FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Adding a treewidth support to the Boost Graph Library
Student:	Václav Král
Supervisor:	RNDr. Ondřej Suchý, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2020/21

Instructions

- Get familiar with the software policies of the Boost Graph Library [1].
- Get familiar with the concept of tree decomposition and tree-width [2].
- Analyze the possibilities to add support for tree decompositions and tree-width into the library, following its software policies.
- Implement an algorithm for obtaining a tree decomposition.
- Implement an algorithm for turning an arbitrary decomposition into a nice one of the same width.
- Implement a simple algorithm that uses dynamic programming over tree decompositions.
- Keep the algorithms as generic as possible, using the template and hook approaches usual in the library.

References

[1] https://www.boost.org/doc/libs/1_66_0/libs/graph/doc/table_of_contents.html
[2] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, Saket Saurabh: Parameterized Algorithms. Springer 2015, ISBN 978-3-319-21274-6, pp. 3-555

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 22, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Treewidth support for the Boost Graph Library

Václav Král

Department of Software Engineering
Supervisor: RNDr. Ondřej Suchý, Ph.D.

May 6, 2020

Acknowledgements

First of all, I would like to thank my supervisor RNDr. Ondřej Suchý, Ph.D. for all his advice, patience, and the time he dedicated to helping me with this thesis. I would also like to thank my friends and my family who helped me during my studies and provided psychological support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 6, 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Král, Václav. *Treewidth support for the Boost Graph Library.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstract

The aim of this thesis is to extend the C++ Boost Graph Library (BGL) with algorithms for obtaining a tree decomposition of a graph and an example of an algorithm, that uses the tree decomposition. In this thesis the reader will get familiar with not only the algorithms and their usage, but also with the library itself. Further in this thesis the successful implementation of the algorithms is discussed, where the main focus is the compliance with conventions of BGL and rules of generic programming. The quality of implementation and possible enhancements are discussed at the end of the thesis. The result is a working extension of BGL.

Keywords Boost Graph Library extension, tree decomposition, treewidth, weighted independent set, generic programming, C++

Abstrakt

Cílem této bakalářské práce je rozšířit C++ knihovnu Boost Graph Library (BGL) o algoritmy pro získávání stromové dekompozice grafu a příklad algoritmu, který využívá tuto stromovou dekompozici. V práci se čtenář seznámí nejen s jednotlivými algoritmy, ale i se samotnou knihovnou, která bude rozšiřována. Dále se práce zabývá samotnou úspěšnou implementací algoritmů, kde je kladen důraz především na dodržení konvencí knihovny BGL a pravidel generického programování. Závěrem práce hodnotí kvalitu implementace a navrhuje možná zlepšení. Výsledkem práce je funkční rozšíření BGL.

Klíčová slova rozšíření Boost Graph Library, stromový rozklad, šířka rozkladu, nezávislá vážená množina, generické programování, C++

Contents

List of Figures

List of code snippets

List of Tables

Introduction

Even to this day, there are many problems that we are still unable to deterministically solve in polynomial time. While their computational complexity remains very high, some of these problems can be solved more efficiently if the input resembles a certain structure, e.g., the input graph is a tree. An example of such a problem is the computation of the *maximal independent set*—naive brute force algorithm has an exponential complexity [1], however if the input graph is a tree, the problem can be solved in a linear time [2]. This raises the question „*What if the input graph is almost a tree?*“.

The similarity of a graph and a tree can be formally described by the concept of *treewidth*—the smaller the *treewidth*, the more the graph resembles a tree. *Treewidth* is defined by something called a *tree decomposition*. We will get more familiar with both of these terms in the first chapter.

Goals

Many natural problems can be solved in linear time if the input graph has a small *treewidth*. In this thesis we will focus on extending the C++ library *Boost Graph Library* (*BGL*) with algorithms that are used to compute the *treewidth* of a graph or use the computed *treewidth* to run more efficiently. *BGL* was chosen because it is the most established graph library in the C++ community and it provides all the necessary tools that will help us to implement our algorithms. We will put large emphasis on:

1. **Generic programming**. We will be using the C++ templates to maximize the code reuse and provide more generic interface.
2. **Testing**. To ensure the correctness of provided algorithms, we will be using *Unit tests*.
3. **Documentation**. Interface of provided functions will be documented in order to provide a reference guide for the potential users.

Structure

Structure of this thesis follows the traditional *Software Development Life Cycle* (SDLC)—Analysis, Design, Implementation, Testing, and Documentation.

In the first part of Chapter 1 we will get familiar with BGL, its characteristics and behaviour. In the second part we will describe some of the key notions that are necessary to understand treewidth or tree decomposition.

In Chapter 2 we will discuss requirements and code architecture. Also we will introduce the algorithms provided by our library extension (their core and used data structures).

In Chapter 3 we will be fulfilling requirements presented in previous chapter. It will be divided into three sections (one for each algorithm). In each of these sections we will discuss the interface along with the implementation of the function.

Chapter 4 will be divided into two parts. In the first one we will take a look at the testing of functions provided by our library extension. Also we will review the fulfilment of the requirements presented in Chapter 2. The second part is dedicated to the documentation of those functions.

Chapter 1

Analysis

In the first part of this chapter, we will discuss characteristics and behaviour of the Boost Graph Library.

In the second part, we will introduce some key notions to help understand the algorithms. Their implementation will be described in the next chapter—those are algorithms for obtaining (nice) tree decomposition and for the maximum weighted independent set problem.

1.1 Boost Graph Library

Boost Graph Library (BGL) is a part of Boost—a large set of libraries (ranging from math utility, pseudo-random number generator (PRNG), or multi-thread to filesystem or testing libraries) written in C++. Several of those libraries have been accepted into the C++11 [3] or C++17 [4] standards. Most of them are licensed under the *Boost Software License*, which is free and open-source license.

BGL itself is a library focusing on graphs and graph algorithms, which are mathematical abstractions used to solve many types of problems. It provides not only a large collection of interfaces for such algorithms, but also varying data structures used for representing the graphs—we will discuss those later on.

Important feature of this library is that it does not need to be built in order to be used, because it is header-only library.

We will now describe the key aspects of the library.

1.1.1 Genericity

BGL puts heavy emphasis on genericity of its algorithms. Each algorithm and container is written in a data structure neutral way in order to give user full control over data structures he wants to work with. This leads to a reduction of code size of the library from $\mathcal{O}(A \cdot D)$ to $\mathcal{O}(A + D)$, where A is a number of

algorithms and D is a number of data structures. In real situation, where we have 40 algorithms and 10 data structures that would mean difference between 400 functions and 50 functions, which is quite notable. There are three ways in which BGL is generic [5]:

Algorithm/Data Structure Interoperability. Interface of algorithms abstracts away details of particular data structure. For example, there are three different ways of traversing over graph—traversal of all vertices, of all edges, and along the adjacencies of vertices[1]. Each of these patterns has a separate iterator. Thanks to this generic interface template functions such as `depth_first_search()`[2] are allowed to work on a variety of graph data structures, where each of them has different ways of storing vertices or edges.

Extension through Visitors. BGL uses notion of *visitors*, which are function objects with multiple methods, that are triggered at specific event points of traversal (e.g., discovering vertex, examining vertex, finishing vertex).

Vertex and Edge Property Multi-Parametrization. In order to associate values („properties“) with either vertices or edges, we have to define corresponding container, that will hold such values. This container is called *Property Map*, which is an abstraction over existing containers like `std::map`, `std::vector`, etc. Each property has its own separate *Property Map*.

1.1.2 Graph concepts

Graphs are the core data structure of the library. Their interface is designed to define how a certain type of graph should be manipulated with in a data structure neutral fashion. This is the reason why graph interface is not a single graph concept, instead, it is factored in smaller pieces where each piece summarizes requirements for a particular algorithm. Another motivation to factor graph concepts into smaller pieces is that most of the algorithms require only a small subset of all possible graph operations. This way the users can choose which of these smaller graph concepts fits their needs better while using only the needed minimum of the whole interface of *Graph*, which leads to increased reusability of algorithms. Figure 1.1 shows relations between graph concepts defined in the BGL.

[1] Traversing along the adjacency of vertex v is basically iterating over all neighbouring vertices of v.

[2] Function defined in BGL that performs depth-first traversal of a graph.

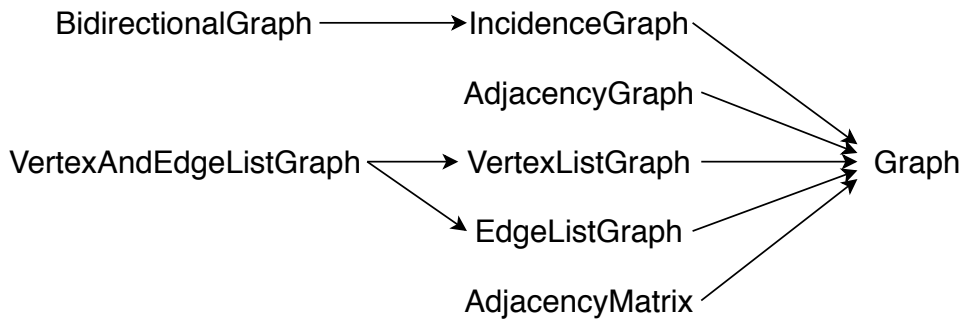


Figure 1.1: The graph concepts and refinement relationships (the image is taken from Boost website [6]).

