FAKULTÄT FÜR !NFORMATIK

M.Sc. Arbeit

Generic Programing for Graph Problems Using Tree Decompositions

ausgeführt am

Institut für Informationssysteme
Arbeitsgruppe Datenbanken und Artificial Intelligence
der Technischen Universität Wien

unter der Anleitung von

Priv.-Doz. Dr. Nysret Musliu
Dr. Fang Wei

durch

Emmanouil Paisios

Wien, 23. Juli 2008

.....	.....
Emmanouil Paisios	Nysret Musliu

FAKULTÄT FÜR !NFORMATIK

Master Thesis

Generic Programing for Graph Problems Using Tree Decompositions

carried out at the

Institute of Information Systems
Database and Artificial Intelligence Group
of the Vienna University of Technology

under the instruction of

Priv.-Doz. Dr. Nysret Musliu
Dr. Fang Wei

by

Emmanouil Paisios

Vienna, July 23, 2008

.....	.....
Emmanouil Paisios	Nysret Musliu

Contents

Kurzfassung

Tree decomposition hat eine zunehmende Bedeutung als Werkzeug zur Lösung von Graphenproblemen, bedingt durch wichtige algorithmische Eigenschaften der Klasse der Graphen mit beschränkter treewidth. Ein wichtiges Theorem von Courcelle [Cou90] besagt, dass Eigenschaften eines Graphen die in MSO Logik beschrieben werden können, in linearer Zeit für Graphen mit beschränkter treewidth entschieden werden können. Dies gibt den theoretischen Hintergrund für die tractability vieler Graphenprobleme, intractability im Allgemeinen und für die oben erwähnte Graphenklasse regte es zu vielen Algorithmen für MSO-definierbaren Problemen an. Bodlaender schlägt in [Bod97] einen Ansatz vor zum Design von Algorithmen mittels dynamischer Programmierung unter der Verwendung von tree decomposition. Dieser Ansatz stellt eine verallgemeinerte Methode vor, mittels der sich auch bereits vor dieser Arbeit vorgestellte Algorithmen entwerfen hätten lassen, wichtiger jedoch diese Methode kann verwendet werden um neue Algorithmen für eine Vielzahl von Graphenproblemen zu konstruieren.

In dieser Diplomarbeit haben wir diesen Ansatz untersucht und implementiert. Das Ergebnis ist ein System, das Bodlaender's Methode verwendet und das es erlaubt spezifische Teile des Algorithmuses mittels Plugins zu beschreiben. Um die Verwendbarkeit und Effizienz des Systems einzuschätzen haben wir einen Algorithmus für Dreifärbung als Plugin implementiert.

Abstract

Tree decompositions have been an increasingly useful tool for solving problems on graphs due to the important algorithmic properties of the class of graphs of bounded treewidth. An important theorem by Courcelle [Cou90] stated that properties definable in MSO logic can be decided in linear time on graphs of bounded treewidth. This provided a theoretical base which revealed tractability of many graph problems, intractable in the general case, for the aforementioned class of graphs and instigated many algorithms solving MSO-definable problems. Bodlaender proposed a dynamic programming approach to designing algorithms using tree decompositions [Bod97]. The approach encompassed several of the algorithms already invented, but was also used as a method for the construction of new ones for a variety of graph problems.

In this thesis we studied and implemented this approach, delivering a system that operates using plug-ins to describe the specific parts of algorithms using Bodlaender's method. Furthermore we implemented a 3 colouring algorithm as a plug-in for the system in order to assess its usability and efficiency.

General Terms Tree decompositions, Dynamic programming.

Keywords Graph theory, dynamic programming, parameterized complexity, tree decomposition, bounded treewidth.

Acknowledgments

Upon the completion of this work, I would like to express my gratitude to a number of kind friends and coworkers. My supervisor Fang Wei, helped me not only with technical advice but also with her guidance and provided me with the confidence I needed for my future pursuits. Professor Musliu, helped me very much during last stage of my thesis, with his comments and suggestions. My friends and fellow students Bruno, Katya, Tomer, Hannes and the other students for their advice and help, and for making my life at the university during my work, fruitful and pleasant. The whole EMCL program and to Prof. Hölldobler and Prof. Pereira who helped me during the unusual beginning of my studies. And to my dear friends Vadim, Sasha and their son Misha, I am grateful, for making me a part of their beautiful life. I am thankful to my parents who provided for me in my every step with love and care, and to my sister who was there for me with a devotion that I do not feel worthy of. Finally to my dear and lovely Ayşe, for all those beautiful moments that were, and the ones that are to come, for her unswerving love and the dreamy world she bestowed on me.

Chapter 1

Introduction

This thesis is concerned with the implementation of a general approach to designing algorithms. This approach is presented in [Bod97] and describes a way to design algorithms solving problems for a family of graphs called graphs of bounded treewidth. This family of graphs has many important algorithmic properties, chiefly due to a very important theorem by Courcelle [Cou90] which stated that all problems expressible in Monadic Second Order logic(MSO) can be decided in linear time on graphs of bounded treewidth[1]. From the well known equivalence of MSO-logic formulas and finite automata one can derive an automated approach to solving these problems using MSO descriptions. Such efforts were made, (e.g. by [Mar06],[KMS02]), yet in some cases proved to be rather difficult to use because of "state explosions" of the generated automata as observed by [GPW07],[Mar06], which had as a consequence either unstable or very lengthy executions. Thus, dedicated algorithms for this class of problems seem to be still important. In [Bod97], Bodlaender observed that many of the algorithms proposed shared a common underlying structure which he formulated and described and which we endeavored to implement.

The result is a generic, pluggable system which provides the necessary background for algorithms following the aforementioned design. The purpose is to have a system on which this class of algorithms can be easily implemented and several algorithms objectively compared. Furthermore, in order to appraise the system, and exhibit its functionality, we implemented an example algorithm solving the 3-colouring problem on this class of graphs.

The thesis is structured in the following way:

Chapter 2 Fundamentals of graph theory are presented briefly, followed by background knowledge of tree decompositions. Some of the algorithms implemented are presented in detail along with the related complexity results.

[1] For a very interesting survey about this and other related meta-theorems see [Gro07]

Chapter 3 The formal description of the generic algorithm approach is presented. The relevant notions are explicated and there is a discussion for further constructions.

Chapter 4 The general structure of the implemented system is presented. Following the structure the implementations corresponding to basic objects and notions of the algorithm are explained and complexity related information for them is provided.

Chapter 5 Chapter 5 is composed of 2 parts. In the first part a 3 colouring algorithm is presented according to the generic design. The necessary notions are defined and their corresponding complexity is sketched. The second part presents in brief the implementation details and proceeds with some test results to provide an example of the algorithm's behavior.

Chapter 6 In chapter 6, several test results regarding the system are made. The measurements are independent of the plug-in and their purpose is to show the overhead of the base system.

Conclusion There is a discussion regarding future work and several conclusions regarding the thesis.

Chapter 2

Preliminaries

In this chapter we cover several of the fundamental knowledge necessary for the thesis. The first part of this chapter will cover some fundamental graph-related topics. The latter part, will include definitions of the main concepts of the generic algorithm approach such as tree decompositions and nice tree decompositions. For a more thorough introduction to graph theory the reader can consult [Die05].

2.1 Basic Definitions

Definition 2.1.1 (Graph)**.** A *graph* G is a pair of sets (V, E) with $E \subseteq V \times V$. The set V is the set of vertices and the set E the set of edges of the graph. Throughout the thesis we consider only *undirected graphs*, that is, graphs where the order of the vertices in each edge is not important.More formally, for each $e, e' \in E$, with $e = (v, u)$ and $e' = (u, v), e' = e$. Two vertices u, v are called adjacent if there is an edge $(u, v) \in E$. The neighborhood of a vertex v, $\mathcal{N}(v)$ is the set of vertices adjacent to v. We will denote the set of vertices/edges of a graph G with $V(G)/E(G)$ respectively. Order of a graph is the number of its vertices $|V|$.

Definition 2.1.2 (Subgraph)**.** A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. An *induced subgraph* of G is a subgraph $H = (V', E')$ where each $e = (v, u) \in E'$ iff $e \in E$ and $v, u \in V'$. We denote the induced subgraph of G, with a set of vertices V' as $G[V']$.

Definition 2.1.3 (Path)**.** Let $G = (V, E)$ be a graph. For $u, v \in V$, a *path* from u to v is a sequence of edges $P = e_1, \ldots, e_n$ such that $e_1 = (u, v_1)$, $e_n = (u_n, v)$, and for each $e_i = (u_i, v_i) \neq e_1$ we have $u_i = v_{i-1}$. The length of the path is n. We call the distance $d(u, v)$ between u and v the length of the minimum path from u to v. A path from u to u is called a *cycle*. A vertex is in a path if it is part of any of the edges of the path. Any edge connecting two non-adjacent vertices in a cycle is called a *chord*.

Definition 2.1.4 (Clique)**.** A *clique* in a graph $G = (V, E)$ is a set of vertices C such that for each $u, v \in C, (u, v) \in E$. We will say that C is k-clique if $|C| = k$.

Definition 2.1.5 (Connected Graph)**.** A graph $G = (V, E)$ is called *connected* if for every pair $u, v \in V$ there is a path from u to v. G is called *acyclic* if it contains no cycles.

Definition 2.1.6 (Tree)**.** A *tree* $T = (V, E)$ is a connected acyclic graph. A rooted tree is a tree where a vertex $r \in V$ is distinguished to be the *root* of the tree. This enables us to define a hierarchy w.r.t to the distance of any vertex from the root. A vertex u is a child of a vertex v if $(u, v) \in E$ and $d(u, r) = d(v, r) + 1$. A *leaf* is a vertex that has no children. A tree is called *binary tree* if each of its vertices has at most two children.

Definition 2.1.7 (k-tree)**.** A graph $G = (V, E)$ is called a *k-tree* if it can be described by the following recursive definition:

- Any $k + 1$-clique is a k-tree.
- Given a k-tree with n nodes, a k-tree with $n + 1$ nodes can be constructed by creating a new node adjacent only to the vertices of a k-clique of the k-tree.

Definition 2.1.8 (Triangulated Graph)**.** A *triangulated graph* $G = (V, E)$ is a graph such that there is no cordless cycle of length greater than 3.

Definition 2.1.9 (Simplicial Vertex)**.** A vertex v in G is called *simplicial* if its neighborhood $\mathcal{N}(v)$ is a clique. Furthermore, we say that G has a *perfect elimination scheme* or *perfect elimination order* if there is a sequence of vertices $e = v_1, \ldots, v_n$ such that each $v_i \in e$ is a simplicial vertex in $G[v_i, \ldots, v_n]$.

The following theorem can be found in [FG65] and [Ros70]. We will need it later to show the correctness of the algorithm for constructing nice tree decompositions.

Theorem 2.1.10. *A graph is triangulated iff it has a perfect elimination scheme. Furthermore for a triangulated graph G, any simplicial vertex can start an elimination scheme.*

Definition 2.1.11. A terminal graph H is a triple (V, E, X) such that (V, E) is a graph, and X is a (possibly ordered) subset of the vertices in V. Each $v \in X$ is called a terminal of H.

The binary operation $\oplus$ is defined on terminal graphs as follows. Given two terminal graphs $H = (V_H, E_H, X_H), H' = (V_{H'}, E_{H'}, X_{H'})$. We say $G = H \oplus H'$ if G is a graph (V, E) such that V is constructed by taking the

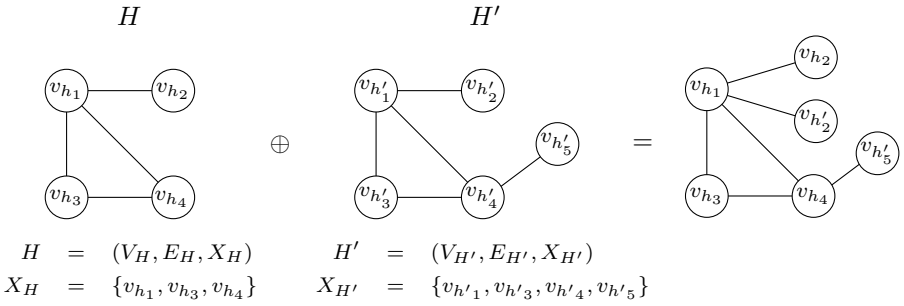


Figure 2.1: Example of the operation $H \oplus H' = G$.

disjoint union of $V_H, V_{H'}$ and joining only those vertices common between X_H and $X_{H'}$. An example of the operation $\oplus$ can be seen in Figure 2.1.

A terminal graph H, is called terminal subgraph of a graph G, if there exists a terminal graph H' such that $H \oplus H' = G$.

2.1.1 Related Algorithms

Calculating Perfect Elimination Schemes

An $O(|V||E|)$ time algorithm for the calculation of perfect elimination schemes, is described in [RT75] and is presented here. We need to define the following:

- A mapping $a : \{1, \ldots, n\} \to |V|$, where $n = |V|$.
- Labels of the form $L(v) = [p_1, \ldots, p_k]$, where $p_1, \ldots, p_k$ are natural numbers. and
- An ordering of the labels such that $L(v) < L(u)$, if for $L(v) = [p_1, \ldots, p_k]$ and $L(u) = [q_1, \ldots, q_l]$ we have:
  - $k < l$ and $p_1 = q_1, \ldots, p_k = q_k$, or
  - There exists $m \leq min(k, l)$ such that $p_1 = q_1, \ldots, p_{i_{m-1}} = p_{j_{m-1}}$ and $p_{i_m} < p_{j_m}$.

We present the actual algorithm in Algorithm 1.

Constructing a k-tree

Given a triangulated graph G_T a k-tree can be constructed in linear time for k constant, using Algorithm 2 which was introduced in [Klo94]. The k-tree produced by Algorithm 2 is also a triangulation of G.

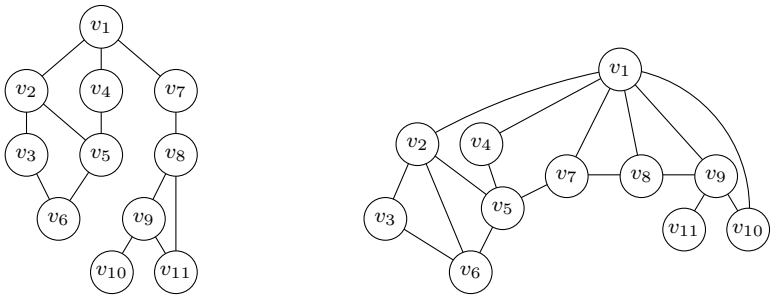
It is important to note here that the assumption of k being constant expresses the parameterized complexity of the algorithm with parameter k. In the rest of the thesis almost all of the graph related complexity results we

Algorithm 1 Perfect elimination scheme for triangulated graphs.

