition, further computations up to the next branch do not make sense which are illustrated by these non-computed nodes.

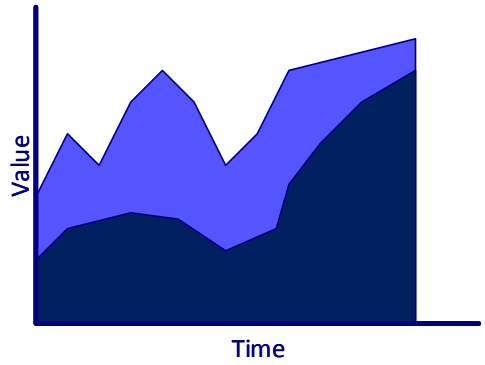


Figure 3.17: Statistic diagram of metadata

3.5.3 Graphical Data Analysis

Graphical data analysis in general describes which data is contained in graphs and diagrams and how they are related. Furthermore, it also includes how visualized elements can be used to get the most information of them and how a detailed analysis works. Therefore, this section deals with the analysis of the data used in the graphs including the relationships between them. In detail, it contains:

- the relationships between particular window types and their included data,
- the analysis of the BDD animation and
- the benefits of visualizing metadata in the statistic diagrams.

For example, multiple tree decompositions of an instance often have commonly used parts, similarities or even same node and edge elements. The same applies for BDDs. Also, the instance graphs have a very close relationship to their tree decomposition. Furthermore there are connections between the BDD and the computation table, since they describe the same solutions. There should be a generally usable highlight feature to mark nodes with all their related contents. Different highlight modes should be available

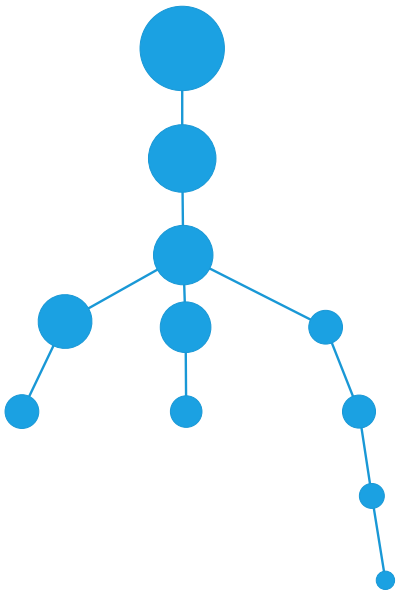


Figure 3.18: Metadata values in tree decomposition

to give the user the best utility to interact with the graphs and the data (as required in Section 3.2.2). The BDDs of a particular tree decomposition should be analyzed and evaluated from the bottom to the top by using an animation with different view modes. To visualize the metadata in a convenient way, interactive statistic diagrams should be used.

Visualizing Relationships between Data Elements

There should be different modes available to highlight the nodes and vertices. Here, we distinguish between "select" and "highlight". The first one means the user action when the user is clicking on a node. The latter is the action done by the software to color the particular nodes.

Furthermore, we clarify the naming of the elements in the various graphs. An instance graph consists of vertices and edges, whereas all other graphs consist of nodes and edges like the tree decomposition or the BDD. Each vertex in an instance graph has an ID. Each node in the tree decomposition has a so-called bag which is a set of vertices. The nodes of the BDD include a variable with a set of vertices, usually they are also containing some more information which is not required for the visualization part. To make the relations between the graphs visible to the user, each node or vertex can have a label visualizing some information like the vertex ID, the bag of the tree decomposition or the set of vertices in the BDD.

Now we introduce 3 highlighting modes by using the example of an instance graph and a tree decomposition.

If we select one or multiple vertices in the instance graph, nodes in the tree decomposition should be highlighted regarding the chosen highlighting mode. One possibility is to highlight these nodes in the tree decomposition which exactly contain the vertices of the selection in the instance graph, let us call this mode “EXACT”, illustrated in Figure 3.19.

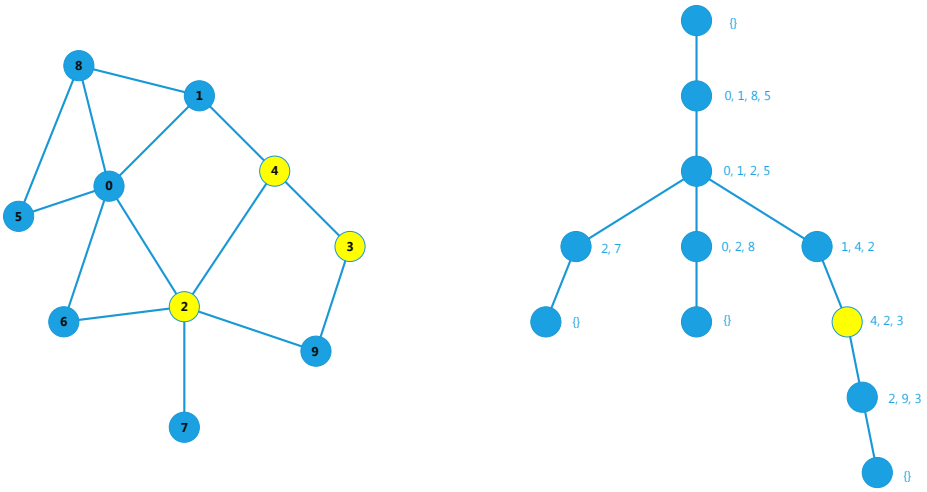


Figure 3.19: Highlighting mode: EXACT

Another option is to highlight all the nodes which contain the vertices of the selection in the instance graph at least. Since this is a simple conjunction, the mode is called "AND" (see Figure 3.20).

The last possibility is to highlight all the nodes which contain one of the vertices of the selection in the instance graph. This is a simple disjunction, the mode is called "OR" (see Figure 3.21).

Instance with Tree Decomposition When we create a tree decomposition, we want to know where the vertices of the instance graph are used in the tree decomposition for better analysis of the algorithms and the configuration. Of course, each node has a bag containing a subset of the vertices of the instance, as already mentioned before. This is also represented by a label. But this alone may not be sufficient for the user, generally when the bags are quite large. Hence, the introduced highlighting feature and different colors would increase the overall clarity.

TD with TD Tree decompositions should be easily comparable to each other. This way, the user can analyse common parts and can get a better overview of the differences between the configurations used to create the tree decompositions. But to compare them, we want to use the introduced highlighting feature again. This allows us to highlight

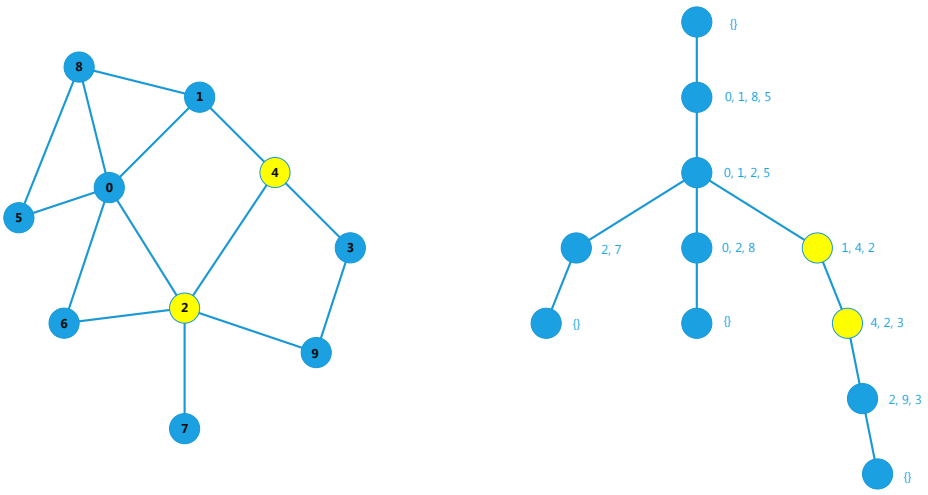


Figure 3.20: Highlighting mode: AND

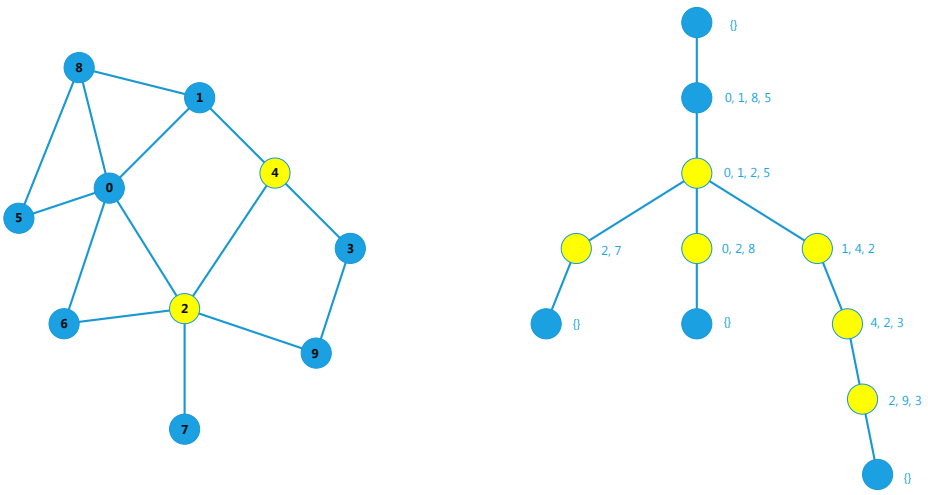


Figure 3.21: Highlighting mode: OR

common parts, by using the “AND”, “OR” or “EXACT” mode. The highlighting of these nodes should be a helpful tool for a detailed analysis and comparison (see Figure 3.22).

BDD with BDD The same approach is required when comparing BDDs. Hence, the nodes of the BDDs should also be interactive and the highlighting should be done in exactly the same way as already described for the comparison of tree decompositions (see Figure 3.23). Then the computations can be easily compared.

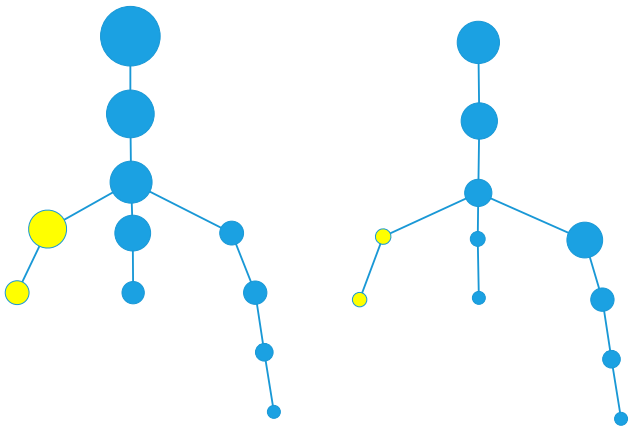


Figure 3.22: Compare tree decompositions

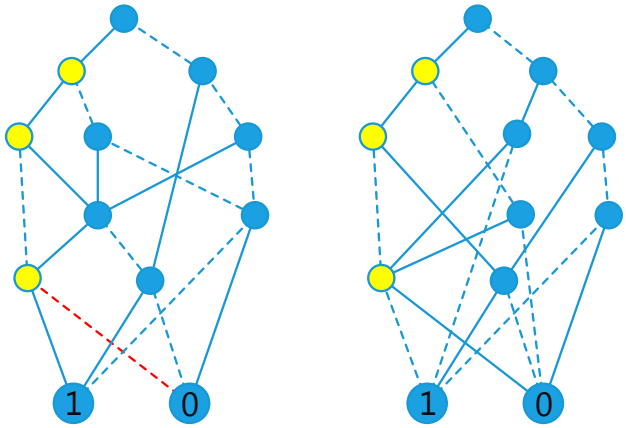
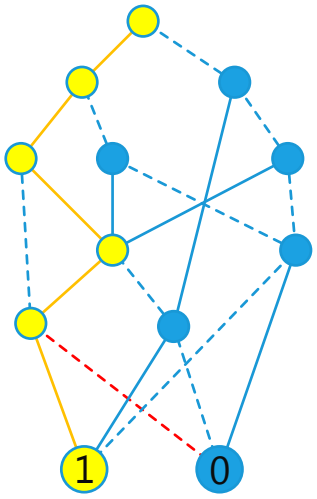


Figure 3.23: Compare BDDs

BDD with Table The BDD and the related computation table should be interactive. When clicking the rows or columns in the computation table, we should get the highlighting in the BDD graph, illustrated in Figure 3.24. Combining these two visualization elements provides an efficient way to understand the solutions since we have a graph visualization in connection with a data table. The BDD is very compact, hence it may contain complement edges/nodes and skipped levels. Therefore the data table should enhance this visualization by providing the solutions in detail. The user can find out the path a model belongs to. All the paths containing an odd number of complement elements are then very clear this way because the user sees paths are ending in the "0" node but they are still models in the computation table. Also skipped levels in the BDD can be seen in the computation table very easily since there are "-" values specified at these positions.



	Variable 1	Variable 2	Variable 3	Variable 4	Variable 5
Model 1	1	1	1	1	1
Model 2	1	1	1	1	0
Model 3	1	1	0	-	1
Model 4	1	1	0	-	0
...	...	...	...	...	...
Model n	0	0	0	0	-

Figure 3.24: Path highlighting in BDD

BDD Evolution Bottom-Up

As we see, BDDs are very compact and contain the partial or even full solutions of our problem. But how are these solutions computed? How are the BDDs related to each other? To make the BDD evolution fully visible to the user, the so-called BDD animation should provide a great way to illustrate that. The animation should start at the very bottom of the tree decomposition. All BDDs in this level should be visible next to each other inside the BDD animation window. Each level in the tree decomposition has an animation step and each step has a BDD for each tree decomposition node (see Figure 3.25). The whole animation is played automatically by walking from the bottom to the top in the tree decomposition. In detail, after a particular time span the BDDs of the next level of the tree decomposition should be visible by doing a slow fade in. The great challenge is to design the animation in a clear way for example by implementing additional effects like the merging of BDDs. This is necessary when two or multiple nodes in the tree decompositions are merged in the next level. Then the related BDDs have to be merged as well (see Figure 3.26). These effects should support the understandability of the general concept of the computation process as well as the generation of the solutions. During the animation, BDDs are usually merged, so that there remains only one single BDD in the end, representing the final solution of the root node.

Depending on the problem and its size, different view modes support a better visibility of the animation process:

- A view is required where all BDDs are consistent in their size during the whole animation process, meaning that the space between two consecutive levels in the

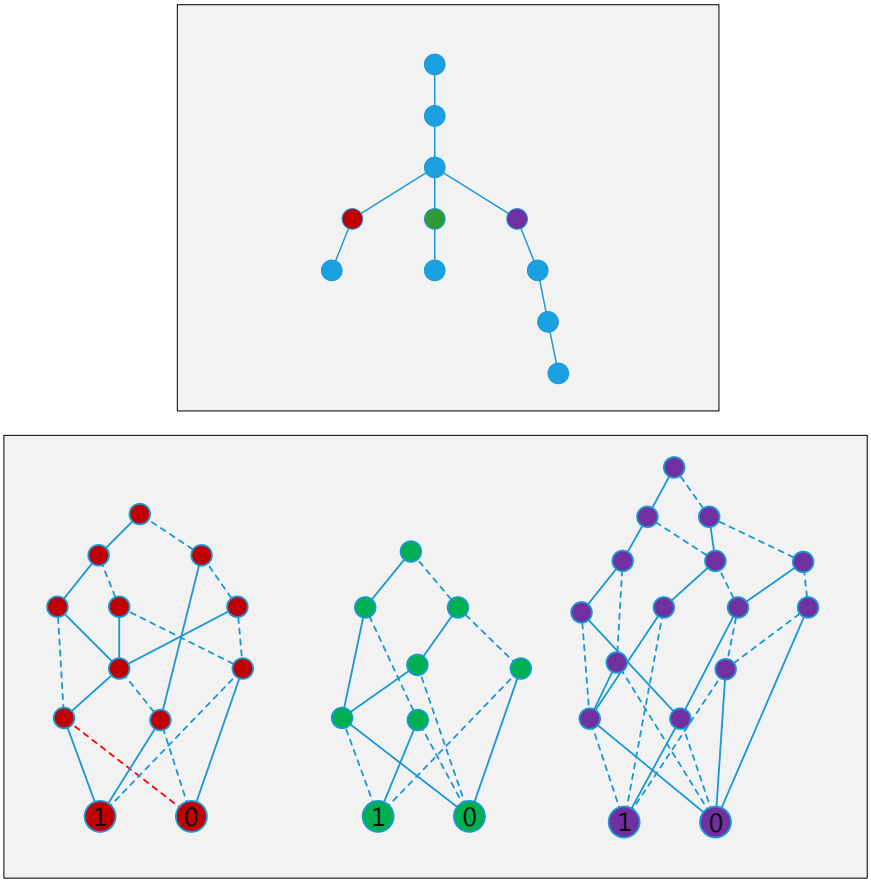


Figure 3.25: BDD animation

BDD should always have the same height.

- Another view would be to display the BDDs bigger and make the BDDs better visible. Each BDD is now displayed independent of the others. Hence, the space between two consecutive levels varies from BDD to BDD and the size is not comparable any longer. This results in a zoomed view, since the BDDs are zoomed to the full window size.

- Another view mode would be to stretch the BDDs to some predefined global order. Each computation includes a global variable order, which can be used for this purpose. Every level in the BDD is moved to the related level of this global variable order since a BDD level always refers to particular variable. It is important to say that only the space may be increased, the order of the variable assignments remains the same. Hence, every level of the BDD may get assigned a new higher level. Using this mode, merging BDDs would be better illustrated because usually the BDDs are interlocking. Also, the levels of the merging BDDs are always referring

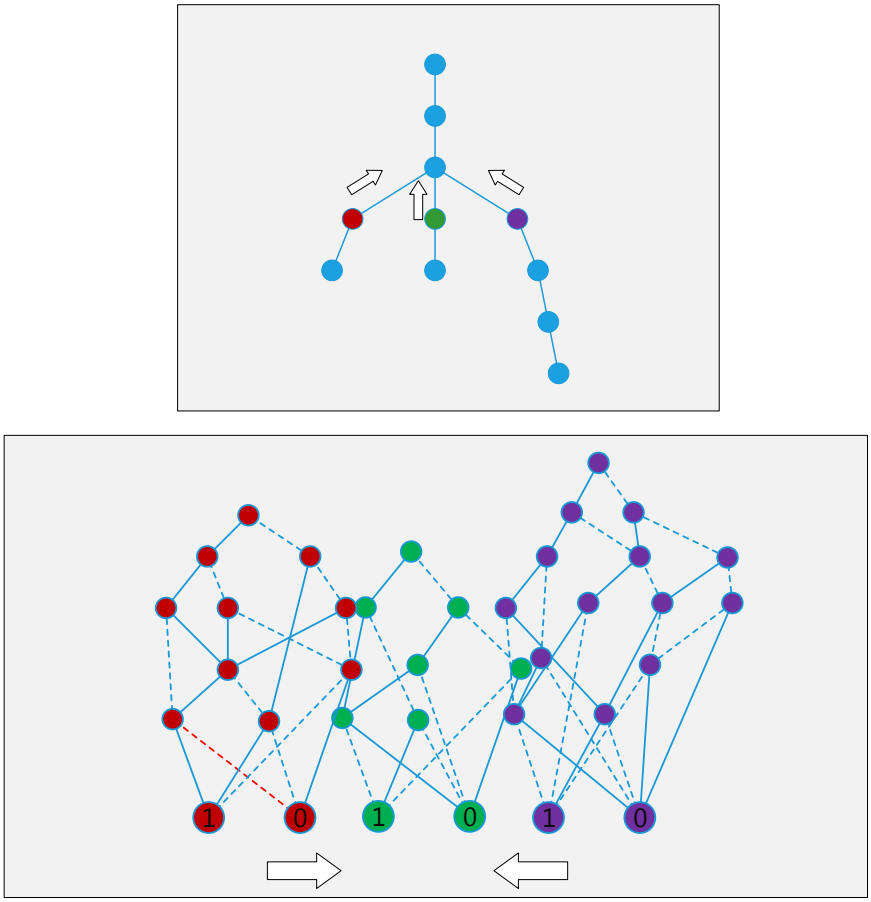


Figure 3.26: Merging BDDs during BDD animation

to the same variable assignment, resulting in a better animation. The disadvantage of this view is that the BDDs are usually displayed smaller, since the variable order defines a fixed number of levels and not all levels may be used for the BDD. In fact, the number of levels in this view is the greatest possible. Therefore the BDDs are usually looking stretched and with bigger spaces between some levels.

Metadata Visualization

Trees We have already described in detail how metadata could be visualized. Our approach is to use trees with different node sizes. Now we want to describe how such tree structures have an impact on the analysis of the metadata. For the development of algorithms, we want to see the duration of the computation, the memory consumption, the number of used variables and so on. When we include all these metadata types with their values in the node size in the tree decomposition, we immediately see the bottlenecks and the parts where many resources are needed. The bigger the nodes, the

bigger the value. It is exactly visible when resources are needed, when they are released and which configurations provide the best results. In fact, this information can also be used to improve the computation process by adjusting the algorithms.

Statistic diagrams To compare the metadata in some other way, we want to use statistic diagrams. With the help of this approach, we can easily compare different metadata types or a specific metadata type of multiple tree decompositions. They should all correspond to the same x-axis which is illustrating the elapsed time. Furthermore, by selecting particular nodes in the tree decomposition, also the appropriate nodes in the statistic diagrams are highlighted. This helps to understand the relationships between the two visualizations of metadata.

CHAPTER 4

Implementation of the Technical Requirements in DecoVis

This chapter contains our realization of the technical requirements specified in Chapter 3 for the development of a graphical visualization tool for dynamic programming on tree decompositions in form of a web application named *DecoVis*.

Section 4.1 gives a short technical overview over technologies, concepts, frameworks and standards used in *DecoVis*. This background information offers the possibility to easily understand the implementation part of *DecoVis*.

Section 4.2 deals with the interfaces used for importing and exporting instances, tree decompositions and computations in *DecoVis*. Especially the particular data format of each component used for importing and exporting of files are covered. Furthermore, project files for the composition of an arbitrary amount of instances, tree decompositions and computations are presented.

The capability of *dynBDD* execution in *DecoVis* is described in Section 4.3. *DynBDD* is a command line tool applying dynamic programming algorithms on tree decompositions. The focus in this section lies on the explanation of the configuration used for dynamic dialog generation of the graphical user interface. Moreover, the steps in *DecoVis* for initiating the *dynBDD* execution with its options and parameters are illustrated.

Section 4.4 gives an overview over the window management in *DecoVis*. Besides the windows for the visualization of instances, tree decompositions and computations, further window types representing BDDs, computation tables of BDDs, BDD animation and statistic diagrams are explained.

Interactive visualization elements are covered in Section 4.5. In more detail, the interactions between different elements from the same window or other windows are illustrated. Especially the different modes for node highlighting are explained. Moreover, the interplay between BDDs and their computation tables is demonstrated.

Section 4.6 describes the two *DecoVis* modes: view mode and admin mode. Similarities and distinctions regarding functionalities are outlined.

4.1 Technical Overview

This section deals with the technical background of *DecoVis* in more detail. In order to make the system easiliy accessible, the decision was made to develop the visualization system in form of a web application. Since this decision entails restrictions in the used technologies, the following subsections as well as Figure 4.1 depict the used concepts, technologies, frameworks and standards that fulfill the requirements and goals in the easiest and best possible way.

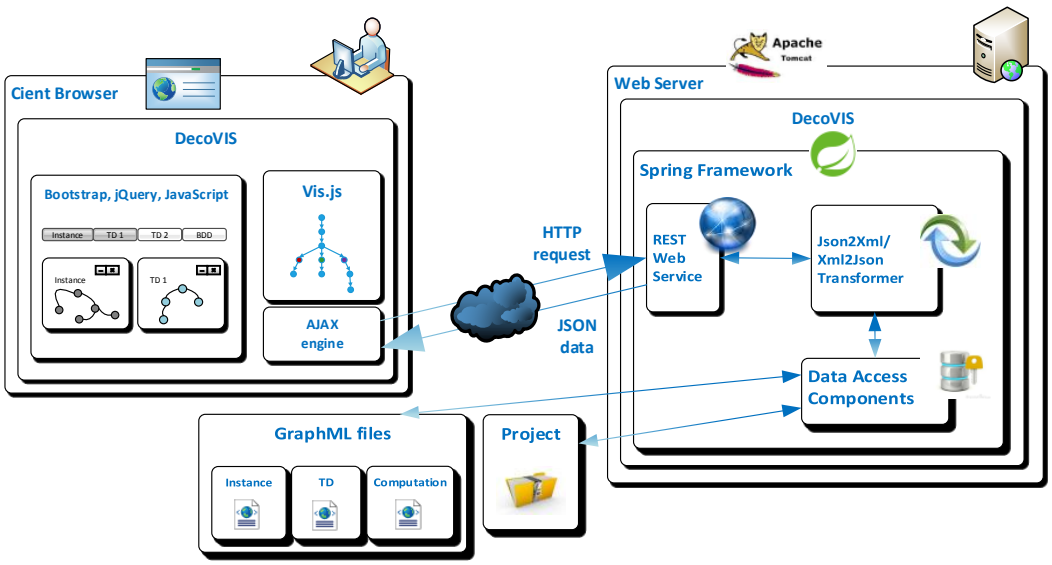


Figure 4.1: Technical overview

In a nutshell, *DecoVis* uses the following dependencies:

- Bootstrap v3.3.2 (`http://getbootstrap.com`)
- jQuery v1.11.1 (`http://jquery.com`)
- Vis.js v4.2.0 (`http://visjs.org`)
- Java v1.7 (`http://www.java.com`)
- Apache Tomcat v7.0 (`http://tomcat.apache.org`)
- Spring Framework v4 (`http://projects.spring.io/spring-framework`)

The following paragraphs describe the used concepts, libraries, frameworks, etc. illustrated in Figure 4.1 in more detail.

Network Architecture *DecoVis* consists of a classical client/server architecture. Since the browser of the client provides the graphical user interface to the user, the application is deployed on the server, or more precisely the web server. By means of the Hypertext-Transfer-Protocol (HTTP) the web server answers to client requests.

Web Server In order to deploy the web application on the server, a web server has to be configured. *DecoVis* uses the open-source web server Apache Tomcat that acts as webcontainer. Since Tomcat implements the specification for Java Servlets and Java Server Pages (JSP), it serves requests to the programmed Java Servlets and JSPs from the DecoVis java project.

Programming languages Since the Tomcat web server runs Java compiled code, the web application is written in Java. For client-side scripting the dynamic programming language JavaScript is used. Furthermore, AJAX (Asynchronous JavaScript and XML) is the technology used for data exchange with the server component and updating of the HTML-page.

Application framework The Spring Framework is an open-source application framework for Java platforms developed for creating high performing and easily testable applications. Since this framework provides a lot of features for the development of web applications, it is the appropriate application framework for *DecoVis*. Furthermore, the Spring Framework is widely spread and provides a comprehensive documentation and library support.

GUI framework Bootstrap is an open-source HTML, CSS and JavaScript framework that is used for the representation of a graphical user interface in a web browser. The decision to use Bootstrap in *DecoVis* is taken because the developed web pages are appropriately rendered on different devices such as smartphones, tablets or computers. Furthermore, web pages can be designed in an easy and fast way.

GUI libraries Vis.js is a JavaScript library for the visualization of graphs and statistic diagrams. Basically it consists of the visualization components: network, timeline, graph2d and graph3d. Since the component network is able to represent undirected graphs and trees, it is used for the visualization of instances, tree decompositions and BDDs. For the statistic diagram in form of a line chart, the vis.js component graph2d is applied. Since the JavaScript library jQuery supports the handling of some Bootstrap elements such as the Bootstrap wizard, it is also included in DecoVis.

Standard Formats As file format for instances, tree decompositions and computations GraphML [59] is chosen. It provides an XML structure and is flexibly extendable. The creation of a new XML schema was not necessary, because GraphML already supports the representation of undirected hypergraphs, hierarchical graphs and application-specific attribute data.

Since JavaScript and a lot of JavaScript libraries, that feed their components with JSON (JavaScript Object Notation) objects, are included in DecoVis, it is obvious to use JSON on the one hand for client side code objects and on the other hand for the data exchange with the web server.

4.2 Data Import/Export

This section gives on the one hand file specifications for the used input and output files and on the other hand importing and exporting features in *DecoVis*. There exist two possibilities for uploading data to *DecoVis*: Either instances, tree decompositions and computations are imported as separated files, or they are imported composed in a project file. The former option assumes, that the user is aware of the relation between tree decompositions and their corresponding computations, whereas the second option contains the information about the links between the files in the project file.

Figure 4.2: Data import/export

Figure 4.2 illustrates the main menu in the main page. Furthermore, access points for file import and project import/export can be found.

4.2.1 File Specification

In order to make information about the components instances, tree decompositions and computations portable, a standardized file format is used in *DecoVis*. Since GraphML [59] offers a file format for graphs with a flexible extension mechanism, this file format is used for each component. In the following the instantiations for the components are illustrated.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<graphml ...>

  <key attr.name="vertex-name" attr.type="string" for="node" id="vname"/>
  <key attr.name="vertex-type" attr.type="string" for="node" id="vtype"/>
  <key attr.name="edge-type" attr.type="string" for="hyperedge" id="etype"/>

  <graph edgedefault="undirected" id="input-instance">
    <node id="v0">
      <data key="vname">a</data>
      <data key="vtype">vertex</data>
    </node>
        :
    <hyperedge>
      <data key="etype">edge</data>
      <endpoint node="v0"/>
      <endpoint node="v1"/>
    </hyperedge>
  </graph>
</graphml>
```

Listing 4.1: GraphML: Instance

Instance

As one can see in Listing 4.1, the outer element **graphml** specifies an XML-file in GraphML-format. Each **key** tag defines an attribute value used for the **key** attribute in the **data** elements. The attribute **attr.name** in each **key** element specifies the name, **attr.type** the data type, **for** the parent tag of the **data** tag and **id** the value in the particular **key** attribute. The definition and usage of the **key** elements stretches over all GraphML-files (Instance, Tree Decomposition, Computation).

The **id** attribute of the **graph** tag indicates the file type, in this case **input-instance** stands for the instance file type. Moreover, a **node** element declares a vertex with its unique identifier, name and type. The unique identifier (**id**) is defined as value of the **id** attribute in the **node** tag. The name and type information can be found in the **data** elements where **vname** and **vtype** attribute values identify the particular information.

The **hyperedge** tag enables the definition of an edge with its type (**etype**) and an arbitrary amount of vertices (**endpoint**). The values of the **node** attributes in the **endpoint** tags refer to the particular **id** attribute values from the **node** elements defined above.

Tree Decomposition

Listing 4.2 specifies a tree decomposition in GraphML-format. The only **key** element defined in this GraphML-file represents the bag (list of vertices) of a tree decomposition node.

As its name implies, the **id hypertree-decomposition** indicates the tree decompo-

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<graphml ...>

  <key attr.name="bag" attr.type="string" for="node" id="bag"/>

  <graph edgedefault="undirected" id="hypertree-decomposition">
        :
    <node id="n3">
      <data key="bag">b, c, d</data>
    </node>
    <node id="n4">
      <data key="bag">a, b</data>
    </node>
        :
    <edge source="n3" target="n4"/>
        :
  </graph>

</graphml>
```

Listing 4.2: GraphML: Tree decomposition

sition file type. The **node** elements with their **id** represent a tree decomposition node. Additionally, the **bag** attribute value in the **data** element depicts the association to instance vertices by means of comma separated vertex-names. **Edge** tags specify the link between a source **node** and a target **node** by means of the **node ids**.

Computation

As Listing 4.3 shows, beside the **key** elements used for **node** and **edge** representation, **key** elements in computation files furthermore consist of metadata definitions used for each tree decomposition node.

The computation graph is identified by **bdd-computation** in the **id** attribute of the **graph** element. Next to the graph tag, a list of variable definitions (**bdd:variable**) follows. Each variable contains an identifier (**id**), an index (**bdd:index**), a type (**bdd:type**), a number (**bdd:number**) and the corresponding vertices from the instance graph (**bdd:vertices**) by means of a comma separated string. The level of the variable in the BDD is represented by the **bdd:index** (details can be found in Section 2.4). If multiple variables in the BDD contain the same content, **bdd:number** defines a counter for these variables. Each **node** element consists of an identifier (should be the same as in the tree decomposition), metadata information and the computed BDD graph.

Metadata information are represented in a **data** element where the **key** attribute represents the metadata type and the element content its value. An arbitrary amount of metadata information can be specified in each **node** element.

Since a BDD is a tree, it consists of an identifier (**id**), nodes (**node**) and edges (**edge**) to its child nodes. Each **node** is represented by an identifier (where the prefix is the id of the BDD followed by "::"). Further **node data** elements can contain information

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<graphml ....>
  <key attr.name="bag-size" attr.type="int" for="node" id="meta-td-bagsize"/>
  <key attr.name="number-of-DD-nodes" attr.type="int" for="node"
     id="meta-bdd-node-count"/>
        :
  <graph edgedefault="undirected" id="bdd-computation">
    <data key="dd-variables">
      <bdd:variable id="var3">
        <bdd:index>3</bdd:index>
        <bdd:type>red</bdd:type>
        <bdd:number>0</bdd:number>
        <bdd:vertices>b</bdd:vertices>
      </bdd:variable>
        :
    </data>
    <node id="n3">
      <data key="meta-td-bagsize">3</data>
      <data key="meta-bdd-node-count">4</data>
        :
      <graph edgedefault="directed" id="n3::">
        <node id="n3::x800a8cc">
          <data key="dd-ncomplement">true</data>
          <data key="dd-nvariable">var3</data>
        </node>
        <edge source="n3::x800a8cc" target="n3::x800a8cb">
          <data key="dd-etype">T</data>
        </edge>
        <node id="n3::x800a8b4Z">
          <data key="dd-nvalue">0</data>
        </node>
        <edge source="n3::x800a8cc" target="n3::x800a8b4Z">
          <data key="dd-etype">E</data>
        </edge>
        :
      </graph>
    </node>
    <edge source="n3" target="n4"/>
        :
  </graph>
</graphml>
```

Listing 4.3: GraphML: Computation

whether the node is a complement node (**dd-ncomplement**), the assigned variable (**dd-nvariable**) and the value in the case of a leaf node (**dd-nvalue**). BDD edges (**edge**) are represented by the parent BDD node (**source**) and the child BDD node (**target**), the edge types (**dd-etype**) THEN (**T**) or ELSE (**E**) and also the information if the edge represents a complement edge (**dd-ecomplement**, not existing in Listing 4.3).

4.2.2 File Import

Basically any number of files can be imported at a time. There are two possibilities for uploading instance files and tree decomposition files over the graphical user interface:

1. Via a big button with the label “select files” in the center of the main page that is shown after the opening of *DecoVis*.

2. Via the ""...”-button that is attached after the tab-list that represent already imported files.

The only thing the user has to bother is establishing the connection between a tree decomposition and its computation. Hence, the file selection of the computation file can be recognizably found in the window of the appropriate tree decomposition.

Particular settings of *DecoVis* can be defined in a global configuration file. The following setting contains the folder used for the file upload defined in the configuration file:

```
temp_dynbddfile_path=/tmp/tmp_files
```

Selected files and also project files are uploaded to this location. Furthermore, the file gets an unique identifier as file name, whereby the mapping between the original filename and the generated identifier gets stored in the web application.

The following sections give an overview over the file types and their information content. The contents of the files are fragmented into important parts that are used for illustration and explanation.

4.2.3 Project Import

A project file is a zip file and furthermore the composition of arbitrary many instances, tree decompositions, computations and additional information. Except for the filename, the zip file includes the input files in their original state. In order to avoid files with different contents but with same filenames, the original filename is renamed. The changed filename is made of a key generator launched from *DecoVis* in the course of each file import.

As the Listing 4.4 shows, the additional information is saved in an ordinary XML file. To each file (graph) extra information such as the indication if the graph was visible (**active**) in *DecoVis* before the project export, the file identifier (**file_id**) to find the corresponding file in the zip folder, the original file name (**file_name**), the window id (**id**) in *DecoVis* and in case of a computation file, the corresponding window id of the tree decomposition (**parent_window_id**).

4.2.4 Project Export

As one can see in Figure 4.2, *DecoVis* provides the generation of a project with a subsequent download. All currently available files from the *DecoVis* session are used for

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<visProject>
    <graphs>
        <active>true</active>
        <file_id>7b1e12a1-94f6-40c2-ab77-5eb7dacaf352</file_id>
        <file_name>6_nodes.lp-hg-pCOL.graphml</file_name>
        <id>1</id>
        <parent_window_id></parent_window_id>
    </graphs>
    <graphs>
        <active>true</active>
        <file_id>50120ba9-fc4e-430b-9183-df146c919fca</file_id>
        <file_name>6_nodes.lp-hg-pCOL-td-seed2-nNONE.graphml</file_name>
        <id>2</id>
        <parent_window_id></parent_window_id>
    </graphs>
    <graphs>
        <active>true</active>
        <file_id>611c347e-9c25-4154-b1ab-7ba2a7afd010</file_id>
        <file_name>6_nodes.lp-hg-pCOL-td-seed2-nNONE-comp.graphml</file_name>
        <id>3</id>
        <parent_window_id>2</parent_window_id>
    </graphs>
</visProject>
```

Listing 4.4: Project file

the project generation. The subsection above illustrates and describes the appended XML file that contains additional information.

The configuration file for the web application contains a publication folder, see example below. *DecoVis* loads and lists all projects found there. The process of project generation followed by project publication can be speed up by specifying the publication folder in the course of a project export. This saves one intermediate step.

```
public_project_path=/tmp/publication
```

4.3 DynBDD Execution

DynBDD is a command line tool for solving complex problems. Furthermore, it uses BDD-based dynamic programming algorithms on tree decompositions for problem solving. More details about *DynBDD* can be found in Section 2.5. Beside the possibility of importing generated tool output externally, *dynBDD* output can be directly generated in *DecoVis*. Moreover, in the course of generation this output can be imported. *DecoVis* provides a wizard for creating instances, tree decompositions and computations.

Figure 4.3 represents the starting page of this wizard. The wizard consists of four tabs: **Instance**, **Tree Decomposition**, **Computation** and **Overview**. In general the first three wizard steps contain input file(s), options (with parameters), a **GENERATE**

button and the generated output files. At the end of the wizard one gets an overview over the generated output files and furthermore an option for importing them into the current session. All imported files and all files available in the current session can then be used for generating new tree decompositions or computations.

4.3.1 Dynamic Dialog Generation

The options and parameters for the execution of the *dynBDD* tool can be chosen in the first three wizard steps. The form of the options in *DecoVis* and their use in the execution tool can be defined in the configuration file.

The first setting that is necessary for the execution is the path where the *dynBDD* tool can be found on the hard disk.

```
dynbdd_path=/home/user/dynbdd
```

By means of the following subsections the configuration settings for dynamic dialog generation and their consequences on the user interface for the generation of the options for instances, tree decompositions and computations are illustrated. The option order in the configuration file is kept for the dialog. The basic syntax for the key of each setting can be found below.

```
syntax: dynbdd.[type].[option].[attribute]
```

As one can see, *dynbdd* represents the beginning of each key in the settings. The *type* depends on the particular output type, i.e. **instance**, **td** and **comp**. The *option* placeholder stands for the option that is used for the execution string for *dynBDD*. *Attribute* represents the usage of the value for the user interface and the execution string. Possible replacements for *attribute* are **option**, **values**, **value**, **label** and **type**.

To explain these possible *attribute* values, the *attribute* **problem** used in each of the first three wizard steps is illustrated. The definitions of the **problem** settings can be found in Listing 4.5, Listing 4.6 and Listing 4.7. Their effect on the elements of each wizard step in the user interface are illustrated in Figure 4.3, Figure 4.4 and Figure 4.5. The option (**option**) in the *dynBDD* execution is called **-p** and has a default value (**value**) **dummy**. The type of the user interface element (**type**) in the wizard dialog is a combobox (**combo**). In this example possible values (**values**) for the combobox are **dummy**, **col**, **ham** and **sat**. The prefixed label for the combobox (**label**) is **Problem**.

Each *dynBDD* execution takes per default an instance file as input. In the case of GraphML-Instance generation (*Tab:Instance*) the input file must be a lp-file, in all other cases a GraphML-file. The definition of the input file format is depicted in the following subsections.

To use a generated tree decomposition file (or an already existing tree decomposition file from the current session) in the *dynBDD* execution as input file, the **value** *attribute* of the declared option must be defined as **{td}**. Moreover, an **option** *attribute* must exist.

In general, the name of the generated file depends on the input file name and is appended by the type and each option (**option**) with its selected value (by default **value**) in the user interface. Options excluded for file name generation are the option names added to the value of the **filenamefilter** key in the configuration file.

4.3.2 Tab: Instance

Listing 4.5 defines the necessary settings for the generation of the **Instance** wizard step in Figure 4.3. As one can see, each key starts with **dynbdd.instance.** followed by the particular *option* and *attribute*.

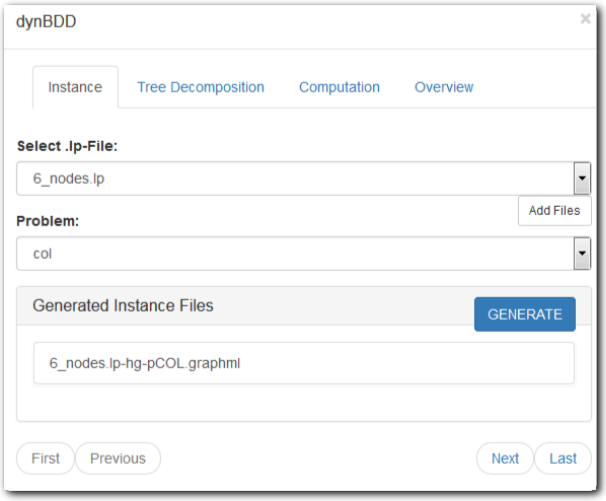


Figure 4.3: Instance window for *dynBDD*

```
dynbdd.instance. ...
problem.option=-p
problem.values=dummy,col,ham,sat
problem.value=dummy
problem.label=Problem
problem.type=combo

output.option=--output
output.value=graphml

only_parse_instance.option=
        --only-parse-instance

print_hypergraph.option=--print-hypergraph

filenamefilter=--output,--only-parse-instance,
        --print-hypergraph
```

Listing 4.5: Instance settings

Since *option* **problem** and the setting **filenamefilter** are already defined above, the remaining settings from Listing 4.5 are explained in the following.

The *dynBDD* options **output**, **only_parse_instance** and **print_hypergraph** have no **type** *attribute* defined. Hence, no user interface elements are generated for them. Each **option** and **value** (if available) can be seen as "static" (fixed) option and parameter for the *dynBDD* execution.

4.3.3 Tab: Tree Decomposition

Beside the already described settings for *option* **problem**, *option* **output** and the general setting **filenamefilter**, further possible settings for the tree decomposition generation are illustrated in Listing 4.6. The generated wizard step in the user interface is shown in Figure 4.4.