Bellow you can find a description for each graph concept from Figure 1.1:

Graph concept. This concept contains only a few requirements that are common for all other graph concepts. Examples of such requirements are `vertex_descriptor` and `edge_descriptor` (indices used to address vertices and edges, respectively).

AdjacencyGraph. This concept provides efficient access to vertices adjacent to a certain vertex in the graph.

IncidenceGraph. This concept provides efficient access to the out-edges of a certain vertex in the graph.

BidirectionalGraph. This concept is similar to *IncidenceGraph*, however it adds requirement for efficient access to in-edges. It is separated from *IncidenceGraph*, because storing not only out-edges but also in-edges requires more storage space and not every algorithm requires in-edges (this is not an issue for undirected graphs, since every in-edge is also an out-edge).

AdjacencyMatrix. This concept provides efficient access to any edge in the graph given the source and target vertices (this is achieved by storing edges in a matrix, hence the name „Adjacency matrix“).

EdgeListGraph. This concept provides efficient traversal of all edges in the graph, i.e., efficient iterating over every edge.

VertexListGraph. This concept provides efficient traversal of all vertices in the graph, i.e., iterating over every vertex.

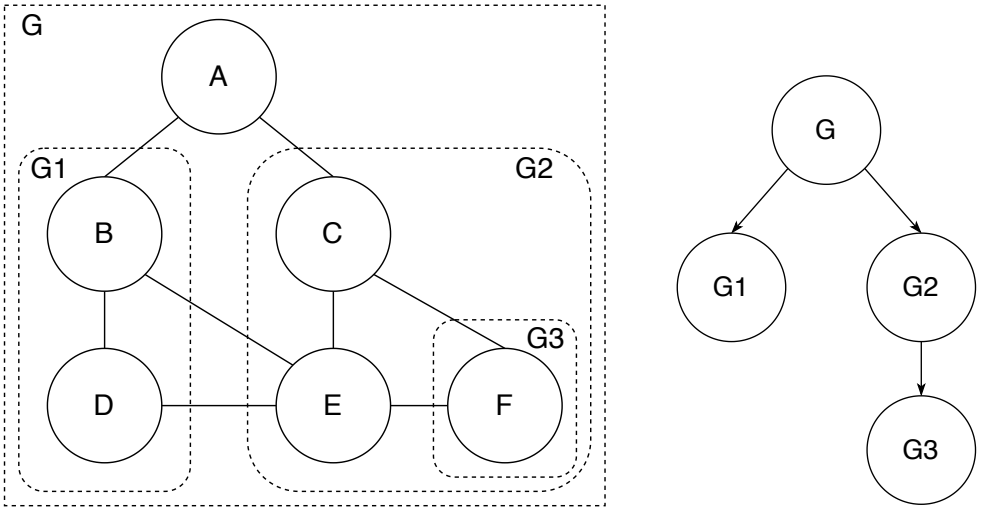
VertexAndEdgeListGraph. This concept combines requirements of *EdgeListGraph* and *VertexListGraph*.

1.1.3 Subgraph

Lot of graph algorithms require the usage of subgraphs, in fact, even the implementation of a tree decomposition algorithm that we will discuss in the next chapter uses subgraphs. That is why BGL provides a class `subgraph` that is used to create subgraphs from a graph.

The `subgraph` class implements induced subgraphs. The main graph and its subgraphs are maintained in a tree data structure. The main graph is the root, and subgraphs are either children of the root or of other subgraphs. All of the nodes in this tree, including the root graph, are instances of the `subgraph` class. The `subgraph` implementation ensures that each node in the tree is an induced subgraph of its parent. The `subgraph` class implements the BGL `graph` interface, so each subgraph object can be treated as a graph [7].

Example of a structure that maintains the parent graph and its subgraphs is shown in Figure 1.2. Each subgraph also has its own vertex and edge descriptor for each vertex and edge, respectively (called *local*) as well as a *global* descriptor, that is used to address the corresponding vertex or edge in the parent graph. That way we can distinguish between vertices and edges of the parent graph and its subgraphs.



(a) Graph G and subgraphs $G1$, $G2$, and $G3$

(b) Tree-like data structure of graph G and its subgraphs

Figure 1.2: Example of data representation used by class `subgraph` to store graph G with its subgraphs $G1$ and $G2$ (which has its own subgraph $G3$). Inspired by [8].

1.2 Notions of used algorithms

In this section we will take a look at formal definitions of general and nice tree decomposition and get more familiar with their properties.

1.2.1 Tree decomposition and treewidth

Tree decomposition is used to represent graph G in a tree structure, allowing us to deduce certain connectivity properties. Also, as we already mentioned earlier, this decomposition is used to improve and speed up certain computationally complex algorithms—example of such an algorithm was given in the introduction of this thesis. We will discuss another example later on.

Now onto the formal definition:

A *tree decomposition* of a graph G is pair $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$, where T is a tree whose every node t is assigned a vertex subset $X_t \subseteq V(G)$, called a *bag*, such that the following three conditions hold:

1. $\bigcup_{t \in V(T)} X_t = V(G)$. I.e., every vertex of G is in at least one bag.
2. For every $uv \in E(G)$, there exists a node t of T such that bag X_t contains both u and v.
3. For every $u \in V(G)$, the set $T_u = \{t \in V(T) : u \in X_t\}$, i.e., the set of nodes whose corresponding bags contain u, induces a connected subtree of T.

The *width* of a *tree decomposition* $\mathcal{T}$ is equals the size of its largest bag minus 1. The *treewidth* of graph G (denoted by $\mathrm{tw}(G)$) is the minimum possible width of a tree decomposition of graph G [9, Chap. 7.2]. See Figure 1.3 for an example. We will be using treewidth as a parameter for an FPT algorithm[3] provided by our library extension.

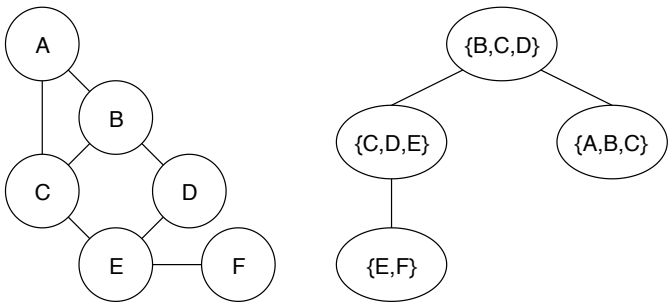


Figure 1.3: Example of a graph (on the left) and its tree decomposition of width 2 (on the right).

[3]Fixed-parameter tractable (FPT) algorithm is an algorithm with complexity exponential only in the size of some fixed parameter k, therefore it works efficiently for a small k.

1.2.2 Nice tree decomposition

Nice tree decomposition is a special type of a tree decomposition, that follows few more conditions in addition to those defined in previous subsection. Those conditions might seem unnatural, but they make the design of dynamic-programming algorithms much easier (as we will see in Section 2.6), which is the primary motivation for introducing nice tree decompositions.

Important property of nice tree decompositions is that if a graph G admits a tree decomposition of width k, then it also admits a nice tree decomposition of width k. In other words, every tree decomposition can be transformed into a nice tree decomposition without increasing its width [9, Lemma 7.4].

We will say that a tree decomposition $\mathcal{T}$ rooted at X_r is *nice*, if following conditions are satisfied [9, Chapter 7.2]:

- $X_r = \emptyset$ and $X_l = \emptyset$ for every leaf l of T. In other words, all the leaves as well as the root contain empty bags.

- Every non-leaf node of T is of one of the following three types:

  - **Introduce node:** a node t with exactly one child t' such that $X_t = X_{t'} \cup \{v\}$ for some vertex $v \notin X_{t'}$; we say that v is *introduced* at t.

  - **Forget node:** a node t with exactly one child t' such that $X_t = X_{t'} \setminus \{w\}$ for some vertex $w \in X_{t'}$; we say that w is *forgotten* at t.

  - **Join node:** a node t with two children t_1, t_2 such that $X_t = X_{t_1} = X_{t_2}$.

For an example of the nice tree decomposition of a tree decomposition presented in Figure 1.3 see Figure 1.4.

From now on, we will have to distinguish nice decomposition from „not nice“ decomposition. Decomposition that does not need to follow additional set of rules presented in this subsection will be referred to as an *general tree decomposition*[4]. The term *tree decomposition* will be used as a generalization of both general and nice decompositions.

[4]Some publications use different term instead of a „general tree decomposition“, e.g., „simple tree decomposition“.

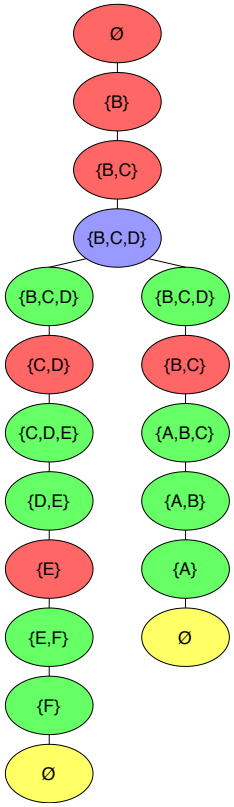


Figure 1.4: Example of a nice tree decomposition of the graph from Figure 1.3 (root is on top, leaf nodes are yellow, introduce nodes green, forget nodes red, and the join node is blue).

1.2.3 Maximum weighted independent set

We will demonstrate the use of the library extension (specifically the nice tree decomposition) on the Maximum Weighted Independent Set problem, which can be described as follows (description is inspired by [9, Chap. 7.1]).