Require: An undirected triangulated graph G.
For each vertex u, assign a label $L[u]$, empty.
for $i = n$ down to $i = 0$ **do**
Select vertex u_m such that $L[u_m] = max(\{L[u], u \in V\})$ and $a(j) \neq u$, for all $1 \leq j \leq n$.
$a(i) \leftarrow u_m$.
for all w such that $a(j) \neq w$, for all $1 \leq j \leq n$ and there exists a path $[u_m, v_1, \ldots, v_p, w]$ **do**
if for $1 \leq k \leq p$, there exists no j with $a(j) = v_k$ and $L[v_k] \leq L[w]$. **then**
Append i to $L[w]$.
end if
end for
end for
Return the array of vertices $a(1), \cdots, a(n)$.

present, refer to the parameterized complexity with parameter the width of the tree decomposition which we will describe in the next section.

An example result of the algorithm 2 is shown in Figure 2.2.



(a) Graph G. (b) a 2-tree of G.

Figure 2.2: Example of a k-tree with k=2.

2.2 Tree Decompositions

Intuitively, a tree decomposition is a mapping from a graph G to a tree. An important related notion we will describe below is the treewidth of a graph. This is due to Courcelle's theorem [Cou90] which states that all properties expressible in monadic second order logic, can be decided in linear time, for the class of graphs of bounded treewidth.

Algorithm 2 k-tree construction from a triangulated graph.

Require: A triangulated graph G_T and an integer k.

```
Using Algorithm 1 construct a perfect elimination scheme for G_T,
{v_1, ..., v_n}.
Initialize G_k to be the k+1 clique using v_{n-k}, ..., v_n.
for i = n down to n - k - 1 do
  Add v_i to G_k.
  for j = i to n do
    Add edge (v_i, v_j) to G_k.
  end for
end for
for i = n - k - 2 down to 1 do
  for all w ∈ N(v_i) and in G_k do
    Add node v_i to G_k.
    Add edge (w, v_i) to G_k.
  end for
end for
```

Definition 2.2.1 (Tree Decomposition [Bod97])**.** Given a graph $G = (V, E)$, a *tree decomposition* is a pair $\mathcal{D} = (S, T)$ where T is a tree (I, F) and $S = \{X_i | i \in I\}$, where each X_i corresponding to a vertex i of the tree, is called bag of i, and we have $X_i \subseteq V$. The following conditions must hold:

- $\bigcup_{i \in I} X_i = V$
- For all edges (u, v) in E, there must be a node i in the tree T of the tree decomposition such that $u, v \in X_i$.
- Given two vertices $i, j \in I$, let $X_\cap = X_i \cap X_j$. Then for any bag X_k with k in the path between i and j, it must be $X_\cap \subseteq X_k$. This condition is known as the connectedness property.

The *width* w of the tree decomposition is the size of the largest of the bags X_i minus 1, that is $w = \max\{|X_i| \, | i \in I\} - 1$. Let $\mathfrak{D}(G)$ be the set of all tree decompositions $\mathcal{D}_i$ of a graph G, we call *treewidth* of G the minimum width over all $\mathcal{D}_i$ in $\mathfrak{D}(G)$. An obvious consequence of the above definition is that given a graph and a tree decomposition of width tw for it, we know that the treewidth of the graph is at most tw.

From here on we will follow the convention to call the members of I nodes. An example of a tree decomposition is shown in Figure 2.3.

Observe that for each node i of the tree T with bag X_i, we can associate a terminal graph $H_i = (V_H, E_H, X_i)$, such that V_H is the union of the bags of the descendants of i and E_H the edges of the induced subgraph $G[V_H]$.

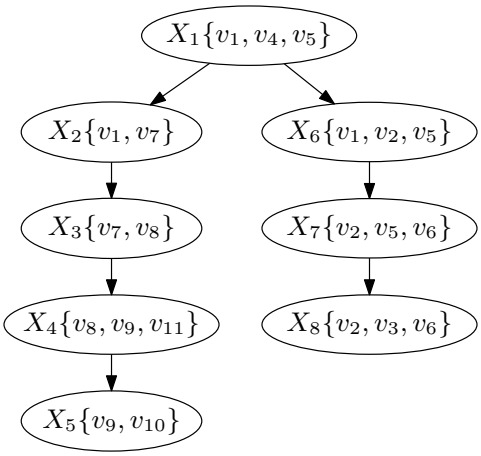


Figure 2.3: Example of a tree decomposition of graph in Figure 2.2 (a).

Notice also, that H_i is actually a terminal subgraph of G, since for the graph $H'_i = (V'_H, E'_H, X_i)$ with $V'_H = (V - V_H) \cup X_i$ with E'_H edges of the induced subgraph $G[V'_H]$, the relation $H \oplus H' = G$ holds. This is due to the connectedness condition of tree decompositions. We will briefly show this by contradiction. Let (v, w) be an edge such that $v \in (V_H - X_i)$ and $w \in (V - V_H)$ then there must exist a bag $X_{i'}$ such that $v \in X_{i'}$ where i' is a descendant of i. Furthermore there must exist a bag X_j such that $w \in X_j$ and j not a descendant of i, and a bag X_k such that both $v, w \in X_k$. If k is not a descendant of i then v must belong to all the bags in the path from k to i' and hence in X_i. If k is a descendant of i then w must belong to all the bags in the path from k to j and hence in i.

Definition 2.2.2 (Triangulation induced by tree decomposition)**.** Given a tree decomposition $\mathcal{D}$ for a graph $G = (V, E)$, the graph $G_T = (V, E_T)$, where $(u, v) \in E_T$ iff there exists a bag X_i such that $u, v \in X_i$. G_T, is called the triangulation of G induced or implied by $\mathcal{D}$.

The graph G_T is indeed a triangulated graph, this is proved in [Klo94].

Definition 2.2.3 (Nice Tree Decomposition)**.** We call a *nice tree decomposition* a tree decomposition that follows a certain structure as described below. A nice tree decomposition may have four kinds of nodes.

1. *Leaf node*: A node that has no children.
2. *Increase node*: A node i whose bag X_i is a superset of at least one of the bags of its children.
3. *Reduce node*: A node i whose bag X_i is a subset of the union of the bags of its children

4. *Stable node*: A node i such that the bags of i and its children are equal.

There can be many kinds of nice tree decompositions, we shall discuss three of them (also implemented in the system).

Definition 2.2.4 (Kloks-type nice tree decomposition)**.** We shall call Kloks-type nice tree decompositions binary tree decompositions with the following types of nodes:

1. *Join node*: A stable node that has two children j, k, such that $X_i = X_j = X_k$.
2. *Introduce node*: An increase node i that has one child j and $|X_i| = |X_j| + 1$.
3. *Forget node*: A reduce node i that has one child j and $|X_i| = |X_j| - 1$.

Definition 2.2.5 (Bodlaender-type nice tree decomposition)**.** We shall call Bodlaender-type nice tree decompositions any Kloks-type nice tree decomposition with the following extra restriction:

1. *Leaf node*: A leaf node χ_i with bag size $|X_i| = 1$.

Definition 2.2.6 (HN-type nice tree decomposition)**.** We shall call an HN-type nice tree decomposition, binary tree decompositions such that the nodes are of the following types:

1. *Leaf Nodes.*
2. *separator node*: A reduce node i with one child.
3. *join node*: An increase node i with two children its bag is the union of the bags of its children.

The following theorems are presented and proved in [Klo94], and themselves are used to show the existence of a nice tree decomposition of width tw for a graph G given a tree decomposition for G of the same width. As this is the foundation for creating nice tree decompositions in our implementation, we will also present the proof, which being constructive, naturally describes an algorithm.

Theorem 2.2.7. *Let $\mathcal{D}$ be a tree decomposition of width tw for $G = (V, E)$ and $G_T = (V, E_T)$ is the triangulation for G induced by $\mathcal{D}$. The tree decomposition $\mathcal{D}$ is also a tree decomposition of G_T.*

From the above theorem we conclude that a tree decomposition for G is also a tree decomposition for G_T and any k-tree constructed from G_T by Algorithm 2.

The following proof shows that given a tree decomposition for a graph G we can construct a HN-type nice tree decomposition of the same width. The proof can be easily transformed to fit the other cases.

Theorem 2.2.8. *Let G be a graph and $\mathcal{D} = (S, T)$ a tree decomposition for G, with width tw. There exists an HN-type nice tree decomposition $\mathcal{D}' = (S', T')$ of G with width tw and $V(T') = O(V(T))$.*

Proof. Initially we construct the triangulated graph G_T induced from $\mathcal{D}$. This obviously takes linear time when the treewidth of the graph is bounded. From the triangulated graph one can find a perfect elimination scheme in linear time and thus construct a k-tree H in linear time using algorithm 2. The rest of the proof shows that a nice tree decomposition for H can be constructed in linear time. Let n be the number of vertices in H. By induction on the order of the graph, the base case being $n = k + 1$, a nice-tree decomposition is possible by taking a tree decomposition of one node with bag size $k + 1$. For the case of Bodlaender-type one needs to use introduce nodes until bag size 1 is reached. Assuming that for any k-tree with $n > k+1$ we can construct a nice tree decomposition, we show that the same can be done for a k-tree of size $n+1$. Let H be the k-tree. Then there must exist a simplicial vertex v in H. Let H' be the k-tree without v. Then there exists a nice tree decomposition $\mathcal{D}_{H'}$ for H'. Take the neighborhood of v, $\mathcal{N}(v)$ which is a clique of size k. There must be a node χ_i in $\mathcal{D}'$ such that $X_i \supseteq \mathcal{N}(v)$. What is left at this point is to show that a nice tree decomposition $\mathcal{D}''$ can be constructed from $\mathcal{D}'$ such that there is a bag X_j in $\mathcal{D}''$ that contains v. For the HN-type nice tree decomposition we have:

- If χ_i is a leaf node, we add two children χ_l, χ_r such that $X_l = X_i$ and $X_r \subset X_i$. Then we add a child χ_c to χ_r such that $X_c = X_r \cup \{v\}$. Observe that we only added three more nodes for this case.

- If χ_i is a unary node with child χ_c. It must be $X_i \subset X_c$. Thus we can proceed to the child until we reach a leaf or a join node.

- If χ_i is a join node with children χ_l and χ_r. We remove the children from χ_i and connect two others χ_{cl}, χ_{cr} such that:

  - $X_{cl} = X_i$,
  - $X_{cr} = \mathcal{N}(v)$.

  Then add a child χ_c to χ_{cr} with $X_c = X_{cr} \cup \{v\}$. Observe again that we have added three nodes to the tree decomposition.

The construction clearly takes linear time since the modifications in all the cases (leaf, unary, binary) can be performed in constant time and the tree decomposition has at most $3(n - 1)$ nodes. □

The proof for the Kloks-type nice tree decomposition can be found in [Klo94] where it is also shown that the size is at most $4(n - 1)$. For the Bodlaender-type the proof is the same while one should observe that the

reduction of leaves to bags of size 1 causes the resulting tree decomposition to be of size at most $4(n-1) + (n/2 \cdot (tw-1))$.

In general nice tree decompositions whose specification follows certain rational restrictions, can be constructed in linear time. One restriction is that the construction of the subset should be possible in linear time in the case of increase nodes, and, correspondingly for the reduce nodes, that the construction of the superset should also be possible in linear time. Furthermore, the specification of the largest size of the subset and the least size of the superset, as well as the type of the reduce or increase nodes with respect to the number of children (unary or binary), affect the size of the tree decomposition and hence, the overall complexity of the construction.

In literature, nice tree decompositions are defined specifically for some problems because they offer advantages in the design of algorithms in a way which we will see in the next chapter. A relatively novel example which we do not cover here, can be found in [DT06], where a *semi-nice tree decomposition* is defined in aid to the design of an algorithm following the general approach we will present in chapter 3, that solves the minimum dominating set in linear time with respect to graphs of bounded treewidth, pathwidth or branchwidth.

Chapter 3

General Algorithms and Tree Decompositions

A very interesting observation regarding the design of algorithms for graphs of bounded treewidth was made by Bodlaender in [Bod97]. Several of the algorithms designed in order to solve problems for graphs of bounded treewidth shared a common underlying structure [BLW87][AP89], [HN02]. Bodlaender expressed this underlying structure in a clear way, presenting it as an approach which can be used as a general design method. In this chapter we explain in detail this approach and define the necessary objects which take part in the design process.

3.1 Structure

We begin with a general description of this underlying structure. Let G be the input graph for which we want an algorithm solving a problem $\mathcal{P}$. The procedure consists of several steps enumerated below.

1. Construct a tree decomposition $\mathcal{D}$ for G

2. Construct a nice tree decomposition $\mathcal{D}'$ from $\mathcal{D}$.

3. For each node in $\mathcal{D}'$ compute a set of structures called characteristics for each node in the tree decomposition. Characteristics, in some sense, describe the properties of possible partial solutions. This computation step is done bottom up and thus, for each node, its set of characteristics should be done effectively given the characteristics of its descendants.

4. If the set of characteristics of the root is not empty the problem is solvable, otherwise there is no solution. If we only need a decision procedure for the problem the algorithm can stop here.

5. Process the characteristics in top-down order to construct the solutions.

In general finding the treewidth of a graph is known to be an NP-hard problem, but when the graphs are known to be graphs of bounded treewidth, their treewidth can be found in linear time using an algorithm presented in [Bod93]. However the aforementioned algorithm is practical for very small instances and therefore there is need for other approaches to the tree decomposition construction. A relatively recent review of the various algorithmic techniques for construction of tree decompositions can be found in [Bod05]. We have already shown that given a tree decomposition, a nice tree decomposition with the same treewidth can be constructed in linear time. The main idea here is that, since the number of nodes in the tree decomposition is linear with respect to the nodes of the graph, then, if the characteristics for a node given the characteristics of its ancestors can be calculated in constant or linear time, we get respectively a linear or polynomial time algorithm that decides the problem. Designing algorithms under this schema, requires that this be considered for the basic notions we will present in the following section.

3.2 Basic Notions

Let $\mathcal{P}$ be a problem we want to solve and G a graph that belongs in the class of graphs of bounded treewidth. The notions we need to define in order to provide a decision procedure for $\mathcal{P}$ are the following:

Solution This is a natural notion describing what kind of object describes a solution for $\mathcal{P}$. We can define a relation $sol_{\mathcal{P}}(G, s)$ which is true whenever s is a term representing a solution for graph G. For example a solution for a subgraph isomorphism problem from G to G' could be a mapping from some of the nodes of G to the nodes of G'. In Figure 3.1 the mapping $\{\phi_1, \phi_2, \phi_3, \phi_4, \phi_5\}$ is a solution for the subgraph isomorphism problem for G, G'.