Since **only_decompose** and **print_decomposition** have no *attribute* **type** definition, no elements in the user interface are generated for these options. They are used as "static" (fixed) options without parameters for the *dynBDD* execution.

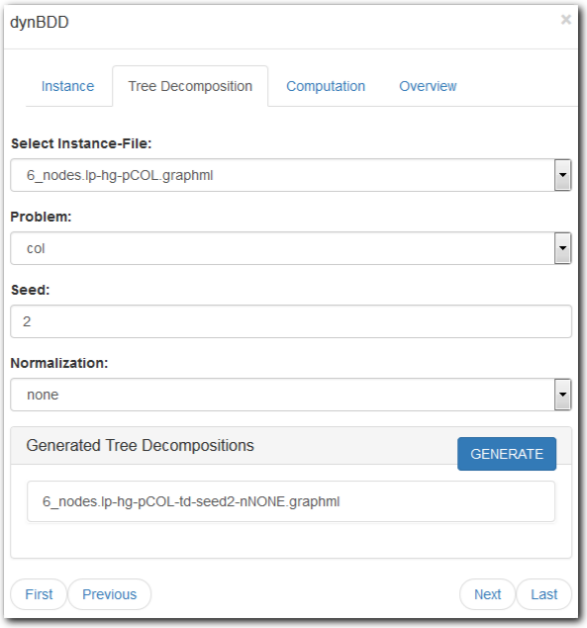


Figure 4.4: Tree decomposition window for *dynBDD*

```
dynbdd.td. ...
problem.option=-p
problem.values=dummy, col, ham, sat
problem.value=dummy
problem.label=Problem
problem.type=combo

output.option=--output
output.value=graphml

only_decompose.option=--only-decompose

print_decomposition.option=
            --print-decomposition

graphml_instance.option=--graphml-instance

seed.option=--seed
seed.value=0
seed.label=Seed
seed.type=field

norm.option=-n
norm.values=none, weak, semi, normalized
norm.value=none
norm.label=Normalization
norm.type=combo

filenamefilter=--output,--only-decompose,
   --print-decomposition,--graphml-instance
```

Listing 4.6: Tree decomposition settings

As already mentioned above, the input file format for the instance file can be chosen. By default, an **lp**-file is used as input for the *dynBDD* execution. Listing 4.6 specifies **graphml_instance** with the value –**graphml-instance**. As illustrated in the combobox for **Select Instance-File:** in Figure 4.4, the input file must be a **graphml**-file from the current session.

The *option* **seed** takes –**seed** as *dynBDD* execution option. The used parameter can be user-defined with the help of a textfield (**field**). Furthermore, the textfield is prefixed with the label **Seed**. Per default, the textfield contains the value 0 (in Figure 4.4 the value is 2).

Norm describes the normalization type by means of a combobox with the possible values **none**, **weak**, **semi** and **normalized**. The default value is **none** and the label for the combobox is **Normalization**. The option for the *dynBDD* execution is –**n**.

4.3.4 Tab: Computation

Listing 4.7 defines possible settings for the generation of the **Computation** wizard step in Figure 4.5. Since *option* **problem**, *option* **output**, *option* **decomp**, *option* **graphml_instance** and the setting **filenamefilter** follow the same schema as the settings explained above, they are not described here furthermore.

The *option* **dm** is visualized in Figure 4.5 as a checkbox (**checkbox**) with the label **Decision Method (Default: ldm)**. The *dynBDD* execution uses –**edm** as option

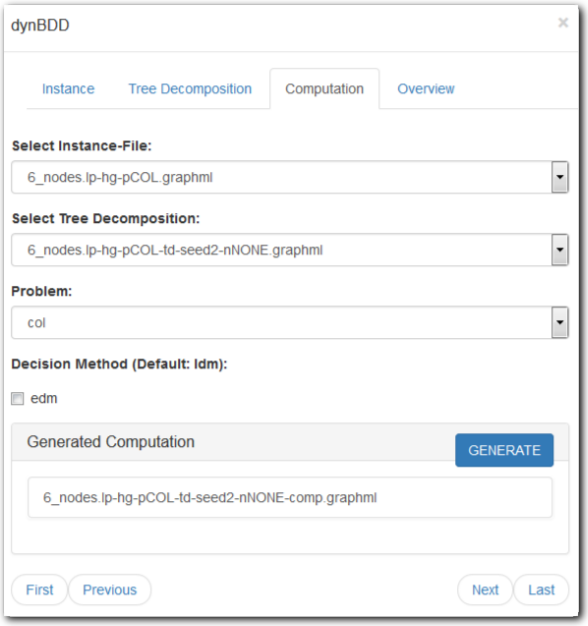


Figure 4.5: Computation window for *dynBDD*

```
dynbdd.comp. ...
problem.option=--p
problem.values=dummy, col, ham, sat
problem.value=dummy
problem.label=Problem
problem.type=combo

output.option=--output
output.value=graphml

dm.option=--edm
dm.label=Decision Method (Default: ldm)
dm.type=checkbox

decomp.option=-d
decomp.value=graphml

graphml_instance.option=--graphml-instance

graphml_td_in.option=--graphml-td-in
graphml_td_in.value={td}

filenamefilter=--output,-d,--graphml-instance,
               --graphml-td-in
```

Listing 4.7: Computation settings

(without parameter) only if the checkbox is selected. Since the checkbox is not selected in Figure 4.5, no option for **dm** information is used in this example.

For computation generation the *dynBDD* execution needs information about the related tree decomposition. The *option* **graphml_td** specifies the "static" (fixed) option for the *dynBDD* execution. **{td}** defines a placeholder for the selected tree decomposition **graphml**-file shown in Figure 4.5. This tree decomposition file (taken from the current session) is furthermore used as input parameter for the *dynBDD* execution.

4.3.5 Tab: Overview

This wizard step gives an overview over the generated output files from the wizard and the files that were already in the DecoVis session before the wizard dialog was started. As one can see in Figure 4.7, the information about the tree decompositions and computations belonging together is emphasized. The **Finish** button imports all newly generated components into the current session. If the wizard is canceled in any step, the generated output files are discarded.

4.4 Window Management

It is necessary to give the user an intuitive and clear window management feature since the most use cases require many windows, which are opened, closed, minimized, reordered

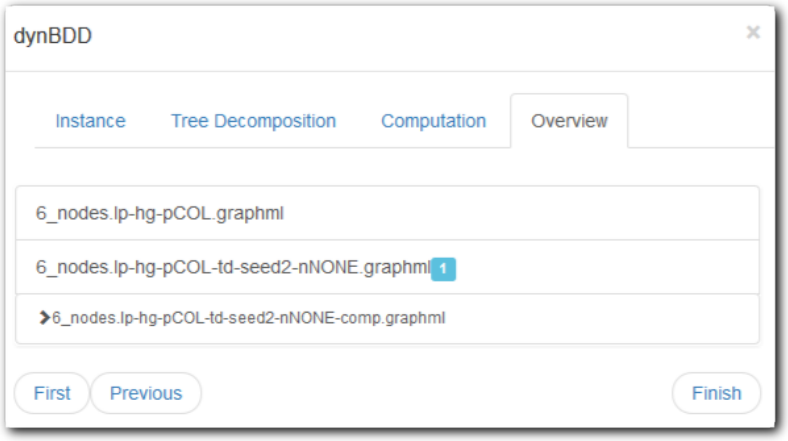


Figure 4.6: Overview window for *dynBDD*

and so on. There are many window types available, interacting with each other. Therefore we invested particular effort in the implementation of a well-thought, easy to use interface including a window management which provides all these features. In addition, our choice for a web application gives us the opportunity to use the new HTML5 and CSS3 features, which support us in the development of an intuitive and user-friendly interface.

4.4.1 Window Handling

The main part of the window management builds the window handling which is responsible for adding, closing, minimizing, reordering windows (see Figure 4.7). For a better overview of the currently available window elements, we implemented a tab bar, showing all visible and minimized windows. They can be closed and minimized from this tab bar. It also includes a button to add a new window, for example to load an instance or tree decomposition file. For all other features which are relying on a window container, the window is created dynamically when it is required.

Below the tab bar, all currently opened windows are visible. They are always adjusted automatically to the browser window. There is no need to adjust the positions manually. We intentionally did this to make sure that always all windows are clearly visible and not overlapping, without wasting empty space. Hence, this saves time for the user to do some positioning of the windows.

In order to increase the usability, the user can specify the maximum number of windows visible at once on the screen. On small screens probably only two or three windows make sense, whereas on big screens up to 9 windows are possible. Additionally, the windows can be reordered individually by drag and drop, but they are always arranged automatically in the grid.

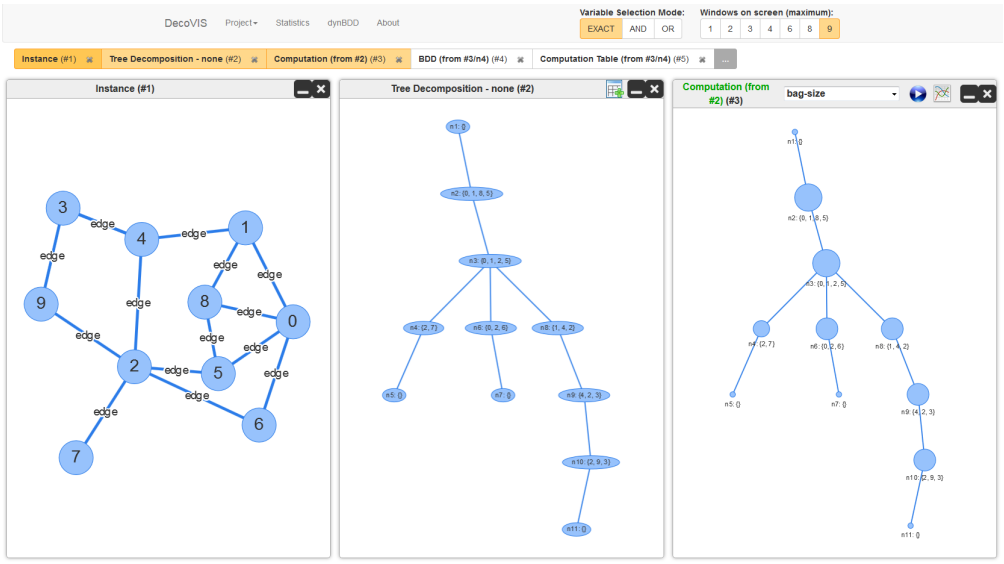


Figure 4.7: *DecoVis*: Window management

4.4.2 Window Types

We implemented 7 different windows types, covering all the required features mentioned in the technical requirements of Chapter 3. Each of them provides a specific use case and also has slightly different user interface elements.

Instance Window

The instance window is used to display the problem instance. The window does not need any special requirements, it just visualizes a simple graph like in Figure 4.8 or a hypergraph like in Figure 4.9. The window has only two buttons, one to close the window and one to minimize it. These two buttons are included in every window, independently of its concrete type. The instance window is either created by loading a *GraphML* file or is dynamically created after the *dynBDD* execution.

Since the chosen library does not support hypergraphs out of the box, a custom graph had to be implemented. There have to be inserted helper nodes having edges to all required nodes, simulating an edge with multiple end points.

Tree Decomposition

This window type is used to visualize the tree decomposition of a particular instance (see Figure 4.10). In the title bar of the window it provides an additional button to add a computation file. Similar to the instance window, the tree decomposition window is created by loading a tree decomposition file or by executing the *dynBDD* process. Hence it does not require any other window or data element and it is completely independent.

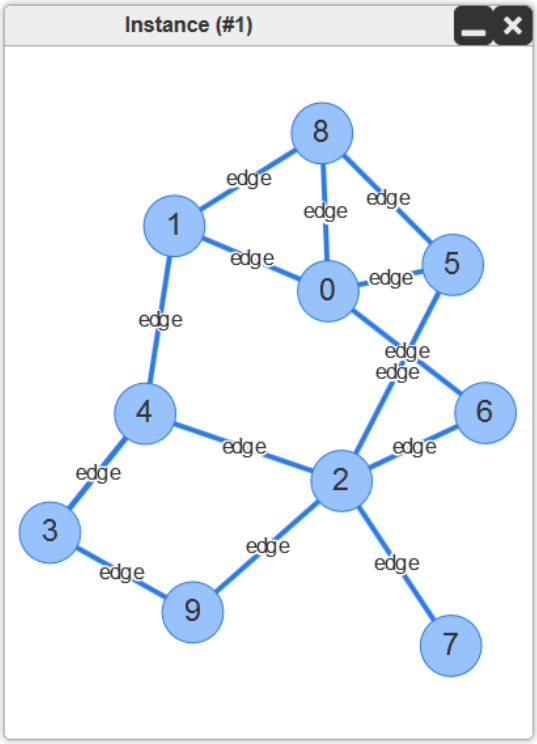


Figure 4.8: Window type: Instance

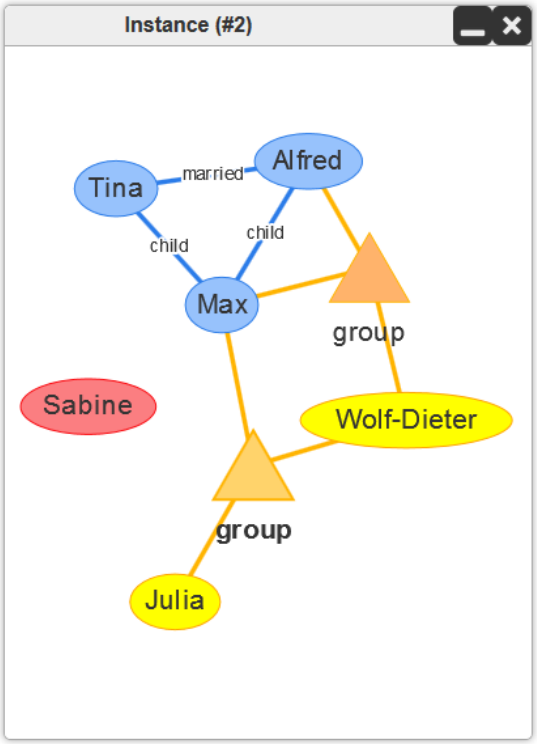


Figure 4.9: Hypergraph example

Computation Window

The computation window is created when loading a computation file in the tree decomposition window or when executing the *dynBDD* process (see Figure 4.11). In contrast to the previous window types, this one depends on some other data, namely the data of the tree decomposition. In fact, it shows the same visualization as in the tree decomposition window, but since it has the computation data included, it has the ability to display some information, i.e. the metadata, about the computation inside the graph. Therefore, this window type has a dropdown field in the title bar to select the particular metadata which should be visualized. Furthermore, this window type provides some more interaction with the graph visualization. By double clicking on a node, the corresponding BDD is displayed in a new window, representing a partial solution of the problem.

BDD Window

As mentioned in the last section, the BDD window is dynamically created when the user wants to display the computation content of a node. Hence, this window type also depends on the tree decomposition window. To let the user know which data the BDD belongs to, the title bar contains the window ID of the tree decomposition and the clicked

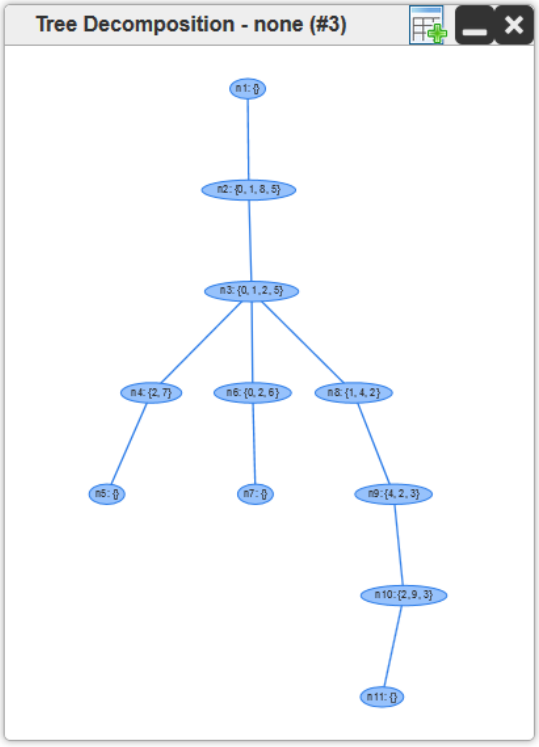


Figure 4.10: Window type: Tree decomposition

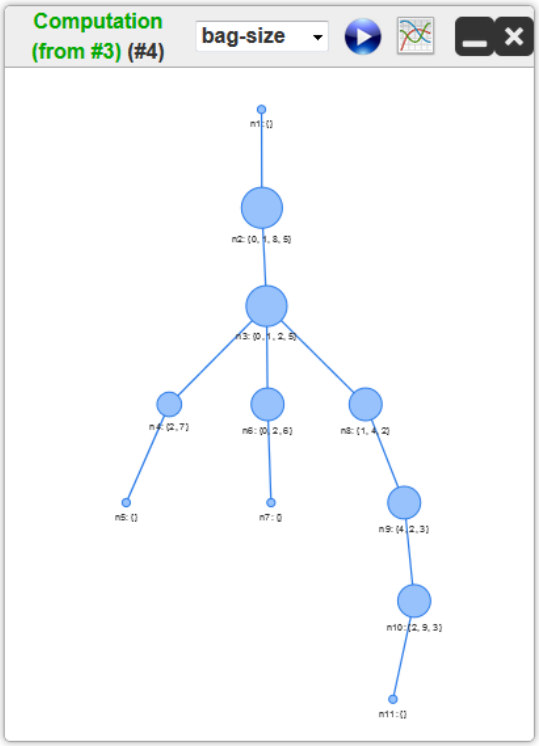


Figure 4.11: Window type: Computation

node ID (see Figure 4.12). Additionally, it contains a button to display the computation table of the current BDD.

Computation Table Window

The computation table window is a little bit different in comparison to the other window types because it does not contain any graph visualization (see Figure 4.13). Instead, it consists of an interactive table element. Since it is created by the BDD window, it relies on it and has some relationships to the BDD. In fact, the computation table enhances the BDD by displaying the solutions in detail. The rows and columns of the table are clickable, which results in highlighting them. Also the relationship between the BDD and the table is used to display some more information to the user. When selecting a row in the computation table, the path in the BDD is highlighted. By clicking on a column, all the nodes of the related vertical level in the BDD are highlighted.

BDD Animation Window

In contrast to the previous window types, this window has many more user interface elements. As mentioned in the requirement analysis, the BDD animation shows the full

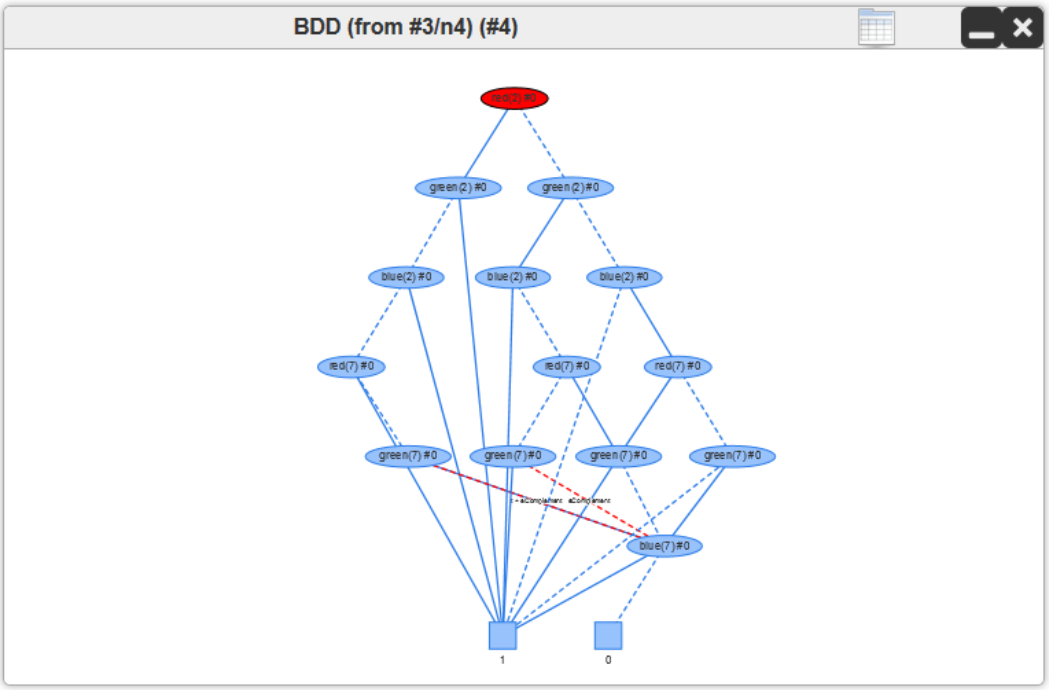


Figure 4.12: Window type: BDD

evaluation of the BDDs from the bottom to the top of the tree decomposition. Usually the animation is done fully automatically, but the user is able to control the animation, of course. Steps can be jumped forwards, backwards, the current step can be paused and the animation stopped. Additionally, the animation speed can be adjusted. All these user interface elements can be found in the title bar as illustrated in Figure 4.14.

There are many transition effects, like fading in, fading out and moving BDDs. With some minor optimizations and calculations this results in a clear animation view by growing, shrinking and merging the BDDs. Hereby, the BDDs representing the partial solutions are transforming to the final solution at the top. In addition, the current animation step is displayed in the tree decomposition window by highlighting the vertical level containing the nodes which represent a BDD each. Usually the user has two opened windows while playing the animation: The tree decomposition window which provides a perfect overview of the overall animation process and the animation window which is visualizing the currently animated BDDs.

Since the animation part is one of the main features of our software, we investigated much time to improve the animation and to make it as fluently as possible. We encountered some performance problems when testing some instances with a large amount of data. An animation step requires to display all BDDs in the current level of the tree decomposition, which results in many nodes and edges which have to be drawn inside the window. Furthermore all the elements are animated, they are moved around and faded in/out.

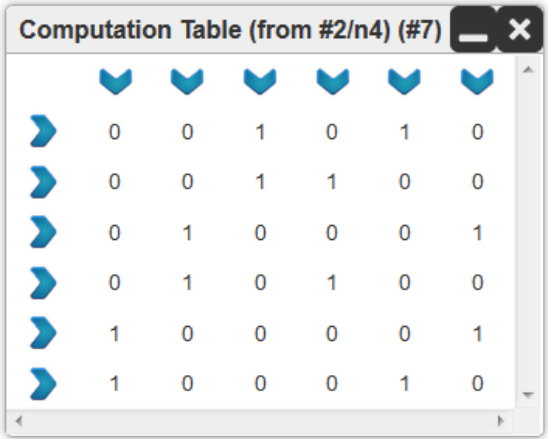


Figure 4.13: Window type: Computation table

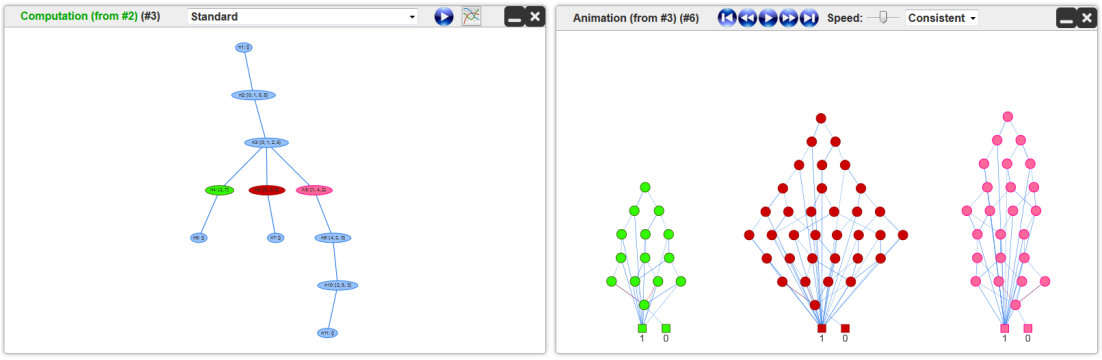


Figure 4.14: Window type: BDD animation

And of course, the next animation step has to be calculated shortly before to provide the transition effects. As we see, there are many calculations to be done. And the more nodes and edges we have, the more overhead is necessary. This could result in an unstable and not very well looking animation due to delays or short freezes. To overcome this problem, we improved the used algorithms for the animation and the related code parts and furthermore, we developed a pre-loading of the animations, so that pictures are taken of the drawn BDD graphs. This is done automatically when running the animation the first time. Hence the first run may be a slow or has some time delays, but the subsequent runs make use of the preloaded graphs. This improves the whole process a lot and results in a smooth and stable animation, even with many nodes and edges without loosing quality or visualization details.

There are different ways how to illustrate the BDDs when we animate them. Depending on the size and data, other views are recommended and useful for a better visualization, analysis and understanding as already described in Section 3.5.3.

The default view mode is called *Consistent View* (see Figure 4.15). As the name may already reveal, this is a view where the BDD size is consistent to all other BDDs independent of the solution and the animation step. Therefore, the distance between two consecutive levels in a BDD is always the same. This makes sense to compare the BDD sizes between the animation steps. However, the BDDs can be smaller in some cases. If a BDD needs a large width to display all the nodes in a particular level, then the height may be smaller and therefore the space between the levels may be also reduced. This makes sure that the nodes and edges of the BDD are still visible and it prevents that the BDD looks like pressed together from the sides.

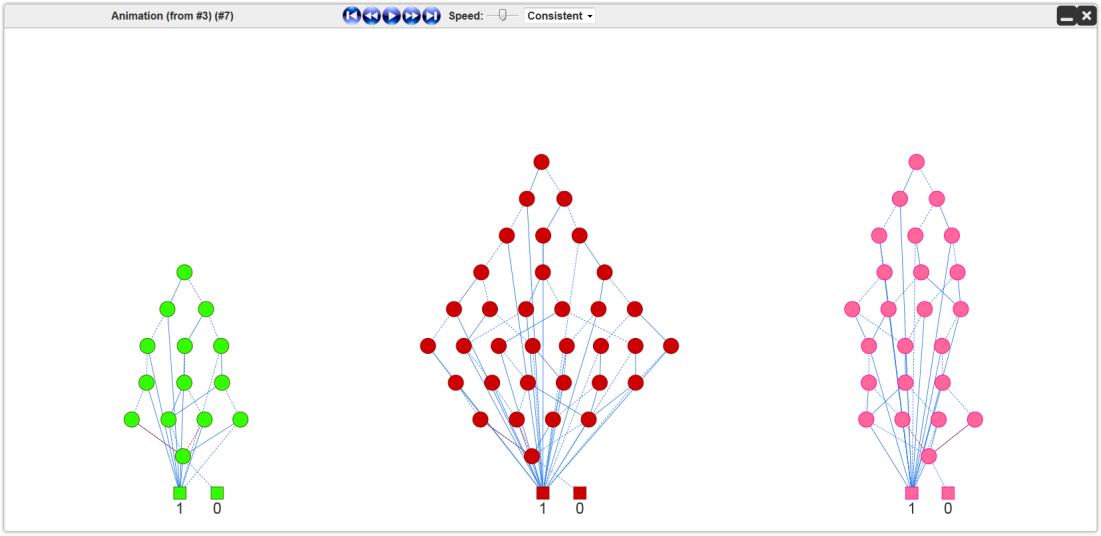


Figure 4.15: BDD animation: Consistent View

The next view mode we implemented is the *Zoomed View* (see Figure 4.16). In contrast to the previous one, the BDDs of the current animation step are always zoomed as far as possible, meaning the BDDs are usually filling the full window. Like in the previous view mode, the BDDs can be smaller in this mode too. If a BDD has a large width, then the height of the BDD may be smaller than the window height. Nevertheless, the advantage of this view mode is that the BDDs are better visible, especially when there are big differences in the height of the BDDs during the animation process. Since each BDD has its own size and is displayed separately, the space between two consecutive levels may be different compared to other BDDs.

The third and last view mode is called *Var Order View* (see Figure 4.16). The name of this view may not be clear first, so let us examine the meaning in detail. When generating the solutions and doing all the computations, we get also a predefined global variable order (see Section 2.4) which is generally used in every BDD visualization. When we traverse the paths in the BDD from the top to the bottom, the occurrences of the variables are exactly in the same order as defined in this global variable order. This is equal for every BDD independent of the view mode. But in the *Var Order View* we apply

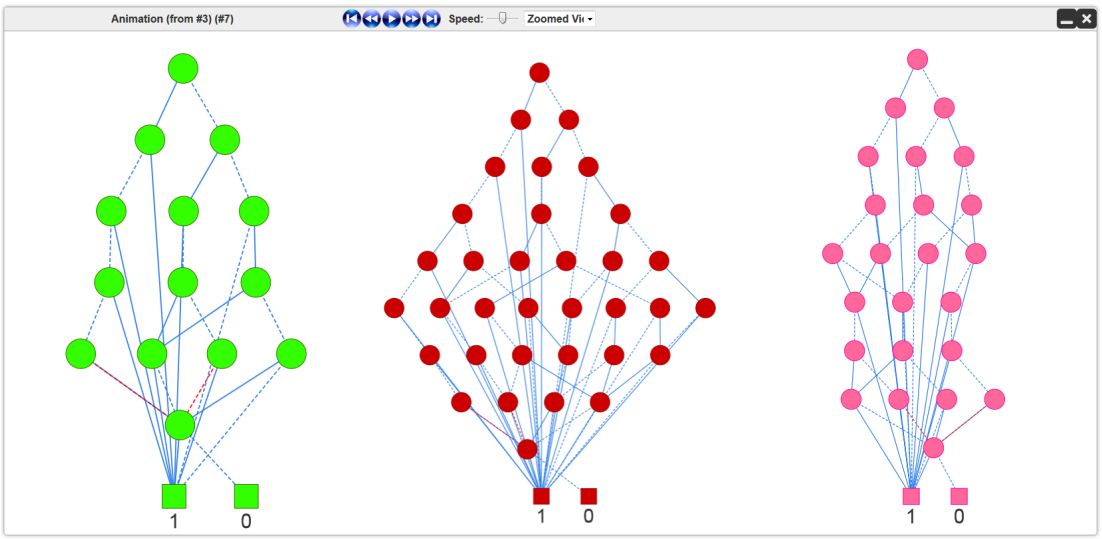


Figure 4.16: BDD animation: Zoomed View

the BDD on the full variable hierarchy. Since not every BDD makes use of all defined variables, we get a stretched BDD. Now, every particular variable assignment is always done on the same level, independent of the BDD size and the current animation step. When a variable is not used in the BDD, the level is just skipped, resulting in a stretched BDD. Hence, every particular level in the BDD may be moved to the relating level of the global variable order. The order of the variable assignments is still preserved, only the space between the levels may be increased due to skipped levels. The great advantage is that the gaps and skipped levels of a BDD may be filled by other BDDs when they are merged. This makes the animation clearer because then the BDDs are interlocking. As mentioned in the other two view modes, the BDDs may be smaller and therefore do not correspond to the variable order. This happens only in very rare cases since the BDDs are much higher than wide in this view mode.

Statistic Diagram Window

The last window type which is used in our software is the statistic diagram window. It contains some other type of visualization: A diagram with x and y axis, which illustrates the statistic of the metadata values of the computation process. The x-axis represents the elapsed time regarding the execution of the computation, the y-axis represents the value of the metadata.

There are two ways how to create a statistic diagram. The first way is to select a metadata type in the dropdown field in the computation window and click the appropriate button to show the statistic diagram. Here, only one single metadata type can be selected as we see in Figure 4.11. But our statistic diagram also supports multiple data sets. Therefore we introduced the *Expert statistic* available in the main menu on the top of

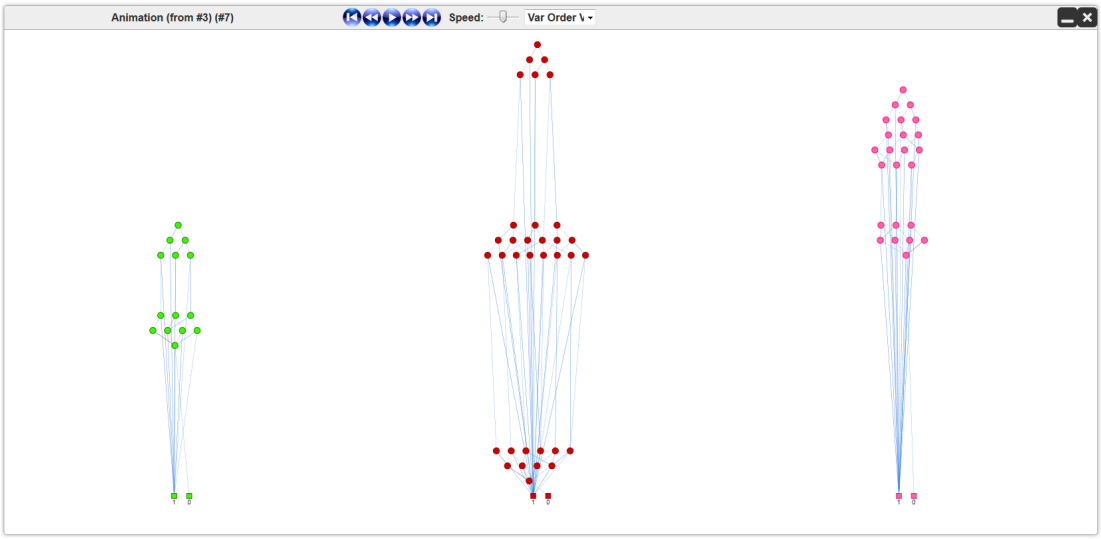


Figure 4.17: BDD animation: Var Order View

the GUI. It provides simple checkboxes of all metadata types of all currently opened computations (see 4.18). It is easy to compare multiple data sets (see Figure 4.19), either different metadata types of a single computation or the same metadata of multiple computations, or even a mix of them. Additionally, the user is able to use a node-based x-axis instead of the time-based one.

As some extra feature, the statistic diagram window contains a button to start a little animation by displaying the nodes one after each other regarding the x-axis.

4.5 Interactive Visualization Elements

The visualization elements embedded in the windows are completely interactive. This section describes all the interactive elements in detail.

For example, the user is able to select nodes and edges, move particular nodes, move the overall graph and zoom in/out. All these features are very helpful, especially for graphs with a large data amount and many nodes. Details about the nodes and edges, like labels and types, may only be visible at a particular zoom level to avoid an overloaded visualization.

4.5.1 Node Highlighting

As already mentioned, nodes in every graph visualization can be selected. By doing that, some other nodes in the same or in other windows may be highlighted automatically depending on the current highlighting mode (also see Section 3.5.3). There are 3 modes available: *EXACT*, *AND* and *OR*. More details about the modes are described in the

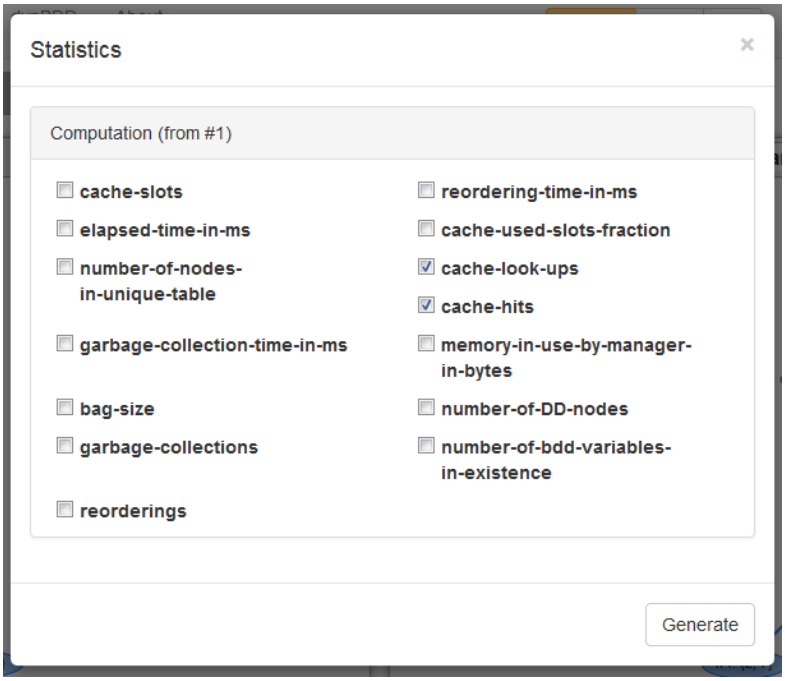


Figure 4.18: Window type: Statistic diagram dialog

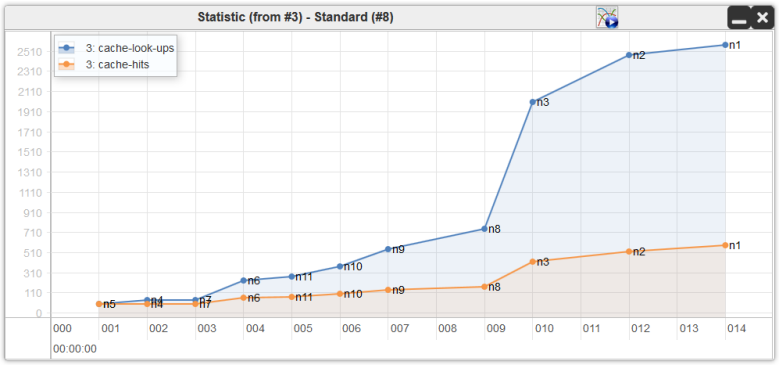


Figure 4.19: Window type: Statistic diagram visualization

following section. Furthermore we want to describe what exactly happens when we select a node in the instance graph, in the tree decomposition or in the BDD window.

Selection Modes

One of the three highlighting modes is called *EXACT*. Like the name may already reveal, only nodes are highlighted which meet the requirements exactly, that is the same vertex or the same set of vertices of the selection. This highlights all the same nodes in other

windows. An example can be seen in Figure 4.20.

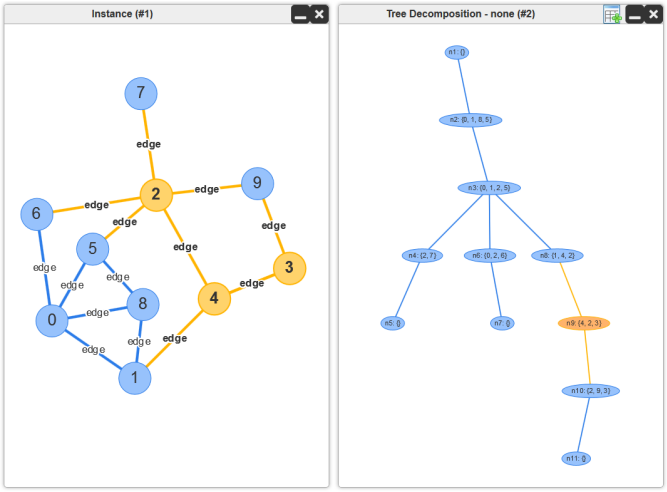


Figure 4.20: Highlighting mode: EXACT

The next mode is called *AND*. Each node may contain a couple of elements. The *AND* mode provides a simple conjunction of these elements. Hence all the nodes are highlighted which contain at least the elements of the currently selected node. Of course, they may contain also other elements, which is the difference to the first mode. In fact, the constraint is less strong than the *EXACT* mode, resulting in more highlighted nodes, illustrated in Figure 4.21.

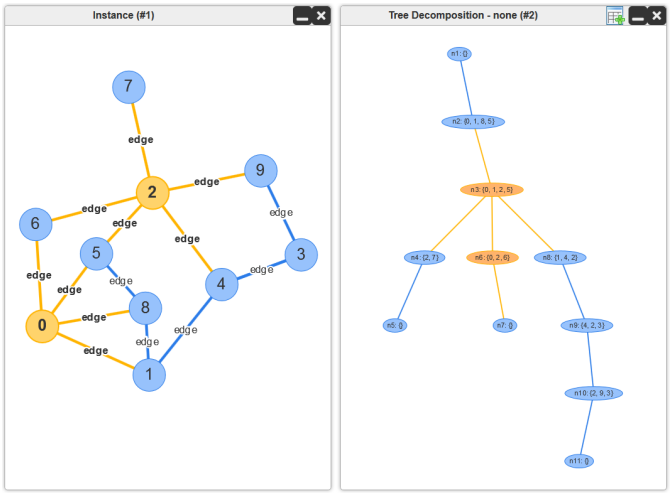


Figure 4.21: Highlighting mode: AND

The third mode sets the weakest conditions for highlighting nodes. It provides a disjunction and highlights all the nodes which contain at least one of the containing

elements in the currently selected node. Therefore, this mode will highlight the most nodes in comparison to the other two modes, see Figure 4.22.

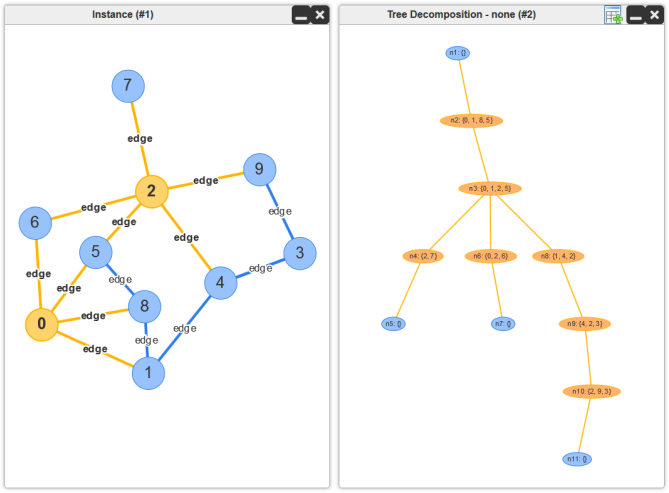


Figure 4.22: Highlighting mode: OR

Instance

Now we want to describe in detail what happens exactly when we select a vertex in the instance graph. When describing the highlighting modes we did not mention some important information: when nodes are selected, the highlighting features act differently for instance graphs and for tree decompositions. That is because instance graphs have only vertices while tree decompositions have nodes containing a set of vertices, called bag. The same applies for BDD nodes which can also have multiple vertices.

The difference for instance graphs only occurs if multiple vertices are selected by the user. This can be done by pressing the *Ctrl* key. In the instance window, multiple selected vertices are still seen as multiple vertices for other instances, but for all the other window types, like tree decompositions, they are seen as a single set of elements.

Let us clarify that by means of an example as shown in Figure 4.23: Assuming we select multiple vertices in the instance graph, like 2, 3 and 4, and we are using the *EXACT* mode. Then other instances will highlight the nodes with 2, 3 or 4, if they are available. But tree decomposition nodes can contain multiple elements and therefore can be matched with the selected vertices as a set of elements. Therefore, in a tree decomposition all the nodes are highlighted which are containing the bag consisting of 2, 3 and 4 in a single node. That is because the selected vertices are seen as a bag in this case. Like already mentioned before, this affects only a selection of multiple vertices in an instance window. A single selected vertex will always have the same behavior in all window types.

Just for information: when we take only instances then it does not matter which highlighting mode we set since they do exactly the same in this case. *EXACT*, *AND* and *OR* fulfill the same requirements because every vertex consists only of the vertex ID. Only when nodes having sets with multiple vertices are used, the modes become relevant, like for tree decompositions or for BDDs.