Imagine that you are a famous TV network hosting a large reality show with no capacity cap. You have already held a few auditions and rated each contestant with a score based on their *fun factor*. Now it is up to you to select contestant, that will make the cut and will be participating in the reality show. When making the list of the cast, you would like to maximize the *total fun factor* of the selected contestants. However you know that it is not a good idea to cast people that know each other, because they will most likely form a „secret pact“ giving them an unfair advantage against other contestants. That is why you will want to avoid this situation.

We model this problem as follows. Assume that relationships between contestants are represented by an undirected graph G. Vertices of G represent contestants and each $v \in V(G)$ is assigned a non-negative weight $w(v)$

that represents the score (i.e., the *fun factor*) given at the audition. Every edge uv of G represents that contestant u knows contestant v (this relation is symmetrical). The task is to find the maximum weight of an independent set in G. This problem is called Maximum Weighted Independent Set. For an example see Figure 1.5.

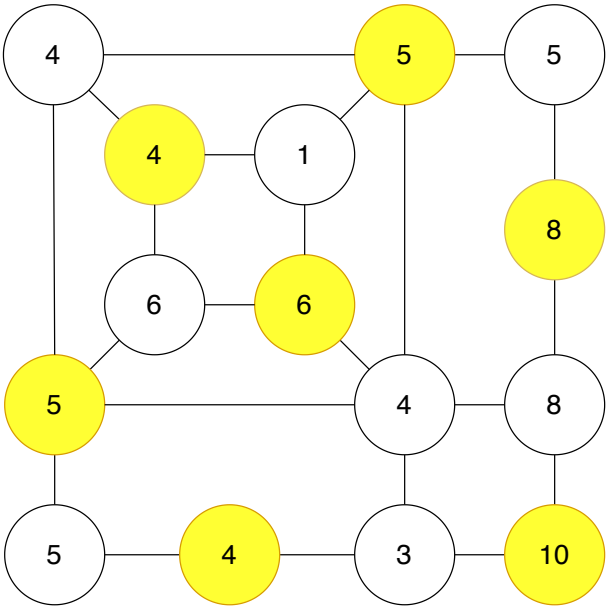


Figure 1.5: Example of a graph and its maximum weighted independent set (represented by yellow vertices) with total weight of 42.

Chapter 2

Design

Since we established the key algorithm notions and characteristics of BGL in previous chapter, we are ready to discuss requirements and code architecture of our library extension. This will also include introduction to the algorithms themselves—their core and used data structures.

2.1 Requirement specification

In the analysis we discussed the importance of genericity and its impact on code design of whole BGL. On one hand, since this work is a BGL extension, it should follow the same principles—in other words, it should also be generic. Take the graph type as an example—we should not limit user to a certain type instance of `adjacency_list`[5]—instead, it should allow user to choose his own desired graph structure. On the other hand, we have to make sure our algorithm is compatible with any graph structure, which is often not possible. That is where concept checking[6] comes in handy. BGL provides concept check library (BCCL), which adds a certain way for the writer of the function template to explicitly specify, *which* model should the template argument (provided by user) conform to. This is achieved by *concept asserts*, which cause compiler error message when incorrect template argument is supplied.

Same principle applies to the *nice decomposition algorithm*. Instead of obtaining decomposition of certain graph inside, we will require a decomposition of a graph as an input and then transform the given decomposition into *nice* one. That way the user can choose, which algorithm for retrieving tree decomposition he will use—be it the one provided in our BGL extension, or a different one.

There are a few more requirements, that will our library extension try to fulfil—those will be divided into functional and non-functional.

[5] `adjacency_list` is specific class implementation of *adjacency list* graph structure.

[6] `https://www.boost.org/doc/libs/1_72_0/libs/concept_check/concept_check.htm`

2.1.1 Functional requirements

F1) Algorithm for obtaining a tree decomposition.

F2) Algorithm for turning a general tree decomposition into a nice one.

F3) Algorithm that uses dynamic programming over a tree decomposition.

F4) Each of algorithms should have generic interface that allows different types of graph as an input.

F5) If user provides incorrect input, algorithms should throw an exception or a compiler error.

F6) *Nice decomposition algorithm* should allow user to select which tree decomposition algorithm will be applied.

2.1.2 Non-functional requirements

N1) Our library extension should follow Boost conventions (code-style, architecture, naming conventions, etc.)

N2) Only C++ Standard Library or other Boost libraries should be used.

N3) Documentation of algorithm interfaces should be provided.

N4) Majority of code should be covered by tests.

2.2 Architecture

As mentioned in non-functional requirement specification (N1), code architecture should follow conventions of BGL. Since BGL is header-only library, the structure of our extension is quite simple and it consist only of header files. Each algorithm has its own separate file, that consist of various non-member functions which are part of `boost` namespace. Interface of each algorithm is directly accessible through that namespace. Other functions, that are not part of the interface and serve as „helper“ function inside the algorithm are inside another nested namespace `detail`, that is supposed to „hide“ the implementation from the user[7].

Each algorithm is covered by tests (Boost test library is used). Tests are located in separate directory, where each header file has its own test-file counterpart named `*headerfile_name*_test.cpp`. Every tested function has its own *test suite*[8] which contains several test cases.

[7] This is an „unspoken“ convention used not only by BGL, please see discussion: `https://stackoverflow.com/questions/26546265/what-is-the-detail-namespace-commonly-used-for`

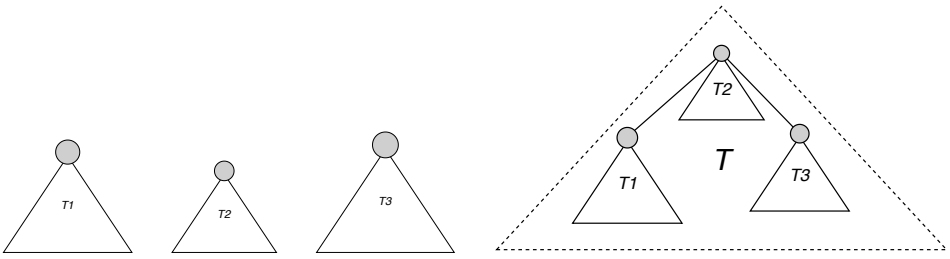
[8] `https://www.boost.org/doc/libs/1_72_0/libs/test/doc/html/boost_test/tests_organization/test_tree/test_suite.html`

2.3 Algorithm for obtaining a tree decomposition

Computing the treewidth of a given undirected graph is NP-hard [10] and there are many different approaches, e.g, exact or approximate fixed-parameter tractable (FPT) algorithms.

In this thesis we will be using the approximate FPT algorithm that mostly follows Reed [11] and is described in the book *Parameterized Algorithms* [9, Chap. 7.6.2]. Given an undirected graph G and integer k, the task of this algorithm will be finding a tree decomposition of a graph G with treewidth at most $3k+4$ or concluding that $tw(G) > k$. Time complexity of this algorithm is $\mathcal{O}(8^k k^2 n^2)$.

As a first step we will have to decide, whether given graph G is connected or not. If it is not, we will apply the algorithm to each connected component of G and connect obtained tree decompositions (see Figure 2.1). From now on, we will assume that G is connected.



(a) Tree decompositions of components of graph G

(b) Tree decomposition of a graph G

Figure 2.1: Tree decomposition of graph G created by connecting decompositions of the components of G (gray vertices are the root of they decomposition they belong to).

The core of the algorithm will be a recursive procedure $decompose(W, S)$, where $S \subsetneq W \subseteq V(G)$ and $|S| \leq 3k + 4$. This procedure tries to decompose subgraph $G[W]$ so that S is contained in one of the bags of the decomposition. Output of this procedure will be a rooted tree decomposition $\mathcal{T}_{W,S}$. Root call of this procedure will be $decompose(V(G), \emptyset)$, which will return a rooted tree decomposition of G.

2.3.1 Procedure decompose

First, we will need to construct a set $\widehat{S}$ with following properties:

1. $S \subsetneq \widehat{S} \subseteq W$
2. $|\widehat{S}| \leq 4k + 5$
3. every connected component of $G[W \setminus \widehat{S}]$ is adjacent to at most $3k + 4$ vertices of $\widehat{S}$.

Constructed set $\widehat{S}$ will be the root bag of the decomposition constructed by this call of *decompose*.

If $|S| < 3k+4$, we take any vertex $u \in W \setminus S$ and construct $\widehat{S} : S \cup \{u\}$. If $|S| = 3k+4$, we iterate through all possible partitions of S into two parts A and B (there is 2^{3k+4} of them) and apply a max-flow algorithm to verify, whether minimum order of a separation of $G[W]$, where A and B are separated, does not exceed $k+1$. This is done by creating a new vertex, that will be connected with every vertex of A—this will be the source for the max-flow algorithm. Similarly we will construct a new vertex connected with every vertex of B—this vertex will be the sink. After applying a max-flow algorithm to find a maximum flow between the newly created source and the sink, we will get a separation (A', B'), where $(A' \cap B')$ represents the cut.

If the computed flow is too big, i.e., such a partition of S does not exist for every pair (A, B), we can conclude that $tw(G[W]) > k$, therefore $tw(G) > k$ and we can terminate the whole algorithm. If we have found a partition (A', B') that satisfies given properties, we can construct $\widehat{S}$ as $S \cup (A' \cap B')$.