Partial Solution A partial solution defines the behaviour of a solution with respect to a terminal graph. Thus, for a node in the tree decomposition a partial solution for the terminal subgraph H related to that node should describe the behaviour of a complete solution when restricted to H. Similarly to the case of solutions we define a relation $psol_{\mathcal{P}}(H, p_H^{\mathcal{P}})$ where H is a terminal graph and $p_H^{\mathcal{P}}$ a term representing a partial solution, that is true whenever $p_H^{\mathcal{P}}$ constitutes a partial solution for the problem. A partial solution for an algorithm that solves the subgraph isomorphism problem can be a mapping of some of the nodes of a terminal graph H of G to some of the nodes of G'. In the example in Figure 3.1, assuming that there exists

in our nice tree decomposition a node i with bag $X_i = \{v_8, v_9, v_{11}\}$ with a successor j, leaf, with $X_j = \{v_9, v_{10}\}$, then the mapping $\{\phi_2, \phi_3, \phi_4, \phi_5\}$ is a partial solution for the terminal subgraph $G[X_i]$.

Extension of a partial solution An extension need not be a specific object. We need though a relation that describes in some way how a partial solution can be extended to a solution. This relation can be defined as $ex_{\mathcal{P}}(G, s, H, p_H^{\mathcal{P}})$ for a graph G, a solution s, a terminal graph H, and a partial solution $p_H^{\mathcal{P}}$, that is true if s is an extension of $p_H^{\mathcal{P}}$ for the given G, H. For the example of subgraph isomorphism, the relation $ex_{\mathcal{P}}(G, s, H, p_H^{\mathcal{P}})$ should be true if s is such that each mapping from nodes H to G in $p_H^{\mathcal{P}}$ is also in s and s is a solution for G. For the partial solution in the previous example, the solution $\{\phi_1, \phi_2, \phi_3, \phi_4, \phi_5\}$ is an extension of the partial solution $\{\phi_2, \phi_3, \phi_4, \phi_5\}$, such that the above predicate is true.

Characteristic of a partial solution The characteristic of a partial solution, should provide minimal information to enable the algorithm to tell whether and how a partial solution can be extended to a solution. An important property that should hold for characteristics is that if two partial solutions p_{sol1} and p_{sol2} have the same characteristic, then p_{sol1} can be extended to a solution if and only if p_{sol2} can be extended to a solution. To fulfill the aforementioned requirement for characteristics it is useful to define a function $ch_{\mathcal{P}}(H, p_H^{\mathcal{P}})$ that evaluates to the characteristic for the partial solution $p_H^{\mathcal{P}}$ of terminal graph H. The characteristic is a more involved notion compared to the ones described above and very important to the inner workings of the algorithm. An example for the subgraph isomorphism problem, for a terminal subgraph $H = (V_H, E_H, X_H)$ and a partial solution p_{sol} could be a mapping from the nodes of X_H to a subset of the nodes of G'. According to the this definition, a characteristic for $G[X_i]$ for our specific example in Figure 3.1, is the mapping ϕ_2, ϕ_3, ϕ_5.

3.3 Algorithm design

When designing the algorithm, apart from defining the notions we described previously, one needs to consider two more aspects.The first one involves formal relationships that should hold between the notions in order to verify the correctness of the algorithm and the second regards the construction of characteristics, where assurances for complexity should be addressed.

3.3.1 Formal rules for notions

For the specific instances of the basic notions several relationships between them can be formally characterized. More precisely the notions for the

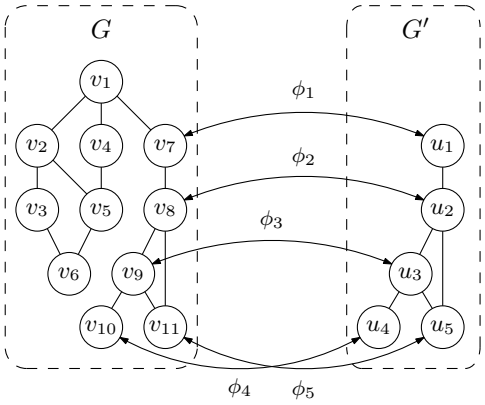


Figure 3.1: Example for basic notions for the subgraph isomorphism problem.

concrete algorithm for $\mathcal{P}$ should fulfill the following logical expressions.

$$ex_{\mathcal{P}}(G, s, H, p_H) \rightarrow \exists H' \left((G = H \oplus H') \wedge sol_{\mathcal{P}}(G, s) \wedge psol_{\mathcal{P}}(H, p_H)\right) \tag{3.1}$$

Relation 3.1 captures the notion of an extension of a partial solution. It expresses that s is an extension of G implies that there exists a terminal subgraph H of G,such that s is a solution for $\mathcal{P}$ on graph G and p_H is a partial solution for H.

$$\begin{aligned} &\forall G, s, H, H'((sol_{\mathcal{P}}(G, s) \wedge (G = H \oplus H')) \\ &\rightarrow \exists p_H(psol_{\mathcal{P}}(H, p_H) \wedge ex_{\mathcal{P}}(G, s, H, p_H))) \end{aligned} \tag{3.2}$$

Relation 3.2 expresses the requirement that for any terminal subgraph H of G and any solution s for $\mathcal{P}$ on G, a partial solution p_H such that s is its extension, exists.

$$\begin{aligned} &\forall H, H', H'', p_H, p_{H'}((ch_{\mathcal{P}}(H, p_H) = ch_{\mathcal{P}}(H', p_{H'})) \\ &\rightarrow \left((\exists s(ex_{\mathcal{P}}(H \oplus H'', s, H, p_H) \leftrightarrow (\exists s'(ex_{\mathcal{P}}(H' \oplus H'', s', H', p_{H'})\right) \end{aligned} \tag{3.3}$$

Relation 3.3 expresses exactly the need that equal characteristics ensure the mutual existence of solutions. That is, p_H has an extension if and only if $p_{H'}$ has an extension.

3.3.2 Complexity Issues

As mentioned before the algorithm performs the calculation of characteristics in bottom up order. One must define how these constructions should be done with respect to the nodes in the nice tree decomposition. For example,

assuming that our algorithm uses Bodlaender-type nice tree decompositions and we would like to solve the subgraph isomorphism problem. We should define how the characteristics are constructed for the leaf nodes and moreover, how the construction of the characteristics takes place in the case of forget, introduce and join nodes given the characteristics of their children.

When defining the above constructions one must keep in mind that the complexity of the construction directly affects the complexity of the whole algorithm. Hence if these constructions can be performed in constant time, the algorithm, would have a linear complexity (under the assumption that the tree decomposition we are using has $O(n)$ nodes and that the input graphs belong to the class of graphs of bounded treewidth).

One last remark regarding complexity is about the construction of solutions. This usually is done in a top down manner, processing the characteristics of the root, constructing partial solutions derived from characteristics while moving towards the leafs until the desired solutions are calculated. The construction of partial solutions should be feasible in constant or linear time for each type of node in our nice tree decomposition. In this case though, since the construction is performed in a top down manner, the partial solutions produced should reflect the behaviour of a candidate solution for the graph $G[\overline{X_i}]$, where X_i is the bag of node i in our tree decomposition and $G[\overline{X_i}]$ the terminal subgraph with vertices the union of all the bags in the path from the root to i.

In practice a problem that should often be addressed is the memory consumption of the algorithm which could grow exponentially in order to keep all the characteristics. A very interesting approach that addresses this problem is the *anchor method* [BNU04]. The method proposes an algorithm that modifies the nice tree decomposition, approximating a solution to a weighted set covering problem and adding some necessary connections for consistency, with the intention of minimizing the redundant characteristics.

Chapter 4

Implementation

This chapter explicates the implementation of the generic algorithm which will henceforth be referred to as *gSolver*. We will discuss the structure and the interface of gSolver and provide a brief introduction on the way plug-ins may be designed.

4.1 General Structure

The central idea of the implementation was to create a program that represents the generic graph algorithm described in the previous chapter, in order to run the corresponding family of algorithms in a unified manner. Since the construction of the algorithms is a non-trivial task and the involved objects of possibly diverse kinds, the approach of a pluggable interface was considered and materialized. The implementation was made in C++ following the generic programming paradigm. Under this perspective, we would require a plug-in providing descriptions of the necessary notions of the algorithm and implementations of the respective algorithmic parts. Given the implementations for these notions gSolver can follow the algorithm using their instances and return the result.

The program flow is described briefly below and in Figure 4.1.

1. Load the selected plug-in.

2. Handle the necessary input, which should be in the minimal case a graph G and its tree decomposition D.

3. From G and D, construct a nice tree decomposition T.

4. Given G and T, we construct the induced subgraphs for each node of T.

5. Calculate the full set of characteristics for each node, bottom up.

6. Construct a solution, if the appropriate information is provided in the plug-in.

7. Output the result.

Loading of the plug-in corresponds the following:

- Checking whether the plug-in is complete, that is, whether the necessary functions exist.

- Loading of the functions declaring the plug-in arguments, tree decomposition type and the necessary algorithm parts.

Regarding the input, the minimal case is a graph instance, a tree decomposition for it and the name of the plug-in chosen. Custom input sources are also available through the option handling interface provided by gSolver.

Construction of the nice tree decomposition is done following the steps described in the previous chapters. More precisely:

1. Construct the triangulated graph induced by the tree decomposition of the input.

2. Extract a perfect elimination scheme for the triangulated graph.

3. Given the elimination scheme, construct a k-tree for the given graph.

4. Construct a nice tree decomposition either base on the descriptions already existing in gSolver or the ones defined by the plug-in.

Finally we have the calculation of the characteristics. Where we call the appropriate functions described by the plug-in w.r.t. the type of the node considered at that moment. The nodes are processed in a bottom up order.

4.2 Basic Classes and Datatypes

The fundamental objects for all the algorithms under the considered schema are graphs and tree decompositions. We will describe in detail their design below.

For graph objects, we use the Boost Graph Library (BGL). The BGL enables us to keep the graph structure as generic as possible and provides a good foundation for the construction of the induced subgraphs. We extended instances of these template classes in our *ugsGraph* class to provide the extra functionality we needed.

For tree decompositions, in order to maintain coherence between our graphs and the tree decomposition trees, we keep the use of BGL graphs but extended in our *treeDec* class.

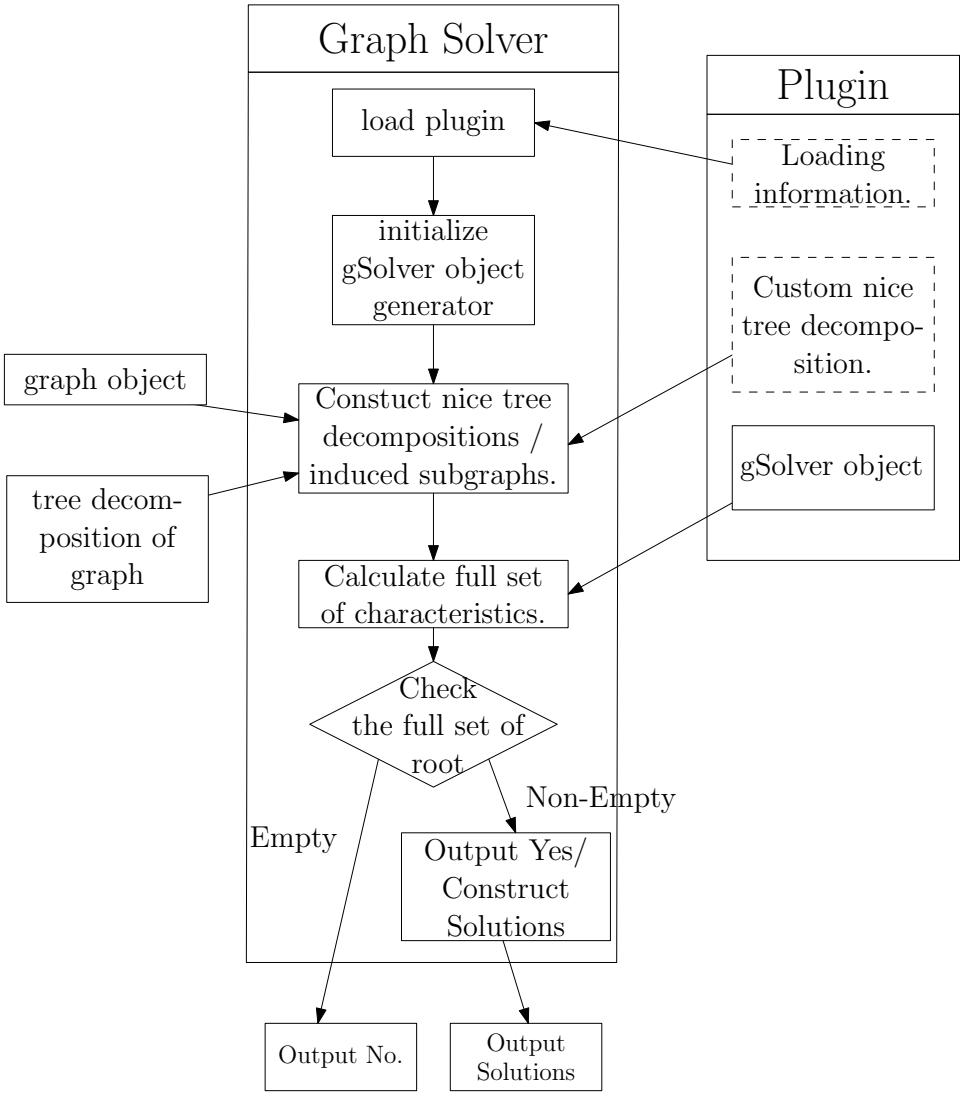


Figure 4.1: General structure of the solver.

4.2.1 The ugsGraph class

The top level graph structure used within the project is called *ugGraph* and is an instance of the BGL subgraph template of an adjacency list using STL lists for vertices and vectors for edges. The choice of lists and other containers is done considering mainly the complexity of the operations used in the algorithms, but also the need for a programmer friendly interface, since plug-ins should be easy to construct. Vertex and edge descriptors and iterators are provided for access to the graph objects by the BGL. The ugGraph class is intended to be as simple as possible, thus it contains only minimal properties for vertices using the *bundled properties* interface provided by boost[1], such as name and id. Properties for edges are defined in a similar way. Further properties can be added as external properties per vertex or edge descriptors.

The main class used in all the transformations is the *ugsGraph* class. It is a child class of ugGraph and provides methods constructing many of the objects needed for the transformations we perform. An important property inherited from ugGraph is that of the subgraph. The graphs used provide vertex creation and removal (accessed from vertex descriptors), edge creation, and vertex existence in constant time. Edge existence and edge removal is done in time $O(V/E)$. Also all the operators used for traversal (i.e. $++$) take constant time. Each object of the class G may have induced subgraphs easily defined from subsets of its vertices. Each subgraph is a ugGraph and maintains a connection to its parent and the root G. When building the output for these graph objects we can produce the complete structure in graphviz format which produces a result as the one depicted in Figure 4.2.