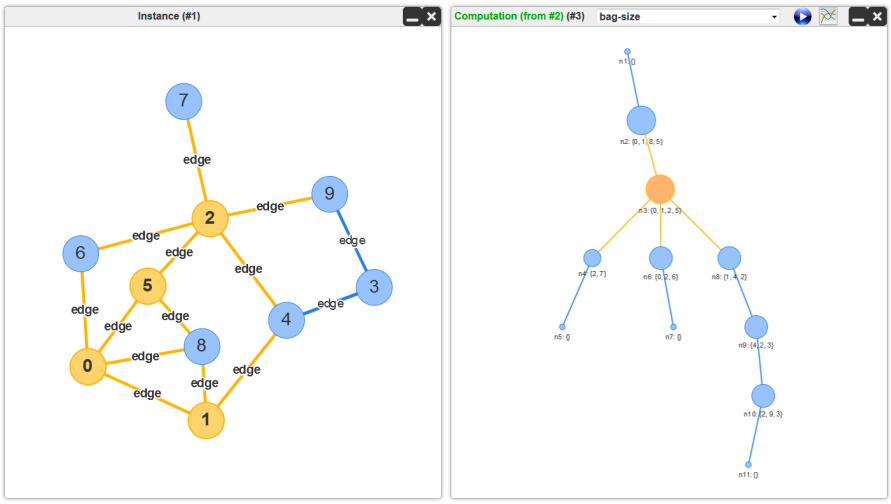


Figure 4.23: Selecting vertices in instance

Tree Decomposition

The behavior of selecting nodes in tree decompositions is similar to that of vertices in the instance graphs (see Figure 4.24). When selecting one or multiple nodes in a tree decomposition, all other nodes in other tree decomposition windows or also in BDDs are matched regarding the current highlighting mode. Each selected node contains a bag of vertices, which is compared with other nodes having also a bag or set of vertices. Like in the previous section, there is a small difference when vertices in instance windows are highlighted. Since the instance vertices only consist of an ID, they are highlighted if they match at least one element in the containing sets in the selected nodes, completely independent of the highlighting mode.

When a statistic diagram is loaded and a tree decomposition node is selected, then the related node in the statistic diagram is also highlighted in this case.

BDD

When selecting nodes in a BDD, the highlighting is done in exactly the same way as already described for tree decompositions (see Figure 4.25) because nodes in BDDs also contain a set of vertices. In BDDs, the nodes contain a little more information than a simple set such as the variable name and an index. This can be seen in the visible

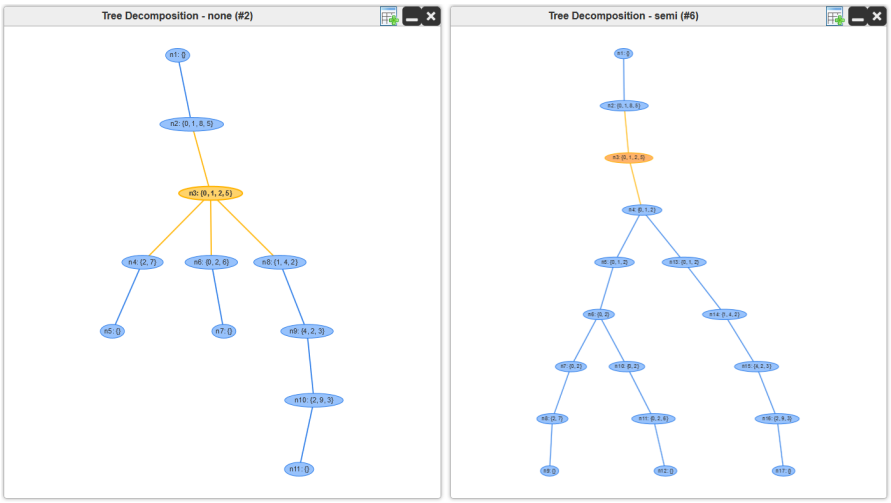


Figure 4.24: Selecting nodes in tree decomposition

label of the nodes. The set of vertices is always defined in the brackets like in the tree decomposition. Instances are the exceptions and their vertices are always highlighted if the element is contained in one of the sets of the selected nodes in the BDD.

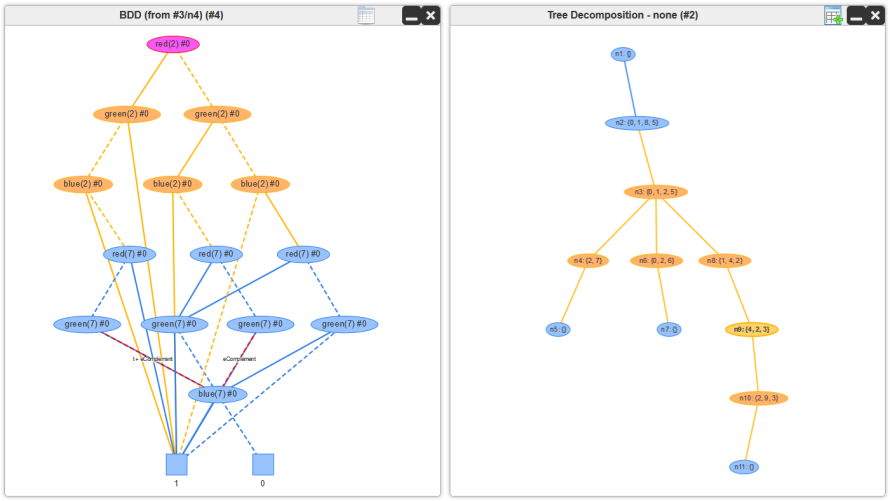


Figure 4.25: Selecting a node in BDD

4.5.2 Computation Table: Path/Level Selection

The computation table provides some enhanced highlighting features. As illustrated in Figure 4.26, by selecting a row, not only the row is highlighted but also the corresponding path in the BDD. Complement nodes and edges are taken into account. Hence, a path

may end in the “0” node, but nevertheless it is a model due to the complement. The highlighting color for complement nodes and edges is slightly different to the normal one to recognize the complement although the nodes are highlighted. As we see in Figure 4.27 the appropriate vertical level in the BDD is highlighted after a column is selected.

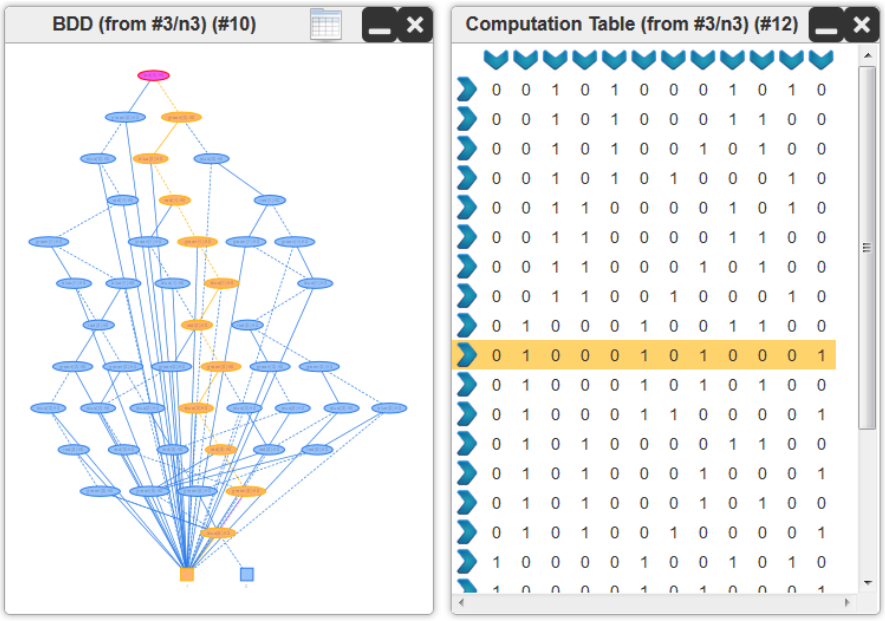


Figure 4.26: Highlighting row in computation table

4.6 User Modes

There are two user modes in the *DecoVis* software: the view mode and the admin mode. These modes can be defined in the config file, by adjusting the following line:

```
### view, admin ###
usermode=admin
```

4.6.1 View Mode

The view mode is intended for the normal website user and is restricted (see Figure 4.28). Only predefined projects can be loaded, there is no way to load own files and *dynBDD* commands cannot be executed either. Also the export feature is disabled. This way the tool works as simple viewer. But the user is able to use all the other features and the entire interactive visualizations are enabled without any limitations. The software is designed in such a way that many people can use the viewer concurrently, without any significant performance issues on the server since the visualization is done completely by

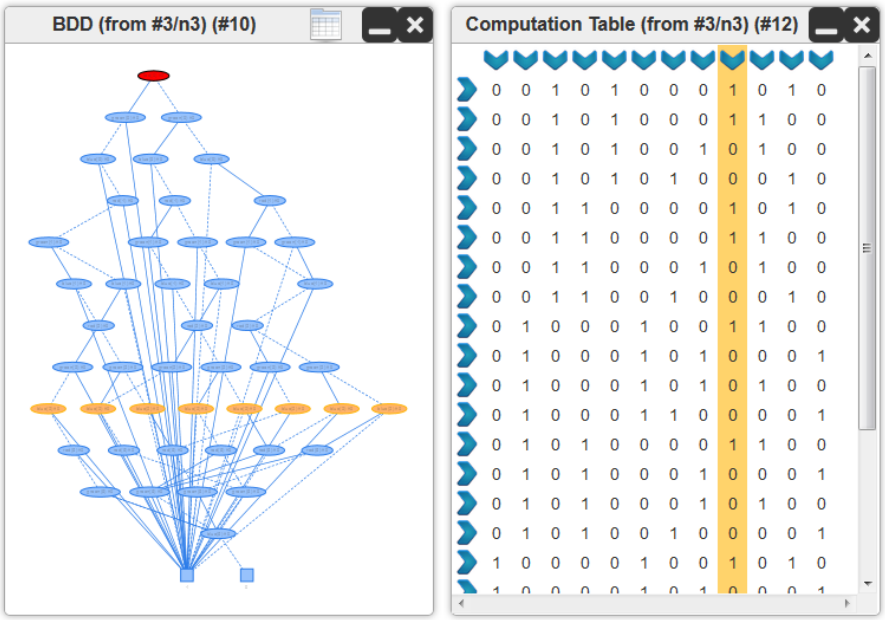


Figure 4.27: Highlighting column in computation table

the browser and therefore on the user's computer. The restrictions of the user prevent that some expensive calculations are done on the server and of course they improve the overall server-side security.

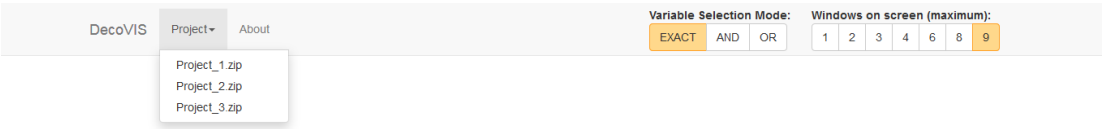


Figure 4.28: View mode

4.6.2 Admin Mode

In contrast to the view mode, the admin mode provides full control of the *DecoVis* software. All features are enabled in this mode like the *dynBDD* execution, the file import and the project import/export. Of course, all other features of the view mode are also enabled. All the exported projects done in the admin mode can be used as predefined projects in the view mode. Only authorized people should get access to this mode since it provides command line executions by running the *dynBDD* process which may affect the security and accessibility of the server when misused.

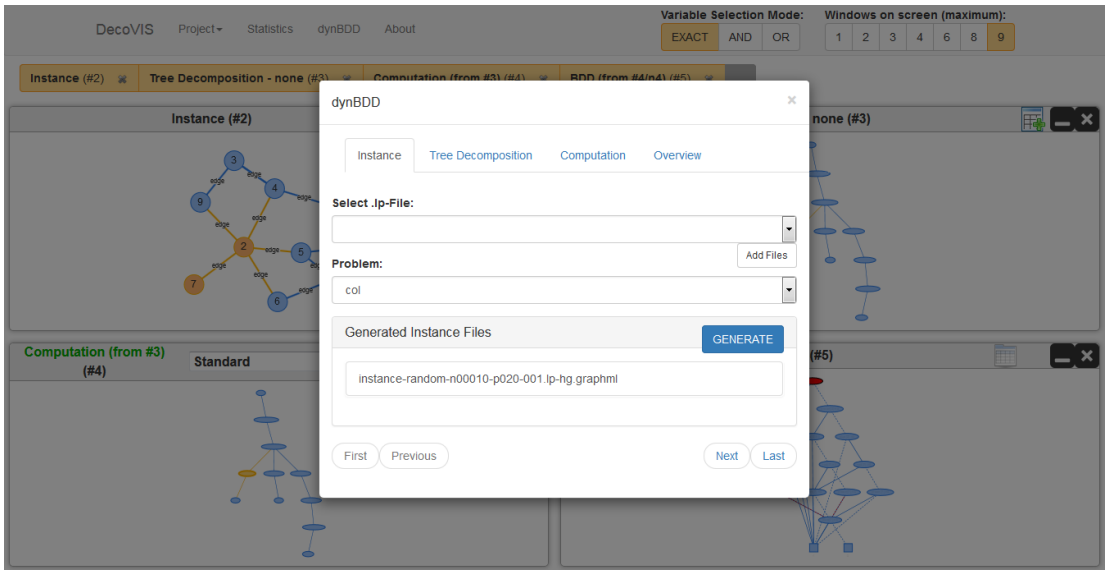


Figure 4.29: Admin mode

CHAPTER 5

Comparison of TD-based DP Systems

In this chapter we present the differences of various dynamic programming systems on tree decompositions regarding performance and stability. This is done by executing many benchmarks tests and then by analyzing the results in detail. For this purpose it is not necessary to know the internal implementations of all the systems (even though they are available as open source), therefore they are handled like black-box-systems. We feed them with some input data, i.e the instances of different problems, and analyze the output afterwards. Any information about the internal processing is not relevant in this case. To get a meaningful result, many problems and instances have to be considered, of course. Later on, some of the results are examined in *DecoVis* by using the statistic diagram feature and the graph visualizations.

5.1 Challenges and Goals

There are many challenges when dynamic programming systems are compared. First, appropriate problems and instances have to be considered to get a meaningful result. Since it is not possible at all to test each instance of every problem, only particular problems with particular instances have to be taken into account. The challenge is to select only these problems and instances which cover a wide range or are more likely to be used in real world scenarios.

Second, all systems must be able to support the the solving of the selected problems and they also have to offer the same problem type like decision, counting or enumeration problems. Otherwise, the result would not be comparable. Hence, the assortment of appropriate problems may be very restricted. Since the internal overhead of counting problems is often negligible regarding decision problems, the only exception may be the use of decision problems together with counting problems.

Third, the computations of the different systems should be done by using the same tree decompositions because the tree decomposition may have a strong impact on the runtime and memory usage. Therefore, all systems have to provide the feature of defining a particular tree decomposition as input.

Another challenge is to select meaningful criteria and properties when comparing the systems. An example would be to compare the runtime and/or the memory usage during the computations because they can be measured externally without relying on the particular tool. There might also be other data which can be used, for example some BDD specific metadata, but all of them are system dependent and therefore not comparable among each other. The selected properties also have to be illustrated, for example by using a bar or scatter diagram. Attention should be paid to the clarity because this visualization should be used to demonstrate the performance differences and benefits.

Last but not least, it is important to call the systems with the proper options and arguments. Often they provide additional information as output like whole solutions or debugging data which may falsify the result because writing the output (into a file or just to the console) takes time and memory. That is why passing the right arguments which deactivates all these unnecessary output is very important.

As a result, the goal is to get a very accurate and meaningful comparison of the selected dynamic programming systems by solving all mentioned challenges as far as possible. We want to know which system is the best regarding runtime and memory. Furthermore it would also be very helpful to get to know whether there are performance differences between the selected problems and instances. Some systems may be better than others only for some particular problems or instances. Therefore the goal is not only to get one best system but rather to get the information about which tool is well suited for which problem or which type of instance in detail. All the performance information gathered during the various benchmark tests should be illustrated in meaningful and clear diagrams.

5.2 Requirements

Before we are able to run the benchmark tests, we need to prepare the dynamic programming systems, the benchmark server and also choose the right problems, i.e. the type or category of the problem, the problem itself and the corresponding instances.

5.2.1 Choosing the Problems and their Instances

There exist several problem types such as decision, counting, optimization and enumeration problems. The output of decision problems is only *true* or *false*. The result of counting problems is the number of available solutions. Enumerating problems produces a list of all available solutions, whereas optimization problems result in only optimal solutions with regard to an optimization criterion (for example the best or worst solution

regarding the minimal or maximal number of vertices). For our purpose, only the first two problem types make sense, because the optimization problem can only be used for particular problems and the enumeration problem would not be suitable for a comparison. Enumerating all solutions of a problem would result in a memory and time issue since writing to an output (file or console) is quite expensive and the more solutions are available, the more affected would be the result. Generally, enumerating the solutions internally, even without writing them out to the console, takes exponential time. That is why we decline this problem type, but nevertheless, that is not relevant for a meaningful comparison of dynamic programming systems. It is only important that we get to know whether one or multiple solutions are available. Since we assume that all the solutions are correct, we usually do not need to verify them and also do not need to compare them with other solutions produced by different tools either. Nevertheless, we have a look at the results and quickly compare them just to make sure that the encodings of the problems and also the options for the particular tools work correctly.

Now we introduce the concrete problems we want to use in our benchmark tests: *3-Colorability*, *Hamiltonian Cycle*, *Stable Extension*, *Vertex Cover* and *Preferred Extension*. We use many different instances for all these problems:

- random graphs,
- subsets of grid-based graphs and
- real-world-instances

The first two groups will be tested with many different instances each. In detail, for the random graphs we start from 10 vertices up to 75 vertices per instance. It is important to say that the treewidth for random graphs is only limited by the number of vertices. For the grid-based graphs we use instances starting from 10 vertices up to 1000 vertices. In detail, we use grid-based graphs where the vertices are arranged in a grid and each vertex has up to 8 neighbor vertices: left, right, top, bottom and the 4 diagonal vertices. But we always use a subset of the edges of this grid-based graph to generate an instance. Here, we use a fixed treewidth of 5 and 20 to get to know how the treewidth has an impact on the performance.

As listed in the following, we also take real world instances into account:

- Metro Beijing (231 vertices and 256 edges)
- Metro Berlin (174 vertices and 187 edges)
- Metro London (304 vertices and 410 edges)
- Metro Munich (97 vertices and 134 edges)
- Metro Shanghai (292 vertices and 332 edges)

- Metro Singapore (127 vertices and 143 edges)
- Metro and Tram of Vienna (138 vertices and 328 edges)
- The Graph and Digraph Glossary (72 vertices and 118 edges) [60, 61]
- The Bibliography for the Book "Product Graphs" (674 vertices and 613 edges) [62, 63]

The last two are based on the Pajek datasets by Batagelj and Mrvar [64]. Especially these instances are very important since they represent scenarios which are more likely to be used in reality and cover a completely different domain comparing to the randomly generated graphs. Therefore these instances have a high relevance when doing meaningful comparisons. All of them have a treewidth which is not greater than 8.

5.2.2 Dynamic Programming Systems

We want to use four different dynamic programming systems on tree decompositions for our comparison: *dynBDD*, *D-FLAT*, *D-FLAT^2* and Sequoia. All these tools are already introduced in Section 2.5 where the most important features and some details about already available comparisons are revealed. Now we describe the requirements for our benchmark tests. Every tool has different command line options which may have an impact on the result. We show the concrete calls for every tool which are used for all tests. As mentioned before, there are many different types of problems. Not every tool supports all types and the same applies for concrete problems.

Therefore we are unable to use the same problem type and problem for all tools, but in contrast to that, we are trying to do several comparisons, each having a different focus. All tools can handle decision problems. Enumeration and counting problems are supported by *D-FLAT* and *D-FLAT^2*. *Sequoia* and *D-FLAT* are both capable of solving optimization problems. As mentioned before, enumeration problems can not be used. We are not planning to use any optimization problems, hence this problem type is also irrelevant for us. The overhead of counting problems in contrast to decision problems can be neglected. If there is at least one solution available, this equals to *true* for decision problems. If there is no solution available, it equals to *false*. These assumptions result in the following comparisons:

- The *3-Colorability* and *Hamiltonian Cycle* problems are compared in the course of a decision problem using the tools *dynBDD*, *D-FLAT* and *Sequoia*.
- The *Stable Extension* problem is solved by *D-FLAT* and *dynBDD*, also done by decision problems since *dynBDD* is only able to handle this problem type.
- In order to show the benefits of processing in two stages for minimization/maximization problems, the *Preferred Extensions* problem is used. Here, *D-FLAT* and *D-FLAT^2* are compared, where the former provides only a one-stage computation and the latter has support for two-staged computations. For this comparison the number of solutions regarding minimization are considered.

5.2.3 Comparing Data

We have already specified the problem type, the concrete problems, their instances and also the relating tools which are capable of solving the problems. Now we want to describe how the results are being compared. First, for every problem, together with a particular tool, we define a benchmark run. Then, we need the data of the output result of each run. Since, each tool has a different output, we need to collect them and evaluate them afterwards. If the tool was not able to find a solution in some predefined time or memory or it produced an error, then the result is marked as invalid and it cannot be taken into account. In detail, we used a time limit of 3 minutes and a memory limit of 4GB. Higher time limits are not very practicable for this number of instances. When the tool produces a solution within the time and memory limit, we have a quick look whether the solutions are correct (by checking them with the outputs of the other tools) and take the used time and memory of this run as result. Next, all results are compared with the help of diagrams. For each problem and data type (time or memory) a separate diagram will be used containing all the different tools which were used for this particular problem. Every tool is then depicted by a tool-specific color.

A line chart is used for the grid-based and random graphs. For each instance we generate one single tree decomposition which is then used in all tools. The x-axis contains the number of solved instances, in detail a particular x-value describes one single instance. The y-axis illustrates either the memory or time usage. It is important to say that the x-value does not describe the same instance for all systems. The x-values of a particular system are sorted by their y-value. Hence, the x-value can not be compared among the dynamic programming systems. But in contrast to that, the values (runtime or memory) of a particular dynamic programming system are now monotonically increasing in the diagram. Otherwise we would get confusing diagrams since the values usually fluctuate extremely even if the number of vertices is increasing. Additionally, it is easy to see when there were occurrences of errors like timeouts or an out-of-memory because such results are missing in this case.

For the real-world instances we use a bar chart instead of a line chart. We generate 10 different tree decompositions for each instance which are then used in all tools. The reason for this is, that one single tree decomposition may not be suitable for a particular dynamic programming tool, because tree decompositions can have a strong impact on the runtime. Since we do not have as many real world instances as random and grid-based ones, the output would falsify the end result much more if one tree decomposition is "bad" for a computation. That is why we use many different ones to eliminate this issue. For the bar chart we now use the minimum and therefore the best result. The reason why we will use the minimum value and not the average value is, that if one of the 10 runs fails, we would not be able to calculate the average value. Every instance is grouped by the tools and the related minimum time and minimum memory data is illustrated by a bar. Each tool has a different color. The x-axis shows the name of the particular instance and the y-axis shows the value of the used time or memory.

5.2.4 Benchmark Server

For the overall benchmark tests we used a server at our university which is principally designed for benchmark purposes. The server runs Linux as operating system and provides an octa-core Intel Xeon processor with 3.5 GHz per core (Intel(R) Xeon(R) CPU E5-2637 v3 @ 3.50GHz). About 256 GB of RAM are available. Only one single core is used for the benchmarks test and each test is executed after another.

5.3 Benchmark Preparation

Instances

As mentioned before, we want to use random instances, grid-based instances and real-world examples. Now we want to generate all needed instances. For this purpose we are using a tool, called *GraphGenerator.jar* that is written in Java, to generate a graph. It takes the following arguments:

- **-g**: graph type (random, grid-based...)
- **-n**: number of vertices
- **-t**: maximal treewidth (only for grid-based)
- **-p**: probability to set an edge to a possible vertex

Hence, the tool generates a single graph per run and produces a *.lp* file as output containing *edge* and *vertex* predicates suitable as input for most of our dynamic programming tools. By using a shell script we generate 75 random graphs, beginning from 5 vertices up to 79 vertices. The edge probability is decreased with each additional vertex, to prevent that graphs with more vertices get too many edges. In detail, we used the formula $1/i * 2$ where i is the number of vertices. This results in building a constant number of edges regarding the overall graph, meaning that for every added vertex approximately 2 edges will be inserted.

Then we create 250 grid-based graphs, beginning from 10 up to 1006 vertices, thus always increased by 4 vertices. Here we use an edge probability of 50%. Since every vertex can only have up to 8 edges (left, right, top, bottom and the 4 diagonals), we have an average of 4 inserted edges for each added vertex. Furthermore, we set the maximal treewidth to 5.

The same is done for another 250 grid-based graphs, but the maximal treewidth is now set to 20.

The real-world instances are already available as *.lp* files, so that they are ready to be used in *dynBDD*, *D-FLAT* and *D-FLAT^2*.

An exception is *Sequoia*: some additional preparations and file conversions have to be done for all instances to be suitable for *Sequoia*, i.e. converting the *.lp* files to so-called *.leda* files which contain the graph representation as *LEDA* format [65]. *Sequoia* only

accepts this type of format as input. Therefore, we used self-written Java tools to convert all the files.

Furthermore, for argumentation problems like *Stable Extension* and *Preferred Extension* we have to alter the instance files a little bit since argumentation problems require *arg* and *att* predicates instead of the *vertex* and *edge* ones. This is done by simple replacements.

Tree decompositions

For every existing instance file we need to generate a tree decomposition which is used for all the dynamic programming tools. For the self-generated graphs, i.e. random graphs and grid-based graphs, we need exactly one single tree decomposition. For the real-world instances we need 10 different tree decompositions each. As mentioned before, this is because a tree decomposition can have a very high impact on the runtime and memory usage. We do not have as many real-world instances available as randomly generated graphs, therefore only one single tree decomposition for each of the 10 real-world instances could falsify the overall result very easily. By using 10 different ones each, we get 100 runs in total for the real-world instances.

As we always use 75 or 250 instances for all the other instance groups, it is enough to use only one single tree decomposition in these cases. The tree decompositions are generated with the help of *dynBDD*.

We need 4 different variables:

- **lp_file**: the LP file which contains ASP encoded instances
- **td_file**: Tree decomposition file
- **td_file_sequoia**: Tree decomposition file for *Sequoia*
- **seed**: Seed value for the tree decomposition

Command to generate the tree decompositions:

```
dynbdd -p col --output graphml --only-decompose --print-decomposition --seed
    ${seed} -n normalized < ${lp_file} > ${td_file}
```

Command to generate the tree decompositions for Sequoia (using Shell, only a single line has to be changed inside the file to be compatible with Sequoia):

```
cat ${td\_file} | sed -e "s/attr.name=\"bag\"/attr.name=\"sequoia_bag\"/" >
    td_file_sequoia
```

Used Tools and Software

To run the benchmark tests, we are using the following tools and software:

- *dynBDD*: Version 1.0.4
  - using CUDD 2.5.0 and Sharp 1.1.1
- *D-FLAT*: Version 1.1.0
  - using Gringo 4.5.3, Clasp 3.1.3 and Sharp 1.1.1
- *D-FLAT^2*: Version unknown, from 14 Dec 2014
  - using Gringo 4.4.0, Clasp 3.1.1 and Sharp 1.1.1
- *Sequoia*: Version 0.9, from 28 Aug 2015
- Several self-written shell scripts and tools:
  - to generate the instances (LP files, containing edges and vertices),
  - to convert the LP files to GraphML files,
  - to generate the tree decompositions,
  - to convert an LP file to the LEDA format,
  - to adjust the existing instance files to be compatible with argumentation problems (e.g. *Stable Extension* and *Preferred Extension*) and
  - to run the benchmark tests including the measurement of runtime and memory usage.

Computations

Now we have everything that is needed to do the computations and benchmarks using the 4 dynamic programming tools.

In our shell scripts we use the following variables:

- **lp_file**: the LP file which contains ASP encoded instances
- **leda_file**: the LEDA file needed for *Sequoia* which contains the instance
- **td_file**: tree decomposition file
- **td_file_sequoia**: tree decomposition file for *Sequoia*
- **problem**: problem name for *dynBDD*
- **problem_lp**: problem definition for *D-FLAT and D-FLAT^2*
- **problem_mso**: problem definition for *Sequoia*
- **seed**: seed value for the tree decomposition (only needed for subset problems)

dynBDD command:

```
dynbdd -p ${problem} --output quiet -d graphml --graphml-td-in ${td_file} <
    ${lp_file}
```

D-FLAT command:

```
dflat -p ${problem_lp} --depth 0 --no-counting --tables --output quiet -d
    graphml --graphml-in ${td_file} < ${lp_file}
```

Sequoia command:

```
sequoia -f ${problem_mso} -e Bool -g ${leda_file} -t ${td_file_sequoia}
```

Sequoia command to use an incidence graph internally (like for *Hamiltonian Cycle*, where an incidence graph is required for the code written in the MSO file):

```
sequoia -f ${problem_mso} -e Bool -g ${leda_file} -t ${td_file_sequoia} -2
```

D-FLAT command for subset problems (like *Preferred Extension*, which will be discussed later on in more detail)

```
dflat -p ${problem_lp} -e arg -e att --tables-max --output quiet --depth 0
    -n normalized --seed ${SEED} < ${lp_file}
```

D-FLAT^2 command for subset problems:

```
dflat -p ${problem_lp} -e arg -e att --tables-max --output quiet -n
    normalized --seed ${SEED} < ${lp_file}
```

Hint: when using subset problems, we define the seed value of the tree decomposition explicitly in the command because *D-FLAT^2* provides no option to define an already existing tree decomposition. It would not make sense to define only the existing tree decomposition for *D-FLAT* alone. Therefore we feed *D-FLAT* and *D-FLAT^2* with the same seed value during the benchmark, resulting in the fact that a particular seed value produces the same tree decompositions in both tools.

Preparations for DecoVis

For the *3-Colorability*, *Stable Extension* and *Hamiltonian Cycle* problem we want to pick one or two instances which look interesting for a detailed analysis and load them into the *DecoVis* tool. We either compare one instance by using two different tools or we compare two instances by using one single tool. For this purpose we need three files: the instance graph as *GraphML* file, the already generated tree decomposition mentioned before (which is already in the *GraphML* format) and the computation files containing metadata about this run. Since only *dynBDD* and *D-FLAT* provide options to output such metadata files, we can only use these two systems for a detailed analysis. For *dynBDD* we use the metadata types called *elapsed-time-in-ms* (representing the current runtime) and *memory-in-use-by-manager-in-bytes* (representing the memory usage in

bytes of the BDDs). For *D-FLAT* we use the *time* metadata (representing the current runtime) and *item-tree-size* (representing the memory usage in bytes of the item tree). It is important to say that *memory-in-use-by-manager-in-bytes* and *item-tree-size* describe only the memory usage of the data structure (BDD or item tree) of a particular node of the tree decomposition and not the overall current memory usage by the system when processing this node. Therefore these metadata types are only restricted comparable for the total memory usage, discussed later on.

5.4 Benchmark Runs

5.4.1 3-Colorability - DynBDD, D-FLAT, Sequoia

The 3-Colorability problem is supported by *dynBDD*, *D-FLAT* and *Sequoia*. The output of the problem states whether a graph is colorable with three different colors whereby two vertices must not have the same color if there is an edge between them.

As mentioned before, we take three different instance groups for our benchmark test: random graphs, grid-based graphs and real-world-instances.

Results & Comparison

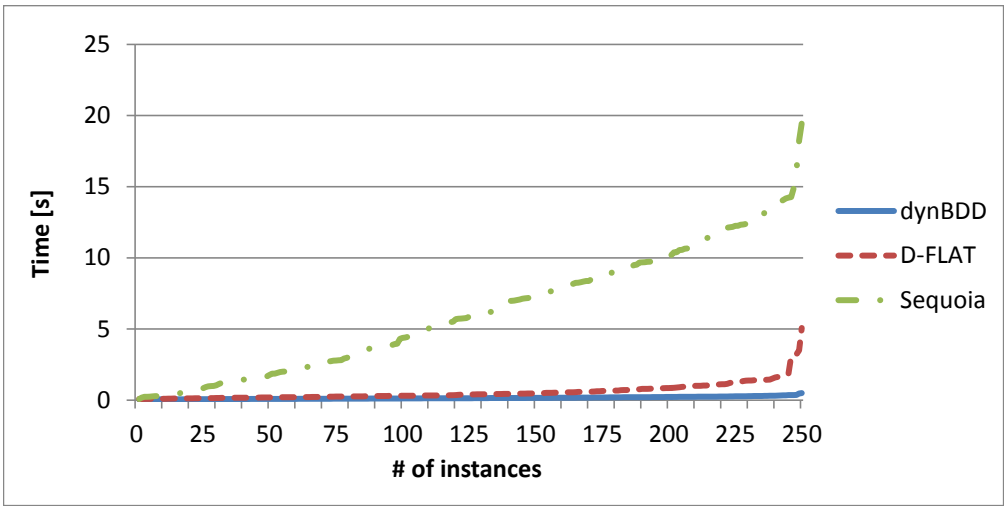


Figure 5.1: 3-Colorability - Time (grid-based instances, treewidth 5)

The first result of the benchmark test is depicted in Figure 5.1. It shows the required runtime for grid-based instances having a maximal treewidth of 5. Altogether 250 instances were tested. As we see, *dynBDD* is the fastest system. *D-FLAT* is a little bit slower. In contrast to *dynBDD* and *D-FLAT*, *Sequoia* takes the most time and is about 10-20 times slower. There were not any timeouts (meaning above the 3 minutes) or any out-of-memory (above the 4 GB limit), which we call shortly "memout" from now.

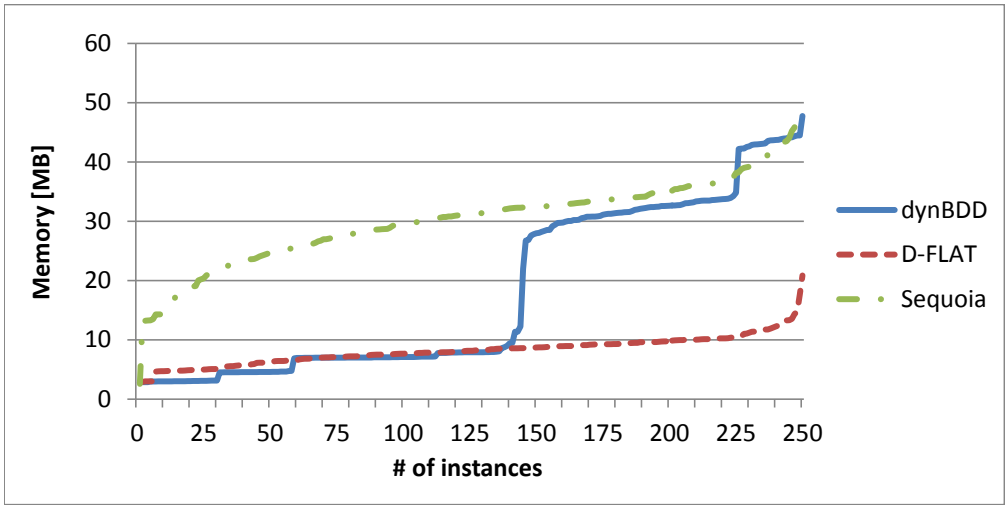


Figure 5.2: 3-Colorability - Memory (grid-based instances, treewidth 5)

Figure 5.2 shows the memory usage of the three systems. Here we have some interesting facts: *dynBDD* and *D-FLAT* have nearly the same memory usage for the first half of the instances, but then *dynBDD* needs about 3 times more for the last half of the instances. *Sequoia* already needs much more memory at the beginning than the other two systems, but at the end, *Sequoia* and *dynBDD* are nearly the same because the memory usage of *Sequoia* does not increase as fast as *dynBDD*. There were no timeouts or memouts.

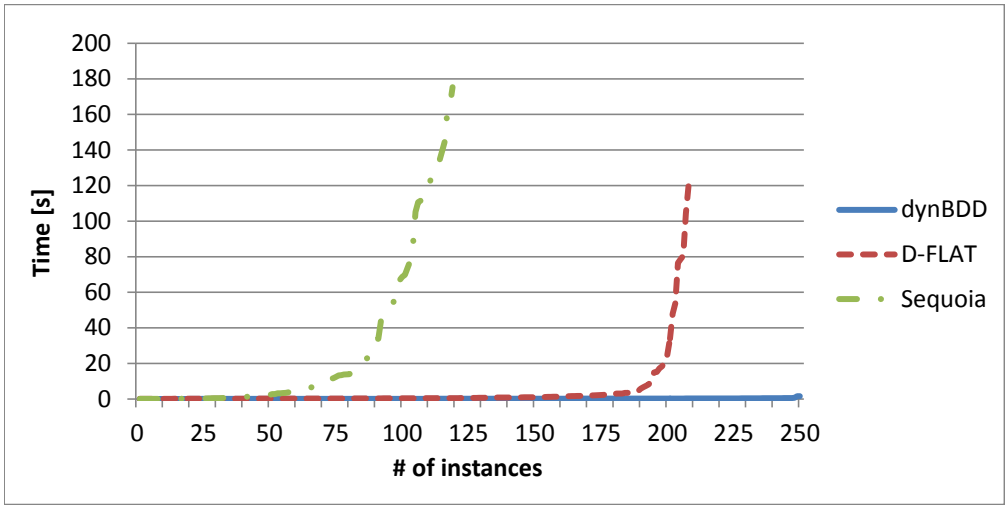


Figure 5.3: 3-Colorability - Time (grid-based instances, treewidth 20)

For the grid-based instances with a maxmimal treewidth of 20, the result is a little

bit different in contrast to the treewidth of 5, as can be seen in Figure 5.3 and Figure 5.4. Here, *dynBDD* is the leader again, not only in time but also in memory usage. Close behind is *D-FLAT*, which has only a deviation after 200 instances. *Sequoia* has already a very big deviation after 100 instances where the time and memory usage increase rapidly at this point. All instances of *dynBDD* were fine, but *D-FLAT* had 42 timeouts during this test, that is why the line is ending at instance 208. *Sequoia* had much more timeouts, in detail 131. Additonally, *Sequoia* had 1 memout.

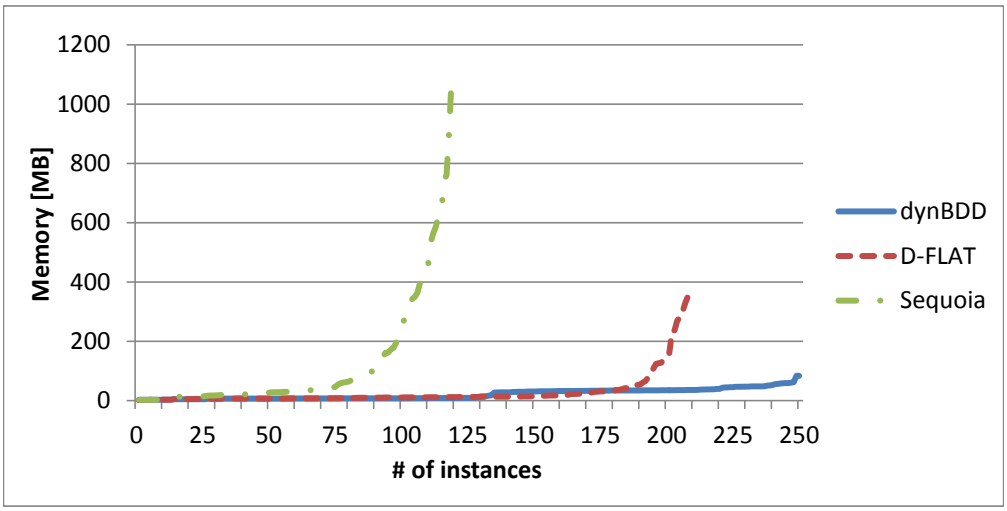


Figure 5.4: 3-Colorability - Memory (grid-based instances, treewidth 20)

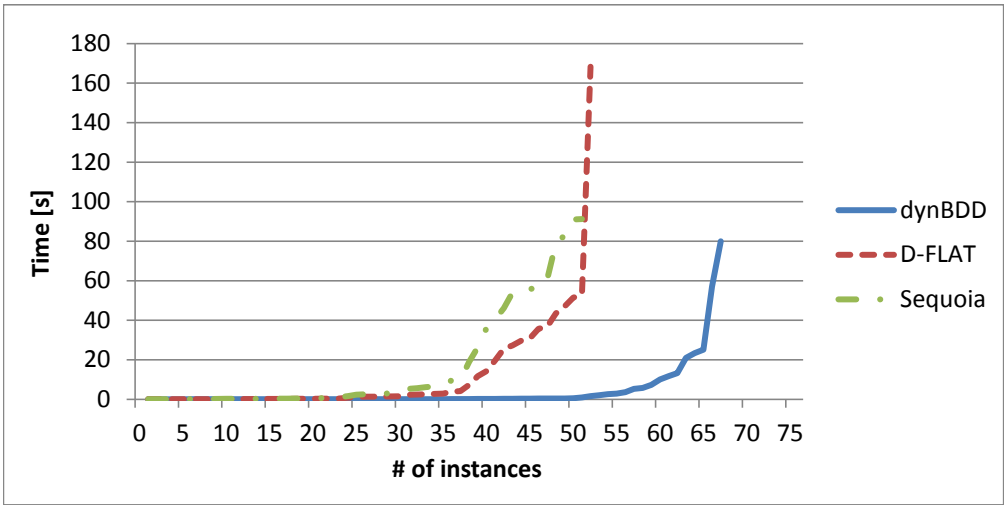


Figure 5.5: 3-Colorability - Time (random instances)

Now we tested random instances and the results are similar to the grid-based instances.

In Figure 5.5 we see the runtime of 75 random instances, beginning from 5 to 79 vertices. For the first 30 instances all three systems perform nearly identical but then *Sequoia* is again the first system that rapidly increases in runtime. It also had 25 timeouts. *D-FLAT* is the next one who drifts away, but only had 21 timeouts. The leader is *dynBDD* again having only 9 timeouts. Memouts did not occur during this run.