After successfully constructing $\widehat{S}$, we create vertex sets $D_1, D_2, .., D_p$ for every connected component $i = 1, 2, .., p$ of $G[W \setminus \widehat{S}]$. For each i we will recursively call $decompose(N_G[D_i], N_G(D_i))$ and let r_i be the root of the decomposition returned by the decompose call. Now we will create root bag $X_r = \widehat{S}$ and for every i we will attach each decomposition $\mathcal{T}_i$ below r using edge rr_i (note that if $p = 0$, then $\widehat{S}$ will be a leaf of the tree decomposition of G). We have successfully created decomposition of $\mathcal{T}_{w,s}$.

2.3.2 Used data structures

Now that we are familiar with the algorithm, our next aim will be choosing the right data representation. Since this work is a BGL extension, we will use existing solutions provided by the BGL.

In Subsection 1.1.2 we discussed the role of graph concepts in BGL, which will now come handy. First step will be to decide, which type of a graph we expect as an input by user. In this algorithm we will need to traverse through all vertices many times (e.g., when we will need to retrieve all neighbours of a certain vertex or while constructing components of a given graph), therefore

the best option is *VertexListGraph*—it is the minimum concept, that includes the operation we will require.

Second step will be to decide which type of a graph will be used to represent the tree decomposition. As opposed to the input graph, we do not require any special operations on this graph—only required operations are adding vertices and edges, which are basic operations supported by every graph concept. Therefore there is no point in limiting user to certain graph concept and we will let him decide, which one he wants to use.

Last step will be choosing the right data structure to represent bags of the decomposition. Since it is more complex, we will dedicate a separate section for it (see Section 2.5).

2.4 Algorithm for obtaining a nice tree decomposition

Main task of this algorithm will be creating a nice tree decomposition d_2 from an general tree decomposition d_1 provided by the user. This will be done by traversing through graph d_1 while simultaneously constructing graph d_2 that follows additional set of rules introduced in the definition of the nice tree decomposition (see Subsection 1.2.2).

Algorithm is described in Figure 2.2. Note that UML activity diagram is significantly simplified in order to maintain readability. From the diagram we can observe that constructing a nice decomposition from a general decomposition will not affect its width, i.e., if a decomposition d_1 has width k, decomposition d_2 will also have width k [9, Chap. 7.2]. To really simplify the algorithm, first, we will construct node with empty bag for every leaf node of d_1. Second, we will insert a few nodes between each other nodes of d_1 where each time there will be one element from node of d_1 introduced or forgotten. That way we will never construct a node with a bag that has more elements than the largest bag of d_1, causing the increased width of d_2.

2.4.1 Used data structures

Data representation will be similar to the one used in the algorithm for obtaining a general tree decomposition. However this time we will not require a graph whose nice decomposition we will be constructing, just the general decomposition.

We will have to decide, which type of graph we will use for both types of decomposition. Since we will be traversing through the vertices of a general decomposition provided by the user, *VertexListGraph* will be the best option.

Similarly to the general decomposition algorithm, the data structure to represent the bags of both decompositions will be discussed in Section 2.5.

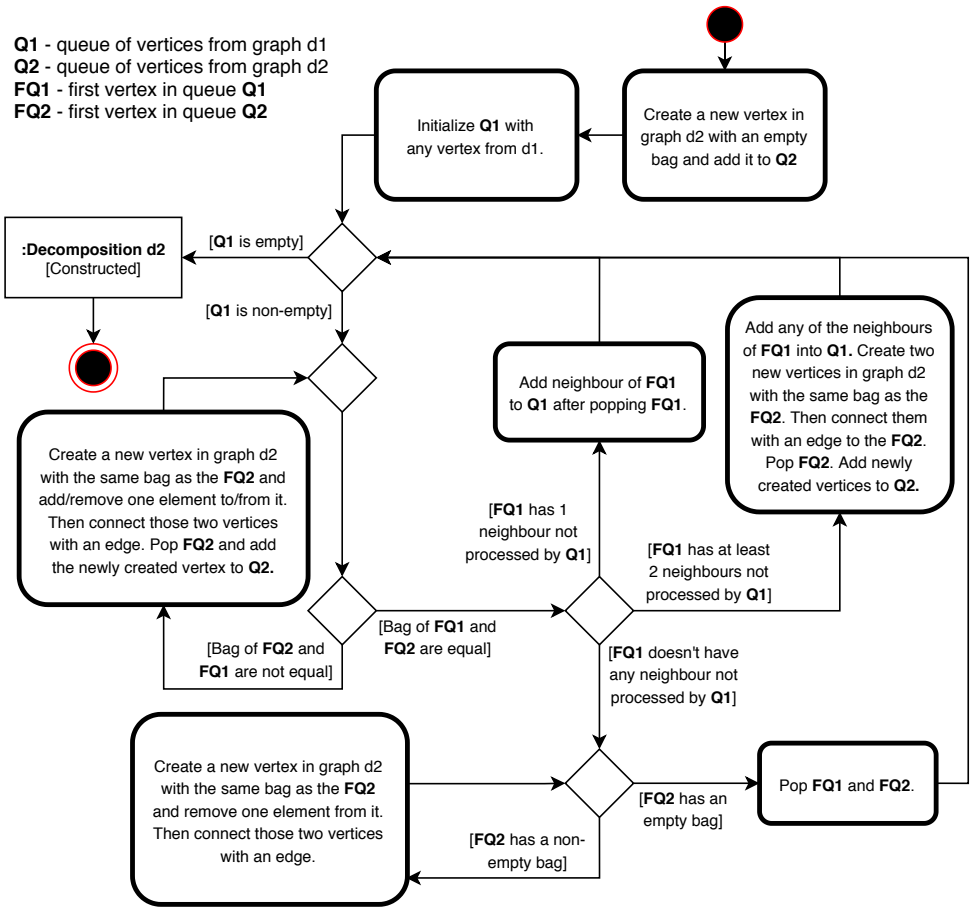


Figure 2.2: UML activity diagram describing construction of the nice decomposition (graph $d1$ is a general decomposition provided by user and graph $d2$ is a nice tree decomposition).

2.5 Representation of mutable bags

Since we want to give the user as much freedom of choice as possible, we will use something called *Read/Write Property Map*[9]—it is a generic interface for associative map, that allows read and write operations (value is retrieved using the key). That way the user can choose a data structure (e.g., `std::map`) as long as it follows the rules of *Read/Write Property Map* concept. The value type of this map will be a mutable container. Graphical illustration of this data structure is presented in Figure 2.3.

[9] `https://www.boost.org/doc/libs/1_72_0/libs/property_map/doc/ReadWritePropertyMap.html`

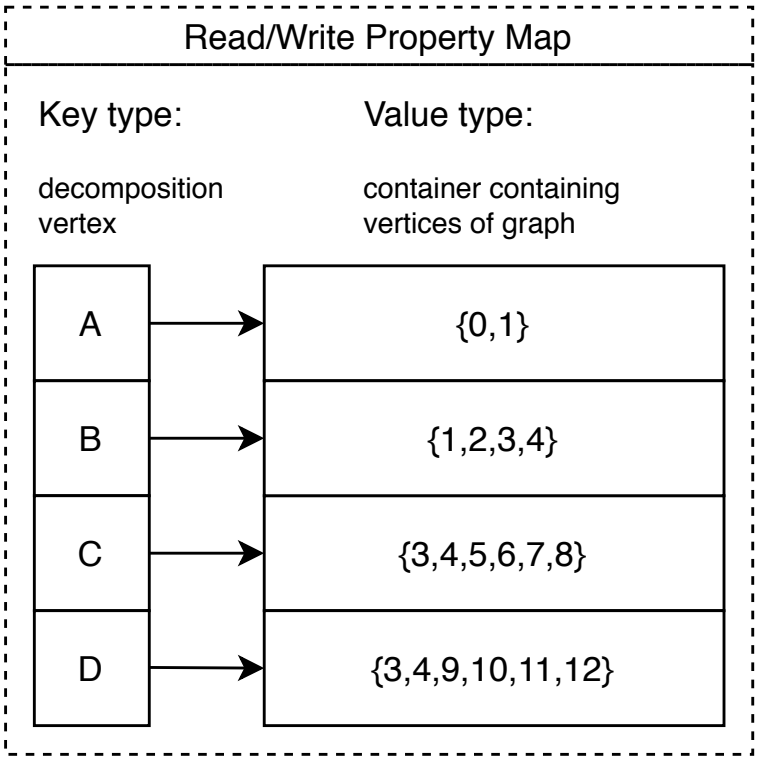


Figure 2.3: Graphical illustration of data structure that represents bags.

Next step will be choosing the right concept of a container that will represent a single bag of node of the decomposition (we will be using list of concepts provided by BGL as a reference [12]). BGL offers a wide range of concepts and the most basic and simple one is *Collection*[10]. It provides methods for accessing and most importantly iterating over the container. However it does not provide a method for inserting elements into the container, which is one of the key methods that we will need (since we will be constructing bags, which requires inserting of elements). Therefore *Collection* concept is not suitable for us.

The most basic concept that provides *insert* method is *Sequence*[11]. However downside of this concept is that it does not support popular containers like `std::set` or `std::unordered_set`.

Since none of the concepts provided by BGL is entirely suitable, we will have to define our own container concept called *InsertCollection*. Its implementation will be discussed in Section 3.3.