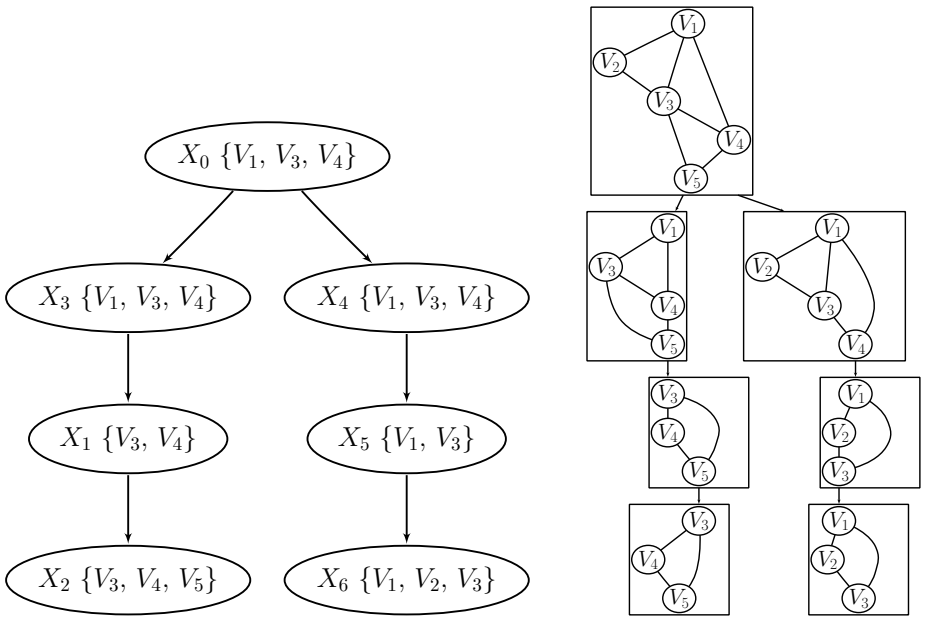
The common way to access specific information on nodes and edges is using vertex/edge descriptors and iterators. That is if we have a ugsGraph object `G`, the vertex descriptor `v` is used in the form `G[v]` to provide access to the actual vertex information. Vertex descriptors for the ugsGraph class are ugVertex objects. The following constructors are provided:

- ugsGraph(**std**::**string** *filename*):
  Reads a graph encoded in the file indicated by the string *filename*. The file should have a simple file format used by the Hypertree project.

- ugsGraph(**const treeDec**& *TD*):
  Constructs the triangulated graph induced by a reference to a tree decomposition *TD*.

- ugsGraph(**ugsGraph**& *G*, **eliminationScheme**& *e*, **size_t**):
  Constructs a ktree from a graph *G* a perfect elimination scheme *e* and

[1]For more information about bundled properties check `http://www.boost.org/doc/libs/1_35_0/libs/graph/doc/bundles.html`



(a) Kloks-type nice tree decomposition. (b) Terminal subgraphs

Figure 4.2: Subgraph structures produced for a Kloks-type nice tree decomposition.

a natural number for k.

The methods provided for the ugsGraph class address issues of connectivity and information on the state of the graph. The most important are listed below:

- **bool** getEliminationScheme(**eliminationScheme**& e):
  Constructs a perfect elimination scheme for the graph object in the object referred to by e. The construction of the perfect elimination scheme can be done in linear time using Algorithm 1. A simpler algorithm has been implemented which for the tested purposes produced slightly better results for the tested instances. For this algorithm we use to the property of triangulated graphs presented in Theorem 2.1.10, which states that any simplicial vertex can start a perfect elimination scheme. Thus we iteratively look for simplicial vertices, in graphs that are produced by removing the simplicial vertice found in the previous step, adding it to the elimination scheme. The storage type of the eliminationScheme objects is an std::vector which provides insertion in constant time at the end and random access to elements. Returns true if successful, false otherwise.

- **bool** isClique(**adj_iter** *AdjS*, **adj_iter** *AdjE*):
  Checks whether a set of vertex descriptors, given by two adjacency iterators indicating the first and last of the adjacent nodes of a simplicial vertex, constitute a clique. This check is done in time $O(E/V)$ w.r.t. parameterized complexity, since edge checking in our graphs is performed in time $O(E/V)$ and the cliques may never have size larger than the treewidth of the graph which is bounded.

- isSimplicial (**ugVertex** *v*,**eliminationScheme** *eS*):
  Returns true when vertex descriptor *v* describes a vertex that is simplicial. The complexity corresponds to the above function.

- **void** connect(**std**::**set**<**ugVertex**> *pc*, **ugVertex** *v*):
  Connects a vertex described by vertex descriptor *v* to each node in the set *pc*.

- **int** constructConnectedComponents():
  Builds a structure indexing the connected components of the graph object, returns the total number of connected components. This is a convenience function using the algorithm provided by boost which computes this set in time $O(V + E)$.

- **int** getVertexbyName(**char**∗ *name*):
  Returns a vertex descriptor of the vertex whose name corresponds to the argument *name*.

4.2.2 The treeDec class

Tree decompositions are implemented by the class *treeDec*. As in the case of the ugsGraph class we have a general graph object called *gen_tree_dec*, which is modeled as a directed graph by an adjacency list in a similar way as in the *ugGraph* graphs. We only attach bundled properties for vertices. The properties used are the minimal properties for a tree decomposition and defined for vertices in the *TDNodeProp* structure shown in Listing 4.1.

```
//*************** General TreeDec Properties ***************//
typedef std::set<ugVertex> TDBag;

struct TDNodeProp
{
    TDBag bag;
    ugGraph* terminalSubgraph;
};
```

Listing 4.1: Vertex Properties for tree decompositions.

The member *bag* represents the tree decomposition bags and are represented by STL sets of vertex descriptors for *ugGraph* objects. Every bag of the tree decomposition will eventually define a terminal subgraph corresponding to the vertices of its bag and its ancestors. This connection between the terminal subgraph is materialized by a pointer to terminal subgraphs that initially is set to `NULL`, but will point to the corresponding subgraph when the induced subgraph structures are built.

The *treeDec* class is a child class of *gen_tree_dec*, providing several methods for the programmer of the plug-in. It is used as a base for the class implementing all the types of nice tree decompositions supported by the project, which are represented by the class *niceTreeDec*. An inheritance diagram is shown in Figure 4.3.

A reference to a distinct graph object is also kept, meant to represent the original input graph from which it was derived. The vertex descriptor for the root node and the treewidth are also kept. Several methods providing set operations for the bag contents are provided as well. Some of the important member functions are mentioned below.

- treeDec(**std**::**string** *filename*, **ugsGraph**& *g*):
  Constructor, parses a file in gml format[2], which is used by the Hypertree project. If the graph *g* is empty it constructs *g* as the induced triangulated graph of the tree decomposition read.

- **void** constructAllTerminalSubgraphs():
  for each node of the tree decomposition its terminal subgraph. The *terminalSubgraph* pointer for each node is set appropriately. All the subgraphs created belong to the graph object indicated to by the reference kept in the structure.

- **void** getTerminalSubgraph(**tdVertex** *v*, **TDBag**& *rS*, **ugGraph**& *p*):
  This method is used by the member function constructAllTerminalSubgraphs. It constructs the subgraph for the node indicated by the node descriptor *v*, a set containing all the vertices in the bags of its children and the parent subgraph *p*.

- **void** addNode(**TDBag**& *n*, **tdVertex** *p*):
  Adds a node as a child of *p* with bag contents described by the bag *n*.

- **TDBag** getUnion(**tdVertex**& v_1, **tdVertex**& v_2):
  Returns a bag containing the union of the bags attached to the nodes indicated by v_1 and v_2.

[2]Information about the gml format can be found in `http://www.infosun.fim.uni-passau.de/Graphlet/GML/`

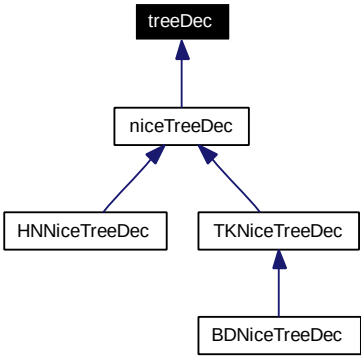


Figure 4.3: `treeDec` inheritance tree.

- **TDBag** getIntersection(**tdVertex**& v_1, **tdVertex**& v_2):
  Returns a bag containing the intersection of the bags attached to the nodes indicated by v_1 and v_2.
- **TDBag** getDifference(**tdVertex**& v_1, **tdVertex**& v_2):
  Returns a bag containing the difference of the bags attached to the nodes indicated by v_1 and v_2.
- **bool** isSubset(**tdVertex**& *vertexSupset*, **tdVertex**& *vertexSubset*):
  Returns true if the bag of the node *vertexSupset* is a superset of the bag of *vertexSubset*.
- **tdVertex** findTDNodeWithClique(**VertexVector** c):
  Returns the node descriptor of the node whose bag contains all the nodes in a clique c.

The niceTreeDec class

The *niceTreeDec* class is the parent class of all the types of nice tree decompositions supported at the moment. The reasoning behind the design stems from the constructive proof (Proof 2.2.8) of existence of a nice tree decomposition. Due to the fact that the construction from type to type changes only with respect to the way to insert the new nodes of the elimination scheme to the intermediate nice tree decomposition, a generic way to design and construct various nice tree decomposition types, can be provided using the visitor design pattern which allows functions to be defined called in predefined control points. So, any custom nice tree decomposition should define these functions which implement the way a vertex of the graph is added to a nice tree decomposition, given a node whose bag contains the neighbourhood of the vertex. The control points correspond to the kind of node we come across.

In our implementation, there are five virtual member functions available for specifying the construction of the nice tree decomposition. The three compulsory ones should define the way that an insertion of a vertex of the elimination scheme in the tree decomposition takes place for leafs, unary and binary nodes.

- **virtual void** LeafOp(**VertexVector** *c*, **tdVertex** *Xp*, **ugVertex** *x*):
  Should define how the vertex indicated by *x* and clique *c* of its neighbours in the so far constructed tree decomposition, is inserted in the nice tree decomposition at leaf node *Xp*.

- **virtual void** UnaryOp(**VertexVector** *c*, **tdVertex** *Xp*, **ugVertex** *x*):
  Similarly with the above member function, defines the insertion in the case of a unary node.

- **virtual void** BinaryOp(**VertexVector** *c*, **tdVertex** *Xp*, **ugVertex** *x*):
  Defines the insertion in the case of a binary node.

Note that we do not distinguish the functions with respect to the actual types of the tree decomposition, since these may vary essentially from definition to definition. What we induced to be necessary, is the processing information in the way that it is actually done during the construction which is the actual time where the type of node is formed. This corresponds also with the various related proofs we have come across.

Apart from these functions there are two more convenience functions that are called before and after the construction to provide some pre- and post-processing on the nice tree decomposition. Also pointers to the triangulated graph and the k-tree constructed in the process as well as the actual elimination scheme are provided to all descendants of *niceTreeDec*. The collaboration graph for the class can be seen in

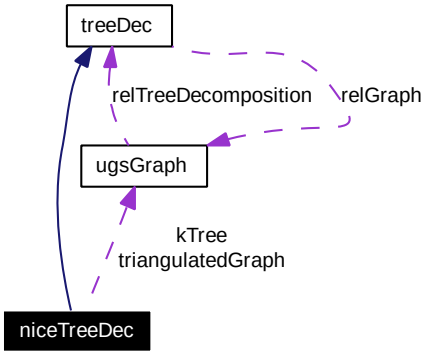


Figure 4.4: `niceTreeDec` collaboration graph.

- **virtual void** preProcessOp():
  Preprocessing of the nice tree decomposition.

- **virtual void** postProcessOp():
  Post processing of the nice tree decomposition.

All three types of nice tree decompositions defined in chapter 2 have been implemented already using this interface and in a very straightforward and concise way. As an example of how something like this is performed within the system we present a widely used type, Kloks-type, which is easily implemented. The actual source code for the case of leafs is shown in Listing 4.2. The other two functions corresponding to unary and binary nodes are implemented accordingly in the *unaryOp* and *binaryOp* functions. Finally, one needs to define the class as a child class of the some descendant of the *niceTreeDec* class and define a constructor calling the *compute* member function of *niceTreeDec* in its constructor.

```
void TKNiceTreeDec::leafOp(VertexVector c, tdVertex Xp,
                                           ugVertex x) {
    if (c.size() < (*this)[Xp].bag.size()) {
        /*
         * If the c is smaller than the bag of the node Xp
         * Add two nodes:
         *  1) rtreeNode, child of Xp with bag equal to c
         *  2) A child  of rtreeNode with bag equal to c+x
         */
        TDBag rBag(c.begin(),c.end());
        TDBag cBag = rBag + x;
        tdVertex rtreeNode = addNode(rBag,Xp);
        addNode(cBag,rtreeNode);
    }
    else {
        /*
         * If the c is equal to the bag of the node Xp
         * Add one node:
         * 1) child of Xp with bag equal to c+x
         */
        TDBag cBag(c.begin(),c.end());
        cBag = cBag + x;
        addNode(cBag,Xp);
    }
}
```

Listing 4.2: Function handling the case of unary nodes in the implementation of the Kloks-type nice tree decomposition.

The pre and post processing functions can be used to perform some initializations and modifications to the nice tree decomposition. In most cases their functionality is natural with respect to the definition of the tree decomposition. An obvious example is the way one can derive the Bodlaender-type nice tree decomposition from the Kloks-type one by defining a post-process function. The implementation should reduce the bags of the all the leaf

nodes using "introduce" nodes until they bags of size 1 are constructed. The actual implementation of the post-process function for the Bodlaender-type nice tree decomposition is shown in Listing 4.3.

```
void BDNiceTreeDec::postProcessOp() {
  /* Check every node in the tree decomposition */
  for(treeDec::vertex_iterator PXp = vertices(*this).first ;
                               PXp!= vertices(*this).second;
                               PXp++) {
    /* If we have a leaf reduce its size until it is one */
    if (countChildren(*PXp) == 0) {
      tdVertex curXp = *PXp;
      while ((*this)[curXp].bag.size() != 1) {
        tdVertex pXp = curXp;
        TDBag newBag = (*this)[pXp].bag;
        /* Remove the first vertex from the bag */
        newBag.erase(*newBag.begin());
        /* Add the new child as a leaf of pXp */
        curXp = addNode(newBag,pXp);
      }
    }
  }
}
```

Listing 4.3: Post processing function to provide a Bodlaender type nice tree decomposition.

4.2.3 Other Operations

There are several other operations implemented to aid in the manipulation of graphs and provide tools for easier programming.

Regarding ugsGraph objects we have implemented the operation $\oplus$ defined in chapter 2. The $\oplus$ operator is implemented using the operator "+" between objects of the ugsGraph class and returns the graph corresponding to the two terminal subgraphs provided. There is also some functionality for finding specific cliques. The function *a_relevant_clique*, returns cliques adjacent to a given node in a subgraph of the main graph object.