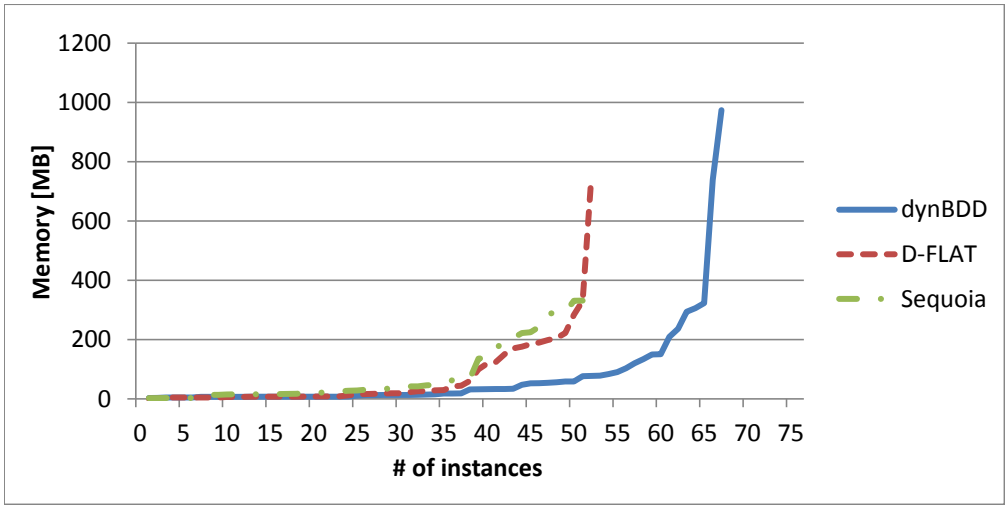


Figure 5.6: 3-Colorability - Memory (random instances)

The deviation of the memory usage is very similar to the runtime and so are the rankings, as can be seen in Figure 5.6.

Next, we analyze realworld instances, which are more interesting. The *dynBDD* system is still the leader in all instances when comparing the runtime (see Figure 5.7). The second and third positioned systems vary from instance to instance. For the metros in Beijing, Shanghai, Singapore and Vienna, *D-FLAT* is better than Sequoia, but for the other metros *Sequoia* is the faster one. If we look at the two additional real world instances, we can see that there is an enormous difference for the *Pajek Glossary*, where *D-FLAT* is better, but in contrast to that, for the *Pajekt Sandi* instance, *Sequoia* is much faster.

The memory usage of the real world instances are only interesting for the *Pajek Sandi* instance, because here *dynBDD* needs the most space, as we can see in Figure 5.8. But all the other instances do not have any special results, since *dynBDD* requires the least memory, followed by *D-FLAT* and then *Sequoia* with the most memory usage.

Analysis in DecoVis

Now we analyse some instances in more detail. We choose 2 instances from the random graphs, where the first one has 51 vertices and the second one has 52 vertices. Our benchmark dataset of *D-FLAT* reveals that the instance with 51 vertices takes about 7 times more runtime and about 5 times more memory usage than the instance with

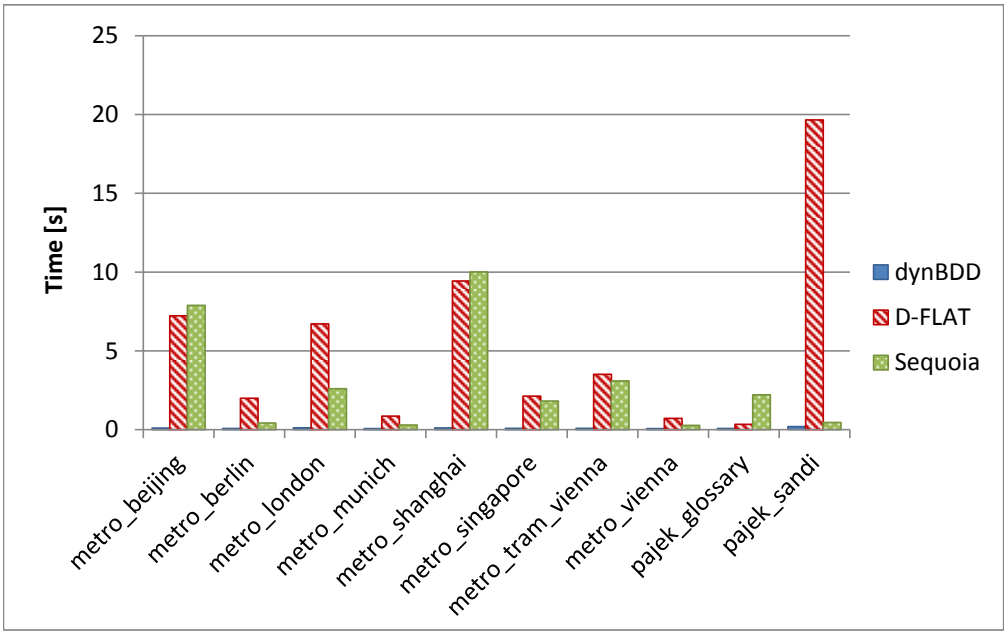


Figure 5.7: 3-Colorability - Time (realworld instances, minimum of 10 TDs)

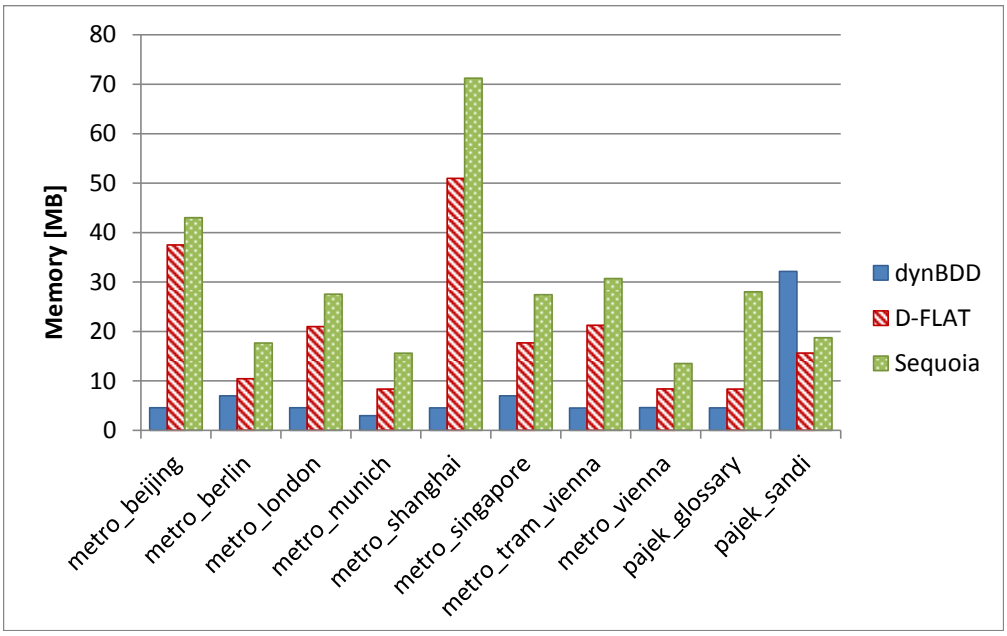


Figure 5.8: 3-Colorability - Memory (realworld instances, minimum of 10 TDs)

52 vertices. We load the instances in *DecoVis* to get more details about what happens exactly during the computation. Figure 5.9 shows the two instances, left the one with 51 vertices and right the one with 52 vertices. As we can see they look quite similar.

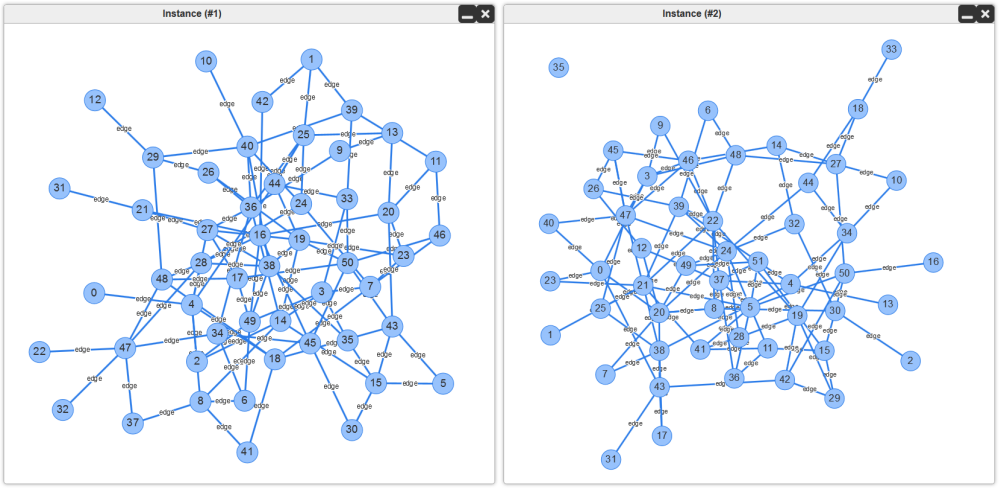


Figure 5.9: Random graphs: 51 vertices (left) vs. 52 vertices (right)

When we load the tree decompositions in *DecoVis* including the runtime metadata (see Figure 5.10), we easily see that the tree decompositions are also similar and both instances took a consistent runtime. In particular, there are no conspicuous values from one node to another. It is important to say that the size of the nodes regarding their value is only consistent per instance. Both have the same size at the top but they are describing different values. Hence, this comparison gives us only details about the distribution of the runtime. In fact, we see that both instances have a quite similar distribution of the runtime during the computation, but the real absolute values in seconds can be completely different.

When we load the item tree sizes into the tree decompositions, we see that the distribution of the memory usage is a little bit different (see Figure 5.11).

To get an overview about the real runtime and memory values during the computation, we use the statistic diagram feature of *DecoVis*. In Figure 5.12 we see the runtime differences, where the blue one is the instance with 51 vertices and the orange one illustrates the instance with 52 vertices. The unit of the y-axis are seconds, the x-axis describes a particular node.

Figure 5.13 shows information about the item tree size. Both diagrams reveal that the higher runtime and memory usage is quite distributed over the whole computation process, but in the middle there is the most difference in runtime as well as in memory.

The last diagram (Figure 5.14) of this analysis shows the memory usage regarding runtime instead of the node index.

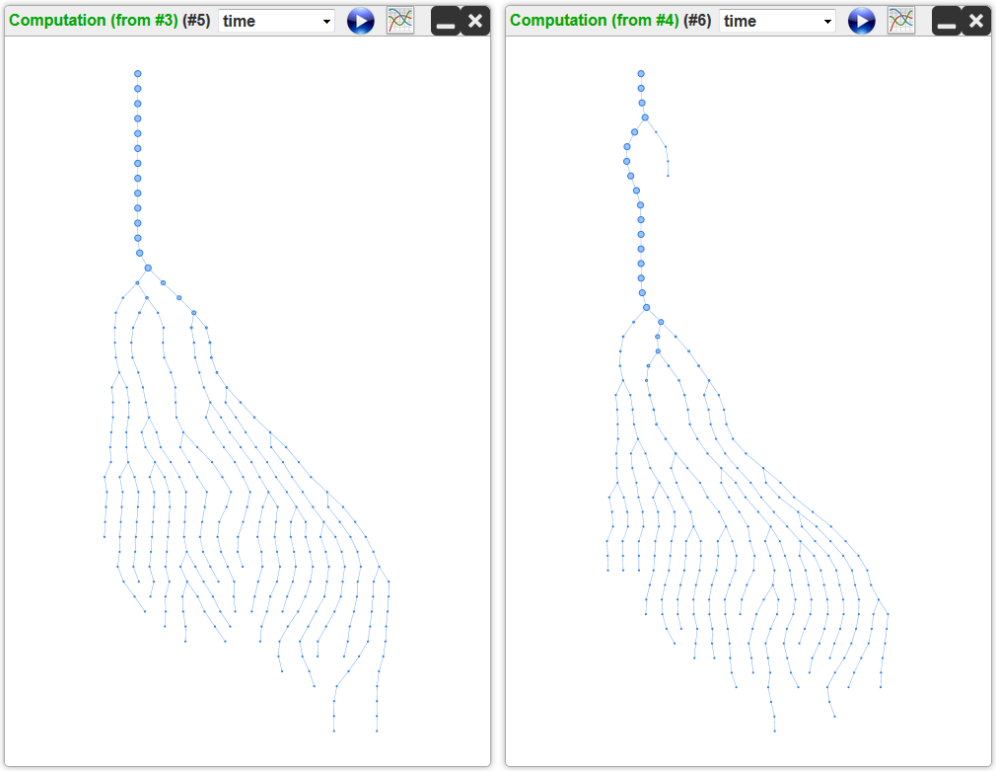


Figure 5.10: 3-Colorability, *D-FLAT*, 2 random graphs: Time

Summary

As we see, *dynBDD* is the most efficient tool regarding runtime and memory usage when comparing 3-Colorability problems. *D-FLAT* takes a little bit longer and usually also needs more memory. The most time and memory is required by the *Sequoia* tool. Here, many instances could not be solved within the predefined time of 3 minutes and the 4GB memory. Figure 5.15 shows once again the timeouts and memouts of the three different tools in each instance group. Furthermore, the numbers of solved instances (satisfiable or unsatisfiable) are illustrated.

5.4.2 Stable Extension - DynBDD, D-FLAT

The *Stable Extension* problem belongs to the area of argumentation [27, 66]. In broad terms, there are arguments and there can be conflicts between them. If there is a conflict between some arguments, there is a binary relation between them. Illustrated as a directed graph, each vertex is an argument and each edge shows the conflict or the so-called attack relation. Furthermore, we can pick a so-called extension which is a subset of all the available arguments. There exist several semantics on how an extensions is selected, like the *Stable Extension* or the *Preferred Extension*. The latter is described

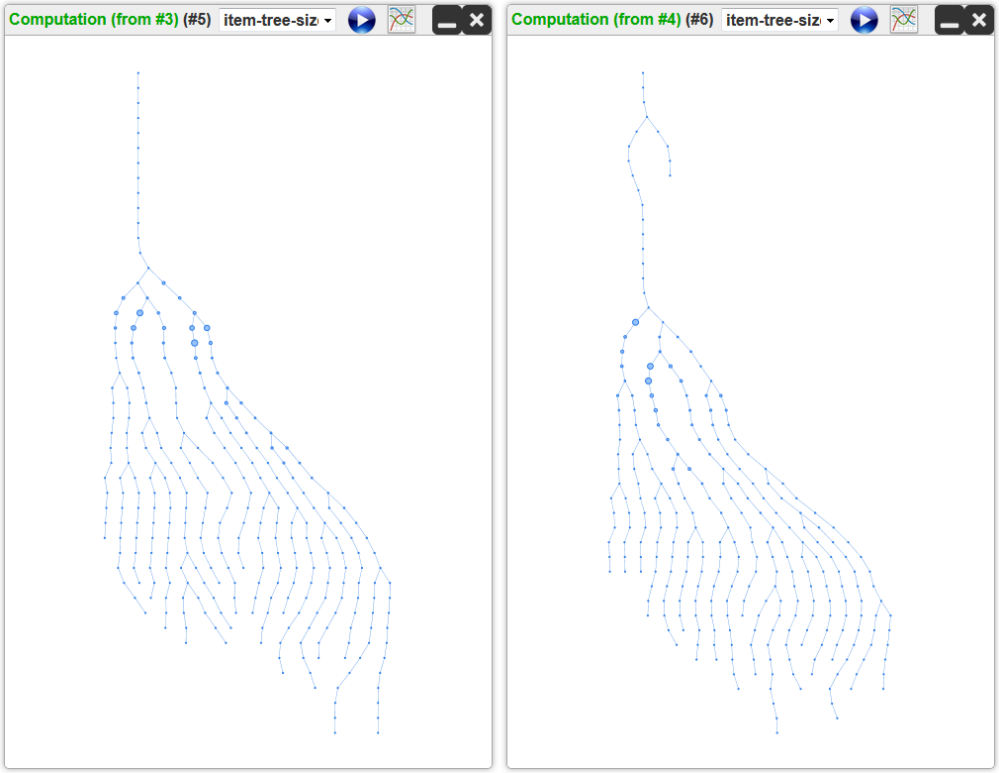


Figure 5.11: 3-Colorability, *D-FLAT*, 2 random graphs: Item tree size

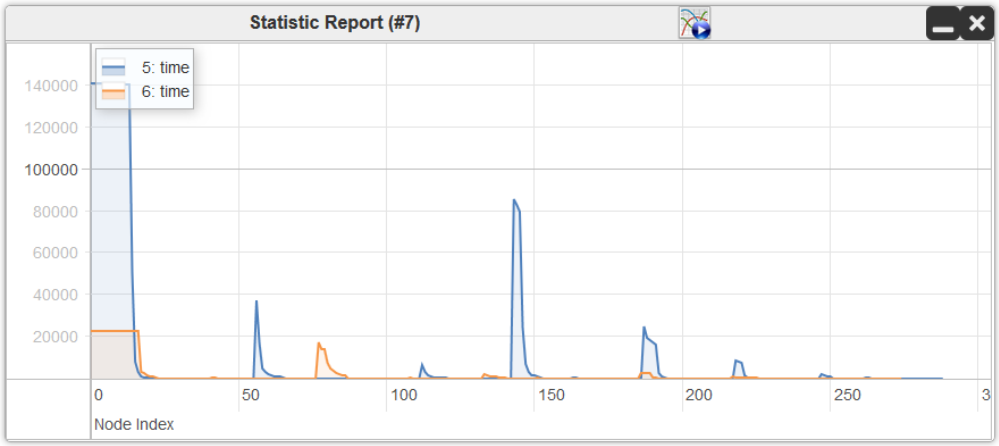


Figure 5.12: 3-Colorability, *D-FLAT*, 2 random graphs: Time

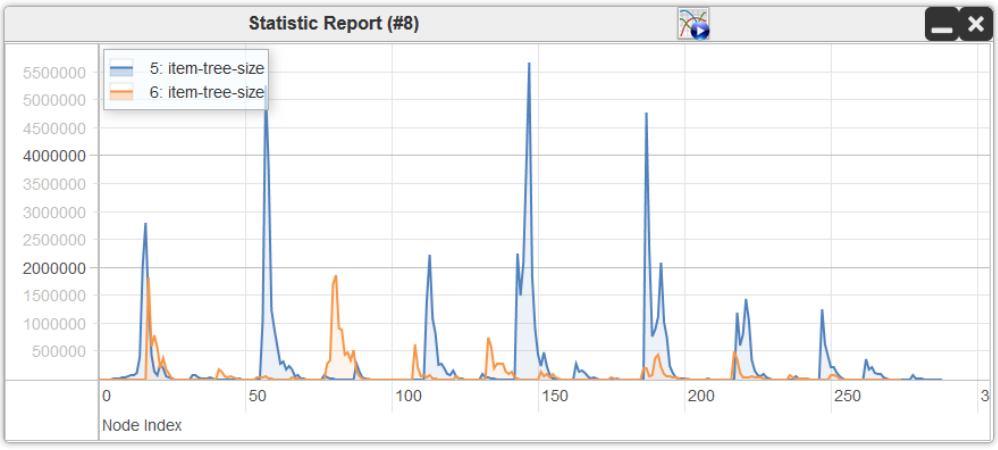


Figure 5.13: 3-Colorability, *D-FLAT*, 2 random graphs: Item tree size

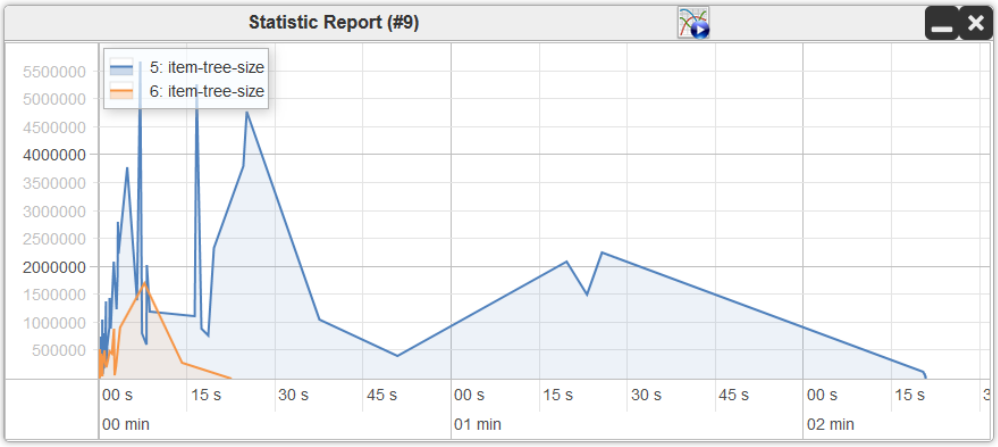


Figure 5.14: 3-Colorability, *D-FLAT*, 2 random graphs: Item tree size

later on. A *Stable Extension* is a set of arguments which is, first, conflict-free (meaning that there is no conflict and therefore no edge between them), and second, this set attacks all other arguments which do not belong to this extension set.

We take the same instances as before to run the Benchmark tests for the *Stable Extension* problem. But in contrast to the *3-Colorability* problem we can only use *dynBDD* and *D-FLAT* because the *Stable Extension* encoding is currently not available for Sequoia. As mentioned before, the instances have to be prepared a little bit to be compatible with argumentation problems because there are not longer *vertex* and *edge* predicates but *arg* and *att* predicates instead (in fact, they are simply renamed to be usable for the argumentation problems).

		Satisfiable	Unsatisfiable	Timeout	Memout
Grid-based Treewidth 5	dynBDD	1	249	0	0
	D-FLAT	1	249	0	0
	Sequoia	1	249	0	0
Grid-based Treewidth 20	dynBDD	13	237	0	0
	D-FLAT	13	195	42	0
	Sequoia	13	106	130	1
Random	dynBDD	53	14	8	0
	D-FLAT	39	13	23	0
	Sequoia	38	13	24	0
Realworld	dynBDD	9	1	0	0
	D-FLAT	9	1	0	0
	Sequoia	9	1	0	0

Figure 5.15: 3-Colorability - Summary of timeouts, memouts and solved instances

Results & Comparison

Figure 5.16 shows the comparison of the runtime of grid-based instances with a maximal treewidth of 5. As we see, *dynBDD* solves the problem much faster than *D-FLAT*. The difference is so big, that the growth of the runtime in *dynBDD* is not visible in the diagram at all.

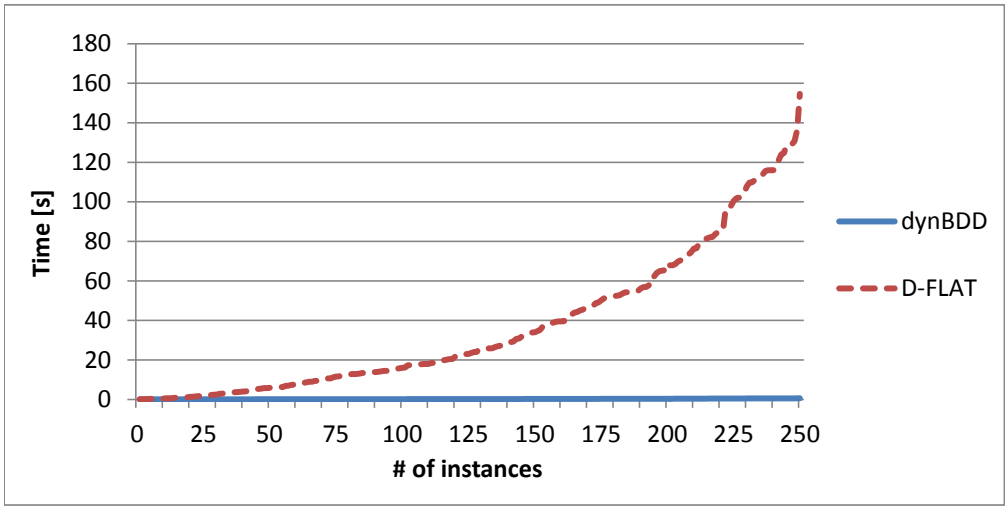


Figure 5.16: Stable Extension - Time (grid-based instances, treewidth 5)

For the memory usage it looks completely different, as can be seen in Figure 5.17. At the beginning *dynBDD* needs less memory, but then there is a quite high jump in memory usage so that *D-FLAT* is better for about 100 instances. At the end *dynBDD* is the leader again because the memory growth is not as high as for *D-FLAT*. The trend of the diagram shows clearly, that *D-FLAT* will take more memory. There were no timeouts or memouts. The anomaly of *dynBDD* will be analyzed later on by using *DecoVis*.

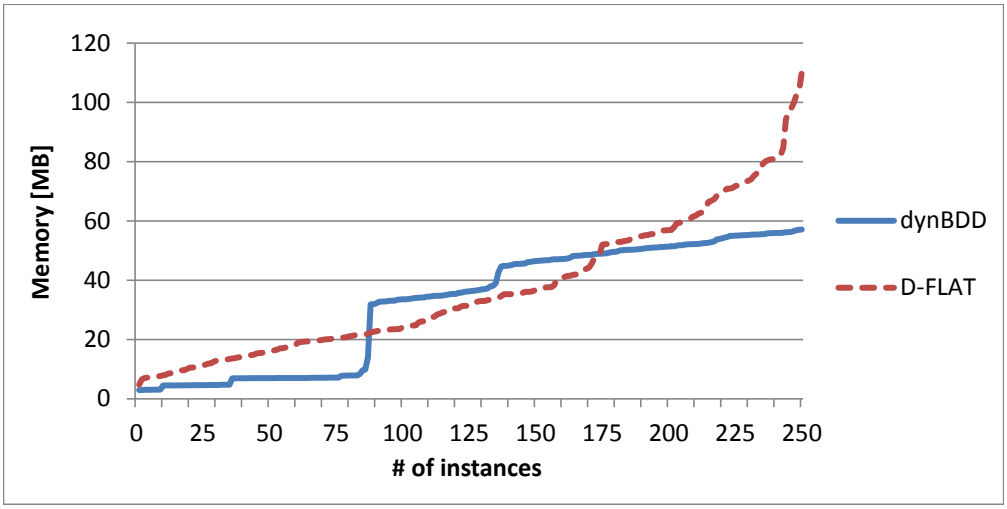


Figure 5.17: Stable Extension - Memory (grid-based instances, treewidth 5)

When we compare the instances with a maximum treewidth of 20 (see Figure 5.18), then we get a very clear result. The runtime of *D-FLAT* begins to grow rapidly after 75 instances, whereas the runtime of *dynBDD* only has a visible growth at the end. *D-FLAT* had 167 timeouts.

The same applies for the memory usage (see Figure 5.19). Once again we see some jumps in memory usage for *dynBDD*, but nevertheless it is the whole time the leader.

In Figure 5.20 and Figure 5.21 we see the differences of runtime and memory usage of random instances. Once again, *dynBDD* is much faster and takes much less memory during the computations. There were 16 timeouts of *D-FLAT*.

Even for all real-world examples, *dynBDD* is the leader, as can be seen in Figure 5.22 and Figure 5.23. In this case, there were no timeouts or memouts.

Analysis in DecoVis

As we have seen in the last section, there are some quite unusual memory jumps occurring in the *dynBDD* computation runs. Now we want to analyze these jumps in more detail with the help of *DecoVis*.

First, we load the two instances of the grid-based graphs having a maximum treewidth of 5, as can be seen in Figure 5.24: one with 294 vertices which is shown on the left and one with 326 vertices which is shown on the right side. Both instances look very slim

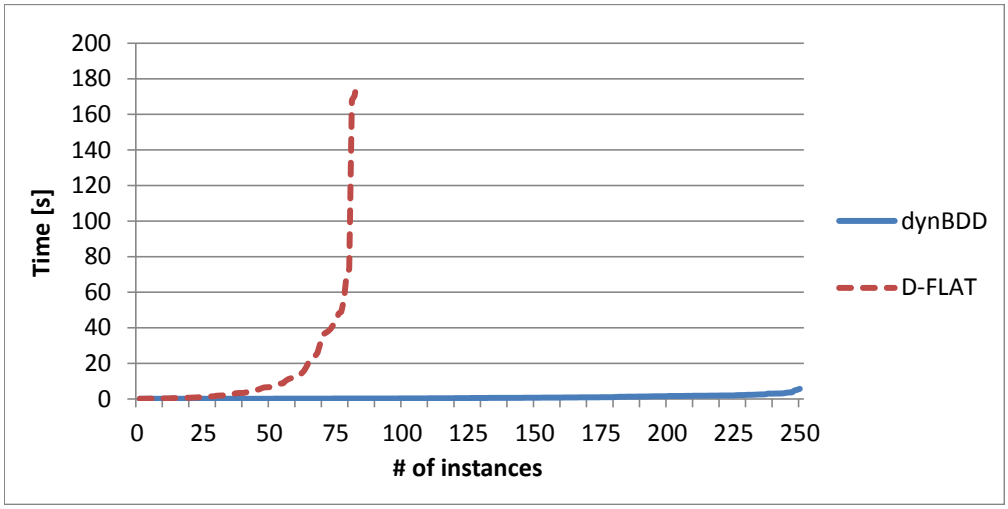


Figure 5.18: Stable Extension - Time (grid-based instances, treewidth 20)

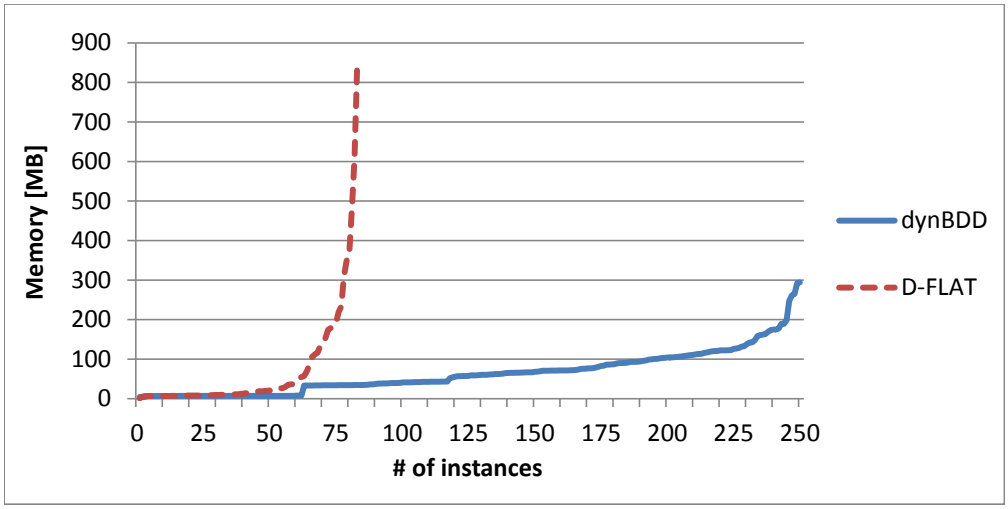


Figure 5.19: Stable Extension - Memory (grid-based instances, treewidth 20)

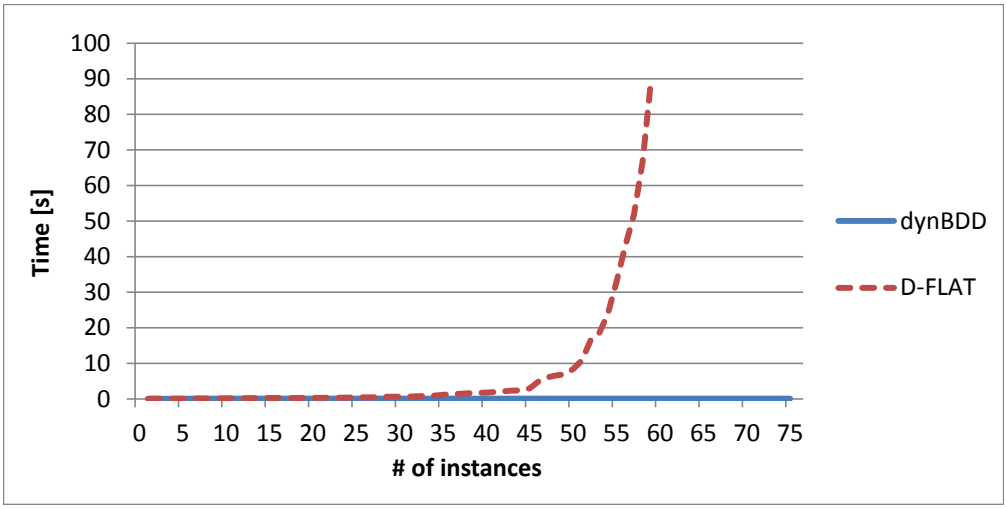


Figure 5.20: Stable Extension - Time (random instances)

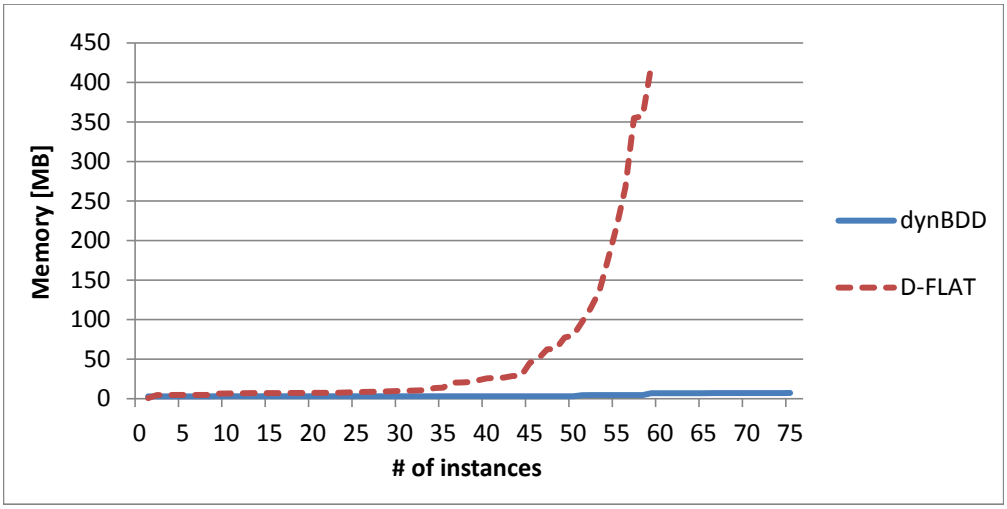


Figure 5.21: Stable Extension - Memory (random instances)

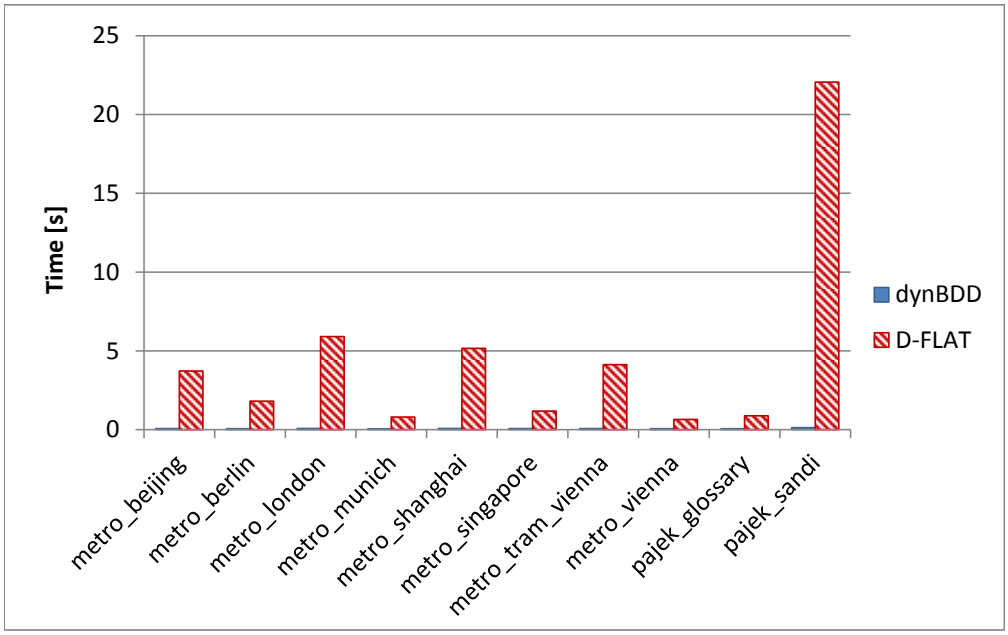


Figure 5.22: Stable Extension - Time (realworld instances, minimum of 10 TDs)

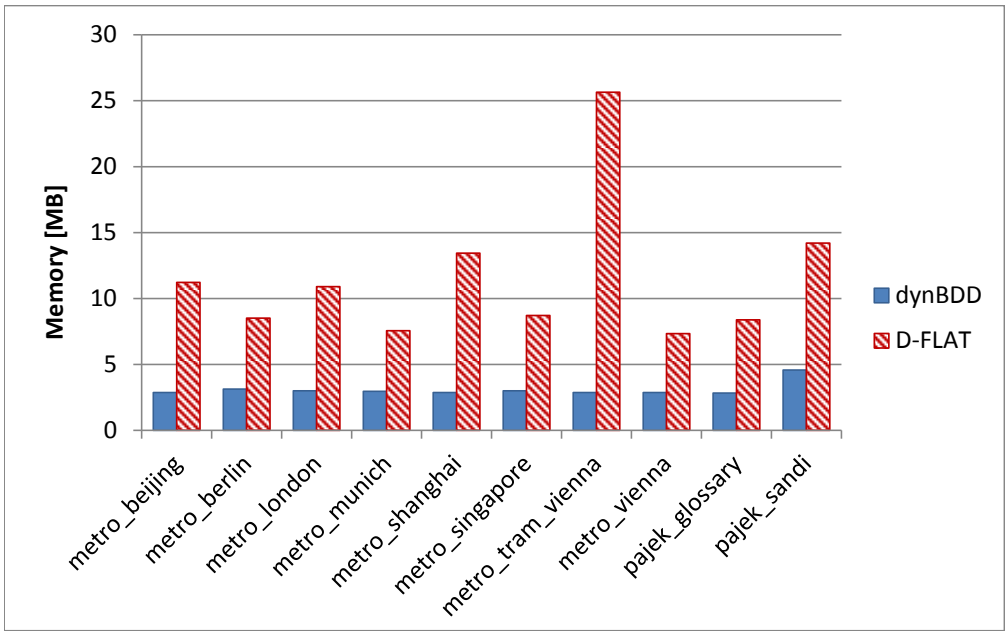


Figure 5.23: Stable Extension - Memory (realworld instances, minimum of 10 TDs)

and long. They are quite similar in total, since they are having a maximal treewidth of only 5. But the left and smaller one (regarding the number of vertices) takes about 4 times more memory usage. The runtime is quite the same.

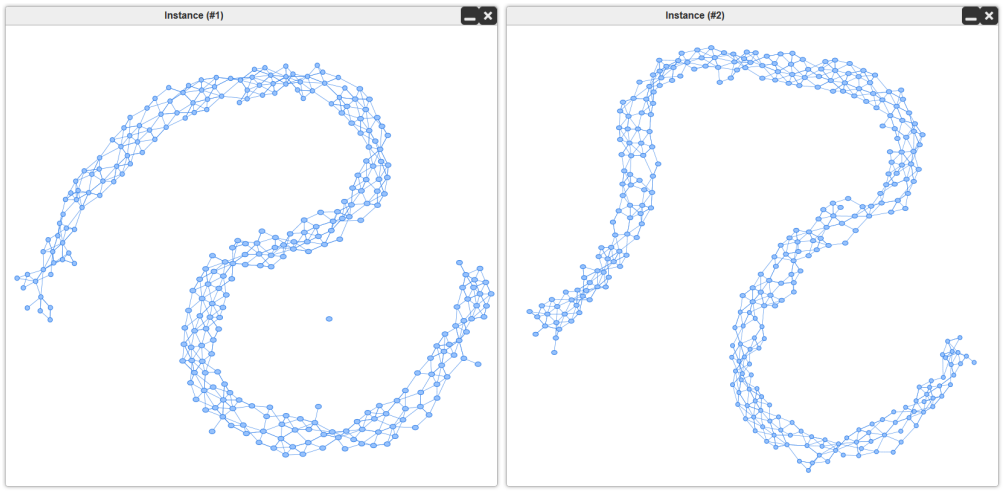


Figure 5.24: Grid-based graphs: 294 vertices (left) vs. 326 vertices (right)

When we now load the tree decompositions including the runtime metadata in *DecoVis* (see Figure 5.25), we can see that the smaller instance (left) needs the most runtime in the right part of the tree decomposition, whereas the bigger instance (right) needs the most runtime in the middle part.

Now we look at the memory usage by filling the tree decompositions with the memory values, illustrated in Figure 5.26. We see, that the distribution of the memory is similar to the runtime. Until now, we cannot see any anomalies.

To get a better analysis of the values in detail, we look at the statistic diagrams of the runtime, shown in Figure 5.27. As expected, they are quite the same.

Now we look at the diagrams of the memory usage, shown in Figure 5.28 and Figure 5.29 (first with the node index as x-axis, then with runtime as x-axis). The metadata produced during the computation does not reflect the real memory usage consumed by the operating system, but only of the BDDs. But here, we cannot see any memory jumps. We get a very surprising information: the metadata of the *dynBDD* computation does not reveal a significant higher memory usage during the whole computation. In fact, the instance with 294 nodes is consuming less memory for the BDDs than the instance with 326 nodes. This is completely contradicting our benchmark results since we expected that there is somewhere a much higher memory usage either at some particular node in the tree decomposition or at least continually over the the whole computation. What is the reason for this behavior? In fact, we cannot find anything wrong, so we replicate the benchmark test. We get the following unexpected result: For the same instance, the final memory usage is jumping from a low value to a four time higher value completely randomly, but never anything between.

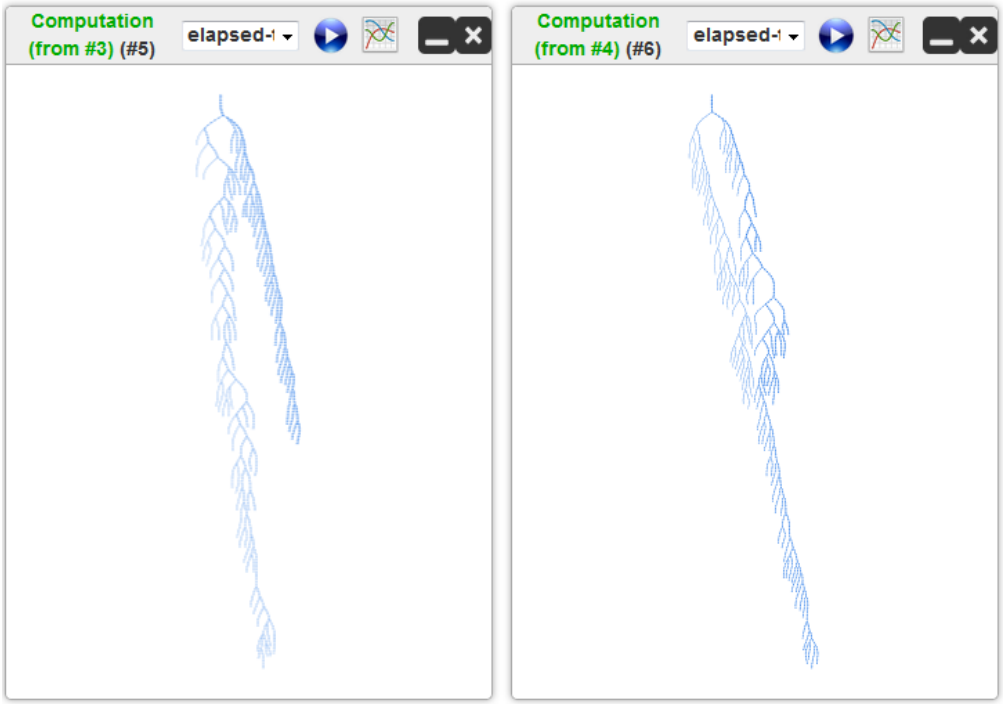


Figure 5.25: Stable Extension, *dynBDD*, 2 grid-based graphs: Time

It seems that sometimes the memory is allocated in quite high steps. It must have something to do on how *dynBDD* allocates the memory. The internal memory usage for the BDDs produced by *dynBDD* as metadata is obviously correct and never shows such jumps. But the total memory usage of the whole application has sometimes much more allocated memory and that is why we get these unusual jumps in the benchmark tests. Unfortunately, we do not know the exact reason for this behavior because we are seeing the *dynBDD* tool as black-box-system and we have no other metadata available which could give us insight in what is going on in detail. But we strongly assume that *dynBDD* and/or its used libraries allocate more memory than they would actually need since this happens completely randomly.