[10] `https://www.boost.org/doc/libs/1_72_0/libs/utility/Collection.html`
[11] `https://www.boost.org/sgi/stl/Sequence.html`

2.6 Maximum weighted independent set algorithm

Solving the Maximum Weighted Independent Set problem is classified as NP-hard [13]. For the naive brute-force approach we would have to generate every possible set of vertices of graph G (that is 2^n permutations) and find the maximum weighted independent set among them—that would result into time complexity $\mathcal{O}(2^n \cdot n^2)$.

However we can approach this problem much more effectively provided that we have a nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(t)})$ of graph G of width k with root r. We will be using the algorithm described in the book *Parameterized Algorithms* [9, Chap. 7.3.1]. This algorithm will allow us to solve problem in time $2^k \cdot k^{\mathcal{O}(1)}$. The core of the algorithm will be a bottom-up dynamic programming based procedure *calculate_weight(t, S)*:

calculate_weight(t, S) = maximum possible weight of the set $\widehat{S}$ such that $S \subseteq \widehat{S} \subseteq V_t$, $\widehat{S} \cap X_t = S$, and $\widehat{S}$ is independent, where V_t is the union of all the bags present in the subtree of T rooted at t, including X_t

If no such set $\widehat{S}$ exists (for example S itself is not independent), the return value of the procedure will be $-\infty$. Root call of the procedure will be *calculate_weight(r, ∅)*, which will calculate the maximum weight of graph G, because $V_r = V(G)$.

2.6.1 Procedure calculate_weight

This recursive procedure defines behaviour for each type of node t we will be possibly dealing with. The base case of this recurrence will be the leaf node. For non-leaf nodes, the result of the procedure depends on the results of their child nodes. Now we will define the behaviour for each type of nodes introduced in Subsection 1.2.2 (in order to retain readability, we will be using the abbreviation *cw* instead of full name of the procedure):

Leaf node. If t is a leaf node, we will return 0.

Introduce node. If t is an introduce node with a child t' such that $X_t = X_{t'} \cup \{v\}$ for some $v \notin X_{t'}$, we put:

$$cw(t, S) = \begin{cases} cw(t', S) & \text{if } v \notin S \\ cw(t', S \setminus \{v\}) + w(v) & \text{if } S \setminus \{v\} \text{ is independent} \\ -\infty & \text{otherwise} \end{cases} \tag{2.1}$$

Forget node. If t is a forget node with a child t' such that $X_t = X_{t'} \setminus \{w\}$ for some $w \in X_{t'}$, we put:

$$cw(t, S) = \max\{cw(t', S), cw(t', S \cup \{w\})\} \tag{2.2}$$

Join node. If t is a join node with children t_1, t_2 such that $X_t = X_{t_1} = X_{t_2}$, we put:

$$cw(t, S) = cw(t_1, S) + cw(t_2, S) - w(S) \tag{2.3}$$

2.6.2 Used data structures

Now we can move onto the choosing of right data structures. Most of them will be similar to those used in previous algorithms.

First we need to choose the right representation for both graph and its nice decomposition. In both cases we will need to traverse through vertices—as we already established in previous sections, best graph concept for this matter is *VertexListGraph*. Both of these graphs can be of different type, however they both have to implement the graph concept *VertexListGraph*.

Next are the bags of the decomposition. Since the algorithm will use the bags of decomposition only for reading, they will be be represented by *Readable Property Map*. The value type of the map will be *Container*[12], which is the most generic interface for a container that provides basic operations for reading.

Each of the vertices of the graph should have defined its weight. For this we will be using similar approach as we did with bags, that is, *Read/Write Property Map*, where we will use the vertex descriptor as a key to access the weight of the said vertex. However the weight has to be a non-negative integer (e.g., `uint`, `ulong`).

The purpose of this algorithm is not only to return the total weight of the found maximal weighted set, but also provide the set itself. This will be achieved by using something called *Color Map*—it is basically another *Property Map*, where each vertex of a graph has assigned a color[13]. In our case we will use two colors—white, if vertex belongs to the maximal weighted set or black otherwise.

[12] `https://www.boost.org/sgi/stl/Container.html`
[13] `https://www.boost.org/doc/libs/1_72_0/libs/graph/doc/ColorValue.html`

Chapter 3

Implementation

This chapter will focus on fulfilling functional and non-functional requirements from the previous chapter. It will be divided into three sections (for each algorithm mentioned in F1, F2, and F3). In each of these parts we will discuss interface, core of the functions and reasoning behind certain decisions that were made.

3.1 Algorithm for obtaining a tree decomposition

In this section we will focus on implementation of the algorithm that constructs general tree decomposition—function `tree_decomposition`. Interface of the function is shown in Code snippet 3.1.

```
template <class Graph, class Decomposition, class Bags>
bool tree_decomposition(Graph & g, Decomposition & d, Bags &
↪  bags, unsigned long k)
```

Code snippet 3.1: Interface of the function `tree_decomposition`.

`Graph g`—graph whose tree decomposition will be constructed.
`Decomposition d`—graph in which will be the decomposition stored (this is an in/out parameter).
`Bags bags`—*Property Map* containing bags for each node of `d` (this is an in/out parameter).
`unsigned long k`—parameter, which will limit the width of decomposition.

Return value of this function will be a boolean—*true*, if tree decomposition of graph `g` exists with width at most $3k + 4$ and *false* if such a decomposition does not exist.

Since most of the parameters of the function are template parameters, we have to assure that the user provided a correct input (preferably during compilation). That is achieved using the *Boost concept asserts* and *Boost static asserts*. If any of these asserts (see Code snippet 3.2) fails, then compilation error will be thrown.

```
//== TYPEDEFS ==========================
//vertex descriptor that is used to address vertices in graph
using Graph_vertex = typename
↪  graph_traits<Graph>::vertex_descriptor;
//vertex descriptor that is used to address vertices in
↪  decomposition graph
using Decomposition_vertex = typename
↪  graph_traits<Decomposition>::vertex_descriptor;
//type of value (container) of PropertyMap
using Bag_value_type = typename
↪  property_traits<Bags>::value_type;
//type of container value, that is stored as value in
↪  PropertyMap
using Bag_inside_type = typename Bag_value_type::value_type;

//== ASSERTS ===========================
BOOST_CONCEPT_ASSERT((VertexListGraphConcept<Graph>)); //A1
BOOST_CONCEPT_ASSERT((InsertCollection<Bag_value_type>)); //A2
BOOST_CONCEPT_ASSERT((ReadWritePropertyMapConcept<Bags,
↪  Decomposition_vertex>)); //A3
BOOST_STATIC_ASSERT((is_same<Graph_vertex,
↪  Bag_inside_type>::value)); //A4
```

Code snippet 3.2: Asserts of the function `tree_decomposition`.

A1) *Graph* must be a model of *VertexListGraphConcept*.

A2) Each bag of *Bags* container must be a model of *InsertCollection* (see Section 3.3).

A3) *Bags* must be a model of *Read/Write Property Map* (with *Decomposition* vertex descriptor as a key).

A4) *Graph* vertex descriptor and value type of container inside the *Bags* must be the same type.

3.1.1 Implementation of the algorithm

Now that we are familiar with the interface of the function, let us take a closer look into the implementation.

Assuming that all concept asserts passed, next step will be retrieving all connected components of graph g provided by user, since procedure *decompose* expects sets of vertices from a connected graph. This is achieved by helper function `get_components`, which returns all components of g in a `std::vector`. However this function requires `subgraph` (see chapter Subsection 1.1.3) as an input, so we have to create empty instance of class `subgraph` and copy graph g into it.

Then, function `decompose` is called on every component. Return type of this function is `boost::tuple`, where first element is `vertex_descriptor` of root node of decomposition, that was constructed by current call of `decompose` and the second one is bool. If the bool = *true*, decomposition was found. Otherwise, decomposition was not found and we can immediately return *false* without further inspection of remaining components.

If the decomposition of every component was successfully found, we will connect each decomposition with an edge to create one connected decomposition. Now we can return *true*, since decomposition of g was found and successfully constructed.

3.1.2 Procedure decompose

Function `decompose` is the core and also the most important part of this algorithm. By recursively calling this function, decomposition of graph g and its bags will be constructed (both *decomposition* and *Property Map* representing the bags are passed by reference). First (aka the root) call of this recursive function will contain empty decomposition, empty bags, `std::set` of vertex descriptors w containing all vertices of graph g and empty `std::set` s.

First goal of this function is to construct set *s1* (corresponding to the set $\widehat{S}$ in Subsection 2.3.1). If size of set s is smaller than $3k + 4$, we iterate over set w in order to find a vertex that is not contained in s and to insert it into set s—this is done by trying to emplace every vertex of w into s using `std::set.emplace()`. If enough vertices were successfully inserted, we stop iterating and carry on with the algorithm.

If the size of s is equal to $3k + 4$, we have to find a vertex separation of s in graph g with minimum order not exceeding $k + 1$. Such a separation is retrieved using the helper function `get_separation` that returns either a set of vertices forming the desired separation or an empty set if such separation was not found—in that case, we can immediately return *false* in the current call of `decompose`, since decomposition of width $\leq 3k + 4$ cannot be constructed.

Now we can insert a new vertex into the decomposition graph—it will represent the root node of the decomposition created by the current call of `decompose`. Since every node of *decomposition* should have its own bag, we will create a new bag for this node and it will contain all vertices from set *s1*.