Finally a method for using power-sets is provided. It is called *get_next_in_powerset* and given a set and one of its subsets, it returns the next subset with respect to an ordering provided by the binary representation of the elements in and out of the set as a number.

4.3 Plug-in design

In this section, we will present a more precise description of the plug-in interface. The plug-in defines various entities and options needed for the calculation of the full sets, the solutions and their presentation. All these

are defined by instances of a class template called *sg_solver*. Each plug-in should construct such an instance and export its constructor and destructor functions via the plug-in interface.

4.3.1 The *sg_solver* template

The plug-in needs to implement an object of the class template *sg_solver* which itself is a subclass of the general *gen_sg_solver* class. The collaboration diagram is shown in Figure 4.5.

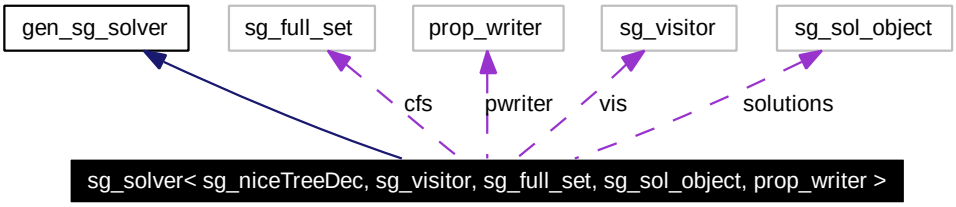


Figure 4.5: `sg_solver` collaboration graph.

The template arguments are described below:

`[sg_niceTreeDec]`: Type that represents the kind of nice tree decomposition we want to use. It may be one of the already implemented ones or any subclass of the niceTreeDec class.

`[sg_visitor]`: A class that corresponds to the visitor concept defining the behaviour of the algorithm on the crucial control points. This behaviour is derived directly from the proofs of correctness for the algorithms as shown in chapter 3.

`[sg_full_set]`: A class that corresponds to the full set of characteristics for all nodes. The classes implemented in the system providing this functionality are mappings from nodes of the tree decomposition to sets of characteristics. Of course other classes may be used when defined in the plug-in.

`[sg_sol_object]`: A class that describes a set of solutions. It should implement the `size()` method and provide an "[]" operator to access elements. No restrictions are made on the type.

`[prop_writer]`: Optional. A class that handles custom output options to output a solution using graphviz.

The following methods are available:

- **virtual size_t** solutionsReturned():
  Returns the number of solutions found by the algorithm. This is done by calling *size()* method for the solution object.

- **virtual bool** existsSolution():
  Returns true if a solution exists, i.e. the full set of characteristics for the root node is not empty. False otherwise.

- **virtual void** constructSolution(**treeDec**& *TD*, **ugGraph**& *g*):
  By default no solution is constructed. And the algorithm returns a yes or no answer to the problem. This method can be overloaded in order to provide a construction of the solution object.

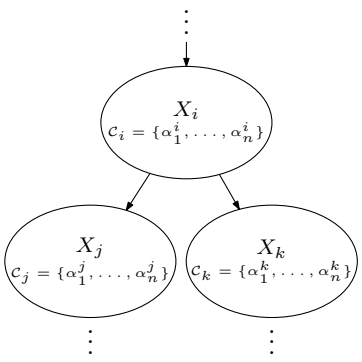
- **virtual sg_property_writer** solutionWriter():
  By default, returns a reference to the *sg_property_writer* object *pwriter*.

- **virtual treeDec** getTreeDec(**treeDec**& *T*):
  Returns a nice tree decomposition from the tree decomposition T, according to the type *sg_niceTreeDec* defined from the template arguments. This method calls a constructor of the *niceTreeDec* class, which we will describe later.

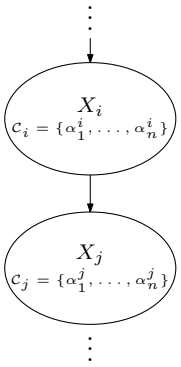
4.3.2 The sg_visitor class

The *sg_visitor* class template is one of the basic objects that a plug-in should implement. It follows the visitor design pattern so that the algorithm can abstract over the more specific structure of the plug-in. Any algorithm following the pattern described in chapter 3, must specify the way the calculation of characteristics is performed. This is always done with respect to the nice tree decomposition node type. Yet the node types may vary from one definition of a nice tree decomposition to another, thus we either need to each time extend the visitor with respect to the defined tree decomposition, or keep the definition of the visitor more abstract. In gSolver the second choice was made, since extending the visitor with respect to each definition of a nice tree decomposition would make the programing of the plug-in more complicated. Any sg_visitor should specify how these characteristics are calculated based on whether the node is a leaf, a unary node or a binary node. Further specialization at the moment is not supported for the reasons explained above and thus the plug-in must distinguish between more refined types of nodes.

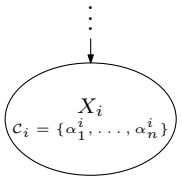
There are several methods to be implemented for an sg_visitor, each one specifying the behaviour of the algorithm at the specified control points, in our case, the basic types of nodes in the nice tree decomposition. An important type needed as a template argument is a type specifying the type of container used as a mapping from each node descriptor of the nice tree decomposition to its set of characteristics. This object is provided as a template parameter for the visitor and is called sg_full_set. According to the possible kinds of node we have:



(a) Binary case: handled by *BinaryOp* .



(b) Unary case: handled by *UnaryOp*



(c) Leaf case: handled by *LeafOp*

Figure 4.6: sg_visitor member functions w.r.t. general node types.

- **void** LeafOp(**tdVertex** *v*, **treeDec**& *T*, **ugsGraph**& *g*, **sg_full_set**&): Defines the behaviour of the algorithm on leaf nodes.
- **void** UnaryOp(**tdVertex** *v*, **treeDec**&, **ugsGraph**&, **sg_full_set**&): Defines the behaviour of the algorithm on the nodes with one child.
- **void** BinaryOp(**tdVertex** *v*, **treeDec**&, **ugsGraph**&, **sg_full_set**&): Defines the behaviour of the algorithm on binary nodes.

Further specialization of the behaviour can be done from within this framework. For example if one would like to distinguish the behaviour from unary introduce and forget nodes one could provide further methods for the visitor called case-wise from within the aforementioned methods.

4.3.3 Notions of the general algorithm

The implementation of gSolver so far does not pose restrictions regarding notions such as partial solutions and extensions. This stems from several reasons. For the partial solutions it is obvious that the corresponding object is no different than the object used for solutions. For example a partial solution for a node in subgraph isomorphism could be a mapping of the

vertices in the node's bag to a subset of the isomorphic graph. Extensions of partial solutions are themselves solutions as notions yet they have the constraint of matching the partial solutions for the nodes in their terminal graphs. More importantly though, for both cases the interface does not need any information for them at any point since decidability stems only from calculations of the characteristics. The extensions and partial solutions play an important role in establishing the correctness of the algorithm and not its actual implementation.

As described earlier, for each node in the nice tree decomposition a set of characteristics is constructed. These characteristics need not be bound to the tree decomposition structure itself and hence the design chosen appoints a new object to manipulate them. Although characteristics may have a variety of types and hence induced structures like their sets should be abstract in order to encompass them, the visitor needs only the notion of the full set of characteristics and some way to access its content, in order to provide it to the control points (leaf, unary and binary nodes). This full set usually corresponds to a mapping from a node descriptor to a set of characteristics that belong to the indicated node. The object that represents the complete set can be chosen in the template class *sg_solver* but the underlying structure is not forced since the actual elements are used only within the visitor functions. Yet in gSolver objects with these properties are implemented to facilitate and speed up programming.

Characteristics

The most important of the implemented notions represents characteristics and is called *sg_characteristic*. It is a class template that has as a template parameter a type representing the objects in the domain of solutions for the problem we want to solve. For example, if one needs to solve a 3 coloring problem the domain would be the three colors, or, in the case of subgraph isomorphism it would be vertices of the subgraph instance. The *sg_characteristic* contains a mapping from the nodes to objects of the domain and offers several convenience member functions that depend in some ways on methods of the domain. The definition of the *sg_characteristic* class is shown in Listing 4.4.

The type *s_map* in each characteristic represents the type of the mapping (i.e. from the vertices to the domain objects) and the member object *m_sMap* is the actual container. Objects of the actual container can be reached by the "[]" operator that is defined as shown in Listing 4.4. Another implemented function is the *set* member function which given the start and end iterators of a bag in the tree decomposition, and a domain object `a`, initializes the mapping mappings from the vertices of the bag to the domain object `a`.

```
/*********** sg_characteristics ****************/
template<typename sg_domain_object>
class sg_characteristic{
protected:
    typedef std::map<ugVertex, sg_domain_object> s_map;
    s_map m_sMap;
public:
    s_map& get_s_map();
    virtual sg_domain_object& operator[](ugVertex v);
    virtual bool operator<(const sg_characteristic& r) const;
    virtual void operator++(int);
    virtual sg_characteristic operator++();
    virtual size_t size();
    virtual bool operator==(sg_characteristic c){return (m_sMap==c.m_sMap);}
    void set(sg_domain_object& a, TDBag::iterator vs,
                                  TDBag::iterator vend);
    virtual ~sg_characteristic() {}
};
```

Listing 4.4: Definition of the *sg_characteristic* class.

The operator "++", can, given a domain for which a "++" operator is defined as well, cycle through possible characteristics. This in some cases provides an easy way to check the characteristics for extensions. For this to work the domain must be finite and the operator "++" of the domain should cycle through it. The advantage of this is that there is no need to generate a big set of possible characteristics and copy the ones we need since the memory consumption would be too large. In general the generating characteristics (if it is possible and not too costly in comparison to the construction of the full set) is more efficient.

Types for the set of characteristics corresponding to a node and the full set can be defined by the following types:

```
sg_plugin_types<customCstc>::sgVertexCSet
sg_plugin_types<customCstc>::sgFullCSet
```

The type *sgVertexCset* is an actual STL set of characteristics meant to be a building block of the *sgFullCSet*. The type *sgFullCSet* would correspond to the `sg_full_set` template argument for the *sg_solver* class template. It is a mapping from node descriptors to *sgVertexCSet* structures containing eventually for each node of the nice tree decomposition its set of characteristics.

Solution objects

In cases where a construction of the solution is necessary, the plug-in can define its own objects in which to store solutions. The solution objects are left on entirely on the plug-in and their construction should be implemented

within the *constructSolution* function. Direct handling of the object is not performed from within gSolver. The actual results can be processed and presented by property writers, which are described below.

4.3.4 Property writers

The purpose of property writers is to allow custom output for debugging and presentation of the solutions. In the case of the solver the property writer passed as an argument to the template is used to define the appearance of a solution. If none is provided the solution will reproduce the graph provided as input without any extra properties.

The string provided by the property writer can provide properties for nodes, edges and for the graph itself. We present the definition of the functor provided for vector properties.

```
void operator() (std::ostream& out, ugVertex v, size_t sn)
```

The properties should be passed to the output stream *out* given the vertex descriptor *v* and the parameter *sn* specifying the solution we are currently processing (if we are indeed processing one).

The output is produced in the *Dot* format of the graphviz project[3], so the options should be appropriately formatted in order to be parsed by the dot parser.

4.4 Other functionality

In this section we present information regarding input and output file types handled by gSolver and profiling functionality. The final part describes the requirements for a plug-in to be loaded by gSolver.

4.4.1 Input and Output

As mentioned earlier, the input graphs can follow the simple file format used by the Hypertree project. The graph is represented in a text file by a series of comma separated edges of the form $E(v_1, v_2)$, where E, v_1, v_2 are the names of the edge and source, and target vertices, respectively, with the last edge in the file ending with a period. The input graphs can be also in graphml format, although gSolver does not use this format to read input at the moment. Parsing specific properties of graphs can be done using special property readers implemented in the plug-in.

The input graphs used for tree decompositions follow the gml file format used by the Hypertree project[4] to output the hypertree decompositions

[3] More information about the format can be found at `http://www.graphviz.org/`

[4] Information on the Hypertree project can be found at `http://www.dbai.tuwien.ac.at/proj/hypertree/`

that are produced. More information on this format plus libraries for parsing and output can be found in `http://www.infosun.fim.uni-passau.de/Graphlet/GML/`. These libraries were also used by gSolver to parse the files. An example of a gml file follows:

```
graph [
  directed 0
  node [
    id 1
    label "{E3, E4}     {V1, V3, V4}"
    vgj [
      labelPosition "in"
      shape "Rectangle"
    ]
  ]
  node [
    id 2
    label "{E1, E2}     {V1, V2, V3}"
    vgj [
      labelPosition "in"
      shape "Rectangle"
    ]
  ]
  edge [
    source 1
    target 2
  ]
]
```

Listing 4.5: Example gml input.

There are various methods constructed to facilitate output of the structures described in section 4.2. The output for graph objects can be either in dot format or text format. All these functions can be found in the gSolveIO namespace. The functions *toDot* can output graphs, tree decompositions, and their properties as shown in subsection 4.3.4. For our ugsGraph objects there is also the possibility to construct the whole subgraph structure as an output as seen in Figure 4.2. In the case of solutions, currently the number of solutions returned can be specified, and the graphs depicting the solutions are exported all into one file. The text functions *toAscii*, apply to all the aforementioned objects and several more, such as specific vertices and edges, vertex sets and vertex vectors.

4.4.2 Profiling

Several functions to aid profiling of the algorithm are implemented.

- Timing: The timing function is called *getcputime()* and returns the time in milliseconds spent by the system on the process. The function

uses as a base the system function *getrusage*, which must exist in the system.

- Memory consumption monitoring. The function *getResidentMemory()* returns the maximum memory size of the process that resides in the main system memory.

4.4.3 Registering the Plug-in

In order for the plug-in to be used by gSolver, there are some interfacing functions needed called class factories. Their names are defined beforehand in favor of effective communication.

1. *gen_sg_solver*∗ sg_solver_create ()
   Returns a pointer to an instance of the solver defined by the plug-in.

2. **void** *sg_solver_destroy* (*gen_sg_solver*∗ *p*)
   Frees the memory of the solver pointed to by *p*.

Since these functions are dynamically loaded during runtime the interface used to do so uses pure C functions and hence the declaration of these functions should be done as presented in the following example:

```
extern "C" gen_sg_solver* sg_solver_create() {
    return new colorSolver;
}
extern "C" void sg_solver_destroy(colorSolver * p) {
    delete(p);
}
```

Listing 4.6: Class factories.

Chapter 5

Reference Plug-ins

In this section we will describe a simple algorithm for the 3 colouring problem that follows this generic approach and show how it was implemented as a plugin for *gSolver*.

5.1 3 Colouring Algorithm

The design of the algorithm follows simply from the generic definitions. We present the corresponding notions and proofs below.