Furthermore, we see that *dynBDD* starts with an already high memory consumption for the first node and then the memory is only slowly decreasing. Probably that is because many variables are instantiated even before the first node is processed.

Summary

In fact, all results were very clear: *dynBDD* is always the leader in runtime as well as in memory usage. There is a small exception for *dynBDD* because there is an unusual memory jump for the grid-based instances of treewidth 5, but overall it does not influence the further trend of these instances. At the end of the benchmarks we also show the

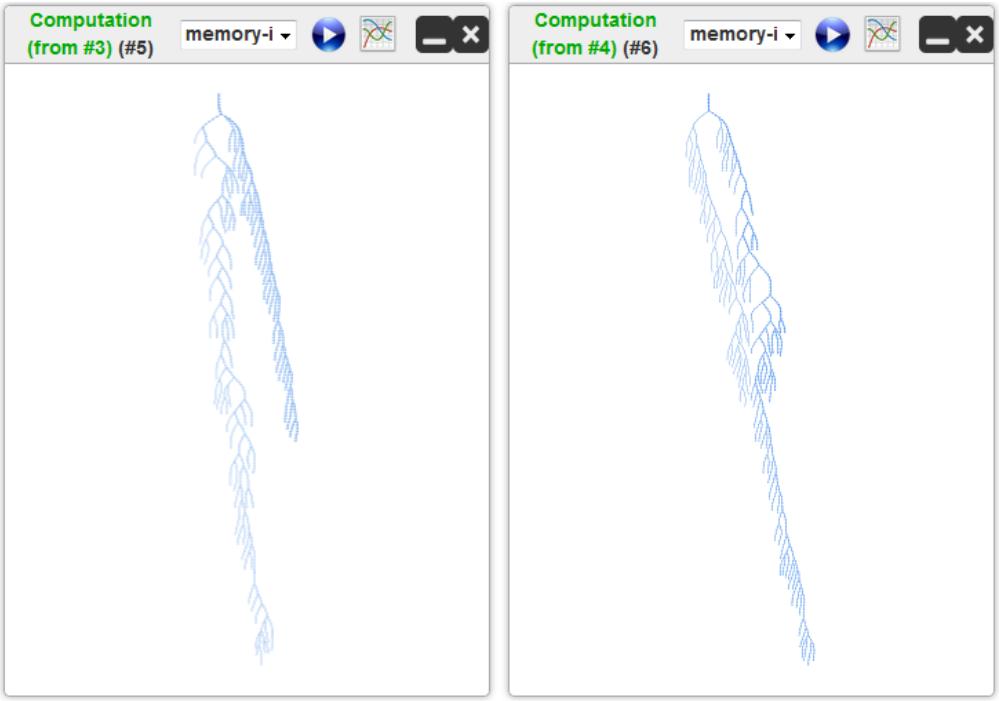


Figure 5.26: Stable Extension, *dynBDD*, 2 grid-based graphs: Memory

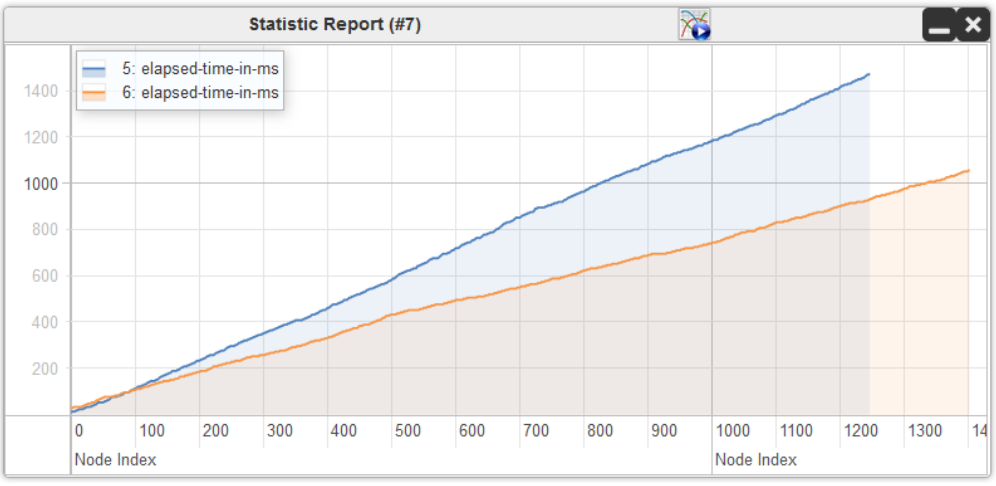


Figure 5.27: Stable Extension, *dynBDD*, 2 grid-based graphs: Time

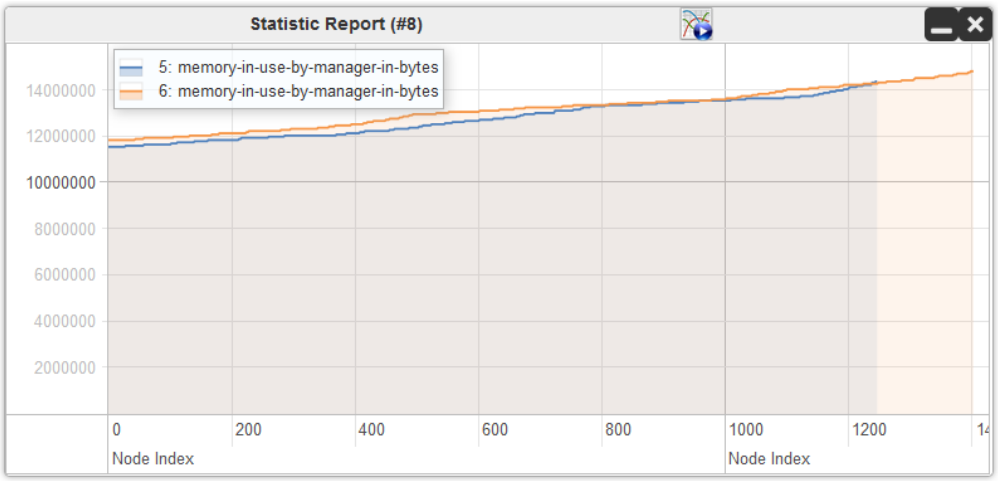


Figure 5.28: Stable Extension, *dynBDD*, 2 grid-based graphs: Memory

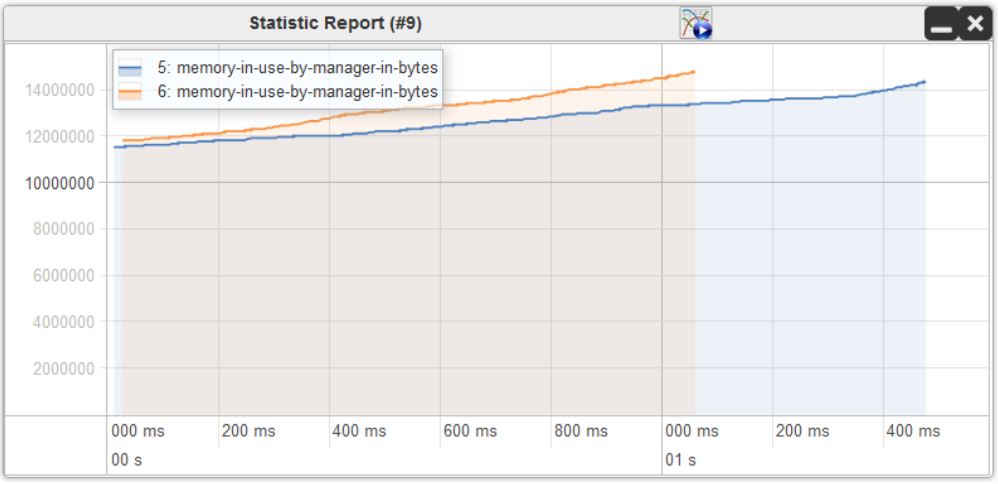


Figure 5.29: Stable Extension, *dynBDD*, 2 grid-based graphs: Memory

overall result of satisfiable and unsatisfiable instances including the timeouts and memouts (see Figure 5.30).

		Satisfiable	Unsatisfiable	Timeout	Memout
Grid-based Treewidth 5	dynBDD	147	103	0	0
	D-FLAT	147	103	0	0
Grid-based Treewidth 20	dynBDD	203	47	0	0
	D-FLAT	72	11	167	0
Random	dynBDD	63	12	0	0
	D-FLAT	51	8	16	0
Realworld	dynBDD	9	1	0	0
	D-FLAT	9	1	0	0

Figure 5.30: Stable Extension - Summary of timeouts, memouts and solved instances

5.4.3 Hamiltonian Cycle - DynBDD, D-FLAT, Sequoia

Hamiltonian Cycle is a path in a graph which visits all vertices - each vertex exactly once [67, 68]. Since it is a cycle, the vertex of the end of the path has to be the vertex of the beginning of the path. This problem is tested with *dynBDD*, *D-FLAT* and *Sequoia* with the same instances as before.

Results & Comparison

We begin with the grid-based instances having a maximum treewidth of 5. In Figure 5.31 we recognize several important facts regarding runtime: First, *dynBDD* is able to solve the most instances in contrast to the other two tools. But for the first 100 instances, *D-FLAT* performs better and takes less runtime. For the next instances, the runtime of *D-FLAT* grows very fast and reaches the timeout limit, whereas *dynBDD* is able to solve 34 more instances. *Sequoia* is only able to solve 4 instances in total. Second, we generally see the overall difficulty of the problem in contrast to the first two problems (*3-Colorability* and *Stable Extension*) where always all instances could be solved by at least one tool. In this case *dynBDD* had 102 timeouts and 1 memout. *D-FLAT* had 128 timeouts. The last one is *Sequoia* with 246 timeouts.

The memory usage shown in Figure 5.32 is similar to the runtime but now *D-FLAT* performs always the best, followed by D-FLAT and *Sequoia* as the last one.

When comparing the grid-based instances with a maximum treewidth of 20, we do not get any changing information regarding performance (see Figure 5.33). The leader is *D-FLAT*, followed by *dynBDD*. We see that both of them can solve quite the same number of instances but *D-FLAT* is much faster. The last one is *Sequoia* again which is

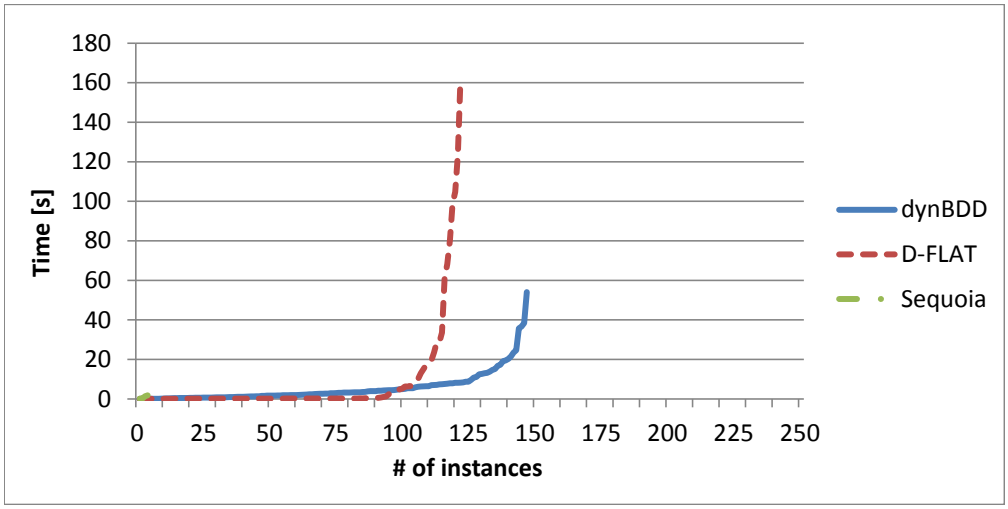


Figure 5.31: Hamiltonian Cycle - Time (grid-based instances, treewidth 5)

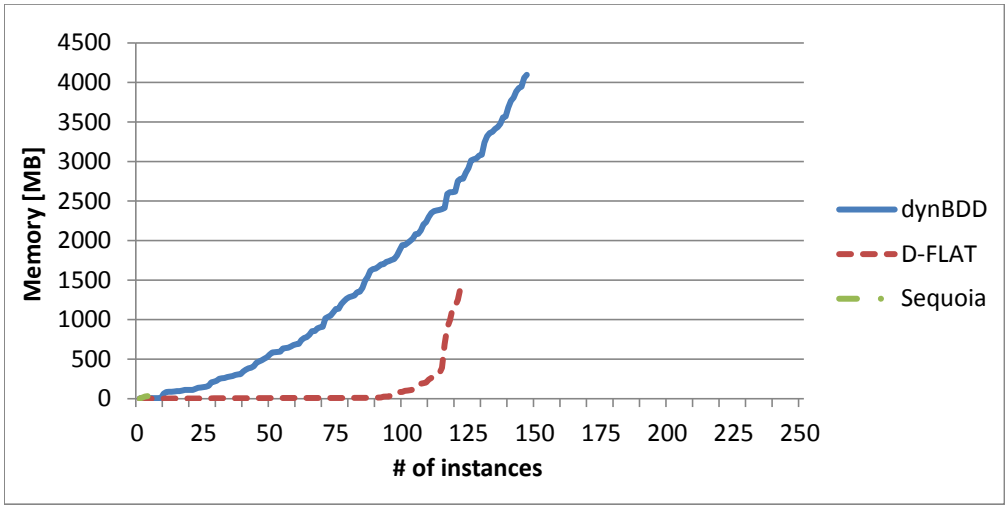


Figure 5.32: Hamiltonian Cycle - Memory (grid-based instances, treewidth 5)

only able to solve 5 instances. In detail, *D-FLAT* had 164 timeouts, *dynBDD* had 59 timeouts with additional 120 memouts and *Sequoia* causes 245 memouts.

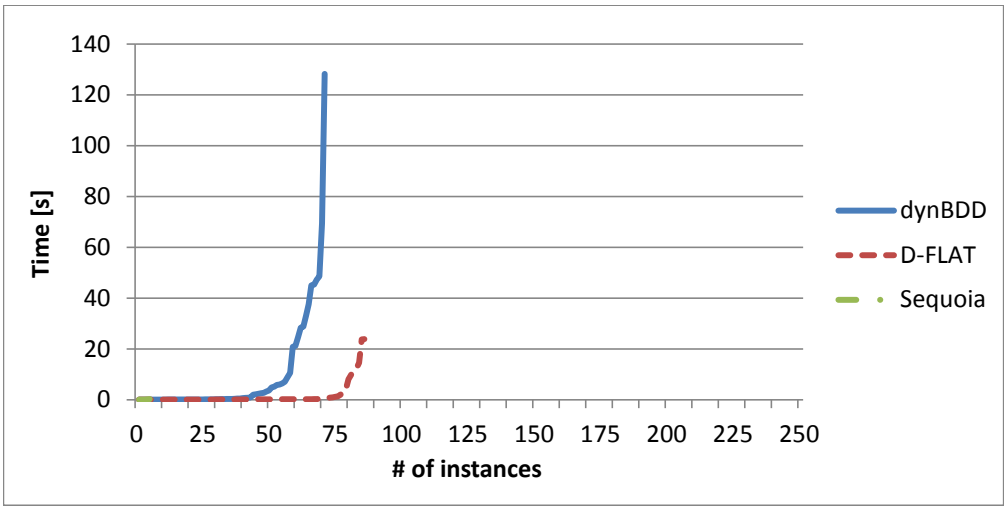


Figure 5.33: Hamiltonian Cycle - Time (grid-based instances, treewidth 20)

The memory usage which is shown in Figure 5.34 is exactly the same regarding the performance rankings.

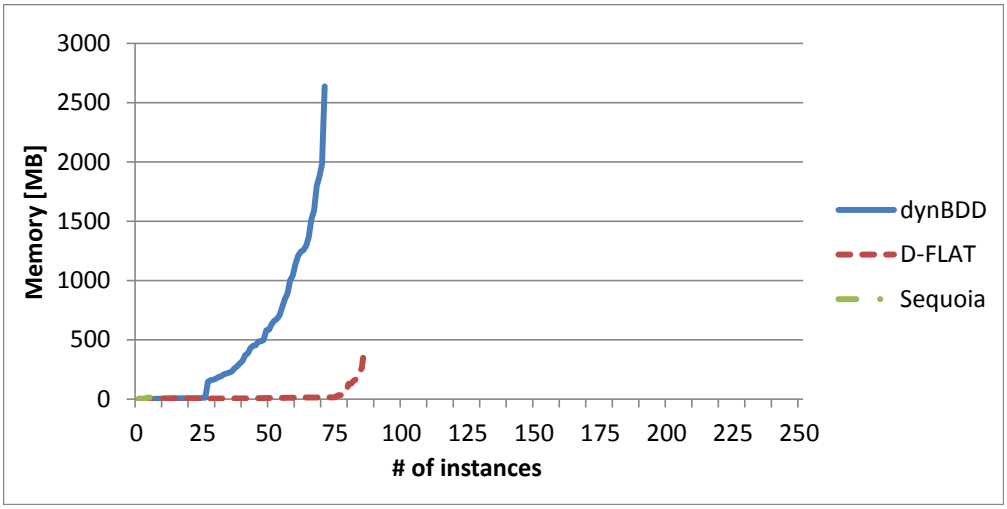


Figure 5.34: Hamiltonian Cycle - Memory (grid-based instances, treewidth 20)

Now we test the random instances, illustrated in Figure 5.35 and Figure 5.36. As we see, the rankings are now a little bit clearer, but they do not reveal any new performance information since they are very similar regarding the grid-based instances. Here, *D-FLAT*

had only 5 timeouts, *dynBDD* had 10 timeouts and Sequoia failed in 59 instances with 3 timeouts and 56 memouts.

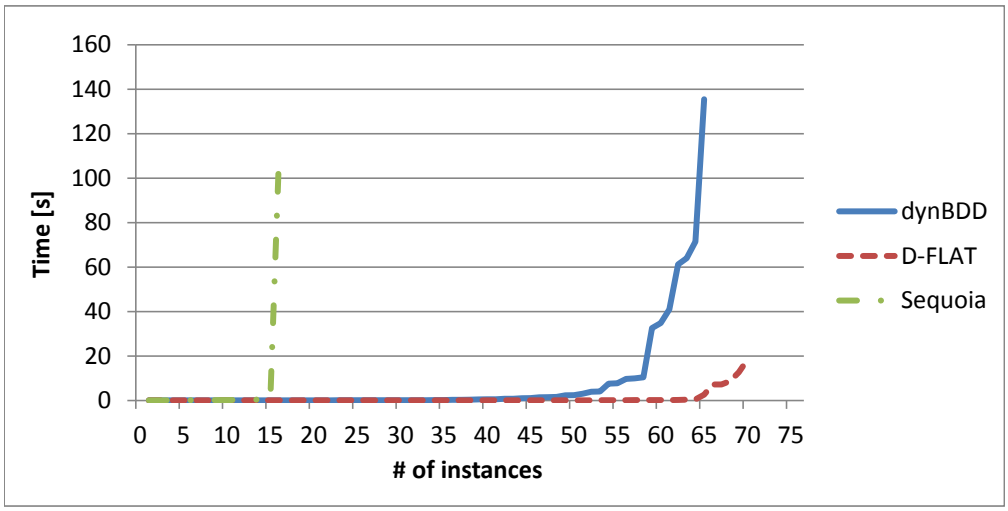


Figure 5.35: Hamiltonian Cycle - Time (random instances)

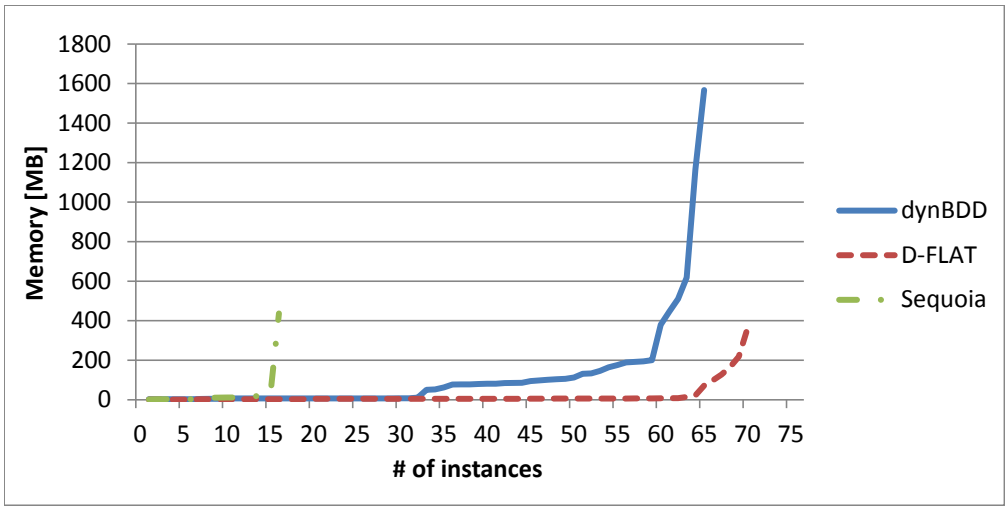


Figure 5.36: Hamiltonian Cycle - Memory (random instances)

Now we analyze the last instance group, the real-world instances. When comparing the runtime (see Figure 5.37) we get some new information regarding performance. For the first 6 instances and also the very last instance, it is very clear that *dynBDD* is much faster, followed by *D-FLAT*. *Sequoia* is not able to solve any of the 10 instances per real-world example, that is why it is marked with an *X*.

The remaining 3 instances are solvable by *Sequoia*, but we see that *metro_tram_vienna*, *metro_vienna* and *pajek_glossary* need very much time in contrast to *dynBDD* or *D-FLAT*.

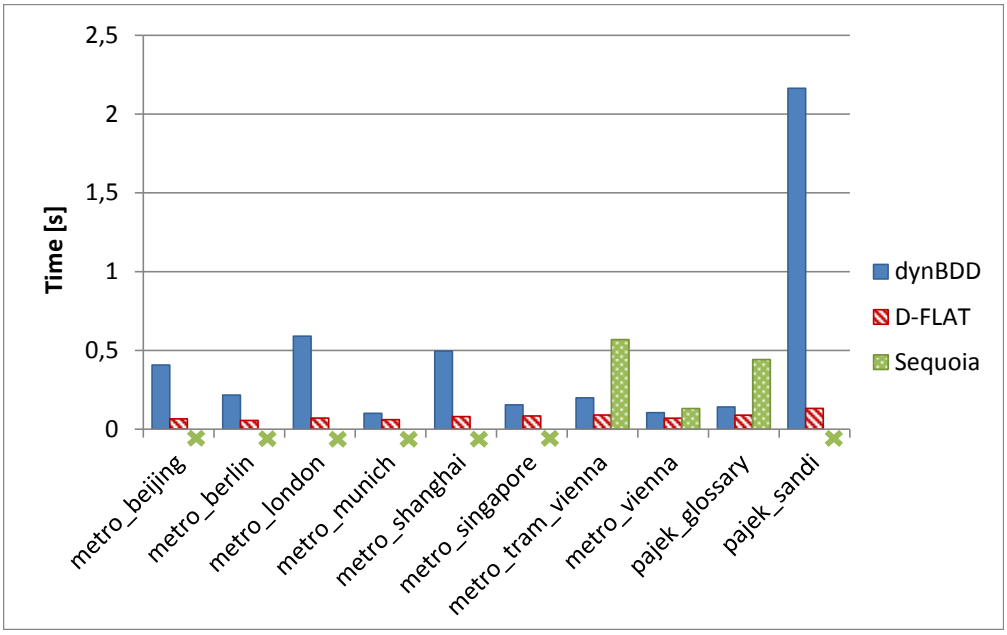


Figure 5.37: Hamiltonian Cycle - Time (realworld instances, minimum of 10 TDs)

Regarding memory usage, *D-FLAT* performs mostly the best and therefore has mainly the lowest memory usage, followed by *dynBDD* (shown in Figure 5.38). Quite interesting is that *Sequoia* can beat *dynBDD* in the two instances *metro_tram_vienna* and *metro_vienna*.

Analysis in DecoVis

For a detailed analysis of instances in *DecoVis*, this time we choose a real-world instance. Furthermore we want to compare and analyze the results between *dynBDD* and *D-FLAT* using one single instance. As instance we choose the *Metro London*.

First, we load the instance in *DecoVis* resulting in the graph shown in Figure 5.39. It is obviously that the graph reflects a metro system.

Now, we load the tree decomposition twice as can be seen in Figure 5.40, once loaded using the runtime metadata of *D-FLAT* (on the left, metadata is named *time*) and once loaded using the runtime metadata of *dynBDD* (on the right side, metadata is called *elapsed-time-in-ms*). We can see, that D-FLAT is spending the most time only at the very left side of the tree decomposition while *dynBDD* is already spending time from the very bottom of the tree decomposition and especially on the right side. Here, we also see how the systems are processing the tree decomposition during the computation. *D-FLAT*

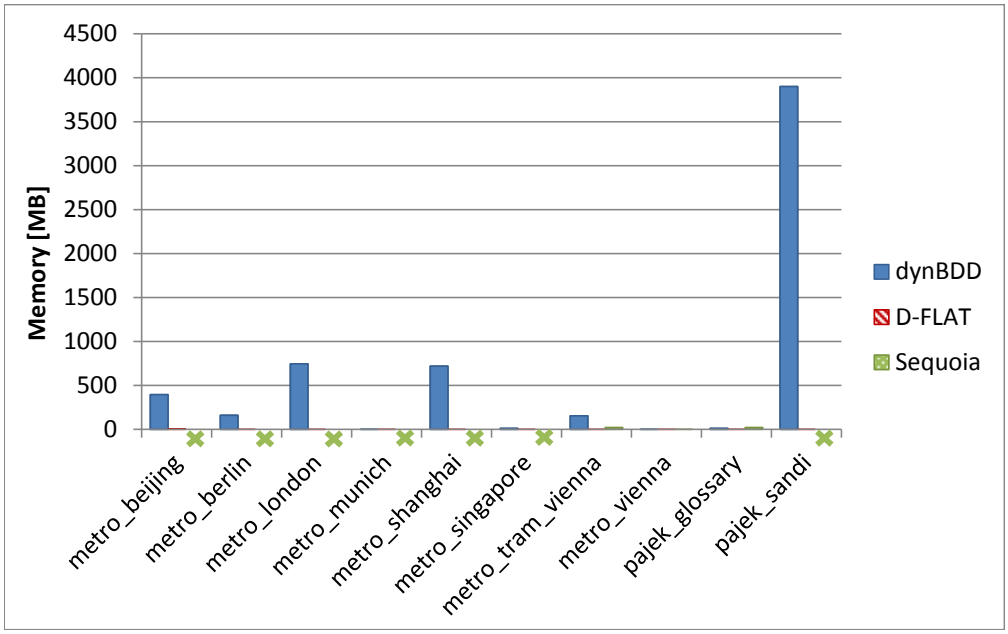


Figure 5.38: Hamiltonian Cycle - Memory (realworld instances, minimum of 10 TDs)

is beginning from the bottom of the very right side, whereas *dynBDD* is beginning from the opposite.

To get a detailed comparison of the memory usage, we change the metadata of the *D-FLAT* tree decomposition to *item-tree-size* and the *dynBDD* one to *memory-in-use-by-manager-in-bytes*, see Figure 5.41. It is very important to say that these two metadata types are not comparable to each other in the manner of absolute values even though they are both given in bytes. The reason is that *D-FLAT* seems to produce the exact item-tree-size for every node of the tree decomposition, but in contrast to that, *dynBDD* seems to do some caching and other things influencing the memory usage during the computation. This falsifies the memory usage when only a particular node of the tree decomposition (and therefore a single BDD) is considered.

But nevertheless, the two metadata types can be used for a comparison of relative values regarding the whole computation, meaning the minimum values of both metadata types are set to the same level and the maximum values are set to the same level. This is how the visualization of metadata values inside tree decompositions in *DecoVis* already works. Each node size visualizes a relative value and that is why we can compare the approximate memory usage of *D-FLAT* and *dynBDD* in these visualizations.

The biggest item tree size (and therefore approximately the most memory) of *D-FLAT* is used in the beginning of the very left path of the tree decomposition. In *dynBDD* the most memory is used at the beginning of all the paths on the right side of the tree decomposition which reaches their maximum on the top when all the paths get merged.

In Figure 5.42 we see the absolute values of the runtime regarding the node index. In

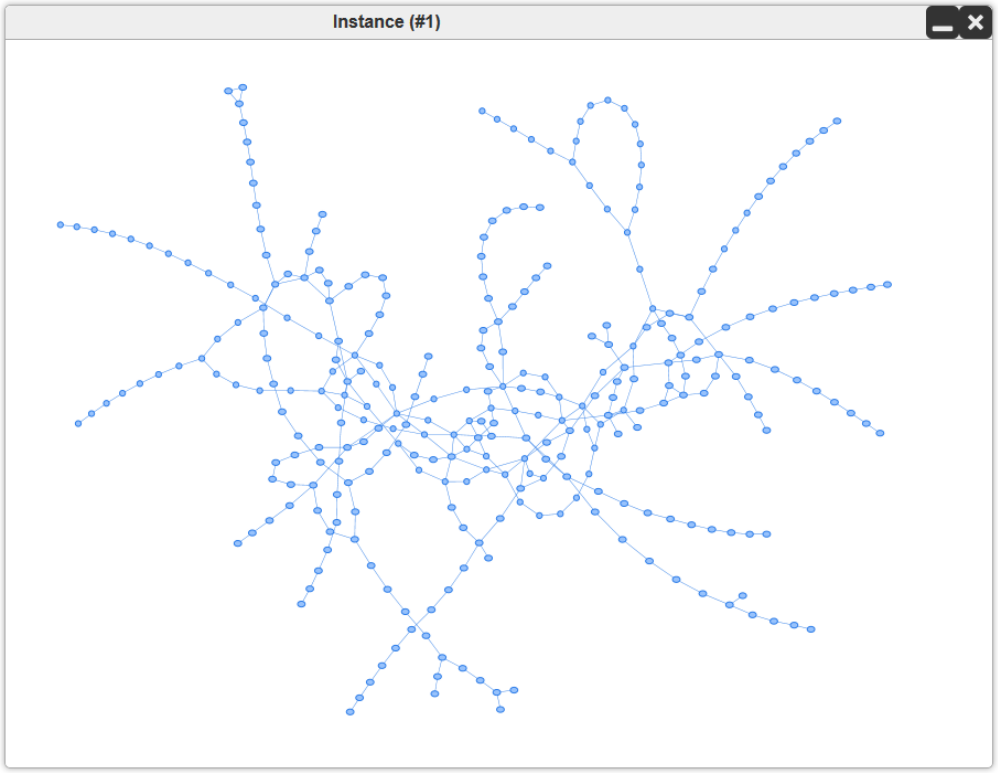


Figure 5.39: Realworld instance: Metro London (304 vertices)

D-FLAT the most time is needed for the first nodes while in *dynBDD* the runtime is constantly increasing from the first node index to the last one.

In Figure 5.43 we illustrate the absolute memory values in bytes. Always keep in mind that they are not really comparable, but nevertheless we want to show the differences. The *dynBDD* tool (orange line) reveals a quite constant memory usage for the BDDs. But the real sizes of the particular BDDs are certainly lower, probably due to caching or some reservation of memory during the computation. *D-FLAT* (blue line) on the other hand has a very low memory usage for the item trees. It is so low that it is not really visible. To give a better insight of the item tree sizes in *D-FLAT*, we illustrate them in Figure 5.44. Here, we see only the item tree sizes in bytes of *D-FLAT*. Only for a few nodes in the tree decomposition an item tree is used at all, for all other nodes the item tree size is 0.

Summary

For the *Hamiltonian Cycle* problem we get different performance results regarding the three tools *dynBDD*, *D-FLAT* and *Sequoia*. Here, *D-FLAT* usually performs best, followed by *dynBDD* and *Sequoia*. This applies to runtime as well as memory usage. Only for some grid-based instances of treewidth 5, *dynBDD* performs a little bit faster

Figure 5.40: Hamiltonian Cycle, *D-FLAT* & *dynBDD*, Metro London: Time

Figure 5.41: Hamiltonian Cycle, *D-FLAT* & *dynBDD*, Metro London: Item tree size vs. memory

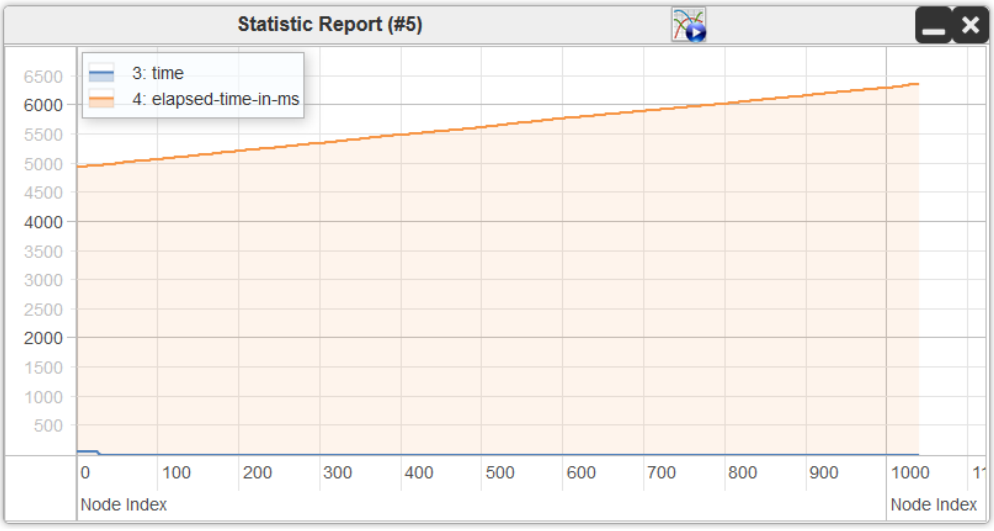


Figure 5.42: Hamiltonian Cycle, *D-FLAT* & *dynBDD*, Metro London: Time

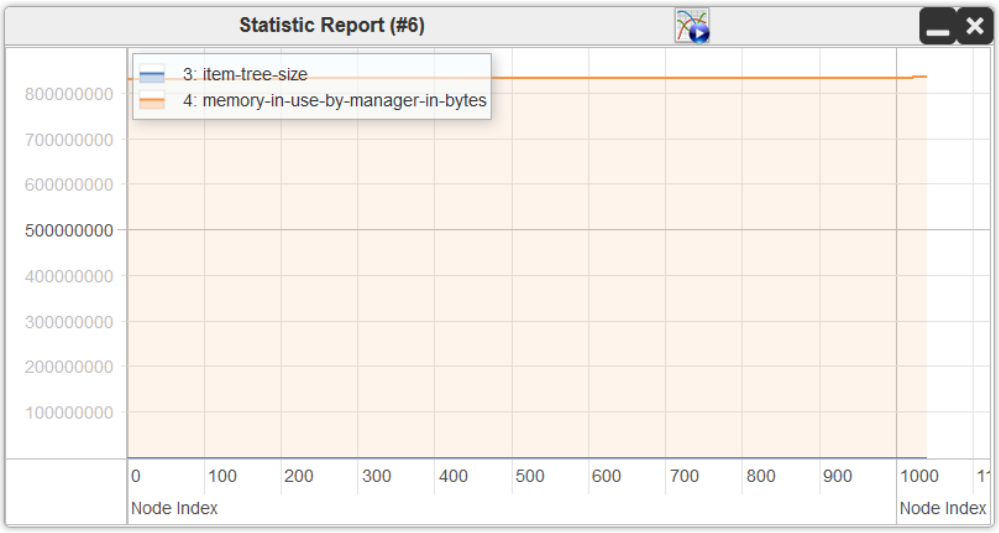


Figure 5.43: Hamiltonian Cycle, *D-FLAT* & *dynBDD*, Metro London: Memory

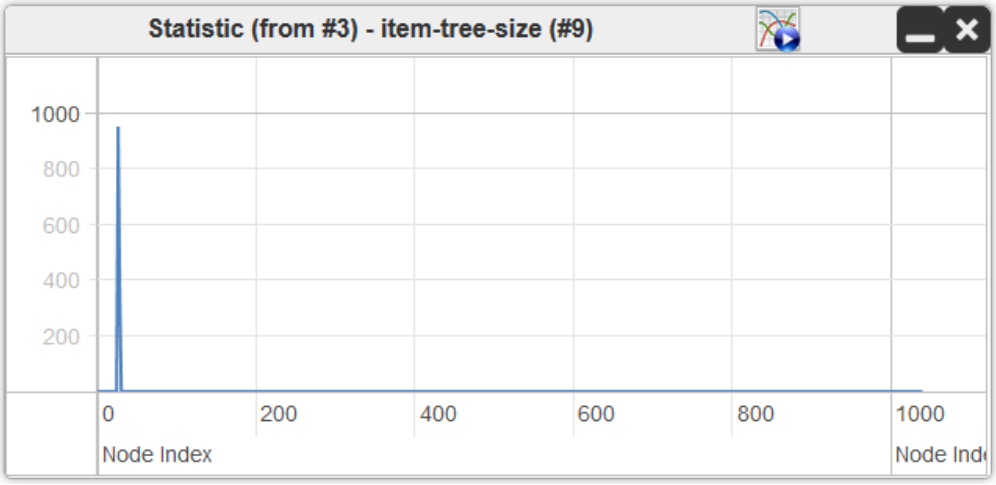


Figure 5.44: Hamiltonian Cycle, *D-FLAT*, Metro London: Memory

than *D-FLAT*. But there is another small exception: for the 2 real-world instances *metro_tram_vienna* and *metro_vienna*, *Sequoia* requires a little bit less memory than *dynBDD*. In Figure 5.45 we see the overview of solved instances in total.

		Satisfiable	Unsatisfiable	Timeout	Memout
Grid-based Treewidth 5	dynBDD	11	136	102	1
	D-FLAT	1	121	128	0
	Sequoia	1	3	0	246
Grid-based Treewidth 20	dynBDD	2	69	59	120
	D-FLAT	0	86	164	0
	Sequoia	0	5	0	245
Random	dynBDD	7	58	10	0
	D-FLAT	6	64	5	0
	Sequoia	4	12	3	56
Realworld	dynBDD	0	10	0	0
	D-FLAT	0	10	0	0
	Sequoia	0	3	0	7

Figure 5.45: Hamiltonian Cycle - Summary of timeouts, memouts and solved instances

5.4.4 Preferred Extension - D-FLAT, D-FLAT^2

The *Preferred Extension* [27, 69] is also an extension of the abstract argumentation framework. First, we want to describe an *Admissible Set* which is required to understand the *Preferred Extension*.

An *Admissible Set* E has two important properties: First, it is conflict-free, therefore there is no conflict between the arguments in E. Second, all arguments are acceptable with respect to E, meaning that if there is an attack from an argument a not included in E to an argument b in E, there has to be at least one argument in E which defends b by attacking a.

A set is a *Preferred Extension* if it is subset maximal among all *Admissible Sets*. Therefore we can use now the *D-FLAT^2* tool which has a focus on subset maximization (and also minimization) problems.

Results & Comparison

In Figure 5.46 we see the comparison of *D-FLAT* and *D-FLAT^2* with grid-based instances having a maximum treewidth of 5. It is easy to see that *D-FLAT^2* is much faster and can handle about twice of the instances in contrast to *D-FLAT*. In detail, *D-FLAT^2* is able to solve 33 instances, whereas *D-FLAT* can only solve 15 instances. All other instances had a timeout.

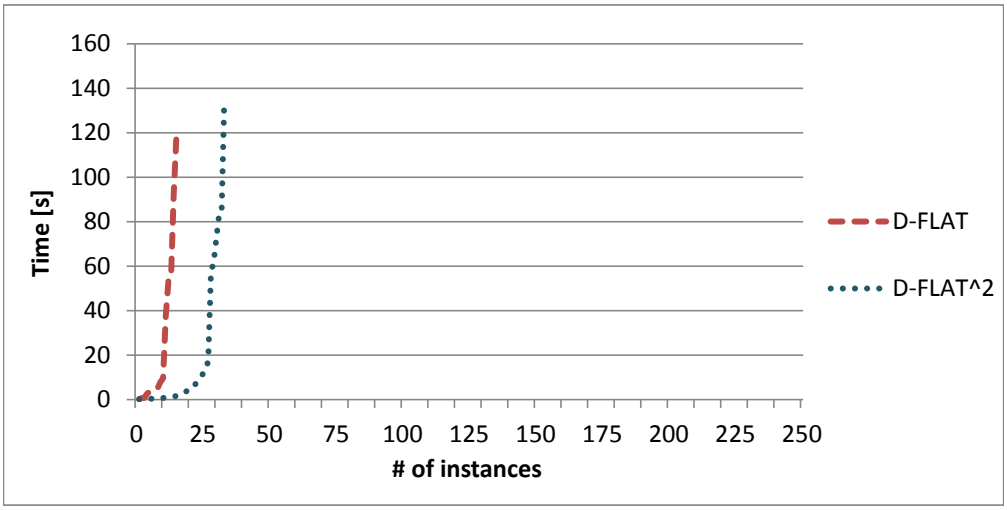


Figure 5.46: Preferred Extension - Time (grid-based instances, treewidth 5)

The memory usage illustrated in Figure 5.47 shows that *D-FLAT^2* needs only a minimum of memory.

Figure 5.48 and Figure 5.49 are showing similar results when comparing grid-based instances with a maxmimum treewidth of 20. Here, *D-FLAT^2* solves 32 instances and *D-FLAT* 23 instances. All the other ones reached the time limit.