Finally, we will create a new subgraph of g, that will contain vertices from $w \setminus s$ (we will call this set d in short). This can be easily done by using `std::set_difference`—it inserts difference of two sets into a new one, in our case set d. Next, we will construct a new subgraph *d_graph* using the set of vertices d obtained earlier. Now we can call recursively `decompose` on every component of *d_graph* (see Code snippet 3.3).

```
//for every component we will recursively call decompose
for (auto component : d_components) {
  tuple<Decomposition_vertex, bool> res = detail::decompose(
    parent,
    decomposition,
    bags,
    detail::get_neighbour_vertices(*component, true),
    detail::get_neighbour_vertices(*component, false),
    k);

  if (!get<1>(res)) return res;
  //connect bag with current root
  add_edge(get<0>(res), root_bag, decomposition);
}
```

Code snippet 3.3: Creation of new recursive calls of the function `decompose`.

3.1.3 Retrieving separation

Function `get_separation` is another very important helper function. Its main purpose is to find a vertex separator of a graph containing vertices from s with minimum order not exceeding $k + 1$. If such a separator does not exist, an empty set will be returned.

First, we will need to generate every possible partition of s into two non-empty parts. This is done by the function `split_set`, which takes a set of vertices s and a seed number *c_number* and returns a tuple of two sets (let us call them a and b). Binary representation of *c_number* will decide, how s will be divided: If nth number of the binary representation of *c_number* is `0`, the nth element of s will belong to a. Otherwise it will belong to b. Implementation of this function is shown in Code snippet 3.4.

```
Graph_vertex_set a, b;
unsigned long long set_size = s_copy.size();
for (unsigned long long i = 0; i < set_size; i++) {
    auto it = s_copy.begin();
    if (c_number % 2 == 0) a.insert(*it);
    else b.insert(*it);
    s_copy.erase(it);
    if (c_number != 0) c_number = c_number >> 1;
}
return make_tuple<Graph_vertex_set, Graph_vertex_set>(a, b);
```

Code snippet 3.4: Implementation of the function `split_set`.

Now that we have every possible partition of s, we can move onto finding a minimum separator that separates a and b. This is achieved by function `max_flow_sep` that applies max-flow algorithm on a graph g, where partition a will be the source and partition b will be the sink, which calculates the vertex separation of the partition (this will be our desired separator). For that, we will be using `edmonds_karp_max_flow` provided by BGL that implements Edmonds and Karp algorithm. However this leads to a few obstacles which will force us to slightly modify the graph g before using `edmonds_karp_max_flow`.

First obstacle is that `edmonds_karp_max_flow` expects directed graph as an input—however we are working with undirected graphs (see Section 2.3). This is easily solved by converting our graph g into a new one (let us call it *g1*) where every undirected edge uv will be represented by two directed edges uv and vu. From now on we will be working with a newly created graph *g1*.

Second obstacle is that `edmonds_karp_max_flow` calculates an edge separator instead of a vertex separator. For each vertex v we will create two vertices v_1 and v_2 connected by directed edge v_1v_2. Every edge that had an end in v will be connected to v_1 and every edge that had a start in v will be connected to v_2 (see Figure 3.1). This method is explained in a proof of Menger theorem [14]. Now if the max-flow algorithm calculates an edge v_1v_2 as a part of separator of $g1$, it means that vertex v is also a part of separator of a graph g.

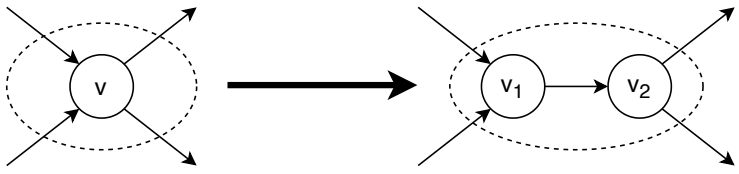


Figure 3.1: „Simulating“ a vertex v with an edge v_1v_2.

Our source for the max-flow algorithm is a set a and the sink is a set b. Both of these sets can contain multiple vertices, however `edmonds_karp_max_flow` expects a single vertex as a source and sink. As described in Subsection 2.3.1, we will have to create two new vertices x_1 and x_2 in a graph $g1$, representing the source and sink, respectively. Vertex x_1 will be connected with every vertex of a and x_2 will be connected with every vertex of b.

As a last step, we have to define capacities for each edge of $g1$. Edges that „simulate“ a vertex from g (described in Figure 3.1) will have capacity set to `1`. Every other edge has capacity of `2` (thus these edges will not be part of the computed separator). Capacities are stored in *Property Map*, that is passed as an argument to `edmonds_karp_max_flow`.

Now we are ready to call `edmonds_karp_max_flow`. If the return value of this function is at most $k + 1$, we have successfully found separation of a graph $g1$—edges of this separation are stored in another *Property Map*. For each edge v_1v_2 contained in *Property Map* we will insert vertex v from g into the set, that will be the return value of function `get_separation`. This set is our desired vertex separation of graph g.

If the return value of `edmonds_karp_max_flow` is higher than $k+1$, function `max_flow_sep` will return an empty set and it will have to be called on another partition of s created by the function `split_set`.

3.1.4 Retrieving components

Earlier we mentioned helper function `get_components` that is used to retrieve all components of a certain subgraph G. This function is basically a wrapper around `boost::connected_components` provided by BGL, which returns number of all components and a map where each vertex from G has assigned an integer which determines in which component it belongs to. However this representation is not suitable for our algorithms, therefore we will rewrite the map into `std::vector` containing pointers to all of the components of graph G.

3.1.5 Retrieving neighbours

Another useful helper function is `get_neighbour_vertices` which returns all of the neighbour vertices of a subgraph G' in a parent graph G (formally $N_G[G']$ or $N_G(G')$).

It is basically a wrapper around the function `boost::adjacent_vertices` provided by BGL with a small modification—our function takes in addition to subgraph G' also a boolean as an input which determines if the neighbours of G' should also include vertices of G' themselves. If the boolean is set to *true*, vertices of G' are included, otherwise they are not (formally $N_G[G']$ and $N_G(G')$, respectively).

3.2 Algorithm for obtaining a nice tree decomposition

In Section 2.4 we discussed the algorithm that constructs a nice decomposition based on a general tree decomposition provided by the user and now it is time to take a look at the implementation. Interface of the function `nice_tree_decomposition` is shown in Code snippet 3.5. Note that both general and nice tree decomposition use the same data type for the graph and its bags.

```
template <class Decomposition, class Bags>
typename graph_traits<Decomposition>::vertex_descriptor
nice_tree_decomposition(Decomposition & d, Bags & bags,
↪  Decomposition & nice_d, Bags & nice_bags)
```

Code snippet 3.5: Interface of the function `nice_tree_decomposition`.

`Decomposition d`—a general decomposition based on which a nice tree decomposition will be constructed.
`Bags bags`—*Property Map* containing bags for each node of `d`.
`Decomposition nice_d`—graph in which will be the constructed nice decomposition stored (this is an in/out parameter).
`Bags nice_bags`—*Property Map* containing bags for each node of `nice_d` (this is an in/out parameter).

Return value of this function is `vertex_descriptor` of root node of constructed decomposition. If `d` contains a cycle, `std::invalid_argument` exception is thrown.

Similarly to function `tree_decomposition` we have to make sure that user provided a correct input. For example if graph concept that does not support traversing through vertices was provided, code would not compile and compiler would produce long error messages, making it much harder for user to find out what he did wrong—this is why we will use *Boost concept asserts*. Below is a list of all used asserts (for more details see Code snippet 3.6):

A1) *Decomposition* must be a model of *VertexListGraphConcept*.

A2) Each bag of *Bags* container must be a model of *InsertCollection* (see Section 3.3).

A3) *Bags* must be a model of *Read/Write Property Map* (with *Decomposition* vertex descriptor as a key).

```
//== TYPEDEFS ==============================================
//vertex descriptor that is used to address vertices in
↪  decomposition graph
using Decomposition_vertex = typename
↪  graph_traits<Decomposition>::vertex_descriptor;
//type of value (container) of PropertyMap
using Bag_value_type = typename
↪  property_traits<Bags>::value_type;
//============================================== TYPEDEFS ==

//== ASSERTS ===============================================
BOOST_CONCEPT_ASSERT((VertexListGraphConcept<Decomposition>));
↪  //A1
BOOST_CONCEPT_ASSERT((InsertCollection<Bag_value_type>)); //A2
BOOST_CONCEPT_ASSERT((ReadWritePropertyMapConcept<Bags,
↪  Decomposition_vertex>)); //A3
//============================================== ASSERTS ==
```

Code snippet 3.6: Asserts of the function `nice_tree_decomposition`.

3.2.1 Implementation of the algorithm

In Section 2.4 we have gotten familiar with a simplified version of this algorithm (especially in Figure 2.2). Actual implementation is similar, however instead of an iterative algorithm with a queue we will be using a recursive function `nice_decomposition_rec`, that works in the same manner. In the next few paragraphs we will be using same terms, as in Section 2.4—d_1 is a general decomposition, d_2 is a nice decomposition, *FQ1* and *FQ2* is (since we do not use a queue) a currently processed vertex from d_1 and d_2, respectively.

Function `nice_decomposition_rec` takes *FQ1* and *FQ2* as a parameter, compares them and if there are any differences between their bags, new vertex is constructed in d_2 and `nice_decomposition_rec` is called with a new *FQ2*. If there are no differences, we can move on to another vertex from d_1 (this vertex will be a new *FQ1*).