5.1.1 Algorithm Description

We start by defining the appropriate notions.

Nice tree decomposition: Bodlaender type nice tree decompositions are used.

Solution: A solution s is a mapping from the set of vertices V of the input graph to the set of colors C.

Partial Solution: Given a terminal graph $H = (V', E', X')$, a partial solution p_H associated to H is a mapping from the nodes in V' to the set of colors C. For a node χ_i in the tree decomposition with bag X_i, we denote a partial solution associated to its terminal subgraph $G_{[\chi_i]}$ with p_{χ_i}.

Extension: An extension e_H of a partial solution p_H is a solution s such that p_H is its restriction to the set V' denoted by $s|_{V'}$.

Characteristic: A characteristic α_{χ_i} of a partial solution of a node χ_i is the restriction of the partial solutions to the bag X_i. We will give a similar proof to the one given in [Bod97] showing that this definition of characteristic corresponds to the notion presented in chapter 3.

In order to show that the notion of characteristic we just presented is correct we need to show that given two partial solutions with the same characteristic one has an extension if and only if the other has an extension.

Lemma 5.1.1. *Take two partial solutions p_{χ_1} and p_{χ_2} that have the same set of characteristics C_χ. We show that p_{χ_1} has an extension if and only if p_{χ_2} has an extension.*

Proof. If p_{χ_1} has an extension e_{χ_1} then there must be a solution s whose restriction to the nodes of $G_{[\chi_1]}$ is p_{χ_1} and moreover its restriction to the nodes of the bag X_1 of χ_1 will be C_χ. We know that the only vertices of $G_{[\chi_2]}$ that are adjacent to vertices in any subset of $V(G) - V(G_{[\chi_2]})$ are those in X_2. This is due to the fact that there must be some bag containing both vertices, say v, u, and all the nodes in the path from this bag to any successor of χ_2 must contain v or u. Hence for χ_1, χ_2 all adjacent nodes must be in $X_1 = X_2$

By the above observation, we have that for the nodes in $V_d = V(G) - (V(G_{[\chi_1]}) \cup V(G_{[\chi_2]}))$, it must be $p_{\chi_2}|_{V_d} = p_{\chi_1}|_{V_d}$. And since the nodes in $G_{[\chi_2]} - X_2$ are disjoint from the rest, the partial solution p_{χ_2} is correct and has as extension the union of the mappings p_{χ_1} and e_{χ_1} restricted to the nodes not in $V(G_{[\chi_2]})$. □

What remains to be shown is that for each of the types of nodes in our tree decomposition it is possible to efficiently calculate the set of characteristics from its children. We can then proceed in describing the procedure for constructing the solutions.

Decision procedure for the 3 colouring algorithm

The case of leaves is trivial since any partial solution is just a mapping from the vertex of its bag to the set of colors and coincides with the set of characteristics.

Definition 5.1.2. Let χ_i be an introduce node in a Bodlaender type nice tree decomposition of $G = (V, E)$. Furthermore, let (X_i, E_{χ_i}) be the induced subgraph $G[X_i]$ and χ_j be the child of χ_i such that $X_i = X_j \cup \{v\}$. Then a mapping $\alpha : X_i \mapsto \{1, 2, 3\}$ belongs in $\mathcal{C}(\chi_i)$ iff

1. for all $u \in X_i \setminus \{v\}, u \mapsto \alpha(u) \in \mathcal{C}(\chi_j)$ and
2. for all $u' \in X_j$, if $(u, u') \in E_{\chi_i}$ then $a(u) \neq a(u')$.

Lemma 5.1.3. *Definition 5.1.2 (i) is correct and (ii) can be constructed in constant time.*

Proof.

(i) For each partial solution p_{χ_j} there is a partial solution of χ_i whose restriction to $G_{[\chi_j]}$ is equal to p_{χ_j}. Also, for any partial solution p_{χ_i}, the restriction $p_{\chi_i}|_{G_{[X_j]}}$ is a partial solution of p_{χ_j}. Thus the characteristics of χ_j correspond one to one to the restrictions of the characteristics of χ_i to X_j.

Observe also that each characteristic of χ_i (as a restriction of a partial solution) must also obey condition 2 since any partial solution is a valid colouring. Finally any mapping of v to $\{1, 2, 3\}$ obeying 2, must be a valid colouring of the terminal subgraph and thus there is a partial solution containing it.

(ii) Since the treewidth is bound, say by tw, the number of nodes to check for adjacency is also bound by tw. Furthermore the size of the set of characteristics for each node χ_i is also bound by 3^{tw}.

□

The corresponding constructions and their proofs for forget and join nodes follow.

Definition 5.1.4. Let χ_i be a forget node, (X_i, E_{χ_i}) the terminal subgraph $G_{[\chi_i]}$, and χ_j the child of χ_i such that $X_j = X_i \cup \{v\}$. Then a mapping $\alpha : X_i \mapsto \{1, 2, 3\}$ belongs in $\mathcal{C}(\chi_i)$ iff it is a restriction of a mapping $\alpha' \in \mathcal{C}(\chi_j)$.

Lemma 5.1.5. *Definition 5.1.4 is correct and the set can be constructed in constant time.*

Proof.

(i) It is obvious that α is a valid colouring, since it is a restriction of α' to X_i. Furthermore any partial solution of χ_j is a partial solution of χ_i and vice versa since they are related to the same terminal subgraphs.

(ii) The restriction can be done in constant time w.r.t. tw since one vertex is removed from each mapping and both the number of mappings as well as the number of vertices are bound.

□

Definition 5.1.6. Let χ_i be a join node, (X_i, E_{χ_i}) the induced subgraph $G[X_i]$ and χ_j, χ_k children of χ_i. Then a mapping $\alpha : X_i \mapsto \{1, 2, 3\}$ belongs in $\mathcal{C}(\chi_i)$ iff $\alpha \in \mathcal{C}(\chi_j)$ and $\alpha \in \mathcal{C}(\chi_k)$.

Lemma 5.1.7. *Definition 5.1.6 is correct and the set of characteristics can be constructed in constant time.*

Proof.

(i) First, observe that any characteristic α belonging to the intersection of $\mathcal{C}(\chi_j)$ and $\mathcal{C}(\chi_k)$ is a characteristic of χ_i, since there must be a partial solution p_{χ_j} and a partial solution p_{χ_k}, such that $p_\psi|_{X_i}$ is equal to α. This is a consequence of the connectedness property of tree decompositions, from which we can derive that for a node w belonging to the terminal subgraph of χ_j but not in the terminal subgraph of χ_k, there is no edge between w and any of the nodes in the terminal subgraph of χ_j. Furthermore, observe that any partial solution of χ_i restricted to the nodes of $G_{[\chi_j]}$, or to the nodes of $G_{[\chi_k]}$ is also a partial solution of χ_j or χ_k respectively.

(ii) The intersection of the two sets, for this specific case, given the fact that the size of each set is bound, can be considered to take place in constant time.

□

Constructing the colourings

The construction of solutions is performed in a top down manner, determined by the kind of node we encounter while processing the tree decomposition. Starting from the root we set as possible solutions all the characteristics of the root node. Let P_{χ_i} be the set of partial solutions constructed associated to the terminal subgraph of G, $G[\overline{X_i}]$, where $\overline{X_i}$ is the set of vertices in the bags of all nodes in the tree decomposition that are not successors of χ, then, for the various types we have:

Forget Node Let χ_i be a forget node, χ_j its child and v the vertex not in χ_i. Assuming we have all partial solutions of the subgraph $G[\overline{X_i}]$, the set of partial solutions of $G[\overline{X_j}]$ consists of partial solutions of $G[\overline{X_i}]$, extended by a valid coloring of v. But from the construction of the characteristics this condition was conserved for the nodes in X_i and from the connectedness condition of tree decompositions, there is no node $u \in \overline{X_i} \setminus X_i$ such that $(u, v) \in E$. Thus the partial solutions P_{χ_j} of $G[\overline{X_j}]$ can be constructed from the partial solutions P_{χ_i} of $G[\overline{X_i}]$ by extending each $p \in P_{\chi_i}$ with any restriction of X_i equal to $c_k|_{X_i}$, for $c_k \in \{c_1, \ldots, c_n\}$ characteristics in χ_j of n partial solutions $p_1 \ldots p_n$, such that $p_k(u) = p(u), u \in X_i$ and $p_k(v) = c_k(v)$.

To give an example, let p_χ be a partial solution for node χ with bag $\{v_1, v_2\}$, such that

$$p_\chi = \{v_1 \mapsto 1, v_2 \mapsto 3, v_3 \mapsto 2\}$$

and characteristics

$$c_1 = \{v_1 \mapsto 1, v_2 \mapsto 3, v_4 \mapsto 2\} \text{ and } c_2 = \{v_1 \mapsto 1, v_2 \mapsto 3, v_4 \mapsto 3\}.$$

The set of partial solutions should now be:

$$p_{\chi_1} = \{v_1 \mapsto 1, v_2 \mapsto 3, v_3 \mapsto 2, v_4 \mapsto 2\}$$

and

$$p_{\chi_2} = \{v_1 \mapsto 1, v_2 \mapsto 3, v_3 \mapsto 2, v_4 \mapsto 3\}$$

Introduce Node This case is much easier since if χ_i is an introduce node and χ_j its child, then $X_i = X_j \cup \{v\}$. Which means that the solutions restricted to χ_i are a subset of χ_j and thus their valid partial solutions for $G[\overline{X_i}]$ must be the same.

Join Node This case is similar to the above as we know for χ_i a join node and χ_j, χ_k its children $X_i = X_j = X_k$. The solutions for the children should clearly be those of the parent.

5.1.2 Implementation

The three coloring algorithm presented was implemented as a plug-in for our system. We will present some information regarding its implementation and several test results.

A first step is to create a class called *colorSolver* based on the *sg_solver* template. Its collaboration graph is shown in Figure 5.1. The details of the

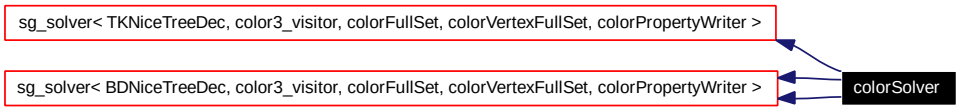


Figure 5.1: Collaboration graph for the *colorSolver* class.

template parameters are given below:

1. The *BDNiceTreeDec* class, that represents Bodlaender type nice tree decompositions. It is already implemented in *gSolver*.

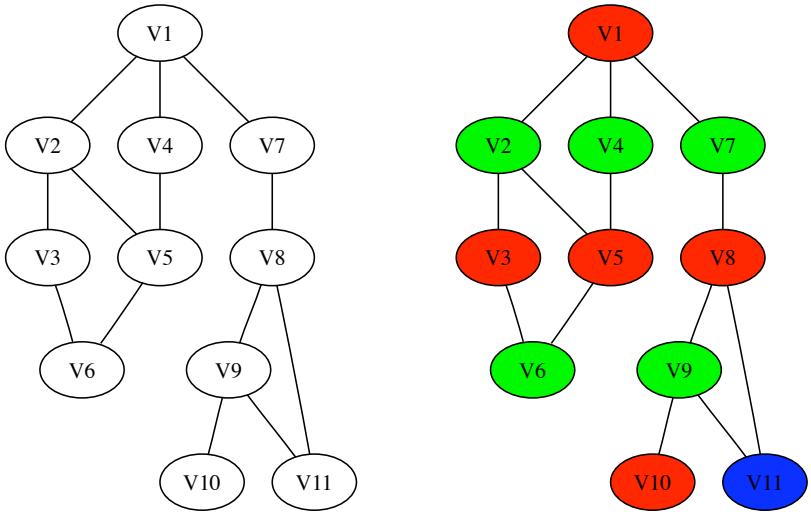
2. A *color_visitor* class, which is the implementation of an *sg_visitor*. The color_visitor mainly implements the algorithms presented in the previous subsection. That is, it only describes how the characteristics should be constructed for the nodes of the tree decomposition. For leaves the process is straightforward since we just have one vertex to map to each of the three colors. For the rest of the nodes we follow the approach described above using the set operations provided. An observation one can make here is that the distinction between *introduce* and *forget* nodes has to be done by the plug-in.

3. A *colorFullSet* class, for the full set of characteristics. In order to construct this class we need to implement within the plug-in an object for characteristics. That object is called *colorCharacteristic* and templates the *sg_characteristic* class with a domain - also implemented - which is comprised of a representation of the three colors and some convenience methods to cycle between colors and compare them. After implementing those two classes the *colorFullSet* class corresponds directly to the predefined by the color characteristics type: *sg_plugin_types*<*colorCharacteristic*>::*sgFullCSet*.

4. A *colorVertexFullSet* class, defines the set of characteristics for a node in the tree decomposition. This class, corresponds directly to the type: *sg_plugin_types*<*colorCharacteristic*>::*sgVertexCSet*.

5. A *colorPropertyWriter* class, extends the *sg_property_writer* and arranges the output of solutions with the colors found.

The implementation of the plugin takes about 600 lines of code. The exact output of the plugin can be seen in Figure 5.2.



(a) Input graph in DOT language

(b) One of the possible colourings found.

Figure 5.2: Example of input-output of the color plugin using *properyWriter* to present the colourings found.

Notes

An important observation here regards the case of the introduce node, where the additional vertex in the bag of the parent is disjoint from all other ver-

tices. The valid characteristics in this case would triple and several consequent cases like this on the tree decomposition structure would make the size of the colorings grow exponentially large thus inducing a correspondingly large memory consumption. This may be handled by several ways especially when the solutions need not be built. In our implementation we simply add a new color which marks that the colouring of the vertex can be any, and transformed slightly the algorithm in the unary and binary cases as well as in the construction of solutions to reflect this.

Another observation is that the notions of partial solutions and extensions are indeed not needed as separate types in this case, since they can be encompassed with the mappings used for expressing characteristics of colorings.

5.1.3 Testing

We did various tests to observe the three colouring algorithm with respect to several parameters. What is of interest is the complexity with respect to the order of the graph given graphs and their tree decompositions of certain width. Of interest is also the behaviour with respect to the width of the input tree decompositions given graphs of equal size. We tested about 9.000 instances of 3-colorable graphs of sizes from 20 nodes to 60 nodes and various widths from 3 to 20 using Joseph Culberson's graph generators[1]. There is a certain variability in the sample size as there is no minimal guarantee on the order of the graph from the generators (that is, requested graphs of order 40 had a maximal order of 40). The graphs were categorized by width as computed using the software from the Hypertree project. The cases where there is no colouring are in principle much lighter since the computation will stop before the root of the tree decomposition. Figures 5.3, 5.4, 5.5, 5.6, show the time needed to calculate the full set of characteristics for various ranges of widths of the input tree decompositions. The time measured corresponds only to the time needed for the calculation of the full set, the construction of the nice tree decompositions and the terminal subgraph structure are not included. The boxplot plots on the left side, provide information on the differences of the results regarding the dispersion, the plots on the right side present in more detail the median of the processing times calculated. We should emphasize here that the calculations were not performed using tree decompositions of width equal to the treewidth of the graph but rather an approximation.