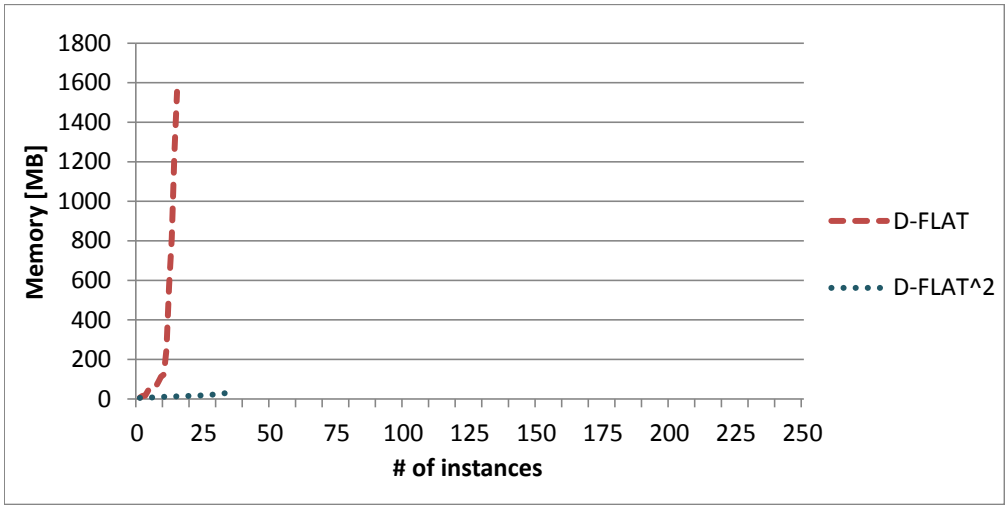


Figure 5.47: Preferred Extension - Memory (grid-based instances, treewidth 5)

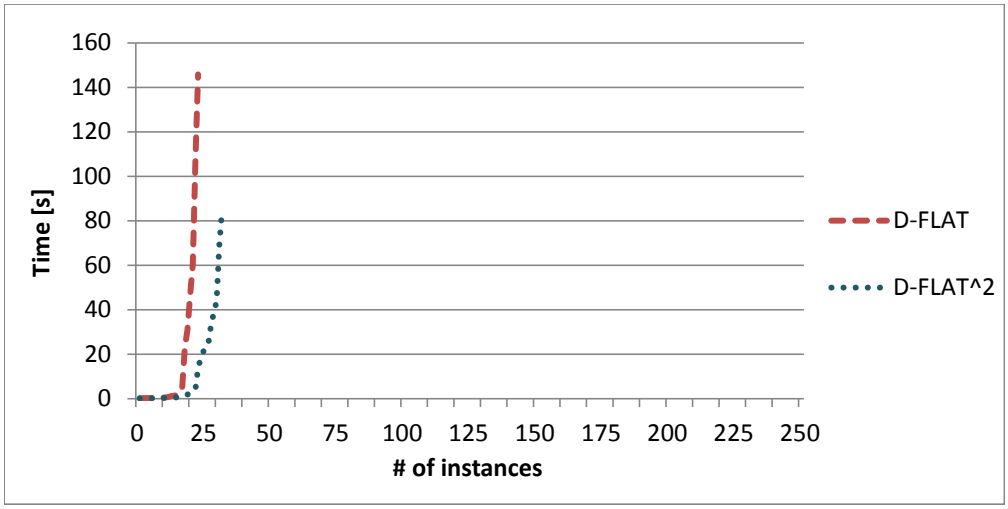


Figure 5.48: Preferred Extension - Time (grid-based instances, treewidth 20)

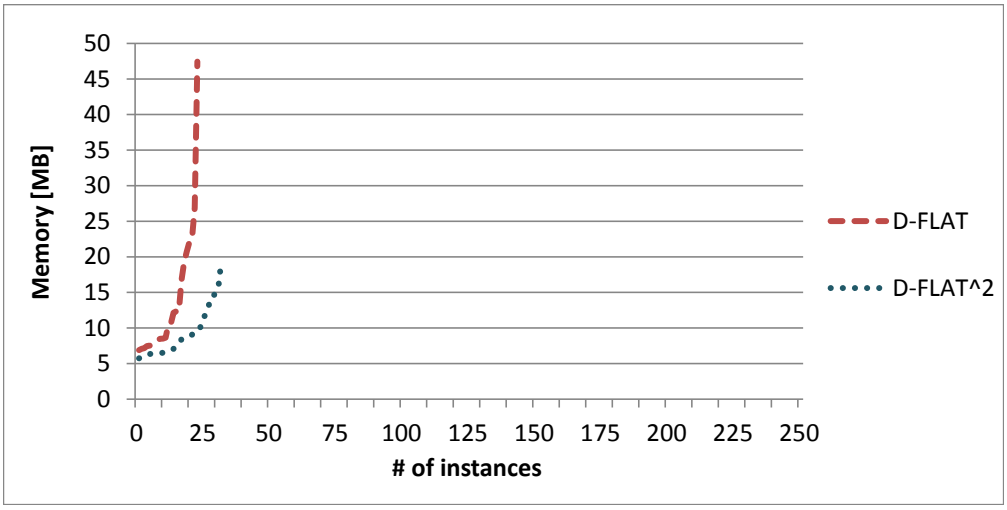


Figure 5.49: Preferred Extension - Memory (grid-based instances, treewidth 20)

The same applies to the random instances shown in Figure 5.50 and Figure 5.51: the growth of runtime and memory happens in quite the same way as before. But in total more instances could be solved. This way *D-FLAT^2* could handle 49 instances. The other 26 instances had a timeout. *D-FLAT* was now able to solve 27 instances. It had 42 timeouts and this time there were 6 memouts.

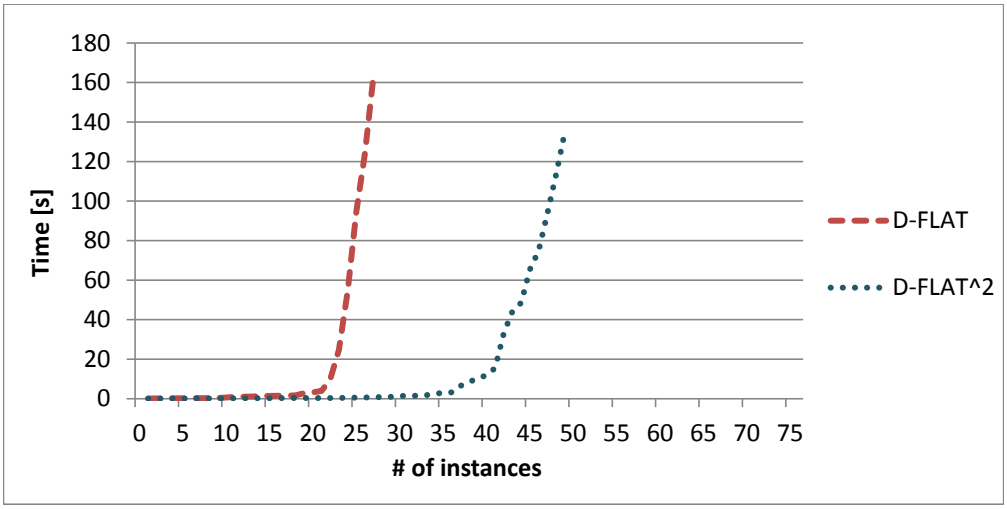


Figure 5.50: Preferred Extension - Time (random instances)

For all the 10 realworld instances *D-FLAT^2* is also able to beat *D-FLAT* in runtime as well as in memory usage (see Figure 5.52 and Figure 5.53). 5 instances could not be solved by *D-FLAT* because of a timeout occurred in all the 10 different tree decompositions of

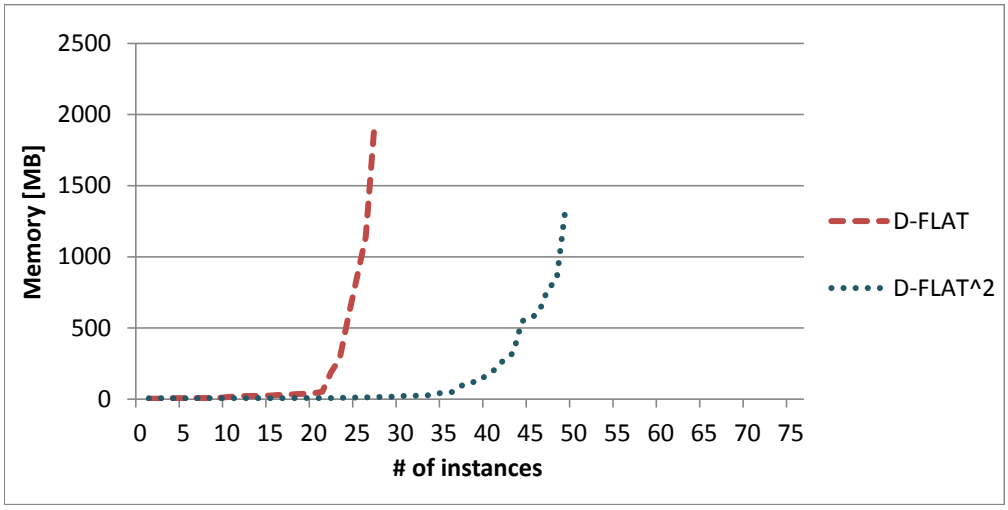


Figure 5.51: Preferred Extension - Memory (random instances)

every instance. *D-FLAT^2* had only 1 timeout.

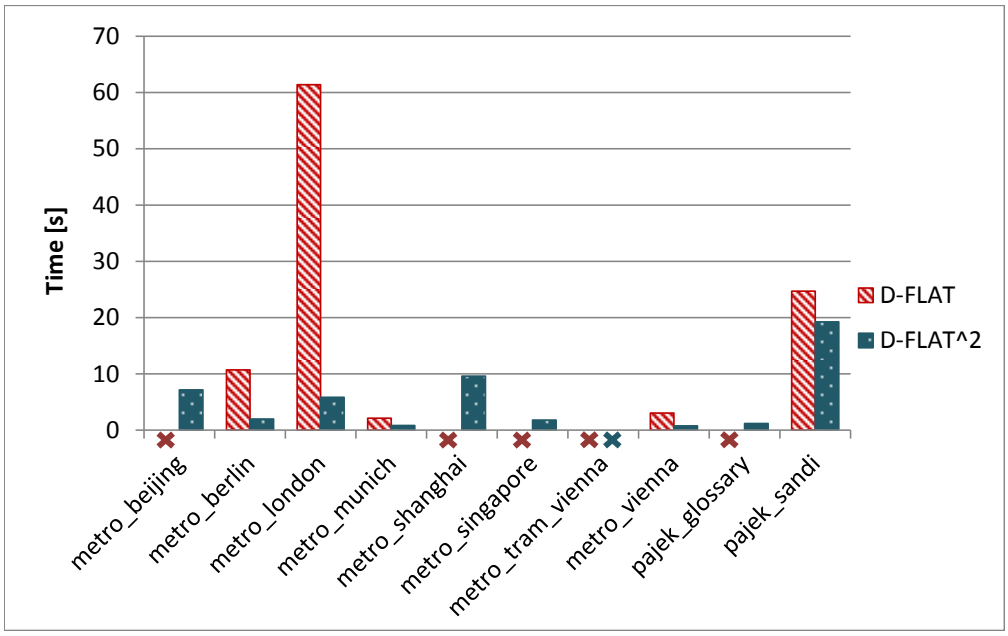


Figure 5.52: Preferred Extension - Time (realworld instances)

In Figure 5.54 we see a summary of all solved instances including timeouts and memouts.

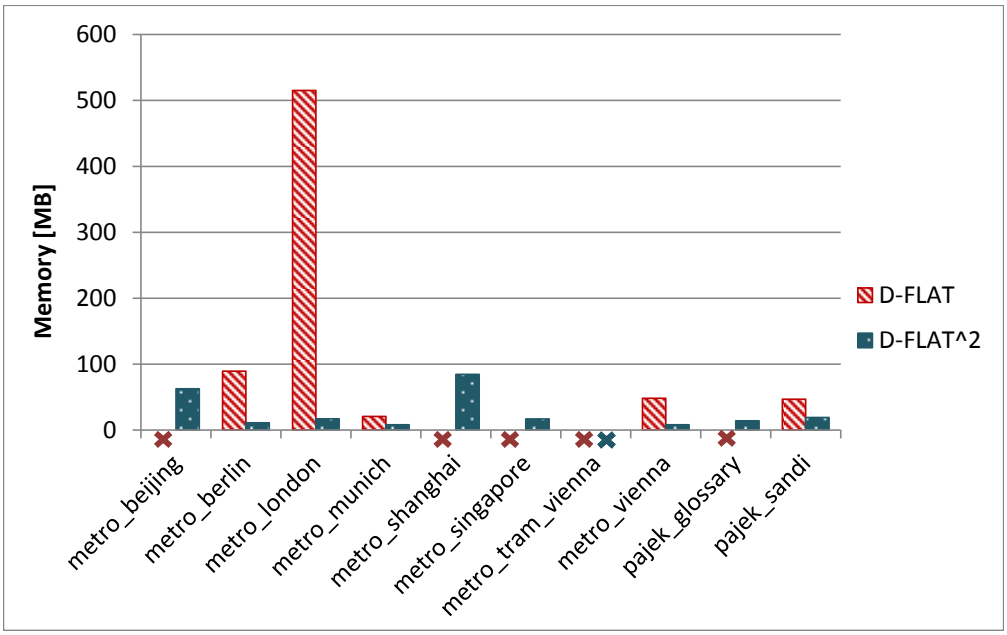


Figure 5.53: Preferred Extension - Memory (realworld instances)

		Solved	Timeout	Memout
Grid-based Treewidth 5	D-FLAT	15	235	0
	D-FLAT^2	33	217	0
Grid-based Treewidth 20	D-FLAT	23	227	0
	D-FLAT^2	32	218	0
Random	D-FLAT	27	42	6
	D-FLAT^2	49	26	0
Realworld	D-FLAT	5	5	0
	D-FLAT^2	9	1	0

Figure 5.54: Preferred Extension - Summary of timeouts, memouts and solved instances

Summary

It is very clear that *D-FLAT^2* is the leader in this benchmark because it could always perform better, not only in runtime but also in memory usage.

5.5 Conclusion

In total, we executed 4 different benchmark tests covering the following problems:

- ***3-Colorability*:** The benchmark revealed that *dynBDD* is the most efficient tool regarding runtime and memory usage. *D-FLAT* is a little bit slower but still powerful. However, *Sequoia* is significant slower in all instances.
- ***Stable Extension*:** Here, *dynBDD* seems to be very efficient too. In all instances it is many times faster and requires extremely little memory in contrast to *D-FLAT*. *Sequoia* was not tested because it does not support this problem.
- ***Hamiltonian Cycle*:** For this problem, *dynBDD* is not the leader any longer since *D-FLAT* is able to solve more instances and is also quite faster and takes less memory. Again, *Sequoia* needs the most time and space and additionally can only solve a very small amount of instances.
- ***Preferred Extension*** *D-FLAT^2* performs very well when the *Preferred Extension* problem is tested. It is faster and takes less memory than *D-FLAT*. We see that the processing in two stages of *D-FLAT^2* indeed has a big performance advantage.

As we can see, there are different rankings, depending on the particular problem. The *dynBDD* tool is the first choice for *3-Colorability* and *Stable Extension* problems, but *D-FLAT* handles *Hamiltonian Cycle* problems better. When we compare subset maximization problems like the *Preferred Extension*, then *D-FLAT^2* is the best choice.

Although *Sequoia* did not score well in all our used benchmark tests, this does not necessarily mean that this applies to other problems.

CHAPTER 6

DecoVis in Practice

This chapter puts the focus on the usage of *DecoVis* for analyzing systems that apply dynamic programming on tree decompositions. Representatively, *dynBDD* is used in this chapter for such dynamic programming tools. Furthermore, this chapter tries to explain particular behaviors of different options and settings provided by *dynBDD*. In addition to the *dynBDD* analysis, conclusions about the suitability of *DecoVis* for daily use are obtained.

Section 6.1 and Section 6.2 provide an introduction to Sections 6.3 - 6.5. The focus in Section 6.1 is placed on the structure of the three sections. The used approaches for analysis are explained in detail.

Section 6.2 gives a short overview over different settings that can be applied on *dynBDD*. These settings represent various approaches and aspects of BDD-based Dynamic Programming on Tree Decompositions. Furthermore, the configuration for each described option in the *DecoVis* dynamic dialog generation for *dynBDD* execution is illustrated.

Sections 6.3 - 6.5 deal with the analysis of *dynBDD* by means of *DecoVis*. Furthermore, the three sections cover the analysis of the three process steps used by systems applying dynamic programming on tree decompositions. Each process step generates an output (Instance, Tree Decomposition and Computation) that can be analyzed in detail by *DecoVis*.

The summary in Section 6.6 examines the analysis of systems using dynamic programming on tree decompositions by means of *DecoVis*. On the one hand this section emphasizes the strengths of *DecoVis* and on the other hand shows the potential for future developments.

6.1 Introduction

DynBDD provides several options and settings that yield various behaviors when computing solutions of problems. This chapter explains particular behaviors by means of *DecoVis*.

For this purpose, instances, tree decompositions and computations are generated, that are raising interesting issues.

Instances are analyzed in Section 6.3 with the help of their instance graphs. Several different instance graphs that are used as basis for further tree decompositions and computations in this chapter are visualized by means of *DecoVis*. Tree decompositions are analyzed in Section 6.3 regarding the different normalization types (definitions and details can be found in Section 2.2). The specific characteristics of each normalization type are illustrated with the help of *DecoVis*. The most comprehensive analyses concern computations (Section 6.4). They are evaluated regarding different decision methods and variable orders. In general, we introduced a consistent approach used for the analysis of computations:

- **Description**
  General information about the particular analysis is listed. Furthermore, used instances, benchmark tests and settings of benchmark tests are described.

- **Instance Selection**
  To get a better overview, benchmark test results are visualized with the help of diagrams. In further consequence, representative runs and their settings used for further analysis are chosen from benchmark test results.

- **Analysis with DecoVis**
  Instances, tree decompositions and computations generated by *dynBDD* are analyzed with the help of *DecoVis*.

- **Conclusions**
  On the basis of the analysis by means of *DecoVis*, conclusions for *dynBDD* are finally derived.

6.2 Different Approaches and Aspects of BDD-based Dynamic Programming

Since *dynBDD* is a tool applying BDD-based Dynamic Programming on Tree Decompositions, it is used in this chapter for computing solutions of NP-hard problems. Furthermore, *dynBDD* provides several approaches for solving problems. To apply specific approaches in *dynBDD* execution, particular options have to be included for running *dynBDD*.

Several concepts are explained in the following and the corresponding options and parameters for *dynBDD* execution are illustrated. Since *dynBDD* can be executed out of *DecoVis* with the help of the dynamically generated *dynBDD* execution dialog, the necessary settings to use the options and parameters are also given in the following.

6.2.1 TD Normalization

As already explained in Section 2.2, several possibilities exist for decomposing graphs. The four tree decomposition normalization types are NONE, WEAK, SEMI and NORMALIZED.

Since *dynBDD* supports each of these normalization types, the dialog in *DecoVis* for *dynBDD* execution should provide an input form for the selection of a particular normalization type.

```
dynbdd.td.norm.option=-n
dynbdd.td.norm.values=none, weak, semi, normalized
dynbdd.td.norm.value=none
dynbdd.td.norm.label=Normalization
dynbdd.td.norm.type=combo
```

Listing 6.1: DecoVis normalization settings

As Listing 4.3 shows, the option for *dynBDD* is **n** and the possible parameters are **none**, **weak**, **semi** and **normalized**.

6.2.2 LDM / EDM

DynBDD defines two different paradigms regarding the dynamic programming algorithm used for solving tree decomposition nodes: Late Decision Method (LDM) and Early Decision Method (EDM). Detailed explanations about the two concepts can be found in Section 2.3.

In contrast to the combobox for the selection of the normalization type above, the decision method can be represented as checkbox in *DecoVis*.

```
dynbdd.comp.dm.option=--edm
dynbdd.comp.dm.label=Decision Method (Default: ldm)
dynbdd.comp.dm.type=checkbox
```

Listing 6.2: DecoVis decision method settings

Since *dynBDD* uses LDM by default, Listing 6.2 defines the selection of the possible option **edm**.

6.2.3 Variable Ordering

Systems using BDD-based Dynamic Programming on Tree Decompositions like *dynBDD* provide multiple concepts for the ordering of variables. A BDD variable defines the assignment of a value to arbitrarily many vertices (details can be found in Section 3.4). Furthermore, each BDD variable assigns a particular BDD level (order) where level 1 specifies the level of the root node and the leaf nodes have the highest level. Variable ordering has a strong influence on the construction of BDDs.

Figure 6.1 serves as example for an instance graph and a tree decomposition for further explanations. *DynBDD* supports the following variable ordering types:

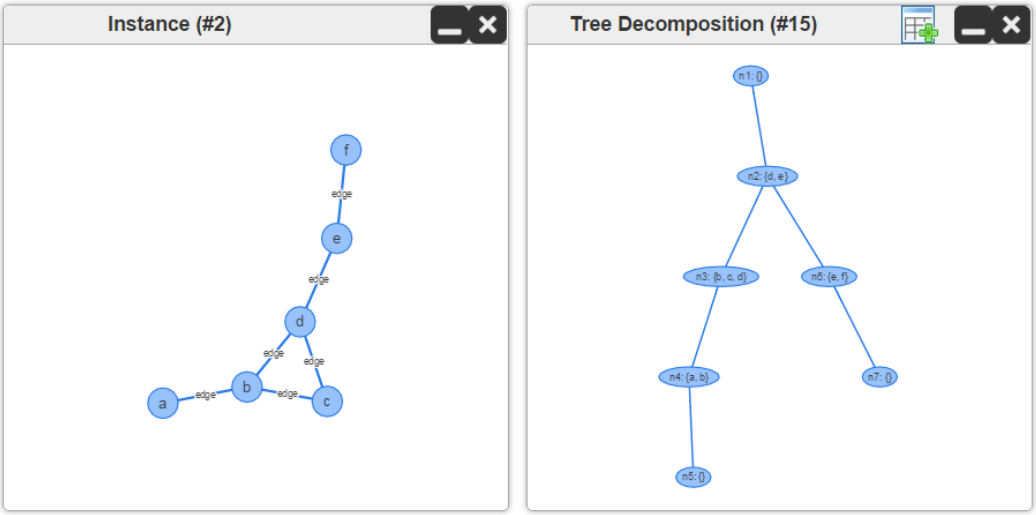


Figure 6.1: Variable ordering example

- **Instance**
  The variable order depends on the vertex order in the instance (source) files. Let us assume the vertices in Figure 6.1 are defined in the instance file in the following order: **f**, **e**, **d**, **b**, **c**, **a**. In instance order this order remains the same.

- **Lexicographic**
  The variable order depends on the alphabetical order of the vertex names defined in the input file. Lexicographic variable order regarding the instance vertices in Figure 6.1 results in the following ordering: **a**, **b**, **c**, **d**, **e**, **f**.

- **Td-first**
  Td-first variable order depends on the traversal order of the tree decomposition when computing the corresponding BDDs. *DynBDD* traverses the tree post-order. The td-first order prioritizes vertex names in the order they occur when traversing the tree post-order. The td-first order for the vertices in Figure 6.1 is defined as follows: **b**, **a**, **d**, **c**, **e**, **f**.

- **Degree-high**
  Degree-high prioritizes each vertex with the help of its degree: When the degree is high, the order is also high. The degree-high order for the vertices in Figure 6.1 is defined as follows: **b**, **d**, **c**, **e**, **a**, **f**.

- **Degree-low**
  This variable order is the inverse of degree-high: When the degree is high, the order

is low. The degree-low order for the vertices in Figure 6.1 is defined as follows: **a**, **f**, **c**, **e**, **b**, **d**.

Since *dynBDD* supports each of these order types, the dialog in *DecoVis* for *dynBDD* execution should provide an input form for the selection of a particular order type.

```
dynbdd.comp.ordering.option=-o
dynbdd.comp.ordering.values=instance, lexicographic, td-first, degree-high, degree-low
dynbdd.comp.ordering.value=instance
dynbdd.comp.ordering.label=Ordering
dynbdd.comp.ordering.type=combo
```

Listing 6.3: DecoVis variable ordering settings

As Listing 6.3 shows, the option for *dynBDD* is **o** and the possible parameters are **instance**, **lexicographic**, **td-first**, **degree-high** and **degree-low**.

6.3 Instance Analysis

In general, instances represent the structure of graphs. Their nodes and edges are used for graph analysis and for the generation of tree decompositions. This section lists several graph instances used as basis for the tree decompositions and computations in the following sections (Section 6.4 and Section 6.5). They make use of *DecoVis* that visualizes the instances on the one hand and emphasizes particular characteristics of each instance on the other hand.

6.3.1 Vienna Metro

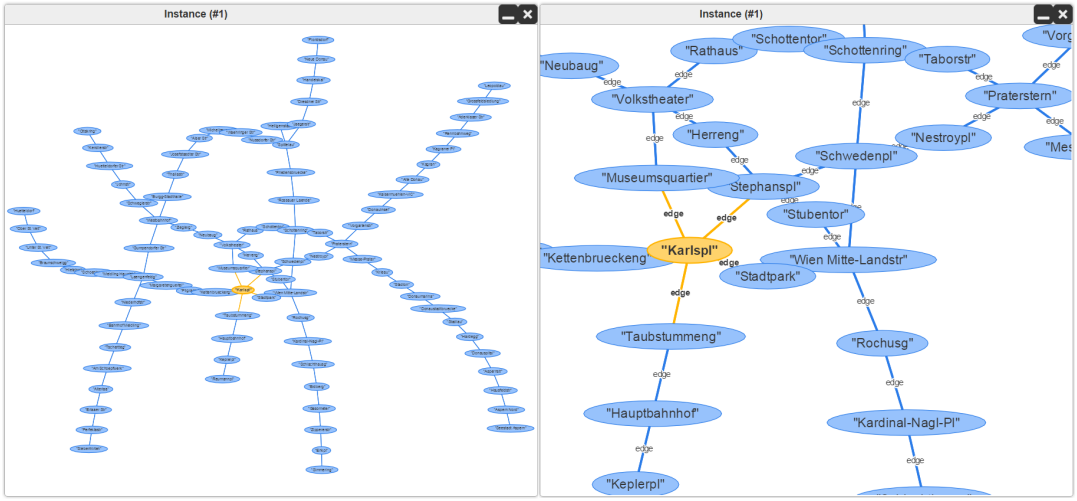


Figure 6.2: Vienna Metro

The first instance illustrates a real-world graph representing the Vienna metro map. It consists of the metro lines U1, U2, U4, U4 and U6 (valid as of November 2015).

Beside the displaying of the metro stations and metro lines in Vienna, Figure 6.2 shows that the stations and lines are also appropriately arranged. That means, that the placements of stations and lines are similar to the placements in official Vienna metro maps (maps available via internet or printed maps). The stations illustrated in the left window of Figure 6.2 are placed manually via *DecoVis*. The right window represents a zoomed view to the stations of the inner districts of Vienna. Furthermore, the metro station Karlsplatz is selected in Figure 6.2.

6.3.2 Grid-based Instances

A grid is a $m \times n$ graph consisting of horizontal and vertical edges where the vertices are aligned in form of a grid. To keep it simple, the grids used in this chapter are $n \times n$ graphs. Basically the following applies: The greater n, the higher the amount of graph cycles inside the $n \times n$ graph. Furthermore, we use this graph type because the treewidth is always proportional to the graph size, i.e. the treewidth of $n \times n$ graphs is always n.

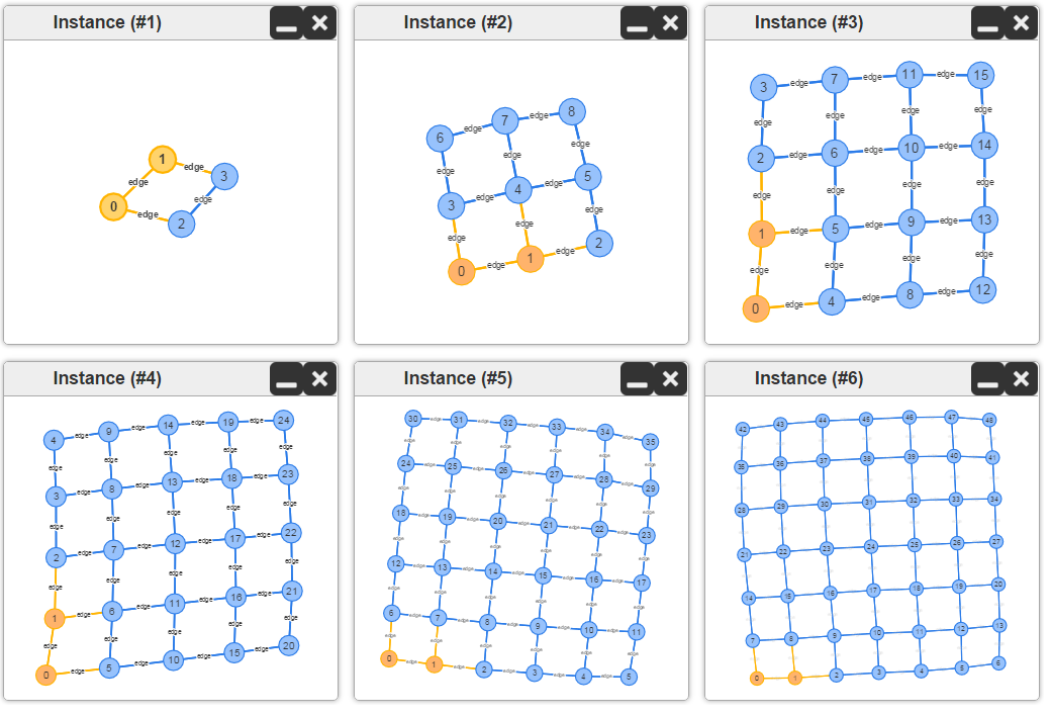


Figure 6.3: Grid instance graphs: 4..49 (node count)

Figure 6.3 illustrates the visualization of the grids where n is between 2 and 7 in *DecoVis*. Furthermore, *DecoVis* tells us about the construction process of the algorithm

that generated the grid graphs. As one can see the algorithm starts with vertex **0** that is set to the corner of each grid. Subsequently, the next vertex **1** is connected via an edge to **0**. To build a line in the grid, this process is repeated until vertex **n-1** is reached. After the creation of n-1 lines with the help of this process, edges are added to connect the lines to form a grid.

6.3.3 Random Instances with Edge Probability

Here we consider a random graph consisting of a fixed amount of nodes and a random number of edges. The number of edges is given by the edge-probability, which represents the probability whether an edge exists between pairs of vertices.

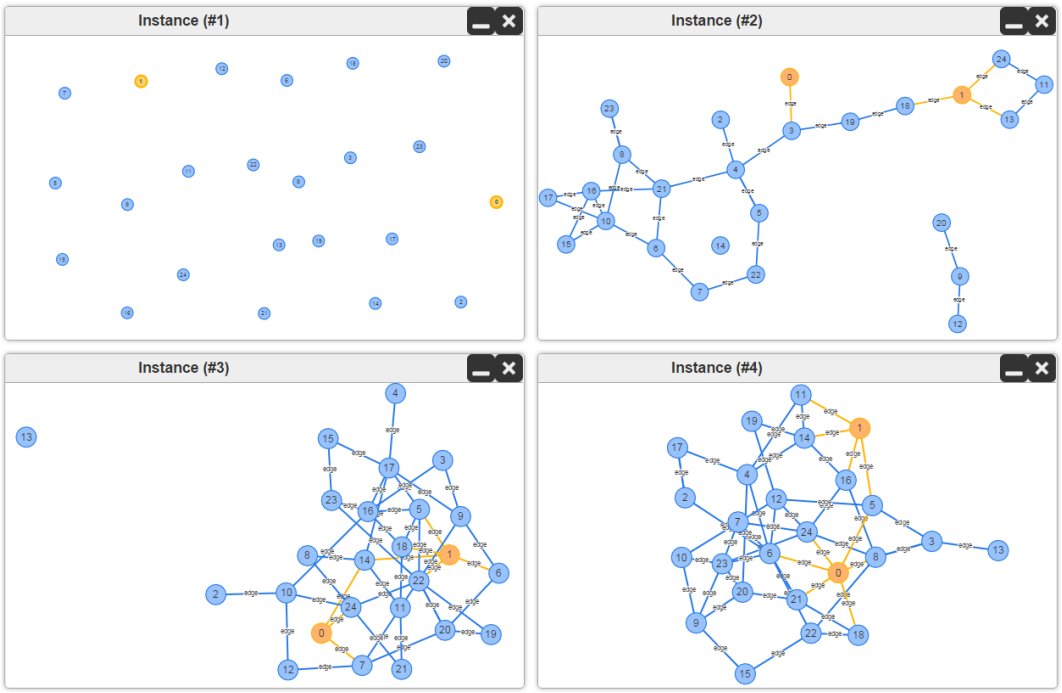


Figure 6.4: Random graphs with edge-probability: 0%, 4% , 7%, 9%

As one can see in Figure 6.4 each instance consists of 25 nodes. The used edge-probabilities are 0%, 4% , 7% and 9%. In order to illustrate that the instance graphs are generated independently to each other, the vertices **0** and **1** are selected in each instance. It is easy to see the neighbors of the selected vertices in a random graph vary from instance to instance and the amount of neighbors increases with the edge-probability.

6.3.4 Clique-based Instances

A clique is a worst case instance regarding the number of edges. A graph is a clique, if every two distinct nodes in the graph are adjacent. As a consequence, the treewidth of the graph is the number of nodes minus 1. Furthermore, a clique consists of the maximal possible amount of graph cycles regarding its node count.

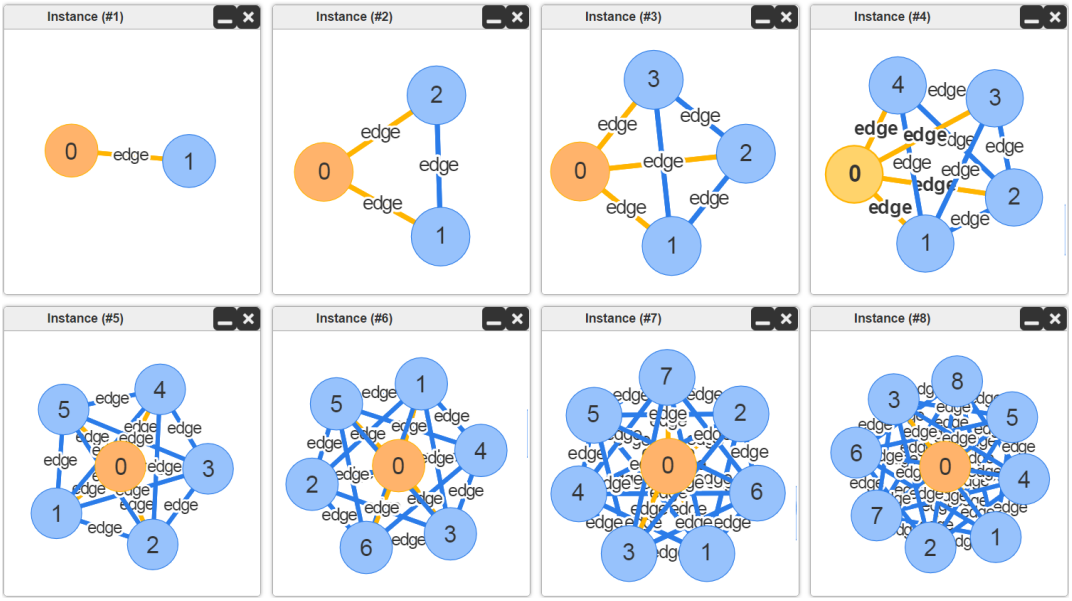


Figure 6.5: Clique with treewidth: 1..8

Figure 6.5 represents cliques where the treewidth is defined between 1 and 8. As one can see with the help of vertex **0**, each vertex retains its neighbors in spite of increasing treewidth.

6.4 Tree Decomposition Analysis

6.4.1 Description

A tree decomposition represents a mapping from the instance graph to a tree. Hence, each tree decomposition node contains several vertices from the instance graph. Furthermore, we differ between four normalization types of tree decompositions: none, weak, semi and normalized. Since *DecoVis* provides the possibility of visualizing tree decompositions generated with *dynBDD*, various interesting issues about tree decompositions can be analyzed with the help of *DecoVis*.

This section analyzes the occurrences of particular vertices from the instance graph in tree decompositions with different normalization types. Furthermore, various characteristics of each normalization type are illustrated by means of *DecoVis*.

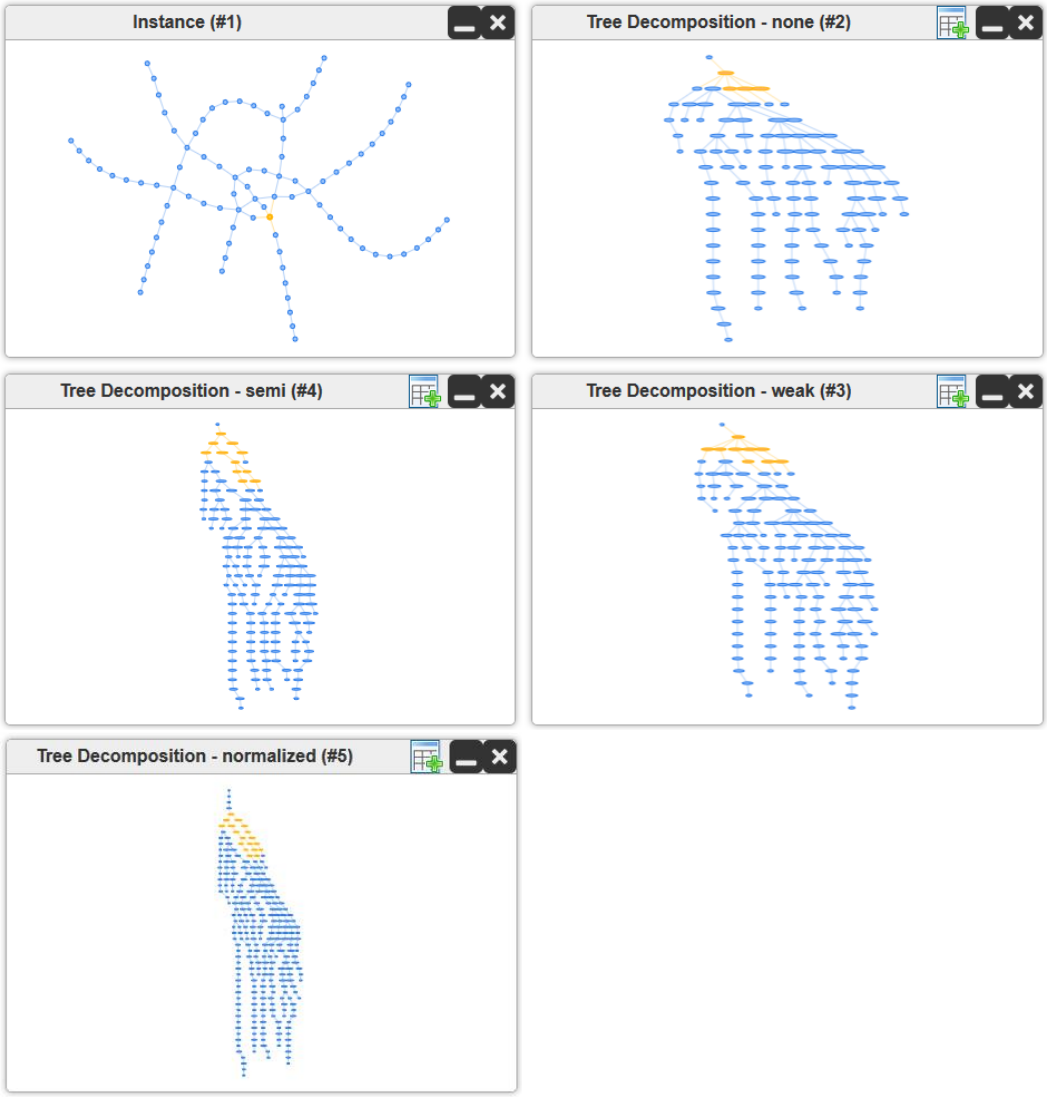


Figure 6.6: Instance + tree decompositions

Figure 6.6 represents the Vienna metro instance from Section 6.3 and a possible decomposition, generated with each normalization type, visualized with the help of *DecoVis*. The difference to the Vienna metro instance from Section 6.3 is the replacement of the metro station names (vertex names) by numbers.

We select vertex **48** (**Wien Mitte - Landstraße**) for further analysis. As one can see in Figure 6.6, **48** can be found several times in the bags of each tree decomposition. We can conclude from this figure, that the occurrences of **48** increases from the none normalized tree decomposition up to the normalized tree decomposition. The differences

between the normalization types are analyzed in the following in more detail.

6.4.2 None vs. Weak Normalized Tree Decompositions

The difference between none normalized tree decompositions and weak normalized tree decompositions lies in the definition of join nodes. For join nodes in weak normalized tree decomposition applies: On the one hand each join node has at least two child nodes and on the other hand each child bag of a join node is equivalent to the join node bag.

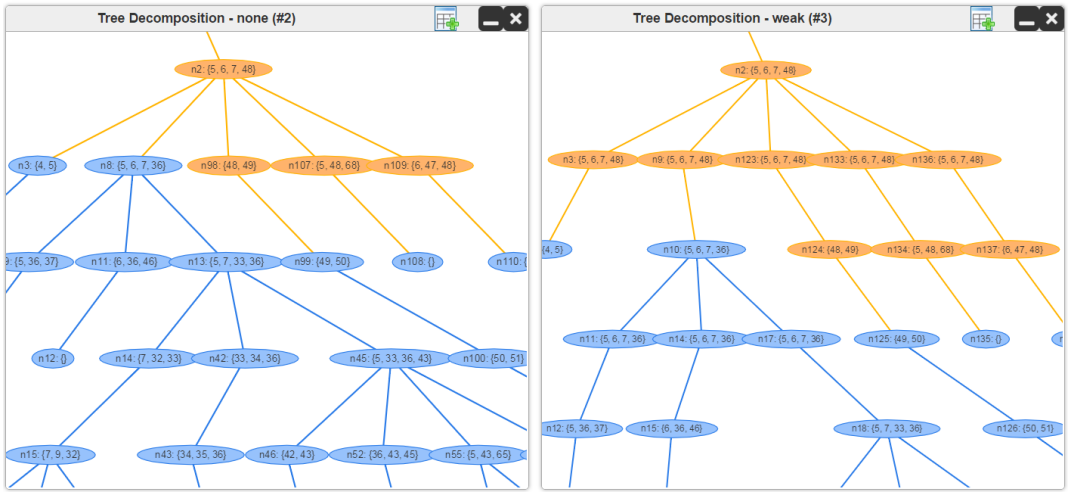


Figure 6.7: None vs. weak normalized tree decompositions

Figure 6.7 represents two zoomed views of the none normalized tree decomposition and the weak normalized tree decomposition of the Vienna metro instance that can be found in Figure 6.6. The two windows consist of the tree decomposition nodes where **48** is a member of the bag. The root nodes of the two tree decompositions and their child nodes serve as example to illustrate the difference between the normalization types none and weak. In contrast to the none normalized tree decomposition, each child node (**n3**, **n9**, **n123**, **n133** and **n136**) of the weak normalized tree decomposition comprise the same bag (**5**, **6**, **7** and **48**) as the root node **n123**. *DecoVis* nicely illustrates that vertices included in the bags of join nodes are more widespread in weak normalized tree decompositions than in the corresponding none normalized tree decompositions.