The root call of `nice_decomposition_rec` will be with a vertex that has an empty bag as *FQ2* (this will be also the root of d_2) and any vertex from d_1 as *FQ1* (it does not matter which one since d_1 is not rooted).

After `nice_decomposition_rec` successfully finishes, d_2 will contain a nice tree decomposition constructed based on d_1. Then the root of d_2 is returned as a return value of the function `nice_tree_decomposition`.

3.3 Concept InsertCollection

In Section 2.5 we discussed the reason behind defining our own container concept and now it is time to take a look at the implementation (presented in Code snippet 3.7). The implementation will be following *Boost Concept Check Library* (BCCL) standards [15].

```
#ifndef BOOST_GRAPH_INSERT_COLLECTION
#define BOOST_GRAPH_INSERT_COLLECTION
template <class C>
struct InsertCollection : Collection<C>
{
 public:
    using Container_value_type = typename C::value_type;

    BOOST_CONCEPT_USAGE(InsertCollection)
    {
        x.insert(x.begin(), e); // require insert method
    }
 private:
    C x;
    Container_value_type e;
};
#endif // BOOST_GRAPH_INSERT_COLLECTION
```

Code snippet 3.7: Definition of concept *InsertCollection*.

InsertCollection is defined using `struct` that is a refinement of BCCL concept *Collection*—this ensures that any data structure that is in conformance to *InsertCollection* is also in conformance to *Collection*. Inside `struct` we will use macro `BOOST_CONCEPT_USAGE` that exercises if method *insert* is defined for the data structure that is being tested. If it is undefined, substitution of template argument `C` will fail and the code will not compile. Interface of method `insert` must be defined as follows:

```
insert ( iterator pos, T& value );
```

Iterator `pos` is a position where the element `value` should be inserted (in case of sorted container it is as close to `pos` as possible). Return value is not tested. This interface was chosen because it is implemented by most of the containers of *C++ Standard Library*. For summary of containers that implement this interface refer to Table 3.1.

Container	Implements `insert`?	Since
`std::array`	No	-
`std::vector`	Yes	C++11 [16]
`std::deque`	Yes	C++11 [17]
`std::forward_list`	No	-
`std::list`	Yes	C++11 [18]
`std::set`	Yes	C++11 [19]
`std::map`	Yes	C++11 [20]
`std::multiset`	Yes	C++11 [21]
`std::multimap`	Yes	C++11 [22]
`std::unordered_set`	Yes	C++11 [23]
`std::unordered_map`	Yes	C++11 [24]
`std::unordered_multiset`	Yes	C++11 [25]
`std::unordered_multimap`	Yes	C++11 [26]
`std::stack`	No	-
`std::queue`	No	-
`std::priority_queue`	No	-

Table 3.1: Summary of *C++ Standard Library* containers and their support of the interface of method `insert` defined in *InsertCollection* concept.

3.4 Maximum weighted independent set algorithm

In this section we will focus on implementation of the algorithm that solves the Maximum Weighted Independent Set problem—i.e., the function `max_weighted_independent_set`. Interface of the function is shown in Code snippet 3.8.

```
template <class Graph, class Decomposition, class Bags, class
↪  Weights, class Colors>
typename property_traits<Weights>::value_type
max_weighted_independent_set(
  Graph & g,
  Decomposition & d,
  Bags & bags,
  typename graph_traits<Decomposition>::vertex_descriptor root,
  Weights & weights,
  Colors & colors)
```

Code snippet 3.8: Interface of the function `max_weighted_independent_set`.

`Graph g`—graph whose maximum weighted independent set will be computed.
`Decomposition d`—nice tree decomposition of graph `g`.
`Bags bags`—*Property Map* containing bags for each node of `d`.
`decomposition_vertex_descriptor root`—root node of decomposition `d`.
`Weights weights`—*Property Map* that contains weights for each vertex of `g`.
`Colors colors`—*Property Map* representing the maximum weighted independent set. Each vertex of `g` is assigned a color—white if they belong to the maximum weighted independent set, black if not. This is an in/out parameter.

Return value of the function is total weight of the computed maximum weighted independent set. If `d` is not a tree, `std::invalid_argument` exception is thrown.

As we can see, the function interface consists of many template arguments. To ensure that the user provided correct input, we will be using *Boost concept asserts*. Below are listed all asserts that have been used (their particular implementation is shown in Code snippet 3.9):

A1) *Graph* must be a model of *VertexListGraphConcept*.

A2) *Decomposition* must be a model of *VertexListGraphConcept*.

A3) *Bags* must be a model of *Readable Property Map* (with *Decomposition* vertex descriptor as a key).

A4) Each bag of *Bags* container must be a model of *Container* (generic interface for container).

A5) *Graph* vertex descriptor and the value of type of container inside the *Bags* must be the same type.

A6) *Weights* must be a model of *Read/Write Property Map* (with *Graph* vertex descriptor as a key).

A7) The value type of *Weights* must be an unsigned integer (e.g., `unsigned int`, `unsigned long`).

A8) *Colors* must be a model of *Read/Write Property Map* (with *Graph* vertex descriptor as a key).

A9) The value type of *Colors* must be a model of *Colors type*.

```
//== TYPEDEFS ==========================================
//type of value (container) of PropertyMap
using Bag_value_type = typename
↪  property_traits<Bags>::value_type;
//type of container value, that is stored as value in
↪  PropertyMap
using Bag_inside_type = typename Bag_value_type::value_type;
//type representing weight
using Weight_type = typename
↪  property_traits<Weights>::value_type;
//type representing color
using Colors_type = typename
↪  property_traits<Colors>::value_type;
//== ASSERTS ===========================================
BOOST_CONCEPT_ASSERT((VertexListGraphConcept<Graph>)); //A1
BOOST_CONCEPT_ASSERT((VertexListGraphConcept<Decomposition>));
↪  //A2
BOOST_CONCEPT_ASSERT((ReadablePropertyMapConcept<Bags,
↪  Decomposition_vertex>)); //A3
BOOST_CONCEPT_ASSERT((Container<Bag_value_type>)); //A4
BOOST_STATIC_ASSERT((is_same<Graph_vertex,
↪  Bag_inside_type>::value)); //A5
BOOST_CONCEPT_ASSERT((ReadablePropertyMapConcept<Weights,
↪  Graph_vertex>)); //A6
BOOST_CONCEPT_ASSERT((UnsignedIntegerConcept<Weight_type>));
↪  //A7
BOOST_CONCEPT_ASSERT((ReadWritePropertyMapConcept<Colors,
↪  Graph_vertex>)); //A8
BOOST_CONCEPT_ASSERT((IntegerConcept<Colors_type>)); //A9
```

Code snippet 3.9: Asserts of the function `max_weighted_independent_set`.

3.4.1 Implementation of the algorithm

Let us assume that asserts from earlier passed. As a first step, recursive function `calculate_weight` (introduced in Subsection 2.6.1) will be called on a root of the decomposition provided by the user.

In every call of this function we will check a memory structure (see Subsection 3.4.2) that stores the results for each combination of parameters t and S (this structure is passed by a reference in every call). If the function was already called with the same combination of parameters t and S, we will use

the memorized results. Otherwise we will compute the weight and maximum independent set and store it into the memory structure mentioned earlier.

After the algorithm ends a function `get_independent_set` backtracks recursive calls stored in memory structure and retrieves a set of vertices that belong to the maximum weighted independent set. Based on this set each vertex from graph g is assigned a color (white color if it belongs to the set, otherwise black color). Then the total weight of the computed maximum independent set is returned.

3.4.2 Memory structure

Memory structure for storing results of calls of the function `calculate_weight` is represented by a 2D map. First dimension that represents parameter t is stored as `std::map` with t as a key. Value of this map is the second dimension representing parameter S stored as `std::unordered_map` with S as a key.

However `std::unordered_map` requires for its key to have a hash object defined, which S (represented by `std::set`) does not have by default. This issue is solved by using the *Boost Functional* library, which defines hash objects for every STL container (including our `std::set`).

Finally, the value of the second dimension is a pair (`std::pair`) which stores weight as a first value and list[14] of vertices of graph g as a second value. Signature of this data structure is shown in Code snippet 3.10.

```
//data structure for storing calculate_weight() call results
std::map<
  Decomposition_vertex,
  std::unordered_map<
    std::set<Graph_vertex>,
    std::pair<Weight_type, std::list<Graph_vertex>>,
    hash<std::set<Graph_vertex>>
  >
> memory;

weight = memory[t][S].first;
backtrack_list = memory[t][S].second;
```

Code snippet 3.10: Data structure used to store results of the calls of `calculate_weight` and an example of access to this structure.

[14] Value of the list is used in backtracking of the recursive calls in the function `get_independent_set`. List is either empty or contains a vertex w in case that t is a forget node and $cw(t, S \cup \{w\}) > cw(t, S)$.

Chapter 4

Testing and documentation

This chapter is divided into two parts. In the first one we will take a look at testing of functions provided in our library extension. Also we will discuss fulfilment of the requirements introduced in Section 2.1.

The second part of this chapter is dedicated to documentation of interfaces provided by our library extension.

4.1 Testing

Since this is an extension for Boost library, the Boost Test Library (BTL) was chosen as an Unit testing framework. Unit tests are supposed to assure that all function provided by this library (and also most of the helper function used by them) are working correctly. Structure of these test will be discussed in the next subsection. In the last two subsections we will discuss a continuous integration that is used to build and run tests automatically and the requirements from Section 2.1.