The first two graphs present a certain linearity when observing the median values though the dispersal of the results is expanding exponentially. For larger tree decomposition widths this image changes, the distribution of

[1]More on the generators can be found in `http://web.cs.ualberta.ca/~joe/Coloring/Generators/generate.html`

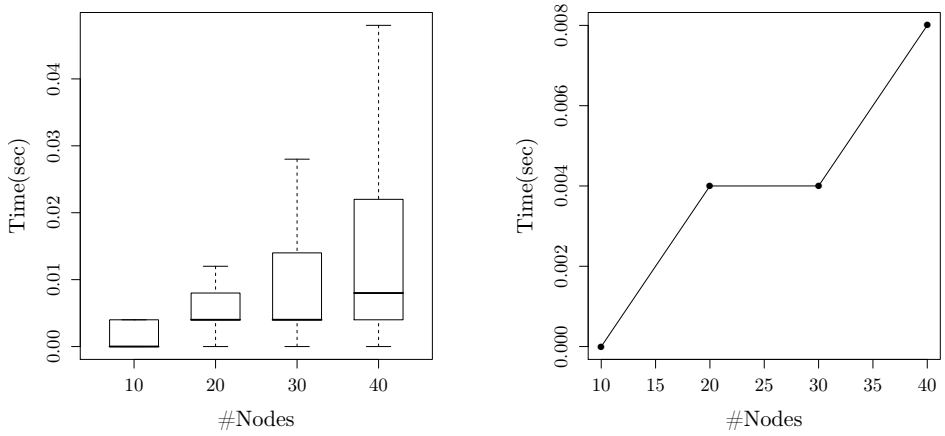


Figure 5.3: Running time for tree decompositions of width $3 \leq tw \leq 5$.

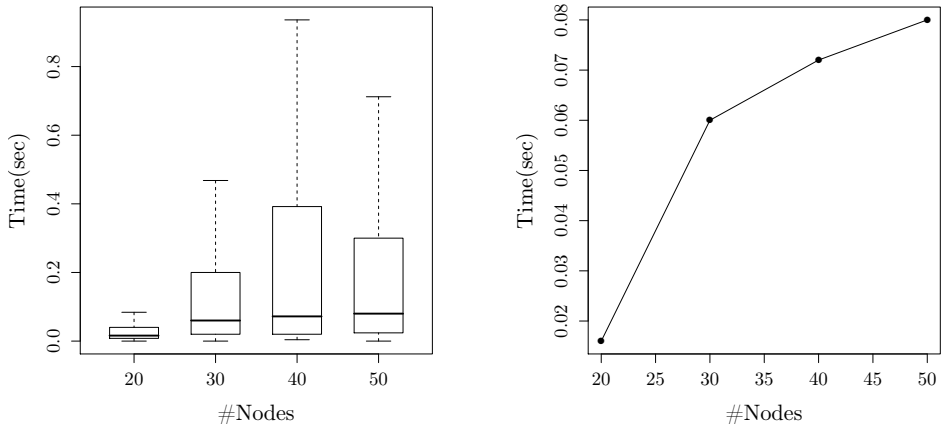


Figure 5.4: Running time for tree decompositions of width $5 \leq tw \leq 9$.

the calculation expands and for the medians one can observe a large variability. In some cases the median values for larger orders seem to drop. This is due to a certain correlation between the number of edges of the graph and the width of the tree decomposition. This observation can be seen clearly in Figure 5.7. Graphs with a relatively large amount of edges will allow less possible characteristics per node and thus the processing time will be reduced.

An issue we would like to raise here is related to the amount of characteristics per node. As mentioned earlier in the case where consecutive introduce nodes occur and the additional nodes are disjoint the number of characteristics triples by each occurrence. It is obvious that the construction of the nice tree decomposition plays a very important role in reducing the growth during construction rather than reducing the growth later. On the

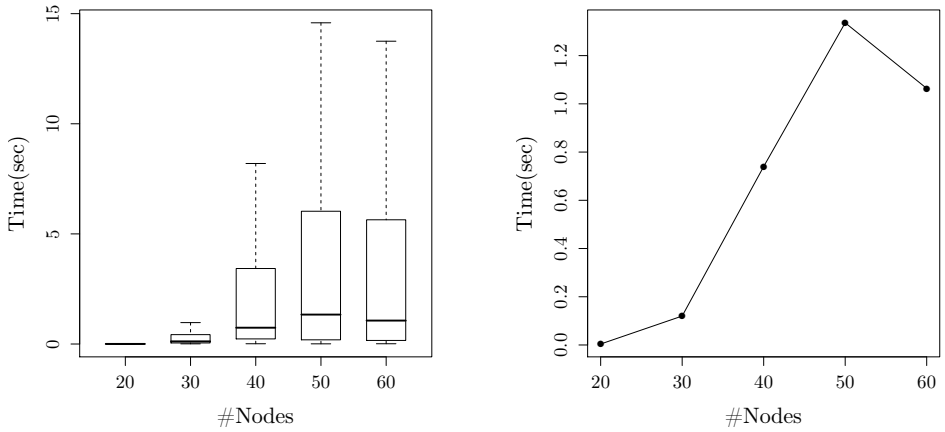


Figure 5.5: Running time for tree decompositions of width $10 \leq tw \leq 14$.

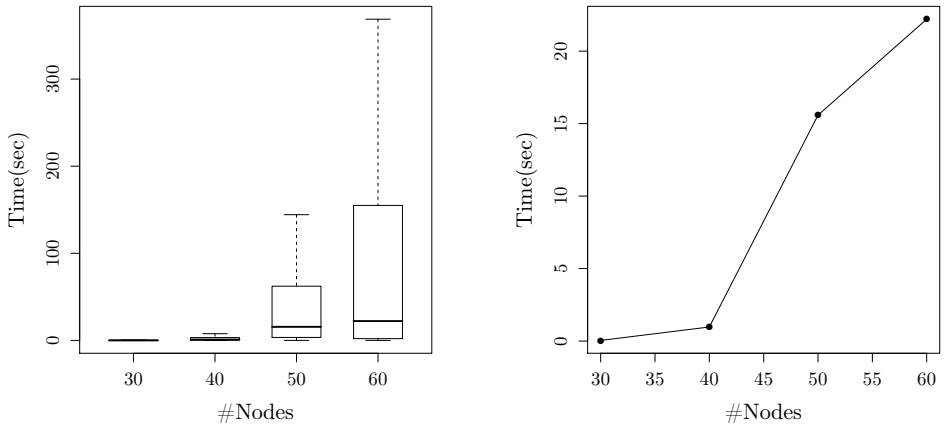


Figure 5.6: Running time for tree decompositions of width $15 \leq tw \leq 19$.

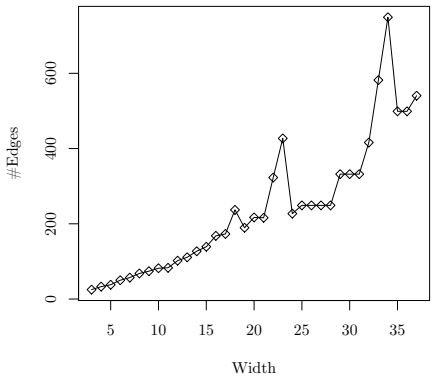
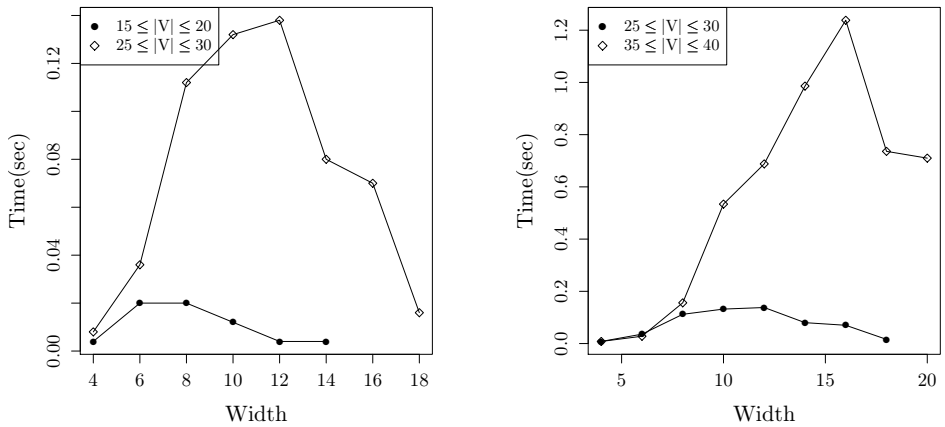


Figure 5.7: Width of tree decompositions with respect to the number of edges.

other hand, the number of characteristics of the root is, in the best case, at least equal to the number of the solutions for the graph. That is, the variation should be expected to be large for graphs with more solutions. Especially using Bodlaender type nice tree decompositions, which enables us on one hand to easily start coloring leafs with one vertex in the bag, but on the other hand is prone to the exponential growth unless the construction is slightly altered.

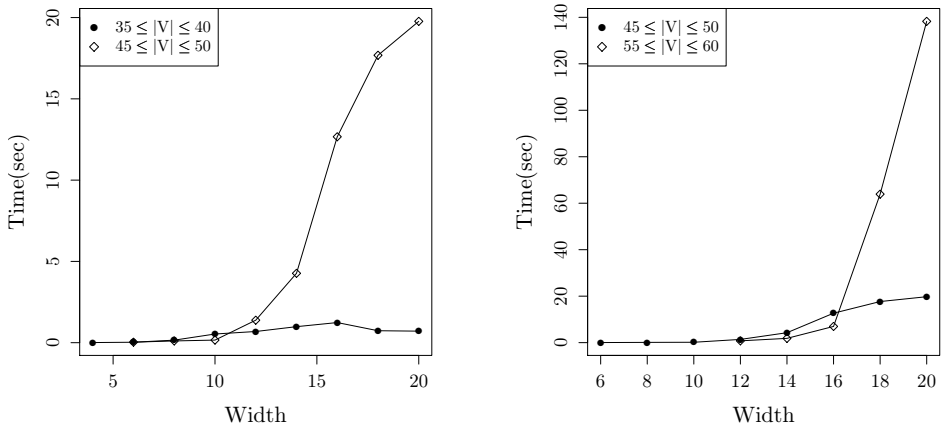
Another perspective is the behaviour of the algorithm with respect to the width of the tree decompositions. The results are presented in Figure 5.8 and Figure 5.9.



(a) Order of graph: $|V| \approx 20, |V| \approx 30$ (b) Order of graph: $|V| \approx 30, |V| \approx 40$

Figure 5.8: Processing time w.r.t. the width of the tree decomposition.

For these graphs we present the median values for the tests. Each width that has been tested has approximately 100 graph instances and the values



(a) Order of graph: $|V| \approx 40, |V| \approx 50$ (b) Order of graph: $|V| \approx 60$

Figure 5.9: Processing time w.r.t. the width of the tree decomposition.

in the figure represent the median of the calculation time. We can observe that there is an exponential growth with respect to the width, until a point where we observe a decrease. This confirms our earlier observation that the calculation time decreases for larger widths due to the correlation between the calculated width and the amount of edges of the instance.

Regarding the construction of solutions, even in graphs of order 40, the sometimes excessive number of possible solutions exceeded the allocation size of the STL containers leading to a abrupt stop of the calculation. Some results were produced and are shown in Figure 5.10 for graphs of order up to 20. The drop w.r.t. time and width is due to to effects. One is the correlation between treewidth and the number of edges presented earlier, which reduces the number of possible solutions. The other is due to the fact that only a few instances were actually solved in the larger cases which resulted in a rather small testing sample for these cases. The behaviour for width up to 14 is depicted in Figure 5.10(b).

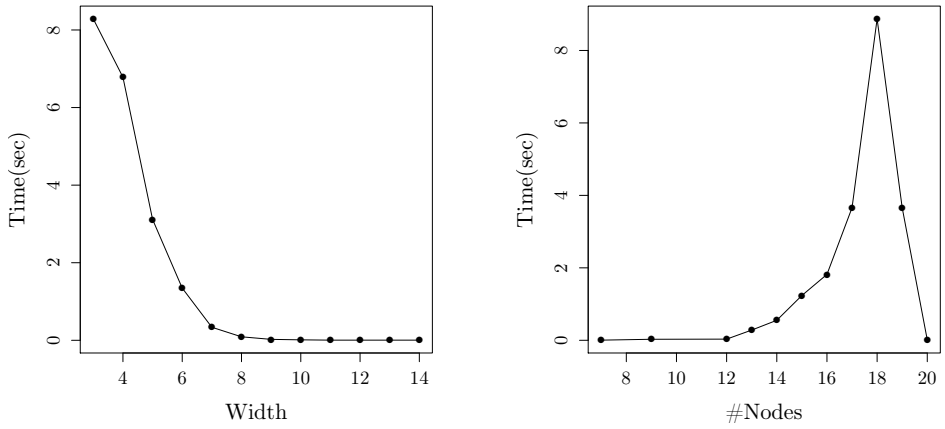


Figure 5.10: Processing time for solutions.

Chapter 6

Experiments

In this chapter we present several experiments regarding the efficiency of gSolver. It is of importance that the fundamental objects assumed in the algorithm, such us nice tree decompositions and terminal subgraphs, are constructed efficiently. We will focus on the time needed for construction of the nice tree decomposition with respect to the width of the original tree decomposition and the time needed for the constuction of the structure of terminal subgraphs. The tests were performed on Mac OS X 10.4.11, running on a MacBook with an Intel Core Duo at 2GHz with 2GB of RAM. The graph instances are a superset of those tested in the 3 colouring example. Graphs of order 20-90 were tested with treewidths ranging from 3 to 60 depending on the order. For each order and each treewidth, approximately 100 test cases were generated.

6.1 Testing construction of basic objects

6.1.1 Nice tree decompositions

The following tests regard the construction of the three implemented styles of nice tree decompositions that is Bodlaender, Kloks and HN style nice tree decompositions. The time measured includes the time for triangulation of the initial graph, construction of the k-tree, and the actual construction of the nice tree decomposition from the k-tree according to the algorithm induced by the constructive proofs for each case as presented in 2.2.8. In each plot the values depict the median of the test cases for each of the styles considering the width as a parameter.