6.4.3 Weak vs. Semi Normalized Tree Decompositions

The difference between weak normalized tree decompositions and semi normalized tree decompositions is, that join nodes in semi normalized tree decompositions have exactly two child nodes instead of more than one child node.

To illustrate the difference between the weak normalized and the semi normalized normalization type on the one hand and to analyze the distribution of vertex **48** on

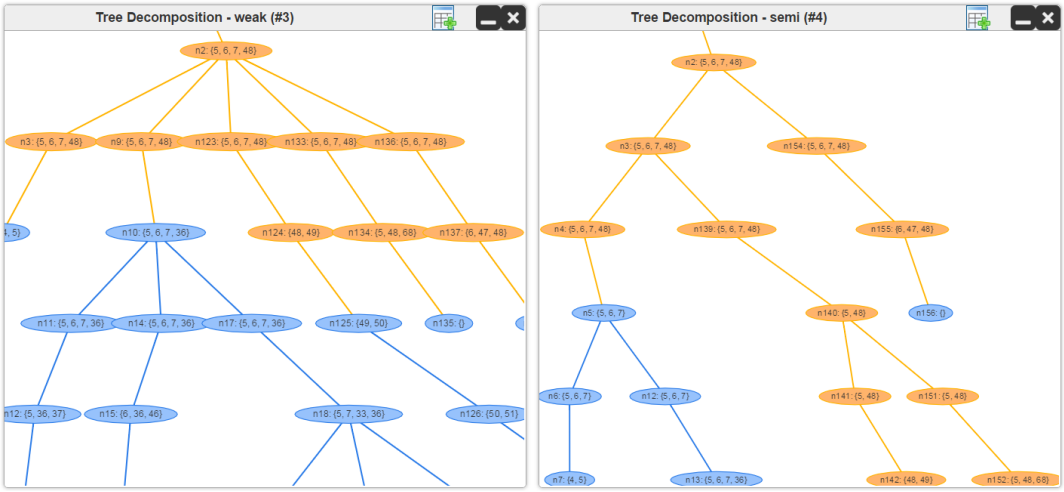


Figure 6.8: Weak vs. semi normalized tree decompositions

the other hand, Figure 6.8 represents the extracts of the weak normalized and the semi normalized tree decomposition where **48** can be found in the bags. It is easy to see, that each join node consists of exactly two child nodes (i.e. **n2** with child nodes **n3** and **n154**). We observe, that vertex **48** can be found in less bags of the semi normalized tree decomposition than in bags of the weak normalized tree decomposition. Furthermore, we see that a vertex available in a bag of the semi normalized tree decomposition join node spreads deeper towards the leaf nodes than the corresponding vertex in the weak normalized tree decomposition.

6.4.4 Semi vs. Normalized Tree Decompositions

The difference between semi normalized tree decompositions and normalized tree decompositions lies in the definition of exchange nodes. For exchange nodes in normalized tree decompositions applies: The bag of the parent node and the bag of the child node only differs in one vertex.

The application of this definition can be found in the comparison of the semi normalized and normalized tree decomposition in Figure 6.9. We choose **n155** from the semi normalized tree decomposition and the corresponding node **n270** from the normalized tree decomposition for further distinctions. With the help of *DecoVis* it is easy to see that the definition above can not be applied to **n155** and **n156** in the semi normalized tree decomposition. The normalized tree decomposition solves this problem by breaking up the bag of **n270** into further tree decomposition nodes **n271** and **n270**. As this example already suggests, normalized tree decompositions normally consist of much more nodes than the corresponding semi normalized tree decompositions. Furthermore, we can conclude from this observation made with *DecoVis*, that the normalized tree

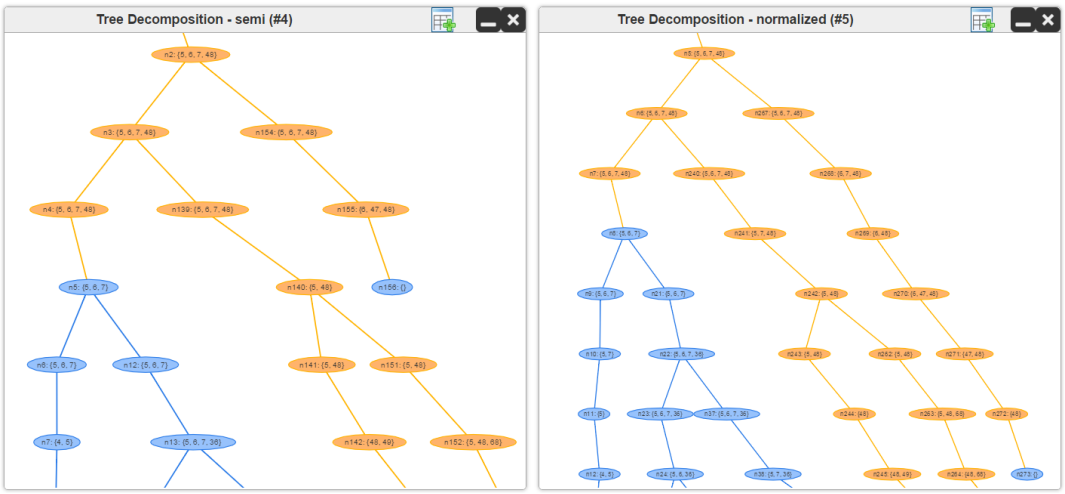


Figure 6.9: Semi vs. normalized tree decompositions

decomposition has a greater tree height than the corresponding semi normalized tree decomposition.

6.5 Computation Analysis

6.5.1 LDM / EDM - BDD Manipulation during the Computation

Description

EDM (Early Decision Method) and LDM (Late Decision Method) represent two different dynamic programming approaches used for tree decompositions. The following explanations give an insight into the process of generating BDDs in each decision method. Furthermore, the following question is discussed with the help of a benchmark test and *DecoVis*: How do the decision methods LDM and EDM differ in the manipulation of the BDDs during the computation?

FIXED:
Instance: Vienna metro
Problem: 3−Colorability
Ordering: Instance
Normalization: Normalized
Nodes: 93

VARIABLE:
Seed: 1..40
Decision−Method: LDM vs. EDM
Measurement: Memory

We use the Vienna metro map without station name specification, shown in Figure 6.6, as instance for the benchmark test (the station names are included in Figure 6.2). The benchmark test gives us information about the memory consumption of *dynBDD* when the 3-Colorability problem is solved by means of LDM and EDM and various seed values. Furthermore, the result of the benchmark test is used for instance selection.

Instance Selection

The diagram in Figure 6.10 represents a scatter chart in combination with a line chart. The memory consumption of the runs, applying different seeds (1..40) and decision methods, are visualized by means of dots in the chart. Furthermore, the average memory consumption across the specified seeds of each decision method is illustrated with the help of a line.

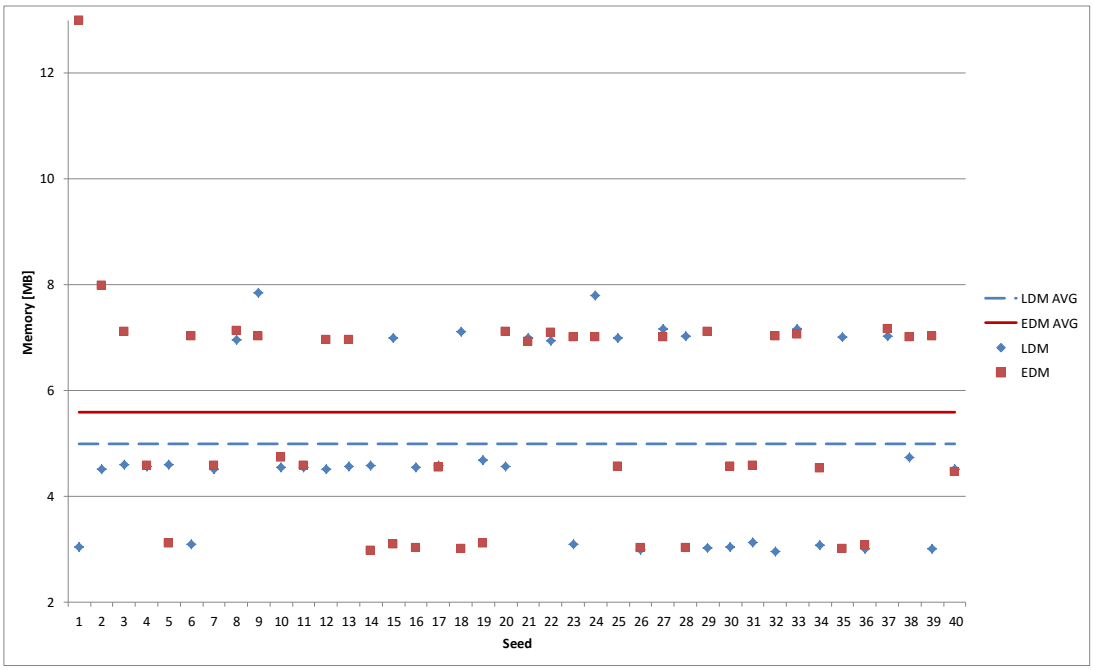


Figure 6.10: EDM vs. LDM: Statistics

We can see in Figure 6.10, that on average EDM requires more memory for solving the 3-Colorability problem for the Vienna metro instance. Furthermore, this illustration of the benchmark test by means of the diagram helps to select computations with the highest difference regarding memory usage during the computation process. Since the memory consumption between the LDM and EDM computation differs most when seed 1 is used, these two computations are used for further analysis.

Analysis with DecoVis

We observe in the result of the benchmark test in Figure 6.10, that on average EDM requires more memory than LDM for solving the 3-Colorability problem for a real-world instance. The following analysis gives one possible explanation for this observation.

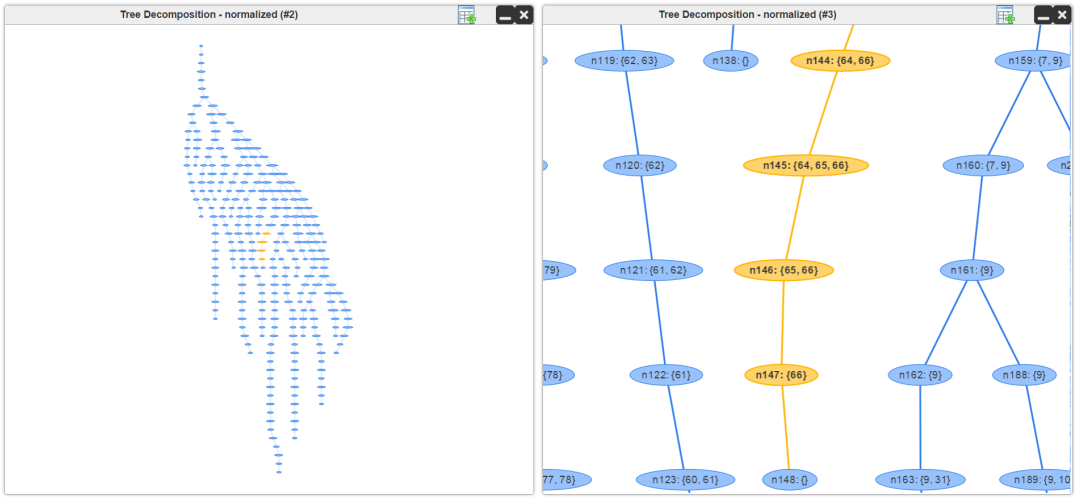


Figure 6.11: EDM vs. LDM (seed 1): Tree decomposition

The left window in Figure 6.11 represents the tree decomposition generated with seed 1 and visualized with the help of *DecoVis*. The right window represents a zoomed view of the tree decomposition consisting of tree decomposition nodes used for further illustrations. **n147**, **n146**, **n145** and **n144** represent the chosen tree decomposition nodes for the explanation of how LDM and EDM differ in the manipulation of their BDDs during the computation process.

Computed by means of LDM, Figure 6.12 illustrates the corresponding BDDs to the tree decomposition nodes **n147**, **n146**, **n145** and **n144**. As one can see, the BDDs of **n147**, **n146** and **n145** only consist of one node, the leaf node TRUE (1). Since vertex **65** gets removed in tree decomposition node **n144**, the corresponding BDD to node **n144** is illustrated in Figure 6.12 with window id **8**. As a consequence, the BDD only consists of variable assignments in the BDD nodes for the vertices **64** and **66**.

The BDDs of the EDM computations can be found in Figure 6.13. We can clearly see here, that each tree decomposition node consists of a corresponding BDD that does not only contain a TRUE leaf node. Furthermore, it is easy to see that each vertex from the bag of a tree decomposition node can be found in the corresponding BDD. For instance, the BDD of **n145** contains BDD variables for the vertices **64**, **65** and **66**.

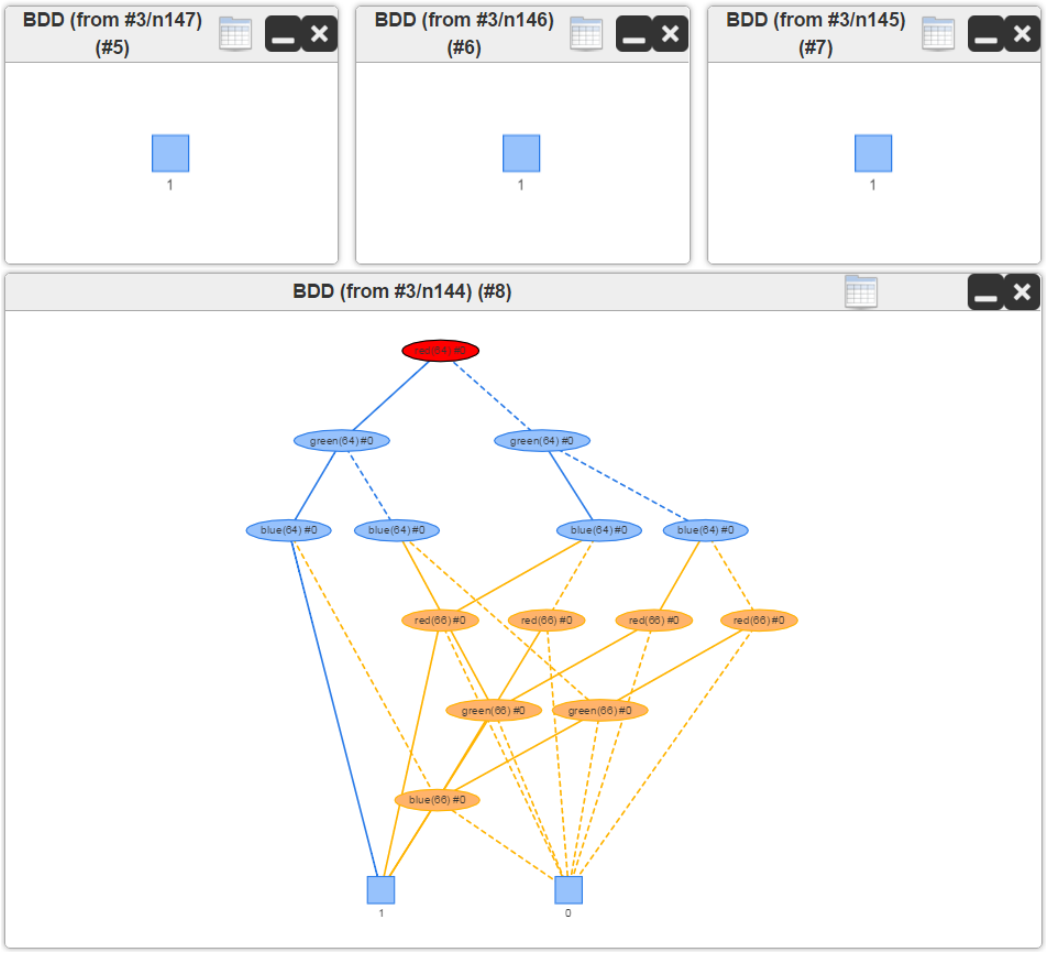


Figure 6.12: LDM - BDDs

Conclusion

DecoVis nicely illustrates the difference of the BDD manipulations during the computation process between the LDM approach and the EDM approach. BDDs only change during the LDM computation if a vertex gets removed from the bag of a tree decomposition node. In contrast to LDM, EDM always generates new BDDs consisting all vertices from the bag of the corresponding tree decomposition node. This observation is one possible explanation for the result of the benchmark test above: On average, EDM requires more memory than LDM for solving the 3-Colorability problem for an instance of real life.

Figure 6.13: EDM - BDDs

6.5.2 LDM / EDM - Instances with High Edge Density

Description

Solving problems with several graph instances, each consisting of a very high edge density, by means of LDM and EDM represents the analysis of worst case scenarios. A benchmark test, using various instances with a high edge density, and *DecoVis* help us to answer the following question: How and why do instances with high edge density differ regarding LDM and EDM?

The details and parameters for the benchmark test are defined in the following:

FIXED:
Problem: Hamiltonian Cycle
Seed: 0
Normalization: NORMALIZED
Ordering: Instance

VARIABLE:
Instance: cliques treewidth=2..9
DM: EDM vs. LDM
Measurement: Memory

Each instance graph used in the benchmark test is a clique, i.e. every two distinct nodes in the graph are adjacent. As a consequence, the treewidth of the graph is the number of nodes minus 1. The benchmark test computes the satisfiability of each instance graph regarding Hamiltonian Cycles with the help of the EDM algorithm and the LDM algorithm.

Instance Selection

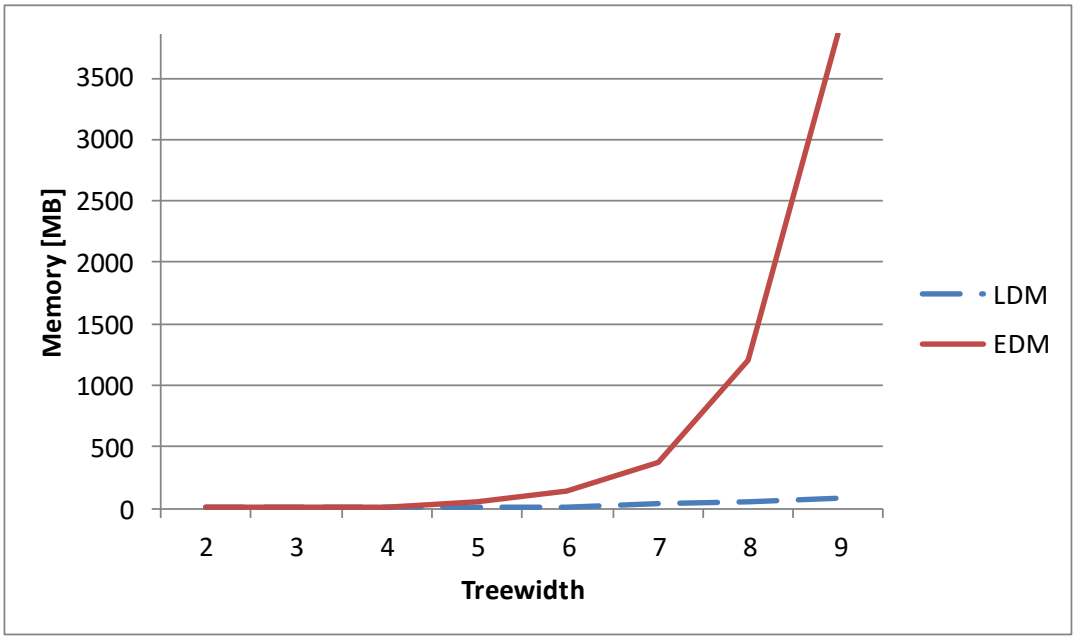


Figure 6.14: EDM vs. LDM: Statistics

Figure 6.14 illustrates the results regarding the memory consumption for solving the problem with its instances defined above with the help of a line chart. The lines represent the decision methods LDM and EDM. The x-axis defines on the one hand the treewidth of the graph and on the other hand the node count that can be computed from the treewidth (node count = treewidth + 1). The y-axis defines the used memory for each run in the unit megabytes.

As one can see, the memory consumption is quite similar when solving instances with treewidth 2, 3 and 4. Since the memory consumption rises sharply from treewidth 5 to treewidth 9, it seems that the consumption of EDM grows strongly exponential whereas the used memory of LDM grows slightly exponential. All instances with a treewidth greater than 9 and computed by means of EDM, result in a timout.

Analysis with DecoVis

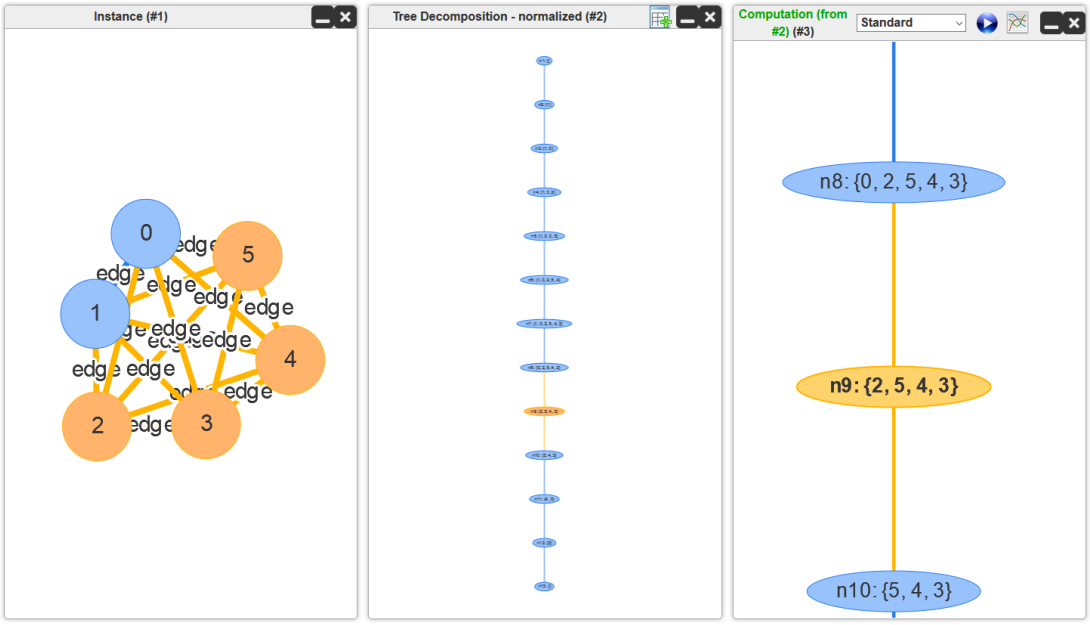


Figure 6.15: Clique with treewidth 5 and its tree decomposition

We want to explain this behavior by analysis with the help of *DecoVis*. Figure 6.15 illustrates in the left window the instance of the clique with treewidth 5 and in the center window the tree decomposition of the instance graph. The right window provides a zoomed view of the node **n9** from the tree decomposition shown in the center window. Since tree decomposition node **n9** is selected for further analysis, each vertex from the bag of the tree decomposition node (Figure 6.15) is marked and **n9** is emphasized in Figure 6.16.

Figure 6.16 represents a statistic report of the LDM (3) and EDM (4) computations regarding the number of nodes in the generated BDDs (y-axis). The x-axis in the line chart represents the tree decomposition nodes in the computation order (post-order), in this tree decomposition: from **n13** to the root node **n1**. As one can see in the statistic report, EDM with the tree decomposition nodes **n9**, **n8**, **n7**, **n6** and **n5** need much more BDD nodes in comparison to their corresponding tree decomposition nodes from LDM. As an example, the BDD of the marked tree decomposition node **n9** from the EDM line

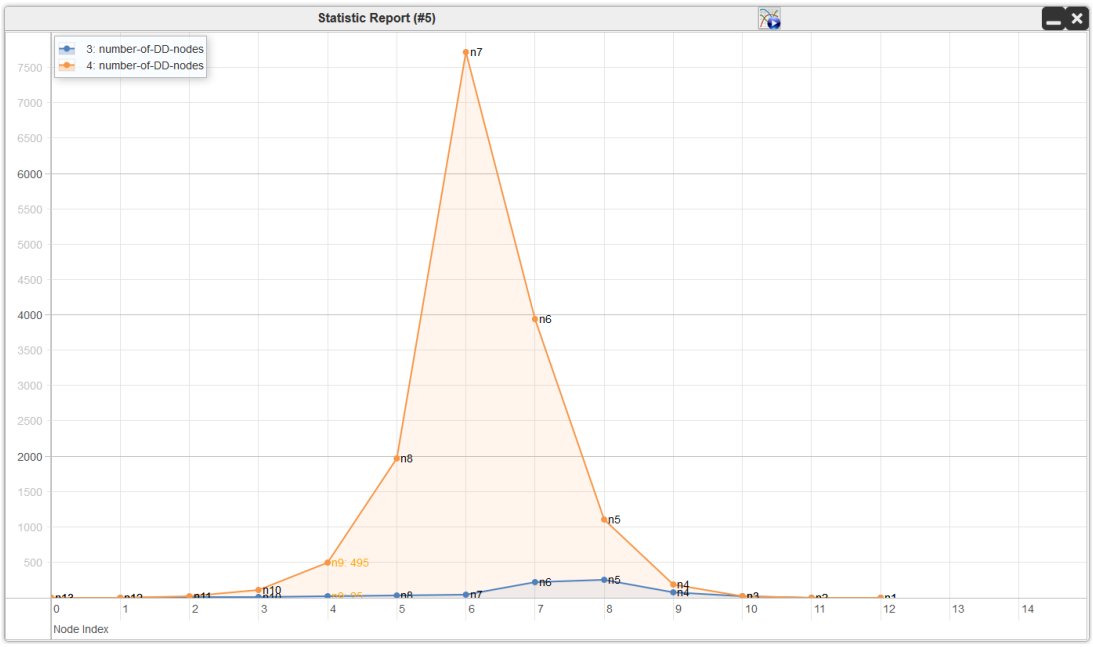


Figure 6.16: LDM vs. EDM: Statistic report (number of DD-nodes)

consists of 495 nodes, the corresponding node from the LDM line only consists of 25 nodes.

Figure 6.17 illustrates the computed BDDs from the LDM and EDM computation of tree decomposition node **n9**. As one can see, the BDD of the LDM computation is build up of way less nodes than the BDD of the EDM computation.

The window on the top of Figure 6.18 represents the corresponding computation table to the BDD computed with the help of LDM shown in the left window of Figure 6.17. The window on the bottom of Figure 6.18 represents the corresponding computation table of the BDD computed with the help of EDM shown in the right window of Figure 6.17. Each line of the particular computation table represents a possible model that is represented by the corresponding BDD.

Conclusion

By means of the benchmark test regarding solving the Hamiltonian Cycle problem of various cliques it is clearly recognizable, that cliques computed with EDM need much more memory consumption than the corresponding LDM computation. Due to these benchmark test results we have a reason to believe, that solving an instance graph with a high edge density by means of EDM results in a much higher memory consumption than solving the same instance graphs with the help of LDM (at least if the Hamiltonian Cycle problem is applied).

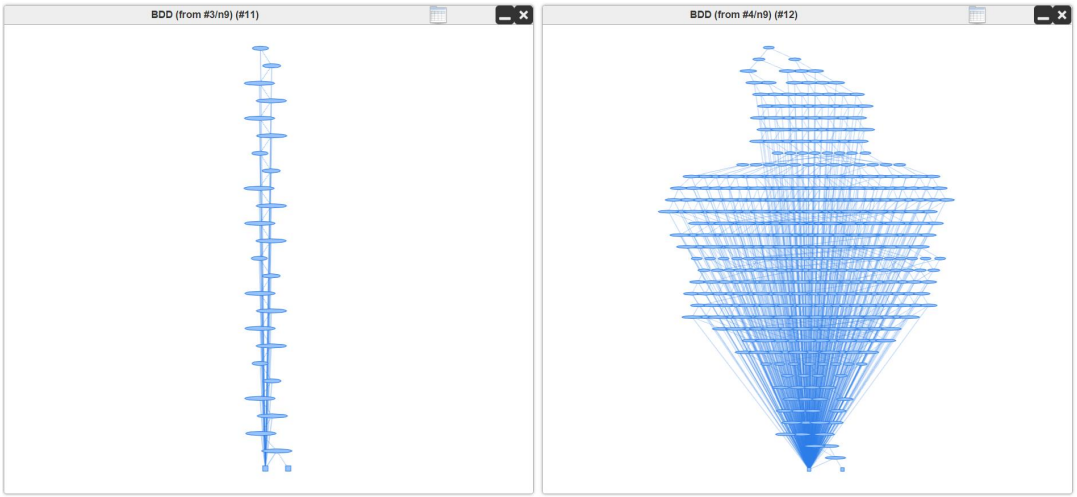


Figure 6.17: LDM vs. EDM: BDDs (node n9)

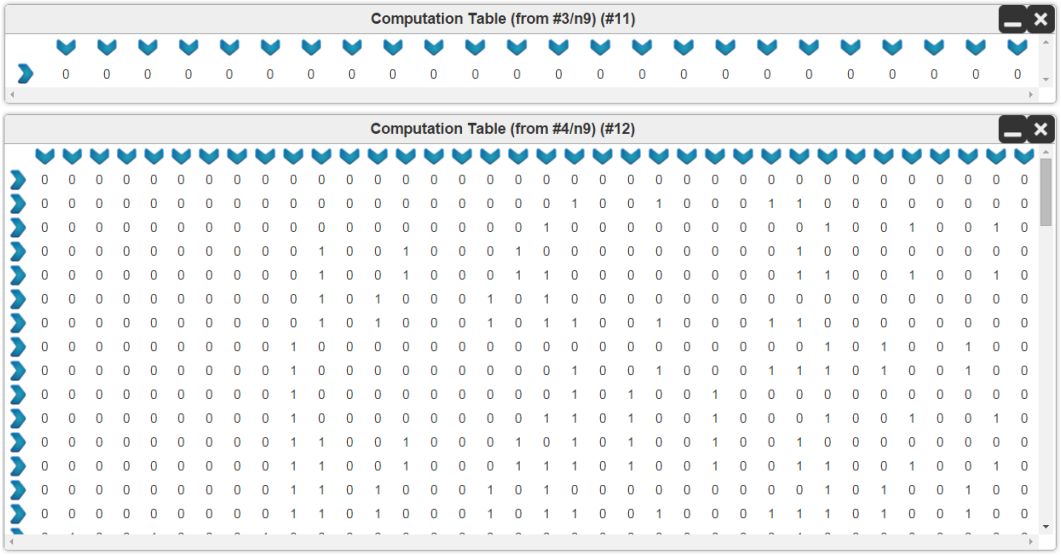


Figure 6.18: LDM vs. EDM: Computation tables (node n9)

The reason for the higher memory consumption can be found in Figure 6.16 and Figure 6.17. Via *DecoVis* we draw the following conclusion: The BDDs computed for each tree decomposition node contain in the EDM case much more variables than in the LDM case, i.e. the BDD consists of considerably more nodes and edges. Hence, in this computation, EDM-BDDs represent much more models than LDM-BDDs. The proof for this statement can be found in Figure 6.18 where the EDM-BDD represents 87 models

and the LDM-BDD just represents 1 model.

6.5.3 LDM / EDM - Satisfiability of Instances

Description

Let us focus on the analysis of LDM and EDM regarding performance of solving problems. For this purpose the analysis of a benchmark test, consisting of serveral instances, gives us information about the runtime of each decision method. Furthermore we will deal with the question: What is the reason for similar or different performances when comparing LDM and EDM?

The details and parameters for the benchmark test are defined in the following:

FIXED:
Problem: 3−Colorability
Seed: 0
Normalization: NORMALIZED
Ordering: Instance
Nodes: 27

VARIABLE:
Instance: random graph, edge−probability: 1...20
DM: EDM vs. LDM
Measurement: Runtime

Each run in the benchmark test checks the 3-Colorability of an instance with the help of the EDM algorithm and the LDM algorithm. Furthermore, the benchmark test consists of several random graph instances, where all instance graphs consist of 27 vertices and differ in their number of edges. In order to make the problem solving more complex from run to run, the amount of edges is increased in each run. The edge amount is calculated by the edge-probability, which represents the probability whether an edge exists between pairs of vertices.

Instance Selection

Figure 6.19 represents the results regarding the runtime for solving the problem with its instances defined above. Each line in the line chart stands for a decision method used for solving the problem. The x-axis defines the probability (in percent) whether an edge exists between pairs of vertices and the y-axis defines the required time (in seconds) for solving the problem.

The instance graphs from the benchmark test where the edge-probability is between 1% and 8% are satisfiable, instances with an edge-probability greater than 8% are unsatisfiable. As one can see in Figure 6.19 the runtime stays quite similar as long as the problem with its instance is satisfiable. From the moment when the runs deliver unsatisfiable as result, the runtime for EDM remains fairly constant in spite of more complex instances. In contrast to the runtime of EDM, LDM entails a longer runtime in

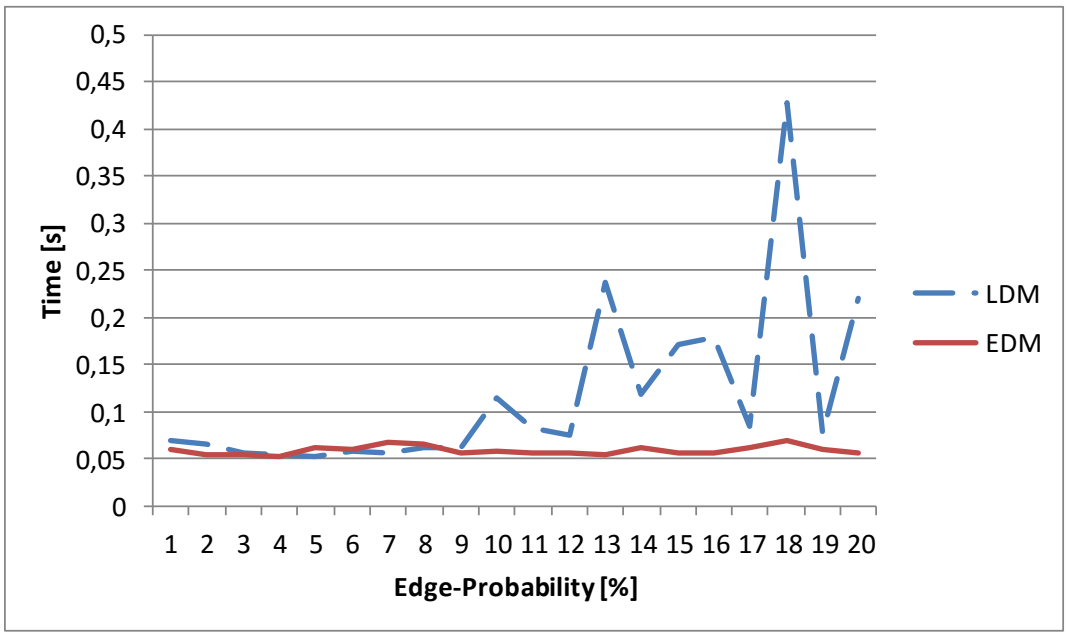


Figure 6.19: EDM vs. LDM: UNSATISFIABLE if probability > 8%

case of unsatisfiability. Furthermore, there is a strong trend that the runtime of LDM increases with the density of edges in the instance.

Analysis with DecoVis

The following explanations try to give an answer to the question mentioned in the beginning: What is the reason for similar or different performances when comparing LDM and EDM? The answer is supported by debugging specific runs with the help of *DecoVis*. Since *DecoVis* provides features for comparing metadata information of computations, particular computations of specific runs from the benchmark test are picked out and compared.

The instance with the edge-probability 8% is used as the basis for the two tree decompositions in Figure 6.20 that illustrates the comparison of the computations generated with the LDM algorithm and the EDM algorithm. Each tree represents the tree decomposition generated from the instance. The metadata value of the particular computation is emphasized in each tree decomposition node. In Figure 6.20 the metadata **elapsed-time-in-ms** is used for the comparison. The higher the corresponding **elapsed-time-in-ms** value in a tree decomposition node, the bigger is the node shape.

Figure 6.21 is used for comparing another two computations with the help of their metadata information. Here the instance with edge-probability 18% is used.

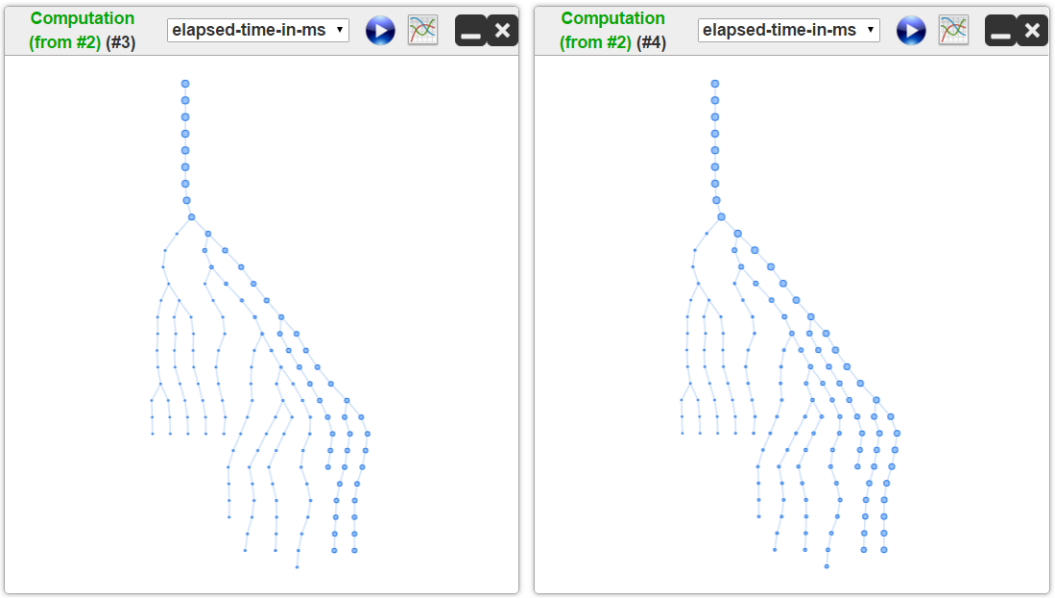


Figure 6.20: LDM vs. EDM (8%): Elapsed time

Conclusion

Figure 6.20 consists of computations resulting when solving a satisfiable problem by means of LDM and EDM. There is no significant difference when comparing the LDM and EDM computations. Hence, we can conclude that if a 3-Colorability problem (with similar execution parameters for *dynBDD*) is satisfiable and there is no recognizable difference between the LDM and EDM computation when the computations are compared by means of *DecoVis*, the runtime for solving this problem is in both cases similar.

In contrast to Figure 6.20, the computations in Figure 6.21 are based on an unsatisfiable instance. As one can see in the LDM computation, the elapsed time (in *DecoVis* illustrated by **elapsed-time-in-ms**) increases when the tree is traversed post-order. Only the last few nodes near to the root node are not computed, i.e. they are not any more considered because of unsatisfiability. As one can see in the EDM computation, the algorithm aborts the process after only a few nodes. This leads to the conclusion that EDM has a better runtime than LDM when solving unsatisfiable problems, because unsatisfiability is recognized at a very early stage and the solving process is, as a consequence, much earlier aborted.

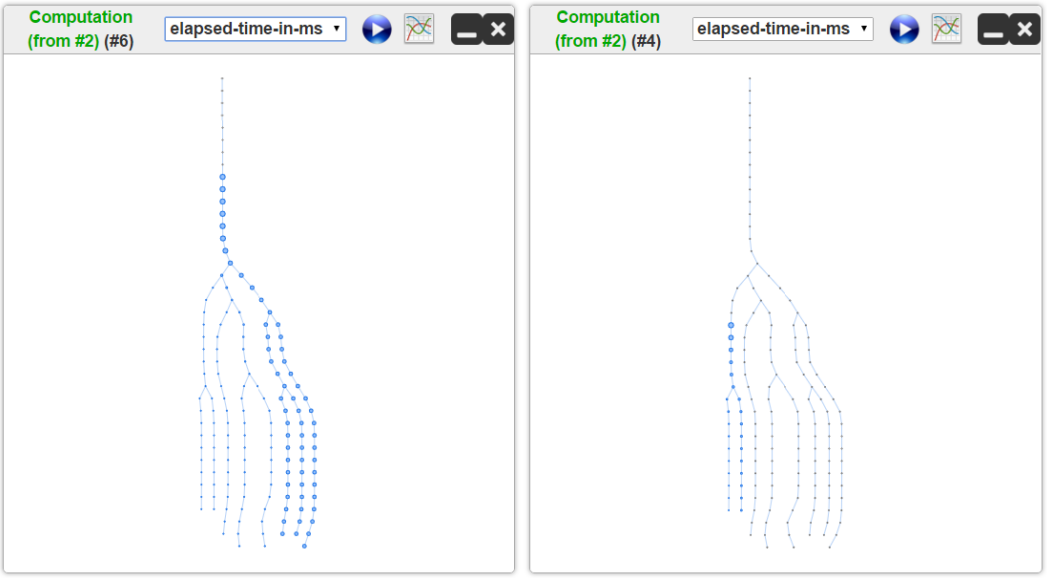


Figure 6.21: LDM vs. EDM (18%): Elapsed time

6.5.4 Variable Order - Impact on Computations

Description

The following analysis focuses on various variable orders of systems using BDD-based Dynamic Programming on Tree Decompositions. Furthermore, different variable order types used to solve a real-world problem are examined. Mainly, the following question is discussed with the help of a benchmark test and *DecoVis*: What is the impact on computations if different variable orders are used for real-world graphs?

FIXED:
Instance: Vienna metro
Problem: 3−Colorability
Normalization: NORMALIZED
Decision−Method: EDM
Nodes: 93

VARIABLE:
Seed: 1..40
Ordering: instance, lexicographic, td−first, degree−high, degree−low
Measurement: Memory

We use the Vienna metro map without station name specification as instance for the benchmark test. The benchmark test gives us information about the memory consumption of *dynBDD* when the 3-Colorability problem is solved by means of different variable

order types and various seed values. The result of the benchmark test helps us in further consequence to compare the computations generated with different variable order types.