When considering the width as a fixed parameter one can observe that the behaviour of the construction is linear, as shown figure 6.2(a). This is also visible for greater widths. We had expected that for "small" values of the parameter the algorithms behave in a linear fashion and this was confirmed by the results. Another observation regarding the actual types of nice tree decomposition is that the Bodlaender style is always higher than

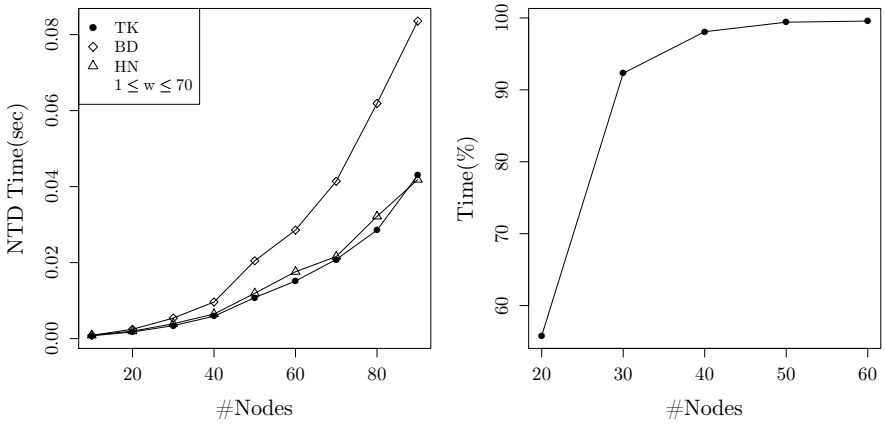


Figure 6.1: Comparisons between various tree decomposition styles (a), and percentage of time used only for the computation of the characteristics w.r.t the total time (b).

the other two types. This is due to the post processing step which also increases the number of nodes in the tree decomposition. We also expect a non-linear general complexity which is also the case considering the results in Figure 6.1(a).

Another test that we deem important is how much time does the construction of the tree decompositions need, compared to the computation of the full set of characteristics. This is important since they are used in each plug-in and should be efficient. The relation can be clearly seen in Figure 6.1(b), where the percentage of time spent in for the calculation of the full set w.r.t to the total time is shown. For graphs larger than 40 nodes almost 100% of the total computation time is spent on the calculation of characteristics. Observe also that for graphs of sizes 10-30 the calculation time is anyway indeed very small. We thus can conclude that the overhead of the base system would not hinder the actual calculations.

6.1.2 Terminal Subgraphs

Similar tests were done for the construction time needed for the terminal subgraph structure corresponding to the nice tree decomposition of the graph. The results of the tests for small ranges of tree width are shown in Figure 6.3. The behaviour is clearly within linear bounds or less in all cases tested.

What is different here is that the construction remains linear without considering the width. This is visible in Figure 6.3(e).

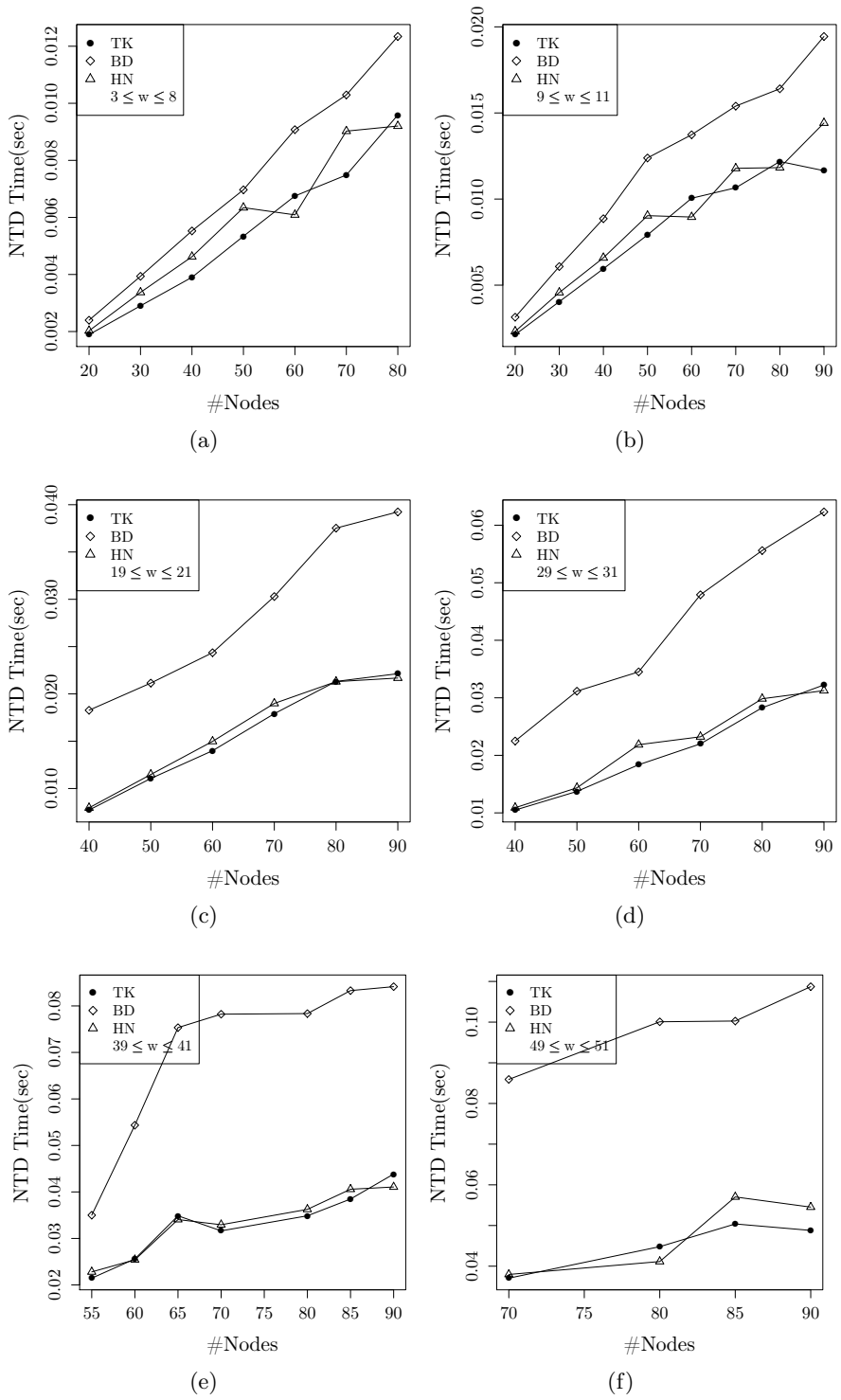


Figure 6.2: Construction time for the implemented nice tree decompositions.

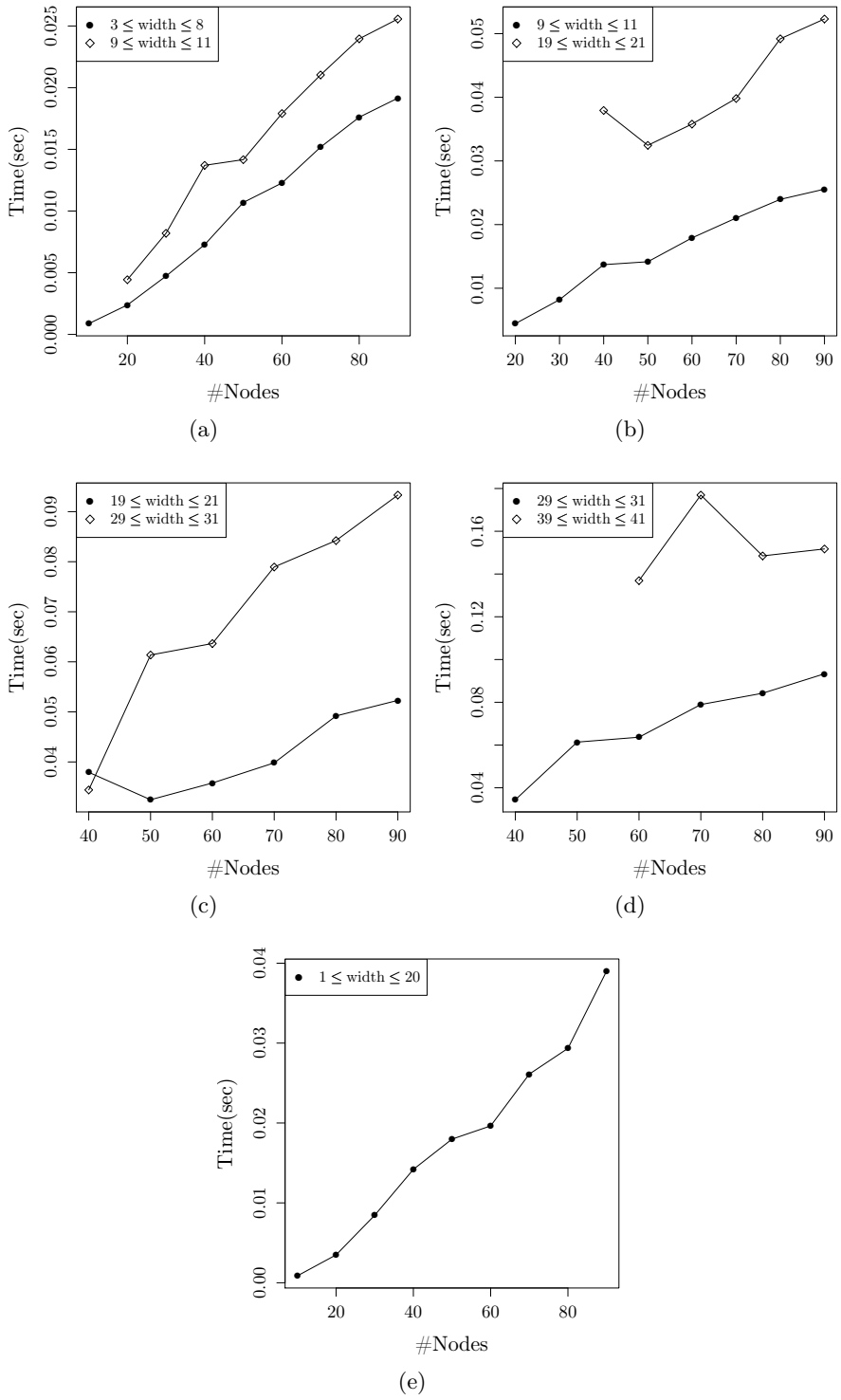


Figure 6.3: Construction time for the terminal subgraph structure.

Chapter 7

Conclusion

We have implemented a generic system for algorithms designed for graphs of bounded treewidth. The base implementation appears to be efficient with respect to possible calculations of the full sets of characteristics and generic enough to accept definitions of many kinds of algorithms for various problems. Several issues related to efficiency of gSolver still remain though.

7.1 Future work

One important factor that inhibits the calculation of the full set of characteristics is the redundancy of the nice tree decomposition with respect to its structure. As observed during the experiments, several bags are kept containing sets of instances of the same characteristics which need to be processed, thus expanding both the memory needed and the calculation time. A solution that we would be eager to implement in the system is the anchor technique [BNU04] mentioned also in chapter 3, which tackles this problem by calculating an approximate solution to a weighted set covering problem of the bags of the nice tree decomposition, This, in turn, provides an approximation to the necessary calculations for the full set.

Another important issue to be addressed is the reduction of the memory needed for the calculations, a problem even more prominent when the construction of a solution is required. A general representation of characteristics is performed with a mapping from the nodes of the graph to some domain. It is usually the case, that many characteristics contain multiple instances of the same mapping which in return leads to larger memory allocation than necessary. This issue can be in some degree addressed using more sophisticated containers using pointers instead of the actual mappings.

Regarding nice tree decompositions, a construction algorithm, should possibly be derived using only its description. What is interesting at this point is that using the description, a theorem prover could be used to prove termination, then we can process the description and generate the appropri-

ate code. Something similar might be possible just to ensure correctness of the algorithm, based on the formulas presented in chapter 3, yet complete code generation should not be possible for this case.

Bibliography

[AP89] S. Arnborg and A. Proskurowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete Appl. Math.*, 23(1):11–24, 1989. 13

[BLW87] M. W. Bern, E. L. Lawler, and A. L. Wong. Linear-time computation of optimal subgraphs of decomposable graphs. *J. Algorithms*, 8(2):216–235, 1987. 13

[BNU04] N. Betzler, R. Niedermeier, and J. Uhlmann. Tree decompositions of graphs: Saving memory in dynamic programming. *Electronic Notes in Discrete Mathematics*, 17:57–62, 2004. 17, 55

[Bod93] H.L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 226–234, 1993. 14

[Bod97] H.L. Bodlaender. Treewidth: Algorithmic Techniques and Results. *Mathematical Foundations of Computer Science 1997: 22nd International Symposium, MFCS'97, Bratislava, Slovakia, August 25-29, 1997: Proceedings*, 1997. v, vii, 1, 7, 13, 39

[Bod05] H.L. Bodlaender. Discovering treewidth. *SOFSEM*, pages 1–16, 2005. 14

[Cou90] B. Courcelle. Graph rewriting: An algebraic and logic approach. *Handbook of Theoretical Computer Science*, 2:194–242, 1990. v, vii, 1, 6

[Die05] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, August 2005. 3

[DT06] F. Dorn and J. Telle. Two birds with one stone: The best of branchwidth and treewidth with one algorithm. *LATIN 2006: Theoretical Informatics*, pages 386–397, 2006. 11

[FG65] D.R. Fulkerson and O.A. Gross. Incidence matrices and interval graphs. *Pacific J. Math*, 15(3):835–855, 1965. 4

[GPW07] G. Gottlob, R. Pichler, and F. Wei. Monadic datalog over finite structures with bounded treewidth. In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 165–174, New York, NY, USA, 2007. ACM. 1

[Gro07] M. Grohe. Logic, graphs, and algorithms. In J. Flum, E. Grädel, and T. Wilke, editors, *Logic and Automata: History and Perspectives*, number 2 in Texts in Logic and Games. Amsterdam University Press, Amsterdam, 2007. 1

[HN02] M. T. Hajiaghayi and N. Nishimura. Subgraph isomorphism, log-bounded fragmentation and graphs of (locally) bounded treewidth. *Mathematical Foundations of Computer Science 2002*, pages 305–318, 2002. 13

[Klo94] T. Kloks. *Treewidth: Computations And Approximations*, volume 842/1994 of *Lecture Notes in Computer Science*. Springer, 1994. 5, 8, 9, 10

[KMS02] N. Klarlund, A. Möller, and Michael I. Schwartzbach. Mona implementation secrets. *Int. J. Found. Comput. Sci.*, 13(4):571–586, 2002. 1

[Mar06] H. Maryns. On the implementation of tree automata: Limitations of the naive approach. *Proc. 5th Int. Treebanks and Linguistic Theories Conference (TLT 2006)*, pages 235–246, 2006. 1

[Ros70] D.J. Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597–609, 1970. 4

[RT75] D. Rose and R. Tarjan. Algorithmic aspects of vertex elimination. In *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing*, pages 245–254, New York, NY, USA, 1975. ACM. 5

Index