Instance Selection

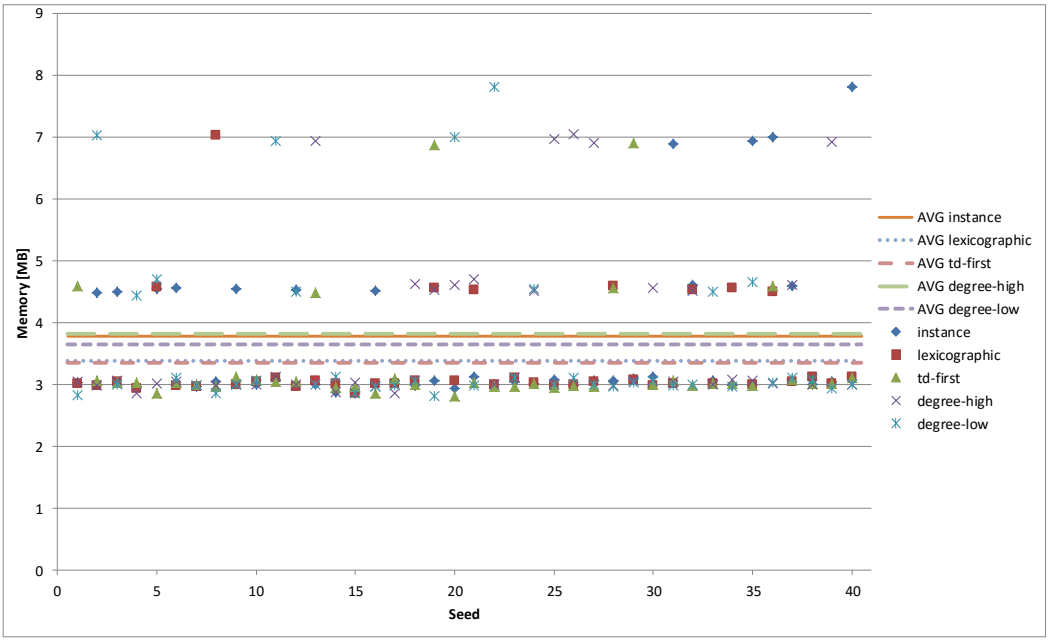


Figure 6.22: Variable ordering: Statistics

The diagram in Figure 6.22 illustrates the used memory consumption of the benchmark test runs applying different seed values and variable orders. As one can see, the average memory consumptions of the variable orders do not differ significantly when various seeds are used as basis for the benchmark test. Since the 3-Colorability problem of the Vienna metro map can be solved in less than 0.1 seconds by means of *dynBDD*, a repeating of the same benchmark test always results in different memory consumption values. Hence, it is easy to see, that no conclusions about "which variable order is best used for real-world problems" can be drawn.

Although no trend is obvious, we select the computations (instance, lexicographic, td-first, degree-high, degree-low) solved with seed 10 from the diagram in Figure 6.22, because they use nearly the same memory (3 MB). Furthermore, they provide an appropriate basis to analyze the impact of different variable orders on computations with *DecoVis*.

Analysis with DecoVis

Figure 6.23 represents a statistic report of computations solved with the variable orders **instance (2)**, **lexicographic (3)**, **td-first (4)**, **degree-high (5)** and **degree-low (6)**. The statistic report illustrates the **number-of-DD-nodes** used for each tree decompositions node. The x-axis in the line chart represents the tree decompositions node in computation oder (post order).

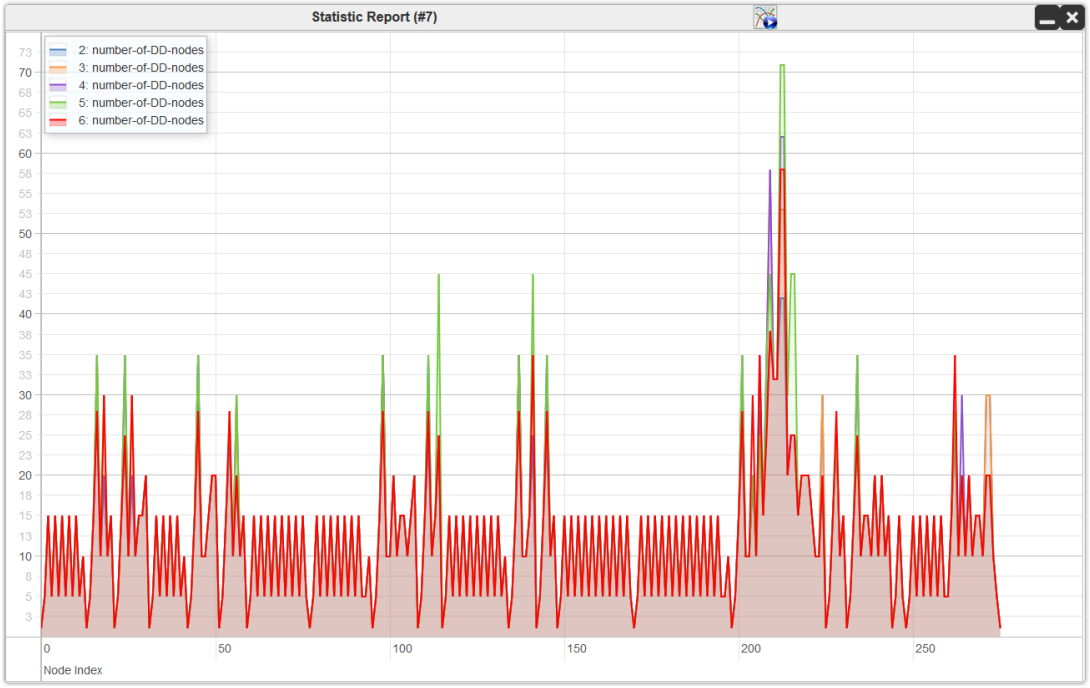


Figure 6.23: Statistic report - number-of-DD-nodes

As one can see in Figure 6.23, the five approaches are quite similar in the number of BDD-nodes. This could be one possible explanation for the similar memory consumption values in Figure 6.22. One visible difference between the variable orders can be found between index 200 and 230. Since tree decomposition node **159** lies between the two values, each BDD (instance, lexicographic, td-first, degree-high, degree-low) of node **159** is used for further analysis.

In order to compare the BDDs of tree decomposition node **159**, Figure 6.24 illustrates in window **Statistic Report (#8)** a zoomed view of the statistic report in Figure 6.23. As one can see, each BDD consists of a different number of BDD nodes. With the help of this view it is easy to order (ascending) the five approaches regarding number of BDD nodes: **instance (2)**, **lexicographic (3)**, **degree-low (6)**, **td-first (4)** and **degree-high (5)**.

Figure 6.24 furthermore represents the BDDs of tree decomposition node **159** gen-

erated with each of the five variable orders. In general, tree decomposition node **159** contains four vertices in its bag: **5**, **6**, **33** and **48**. In order to see the influence of different variable orders in various BDDs, the BDD nodes where **5** occurs are marked. As a consequence, the different placements of **5** in different levels are emphasized.

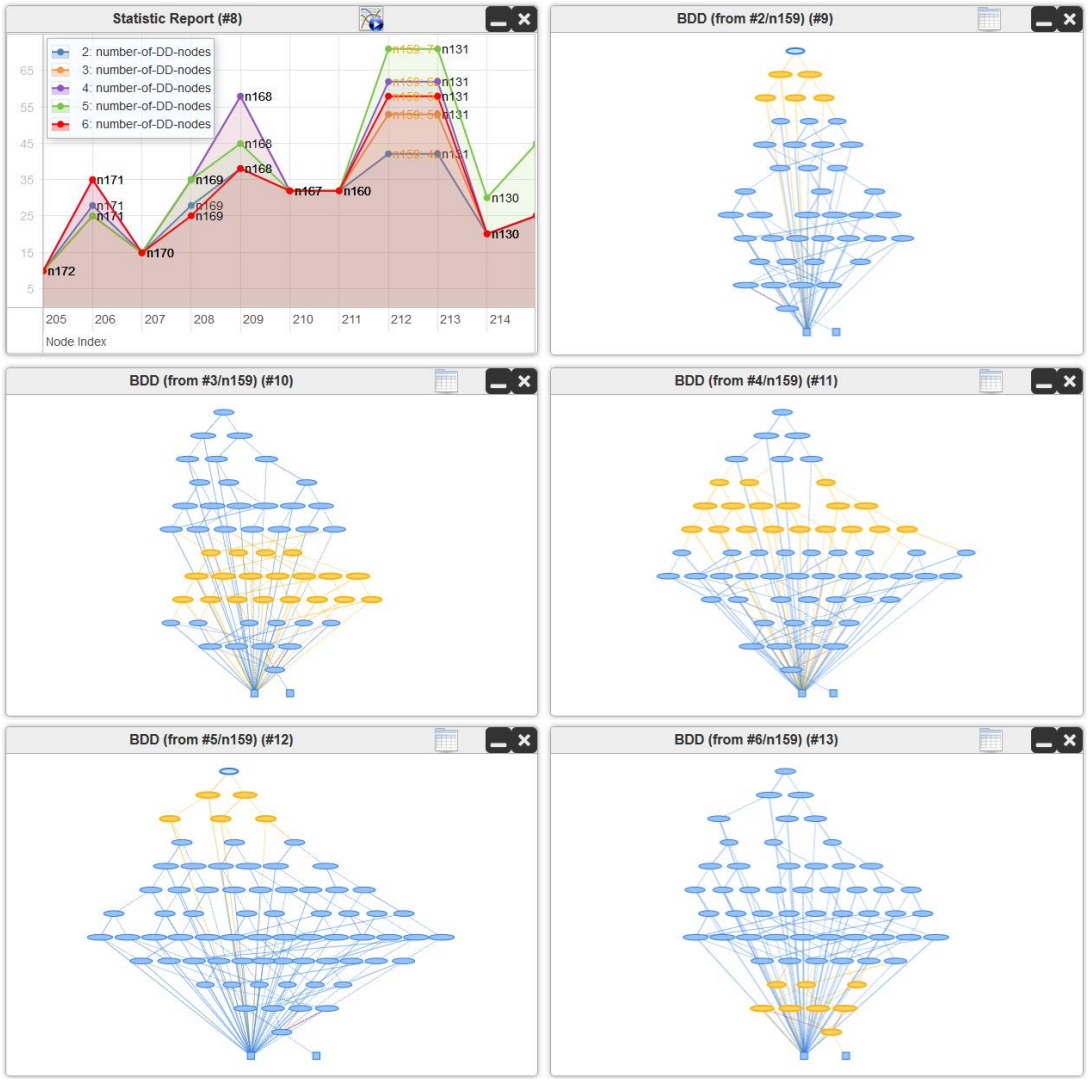


Figure 6.24: BDDs - node 5 is selected

Conclusion

By means of the benchmark test analysis above it is not possible to give general statements about the performance of particular variable orders. As Figure 6.24 shows, a well-chosen variable order has a positive effect even on BDDs consisting of few vertices. It is easy to

see, that the BDD from the second window (**BDD (from #2/n159) (#9)**) computed with instance order is composed of much less BDD nodes than the other BDDs in Figure 6.24.

6.5.5 Variable Order - Impact on Join Nodes

Description

Now we focus on the analysis of BDD evolution in the *dynBDD* computation process when tree decompositions are solved. First, we examine the performance of various variable orders on the basis of several instances with the same node count, but different edge placement, in form of a benchmark test. Next, a representative instance from the benchmark test is chosen for further analysis. In order to compare the join nodes of two computations with different variable orders, the animation feature of *DecoVis* is used to discuss the following question: What is the impact on join nodes when different variable orders are used?

The following settings are used for the benchmark test:

FIXED:
Problem: Hamiltonian Cycle
Seed: 0
Normalization: NORMALIZED
Decision−Method: LDM
Nodes: 10

VARIABLE:
Instance: 20 instances with edge−probability 15%
Ordering: instance, lexicographic, td−first, degree−high, degree−low
Measurement: Memory

Each run in the benchmark test checks the existence of Hamiltonian Cycles of an instance with the help of different variable orders. The benchmark test consists of several random graph instances, where each instance graph consists of 10 vertices. In order to receive a statement about the performance of each variable order type, the hardness of the instances should remain about the same from run to run. Hence, the edges of the instances are generated with an edge-probability of 15 %, i.e. an edge exists between pairs of vertices with probability 15 %.

Instance Selection

We can see in Figure 6.25, that on average each variable order requires pretty much the same memory for solving the Hamiltonian Cycle problem for instances with 10 vertices and an edge probability of 15 %. Nevertheless, **td-first** variable order requires the lowest memory on average in the benchmark test, whereas **instance** variable order uses the most memory on average. This illustration of the benchmark test by means of the diagram helps to select in further consequence representative (with a high difference regarding

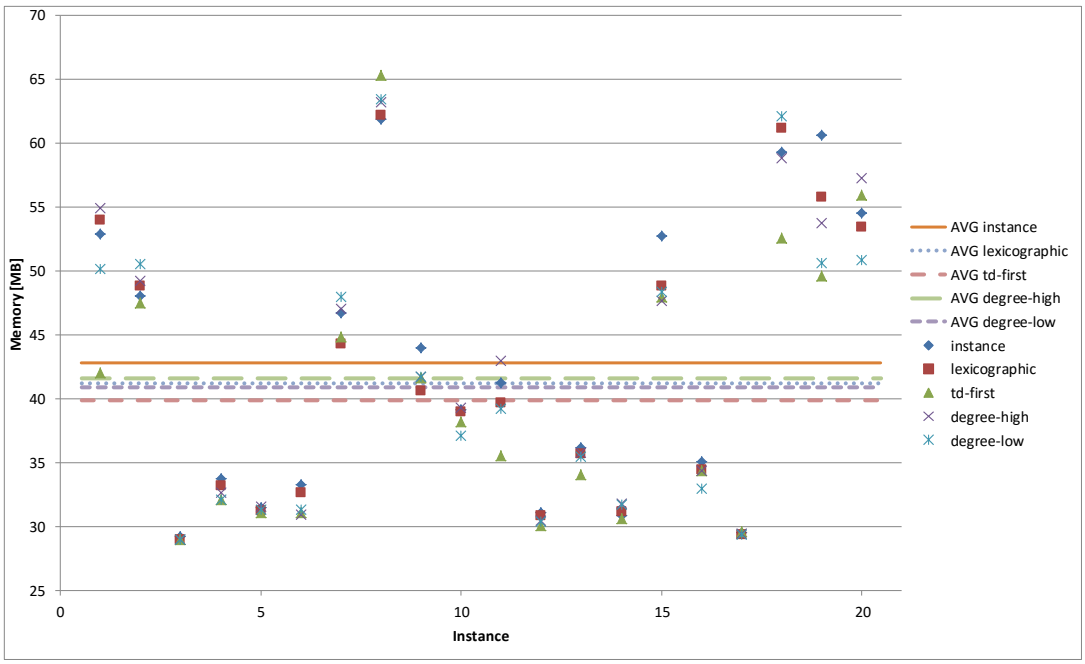


Figure 6.25: Hamiltonian Cycle: Statistics

memory usage) computations. Since the **td-first** variable order performs much better than the other variable order types when instance **1** is used as basis, it is used for further analysis.

Analysis with DecoVis

Figure 6.26 illustrates instance **1** from the benchmark test. Furthermore, the vertices **0**, **1**, **2**, **7**, **8** and **9** included in the bags of the tree decomposition nodes that are used for further analysis are emphasized.

The first window (**Tree Decomposition - normalized (#2)**) in Figure 6.27 represents the normalized tree decomposition of instance **1** in *DecoVis*. Furthermore, the tree decomposition nodes with the bag elements **0**, **1**, **2**, **7**, **8** and **9** are selected. Two of the five emphasized node represent join nodes.

The window in the center (**Statistic Report (#8)**) illustrates a statistic report about the **number-of-DD-nodes** used for each tree decomposition node in the computation process. The right window (**Statistic Report (#9)**) visualizes the zoomed view of the statistics between index 20 and 35 from the **Statistic Report (#8)**.

We can observe with the help of the instance graph in Figure 6.26 and the reports in Figure 6.27, that despite the low number of vertices, the instance is more difficult to solve due to the high edge density than e.g. the Vienna metro map. Furthermore, the statistic report shows that the **td-first** variable order requires the least number of BDD nodes.

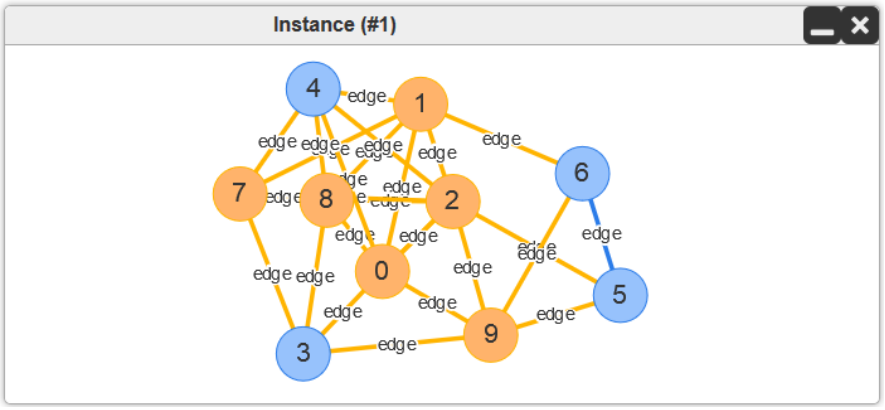


Figure 6.26: Instance 1 with 15% edge-probability

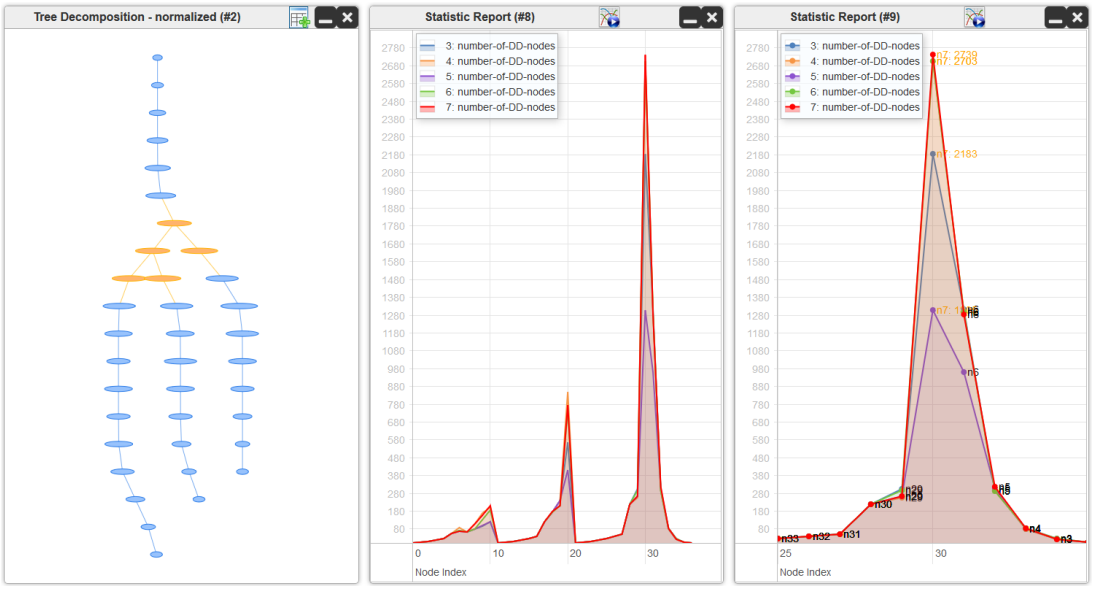


Figure 6.27: Number-of-DD-nodes: instance(3) vs. lexicographic(4) vs. td-first(5) vs. degree-high(6) vs. degree-low(7)

We use the animation feature from *DecoVis* in Figure 6.28 to analyze the evolution of the BDDs generated to each join node of the tree decomposition. Therefore only three levels of the tree decomposition are animated in Figure 6.28. Since **td-first** variable order requires the lowest number of BDD nodes in the statistic report and **degree-high** variable order requires the highest number of BDD nodes, the computations of these two variable orders are compared in the Figure 6.28. The first column in Figure 6.28

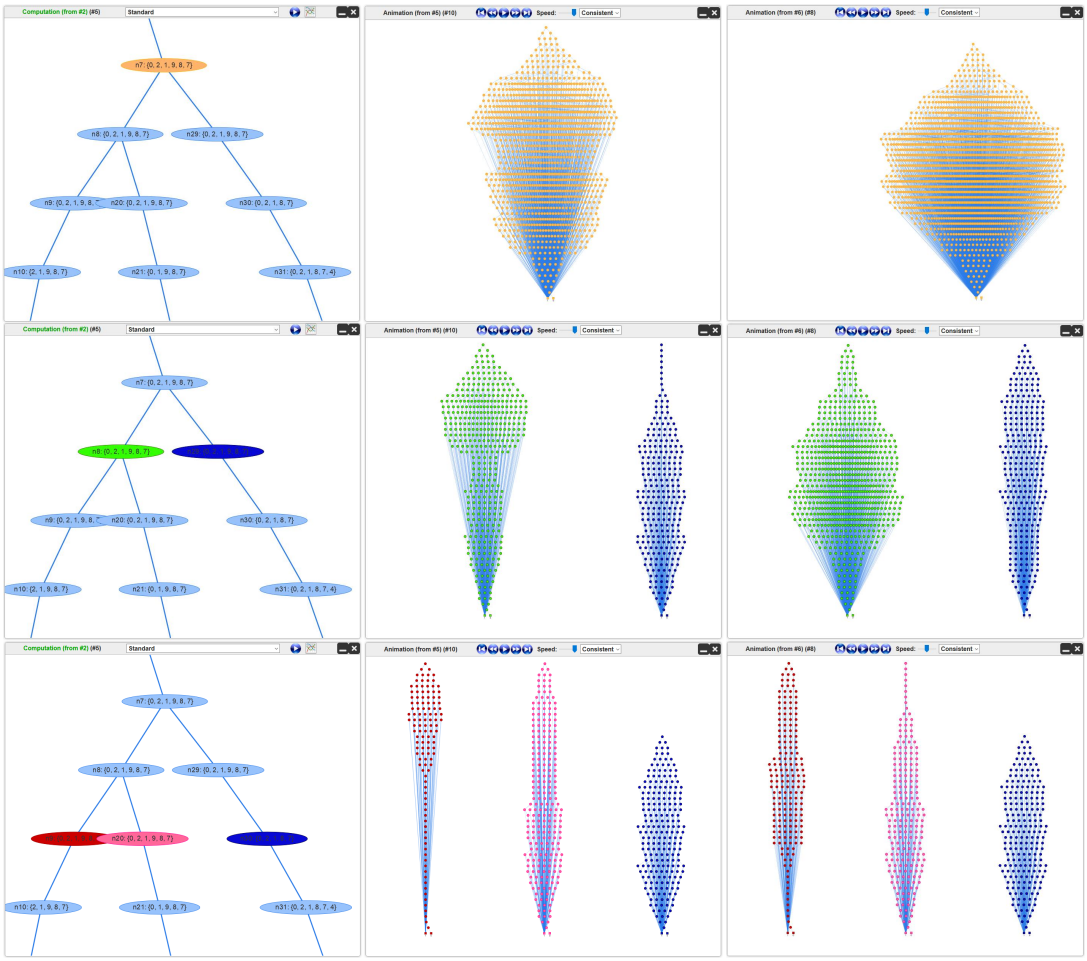


Figure 6.28: BDD animation of td-first(5) and degree-high(6)

represents a zoomed view of the tree decomposition, where the currently animated tree decomposition nodes are emphasized. The window in the center and in the right window represent the corresponding BDDs to the selected tree decomposition nodes in **td-first** variable order and **degree-high** variable order.

Conclusion

One possible conclusion regarding the benchmark test above and the statistic reports in *DecoVis* is, that variable orders with a much lower number of BDD nodes on average (**td-first** in Figure 6.27) imply lower memory consumption (as we see in Figure 6.25).

The impact of different variable orders on join nodes can easily be visualized with the help of *DecoVis*. As we see in Figure 6.28 the merging of two BDDs with a high number of nodes also result in a BDD with a high number of nodes and edges, that represents

the solutions of all child nodes. A consequence of this recognition is, that a well-chosen variable order has a positive impact on the corresponding BDD of a join node.

6.6 Summary

DecoVis supports the analysis of several components regarding dynamic programming on tree decompositions and helps to discuss multiple interesting questions regarding computations. Instance analysis in *DecoVis* deals with the following issues:

- Visualize graph instances representing the structure of graphs (e.g. Vienna metro map, grid-based instances, random instances with edge probability, clique-based instances, etc.).
- Particular characteristics of instances can be emphasized.

Furthermore, *DecoVis* provides several possibilities for the analysis of tree decompositions. As a consequence, various interesting issues about tree decompositions can be analyzed with the help of *DecoVis*:

- Tree decompositions of instances are visualized.
- Occurrences of particular vertices from the instance graph are emphasized in tree decompositions in different normalization types (none, weak, semi and normalized).
- Various characteristics of each normalization type are illustrated.

By means of *DecoVis* several interesting questions regarding computations of solved problems can be discussed and analyzed. The following paragraphs contain on the one hand some of these interesting questions and on the other hand attempts to answer the questions by means of *DecoVis*:

How do the decision methods LDM and EDM differ in the manipulation of the BDDs during the computation?

BDDs only change during the LDM computation if a vertex gets removed from the bag of a tree decomposition node. In contrast to LDM, EDM always generates new BDDs consisting of all vertices from the bag of the corresponding tree decomposition node. These observations are nicely shown by means of *DecoVis*.

How and why do instances with high edge density differ regarding LDM and EDM?

By means of a benchmark test we demonstrated that cliques computed with EDM need much more memory than the corresponding LDM computation. Due to the benchmark

test result, regarding solving the Hamiltonian Cycle problem of several cliques with EDM and LDM, we have a reason to believe that solving an instance graph with a high edge density by means of EDM results in a much higher memory consumption than solving the same instance graphs with the help of LDM. The reason for the higher memory consumption can be appropriately illustrated by means of *DecoVis*: The BDDs computed for each tree decomposition node contain in the EDM case considerably more nodes and edges than in the LDM case.

What is the reason for similar or different performances when comparing LDM and EDM?

With the help of *DecoVis* it turned out that conclusions about the performance of EDM and LDM can only be drawn when unsatisfiable instances are considered. The EDM computation analysis in *DecoVis* illustrated that the computation algorithm aborts the process after only a few nodes. This leads to the conclusion that EDM has a better runtime than LDM when solving unsatisfiable problems, because unsatisfiability is recognized at a very early stage and the solving process is much earlier aborted.

What is the impact on computations if different variable orders are used in real-world graphs?

We saw that no general statements about the performance of particular variable orders can be made with the help of our benchmark tests and the analysis via *DecoVis* when problems of real-world graphs are solved. Nevertheless, *DecoVis* nicely illustrates that a well-chosen variable order has a positive effect on the construction of BDDs.

What is the impact on join nodes when different variable orders are used?

The impact of different variable orders on join nodes can easily be analyzed with the help of *DecoVis*. The animation feature in *DecoVis* visualizes nicely the merging of two BDDs into a join node. Since the join node has to include all solutions of the child nodes, the construction of the join node mostly results in a BDD with more nodes and edges than its child BDD nodes. We can conclude again that a well-chosen variable order has also a positive impact on the corresponding BDD of a join node.

CHAPTER 7

Summary and Future Work

This chapter gives a short summary about the approach for developing a tool for the analysis of systems using dynamic programming on tree decompositions for solving complex problems. Additionally, we look back to the applications of our implemented tool *DecoVis*. In order to improve existing analysis features and support further analysis possibilities, this chapter furthermore deals with ideas for future work.

7.1 Summary

In general, this thesis puts the focus on the development of a tool for analyzing systems using BDD-based dynamic programming on tree decompositions for solving complex problems. In order to understand the concepts and approaches of such BDD-based dynamic programming systems, we first explained the concept of dynamic programming on tree decompositions in detail and then introduced Binary Decision Diagrams (BDDs). In addition to the description of already existing dynamic programming systems, we examined available visualization concepts (graph libraries, frameworks, etc.) in order to develop an analysis tool for such systems.

After having specified the technical requirements via UML diagrams, we implemented our system in *DecoVis*. Beside the visualization of instances, tree decompositions and computations, *DecoVis* provides several possibilities for analyzing dynamic programming systems. Furthermore, a technical overview, all features and two different modes were explained in detail.

In order to present differences of various systems applying dynamic programming on tree decompositions regarding performance and stability, several benchmark tests were generated and the results analyzed in further consequence. For this purpose, the dynamic programming systems *dynBDD*, *D-FLAT*, *D-FLAT^2* and *Sequoia* were compared in form of black-box-systems. We executed and analyzed different benchmark tests covering the problems 3-Colorability, Stable Extension, Hamiltonian Cycle and Preferred Extensions.

To examine the usage of *DecoVis* in practice, we analyzed different behaviors of various options and settings of systems applying BDD-based dynamic programming on tree decompositions. Emphasis of particular characteristics of instances supported the analysis of specific instances. The visualization of various characteristics of normalization types and emphasis of the occurrences of particular vertices from the instance graph represented the main issues when tree decomposition were analyzed. Based on specific instances that are selected from the results of benchmark tests, different decision methods (LDM and EDM) and variable orders (instance, lexicographic, td-first, degree-high and degree-low) were compared and analyzed in detail by means of *DecoVis*.

As a result, one of the main contributions of this master thesis is the development of a completely new system for the analysis of dynamic programming to understand, compare and improve the involved algorithms. Such a tool was not available yet. Another contribution is that we implement this system as a web-based tool, which has the big advantage that the concept and basic understanding of *Dynamic Programming on Tree Decompositions* can be made available to a lot of people very easily. Last not but least, we compared different tools applying DP on TD by means of benchmark tests. While doing that we pointed out the strengths and weaknesses of these systems. Furthermore the different configurations and settings for BDD-based DP on TD were investigated. All these results can now be used for the improvement and further development.

The *DecoVis* software can be found at:
`http://decovis.dbai.tuwien.ac.at`

7.2 Future Work

In the course of the *DecoVis* implementation, the comparison of various systems applying dynamic programming on tree decompositions and the analysis of *DecoVis* in practice, we identified great potential for future work. Some improvements regarding *DecoVis* are enumerated in the following:

Standalone tool for large instances and computations Some interesting instances and computations were generated during the comparison of different dynamic programming systems and the analysis of *dynBDD* by means of *DecoVis* that could not be visualized with *DecoVis*. The reason was that the graph library and the browser cannot deal with huge input graphs. Hence, we concluded that a browser is not the right place to compute and visualize such complex information. The main problem of web browsers is, that they are usually single-threaded. One possible solution to use full processor power is to visualize the information in a standalone tool. The disadvantage of this approach is, that the conveniences of web applications would not be provided any more.

Session-scoped web-application In order to shift several processor intensive functions and data storages from the browser to the web-server, the web-application has to

be adapted with the session-scoped feature. This extension leads to the availability of client specific data on the web-server. One positive effect and example would be that large BDDs would not be loaded from the web-server until their selection in the graphical user interface.

File-refactoring of instances, tree decompositions and computations Instances, tree decompositions and computations are currently stored in GraphML-format (XML). To save time for the transformation into JSON format as used in the visualization library, all components could be stored in JSON-format. Furthermore, the way of representing BDDs of tree decomposition nodes in computation files can be optimized. Since identical BDDs (with exactly the same BDD nodes and edges) are often stored redundantly, BDDs could be listed after the tree decomposition nodes in the file, i.e. each tree decomposition node contains a reference to the corresponding BDD.

Animate multiple computations concurrently To compare two computations with the help of the current *DecoVis* version, computations have to be animated sequentially. Saved animation steps (can be downloaded in form of PNGs) can be compared afterwards. The parallel animation of several computations is desirable, where multiple computations have to be preselected. Afterwards, the computations are animated concurrently in multiple windows.

Instance generation and manipulation We often need to analyze several computations with instances as basis, that only differ minimally, i.e. one vertex or edge is added or removed in contrast to the previous instance. Furthermore, a possibility for automatically generating well-known graphs such as grid-based graphs, cliques or graphs with fix vertex count and edge probability would be required. Hence, *DecoVis* could be enhanced with instance generation and instance manipulation.

Support of several systems applying dynamic programming on tree decompositions The current *DecoVis* version provides the dynamic selection of a tool and its execution parameters and options with the help of a property file. Since further systems applying dynamic programming on tree decompositions can implement the generation of instances, tree decompositions and computations fitting to the interface of *DecoVis*, the selection between several integrated systems could be supported.

Support of further data structures *DecoVis* could support the interpretation and visualization of further data structures (e.g. “Item trees”).

Benchmark-Environment Since most of the analyzed components in *DecoVis* are obtained and chosen in the course of benchmark tests, a benchmark-environment could be embedded into *DecoVis*. Programs to be benchmarked, used options and parameters, instances, time of execution, iterations, used servers and much more could be controlled

by means of *DecoVis*.

Some future work may also include more benchmark tests, for example benchmarks having a focus on other problems, more instances and other metadata types for comparison. Also new approaches and algorithms including different configurations and settings could be investigated by using the *DecoVis* system.

Bibliography

[1] Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[2] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[3] Günther Charwat and Stefan Woltran. Efficient problem solving on tree decompositions using binary decision diagrams. In Francesco Calimeri, Giovambattista Ianni, and Miroslaw Truszczynski, editors, *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2015)*, volume 9345 of *Lecture Notes in Computer Science*, pages 213–227. Springer, 2015.

[4] Joachim Kneis, Alexander Langer, and Peter Rossmanith. Courcelle's theorem - A game-theoretic approach. *Discrete Optimization*, 8(4):568–594, 2011.

[5] Alexander Langer, Felix Reidl, Peter Rossmanith, and Somnath Sikdar. Evaluation of an MSO-solver. In David A. Bader and Petra Mutzel, editors, *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments (ALENEX 2012)*, pages 55–63. SIAM / Omnipress, 2012.

[6] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. The D-FLAT system for dynamic programming on tree decompositions. In Eduardo Fermé and João Leite, editors, *Proceedings of the 14th European Conference on Logics in Artificial Intelligence (JELIA 2014)*, volume 8761 of *Lecture Notes in Computer Science*, pages 558–572. Springer, 2014.

[7] D-FLAT system. `http://www.dbai.tuwien.ac.at/proj/dflat/system/`. [Online; accessed 24-July-2015].

[8] Bernhard Bliem, Günther Charwat, Markus Hecher, and Stefan Woltran. D-FLAT^2: Subset minimization in dynamic programming on tree decompositions made easy. In *Proceedings of the 8th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2015)*, 2015.

[9] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–21, 1993.

[10] Hans L. Bodlaender. Discovering treewidth. In Peter Vojtás, Mária Bieliková, Bernadette Charron-Bost, and Ondrej Sýkora, editors, *Proceedings of the 31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2005)*, volume 3381 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

[11] Markus Aschinger, Conrad Drescher, Georg Gottlob, Peter Jeavons, and Evgenij Thorstensen. Structural decomposition methods and what they are good for. In Thomas Schwentick and Christoph Dürr, editors, *Proceedings of the 28th International Symposium on Theoretical Aspects of Computer Science (STACS 2011)*, volume 9 of *LIPIcs*, pages 12–28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[12] Martin Grohe. Descriptive and parameterized complexity. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Proceedings of the 13th International Workshop on Computer Science Logic (CSL 1999) and the 8th Annual Conference on European Association for Computer Science Logic (EACSL 1999)*, volume 1683 of *Lecture Notes in Computer Science*, pages 14–31. Springer, 1999.

[13] Thomas Hammerl, Nysret Musliu, and Werner Schafhauser. Metaheuristic algorithms and tree decomposition. In Janusz Kacprzyk and Witold Pedrycz, editors, *Springer Handbook of Computational Intelligence*, pages 1255–1270. Springer, 2015.

[14] Marko Čepin. Binary decision diagram. In *Assessment of Power System Reliability*, pages 101–112. Springer, 2011.

[15] Sheldon B Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 100(6):509–516, 1978.

[16] Henrik Reif Andersen. An introduction to binary decision diagrams. *Lecture notes, IT University of Copenhagen*, 1997.

[17] Randal E Bryant and Christoph Meinel. *Ordered binary decision diagrams*. Springer, 2002.

[18] Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[19] Joost-Pieter Katoen. Reduced ordered binary decision diagrams. *RWTH AACHEN, Lecture #13 of Advanced Model Checking*, 2010.

[20] Ales Casar, Robert Meolic, Z Brezocnik, and B Horvat. Representation of boolean functions with ROBDDs. *Clanek sprejet za objavo*, 1993:94, 1992.

[21] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. D-FLAT: Progress report. Technical report, DBAI-TR-2014-86, Vienna University of Technology, 2014.

[22] Piero A. Bonatti, Francesco Calimeri, Nicola Leone, and Francesco Ricca. Answer set programming. In Agostino Dovier and Enrico Pontelli, editors, *A 25-Year Perspective on Logic Programming: Achievements of the Italian Association for Logic Programming (GULP 2010)*, volume 6125 of *Lecture Notes in Computer Science*, pages 159–182. Springer, 2010.

[23] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo: A new grounder for answer set programming. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007)*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2007.

[24] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp*: A conflict-driven answer set solver. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007)*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.

[25] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Benjamin J. McMahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In Alexander F. Gelbukh and Eduardo F. Morales, editors, *Proceedings of the 7th Mexican International Conference on Artificial Intelligence (MICAI 2008)*, volume 5317 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2008.

[26] Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran. ASPARTIX: implementing argumentation frameworks using answer-set programming. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *Lecture Notes in Computer Science*, pages 734–738. Springer, 2008.

[27] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial intelligence*, 77(2):321–357, 1995.

[28] Sequoia: A fast EMSO-solver for graphs of small treewidth. `http://sequoia.informatik.rwth-aachen.de/sequoia/`. [Online; accessed 24-July-2015].

[29] Fabio Somenzi. CUDD: CU decision diagram package-release 2.4.0. *University of Colorado at Boulder*, 2009.

[30] Wolfgang Dvorák, Michael Morak, Clemens Nopp, and Stefan Woltran. dynPARTIX - A dynamic programming reasoner for abstract argumentation. In Hans Tompits, Salvador Abreu, Johannes Oetsch, Jörg Pührer, Dietmar Seipel, Masanobu Umeda, and Armin Wolf, editors, *Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011) and 25th Workshop on Logic Programming (WLP 2011)*, volume 7773 of *Lecture Notes in Computer Science*, pages 259–268. Springer, 2011.

[31] Robert W Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics (TOG)*, 5(2):79–109, 1986.

[32] Valerie Quercia and Tim O'Reilly. *Volume Three: X Window System User's Guide*. O'Reilly & Associates, 1990.

[33] Phillip Beaudet, Eduardus AT Merks, Martin Rendall, and Roger Spall. Window management system with a hierarchical iconic array and miniature windows, February 13 1996. US Patent 5,491,795.

[34] Eric Masselle and Patrick McGowan. Method of window management for a windowing system, August 19 2004. US Patent App. 10/922,189.

[35] jQuery User Interface. `https://jqueryui.com/`. [Online; accessed 14-September-2015].

[36] jQuery. `https://jquery.com/`. [Online; accessed 14-September-2015].

[37] gridster.js. `http://gridster.net/`. [Online; accessed 14-September-2015].

[38] OS.js - JavaScript Web/Cloud Desktop Platform. `http://os.js.org/`. [Online; accessed 14-September-2015].

[39] Ventus: HTML5 & CSS3 Window Manager Experiment. `http://www.rlamana.es/ventus/`. [Online; accessed 14-September-2015].

[40] JUNG - Java Universal Network/Graph Framework. `http://jung.sourceforge.net/`. [Online; accessed 14-September-2015].

[41] JGraphT. `http://jgrapht.org/`. [Online; accessed 14-September-2015].

[42] JGraph. `http://www.jgraph.com/`. [Online; accessed 14-September-2015].

[43] GraphStream - A dynamic graph library. `http://graphstream-project.org/`. [Online; accessed 14-September-2015].

[44] arbor.js. `http://arborjs.org/`. [Online; accessed 14-September-2015].

[45] sigmajs. `http://sigmajs.org/`. [Online; accessed 14-September-2015].

[46] Cytoscape.js. `http://js.cytoscape.org/`. [Online; accessed 14-September-2015].

[47] vis.js. `http://visjs.org/`. [Online; accessed 14-September-2015].

[48] vis.js: Network examples. `http://visjs.org/network_examples.html`. [Online; accessed 20-April-2015].

[49] vis.js: Graph3D examples. `http://visjs.org/graph3d_examples.html`. [Online; accessed 20-April-2015].

[50] vis.js: Graph2D examples. http://visjs.org/graph2d_examples.html. [Online; accessed 20-April-2015].

[51] D-FLAT Debugger. http://www.dbai.tuwien.ac.at/proj/dflat/debugger/. [Online; accessed 24-July-2015].

[52] clavis: Visualizations for clasp. http://www.cs.uni-potsdam.de/clavis/. [Online; accessed 24-July-2015].

[53] Arne König and Torsten Schaub. Monitoring and visualizing answer set solving. *TPLP*, 13(4-5-Online-Supplement), 2013.

[54] Susanne Grell, Kathrin Konczak, and Torsten Schaub. nomore$^<$: A system for computing preferred answer sets. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning, (LPNMR 2005)*, volume 3662 of *LNCS*, pages 394–398. Springer, 2005.

[55] nomore$^<$: graphs and colorings for answer set programming with preferences. http://www.cs.uni-potsdam.de/wv/nomorepref/. [Online; accessed 24-July-2015].

[56] Sebastián Escarza and Martín L. Larrea and Silvia M. Castro and Sergio R. Martig. DeLP Viewer: A defeasible logic programming visualization tool. http://www.academia.edu/1082009/DelP_Viewer_a_Defeasible_Logic_Programming_Visualization_Tool. [Online; accessed 24-July-2015].

[57] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.

[58] Yifan Hu. Efficient, high-quality force-directed graph drawing. *Mathematica Journal*, 10(1):37–71, 2005.

[59] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M. Scott Marshall. GraphML progress report. In *Graph Drawing*, pages 501–512, 2001.

[60] Pajek Network Datasets: The Graph and Digraph Glossary. http://vlado.fmf.uni-lj.si/pub/networks/data/DIC/TG/glossTG.htm. [Online; accessed 12-October-2015].

[61] Bill Cherowitzo: Graph and Digraph Glossary. http://math.ucdenver.edu/~wcherowi/courses/m4408/glossary.html. [Online; accessed 12-October-2015].

[62] Pajek Network Datasets: The Bibliography for the Book "Product graph". http://vlado.fmf.uni-lj.si/pub/networks/data/2mode/Sandi/Sandi.htm. [Online; accessed 12-October-2015].

[63] Wilfried Imrich and Sandi Klavžar. *Product graphs, structure and recognition*, volume 56. Wiley-Interscience, 2000.

[64] Vladimir Batagelj and Andrej Mrvar (2006): Pajek datasets. `http://vlado.fmf.uni-lj.si/pub/networks/data/`. [Online; accessed 12-October-2015].

[65] Kurt Mehlhorn and Stefan Näher. *LEDA: A platform for combinatorial and geometric computing*. Cambridge University Press, 1999.

[66] Sanjay Modgil and Martin Caminada. Proof theories and algorithms for abstract argumentation frameworks. In *Argumentation in artificial intelligence*, pages 105–129. Springer, 2009.

[67] J. Plesník. The NP-completeness of the Hamiltonian cycle problem in planar diagraphs with degree bound two. *Information Processing Letters*, 8(4):199–201, 1979.

[68] Norishige Chiba and Takao Nishizeki. The Hamiltonian cycle problem is linear-time solvable for 4-connected planar graphs. *Journal of Algorithms*, 10(2):187–211, 1989.

[69] Yannis Dimopoulos, Bernhard Nebel, and Francesca Toni. Finding admissible and preferred arguments can be very hard. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR 2000)*, pages 53–61. Morgan Kaufmann, 2000.