4.1.1 Structure

Three different usages of the BTL [27] are supported—header-only variant, static library variant, and shared library variant. We will be using the static library variant, because it is much simpler to use and it allows us to split the tests into multiple files.

```
#define BOOST_TEST_MODULE Treewidth support for bgl tests
#include <boost/test/included/unit_test.hpp>
```

Code snippet 4.1: File `test/main_test.cpp`.

First, we will need to define the main translation unit, that will run all other test files. In our project it is file `test/main_test.cpp` (see Code snippet 4.1). Now that we have defined our main translation unit, we can finally create test files for each function implemented in Chapter 3. Each test file will consist of *Fixture* (we will discuss them in detail in Subsection 4.1.2) and *Test suites* [15]. The tests will have to include library `graph_utility.hpp`, otherwise the main translation unit will not be able to run them.

Inside the test file there will be one *Test suite* for each function tested (mostly helper function used by the main function). *Test suites* can contain multiple test cases. For a better understanding of the test structure, please see Figure 4.1.

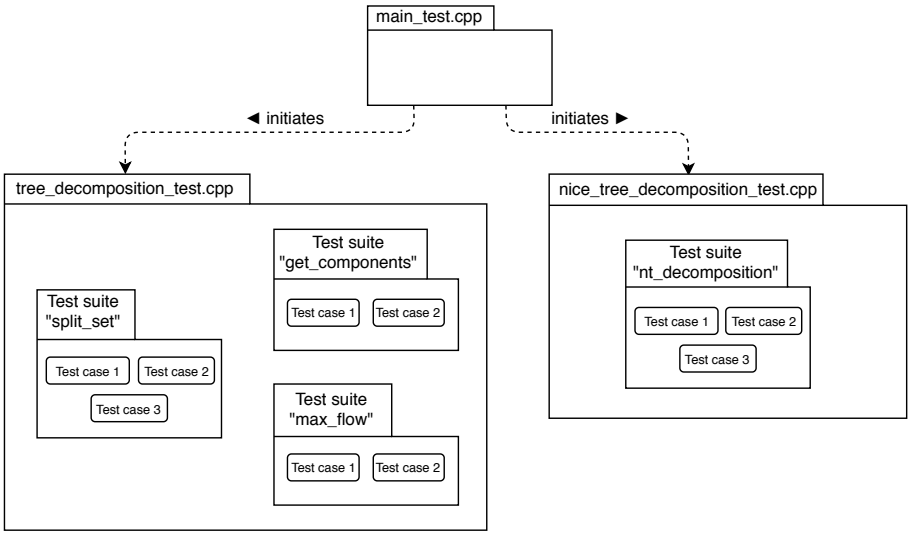


Figure 4.1: Structure of tests used in our library extension.

4.1.2 Fixtures

In general terms a test fixture or test context is the collection of one or more of the following items, required to perform the test [28]:

- Preconditions.
- Particular states of tested units.
- Necessary clean-up procedures.

[15] *Test suite* can be understood as a some kind of wrapper, that keeps all test cases of certain function, that is being tested, in one bundle. More information available from: `https://www.boost.org/doc/libs/1_72_0/libs/test/doc/html/boost_test/tests_organization/test_tree/test_suite.html`

The main advantage of fixtures is in their re-usability—say, we want to test several different functions on the same graph. Instead of re-declaring the graph in each function (which is against DRY[16] principle), we will define the graph only once in the fixture and then pass the fixture inside a *Test suite*, where it can be used in multiple test cases.

In BTL, a fixture is represented by a `struct`. Fixture data are stored in `struct` as *members*. Each time a fixture is required by a *Test suite*, procedure `setup` is initiated (in case of BTL, the constructor of `struct`). After finishing all test cases in a *Test suite*, procedure `teardown` is initiated (destructor of `struct`).

In our case, fixtures are mostly used to define multiple graphs or decompositions and their bags, that will be used in a *Test suite*. Example of a such a fixture is shown in Code snippet 4.2.

```
struct Fixtures_d {
  Fixtures_d() {
    construct_g1();
    construct_g2();
  }
  ~Fixtures_d() = default;

  void construct_g1() {...}
  void construct_g2() {...}
};

BOOST_FIXTURE_TEST_SUITE(split_set, Fixtures_d)
...
BOOST_AUTO_TEST_SUITE_END();
```

Code snippet 4.2: Example of a *fixture* that is „injected“ into a *Test suite*.

4.1.3 Testing of the functions

In each test case we are testing the result of a single call of a function that is being tested. Take a testing of helper function `max_flow_sep` (described in Subsection 3.1.3) as an example—inside the test case we are asserting that the retrieved separation has a correct size and contains expected vertices (see Code snippet 4.3).

[16]DRY (Don't repeat yourself) principle is a principle that focuses on reducing the repetition of information of all kind (e.g., code duplication).

```
BOOST_AUTO_TEST_CASE(maxflow_g2_case2)
{
  subgraph<Graph> sg;
  copy_graph(g2, sg);
  auto separation = detail::max_flow_sep(...);
  BOOST_CHECK_EQUAL(1, separation.size());
  BOOST_CHECK_EQUAL(1, separation.count(2) +
  ↪  separation.count(3));
}
```

Code snippet 4.3: Testing of the function `max_flow_sep`.

Some of the functions are a bit too complex to test them in such a simple manner—one of those functions is `tree_decomposition`. As we learnt in Section 3.1, its purpose is to find the decomposition of a given graph. However decompositions are not generally unique, there can be many different decompositions of the graph (e.g., trivial decomposition contains all vertices in a single root node). That is why instead of comparing retrieved decomposition with a reference decomposition, we will test, if it meets following conditions that basically define tree decompositions:

1. Width of the decomposition is at most $3k + 4$.
2. Every vertex is in at least one bag.
3. Every edge is in at least one bag.
4. Set of nodes containing any vertex v induces a connected component.
5. Decomposition is a tree.

If all of the listed conditions are met, the decomposition is valid. The same approach is used when testing the function `nice_tree_decomposition` (more conditions are added since a nice tree decomposition is more strict than general decomposition).

4.1.4 Continuous integration

Since our library extension is hosted on GitLab, managing continuous integration is relatively easy. With each push or merge request GitLab runs a pipeline of scripts to build and test our library extension. The pipeline and its content is defined in file `.gitlab-ci.yml` located in the root of the project.

Our pipeline is divided into two stages:

1. **Build**. In this phase all of the source files are compiled using the `make compile` command.
2. **Test**. In this phase the tests are run using the `make test` command.

If any of these two stages fails, GitLab prevents merging into *master* branch until the errors are fixed and both of the stages pass (this is a default behaviour for protected branches [29]). This prevents from pushing faulty code into the *master* branch. The status of the pipeline can be viewed on GitLab website (see Figure 4.2).

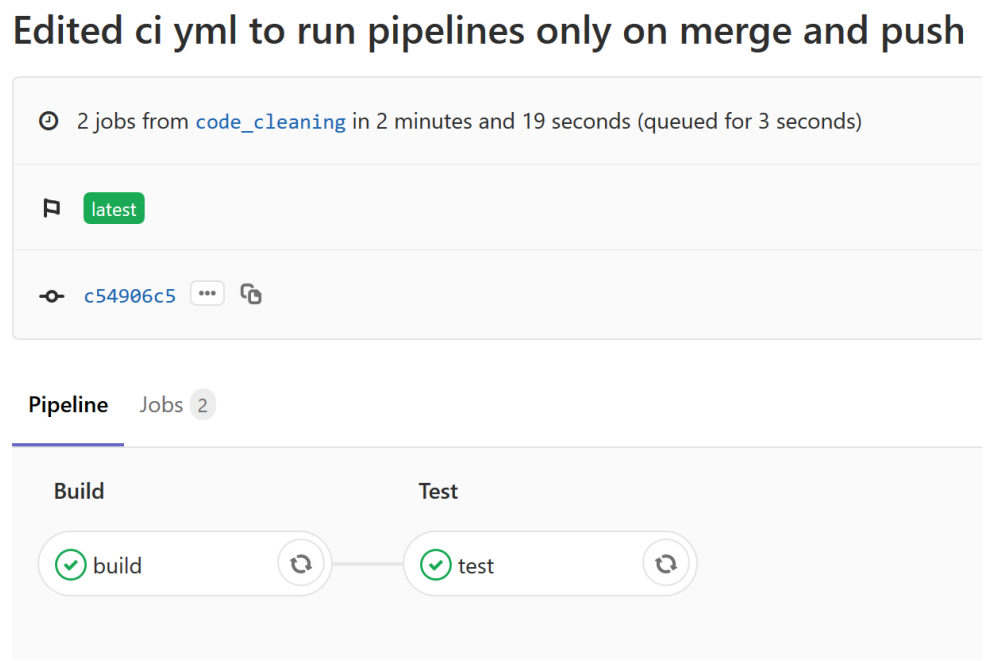


Figure 4.2: Status of the pipeline and its stages.

4.1.5 Fulfilling of the requirements

In this subsection we will review the fulfilment of the functional and non-functional requirements that we have defined in Section 2.1. Each requirement will be discussed in a separate paragraph:

Requirement F1. Algorithm for obtaining a tree decomposition was successfully implemented in a file `src/tree_decomposition.hpp`. Its implementation is discussed in Section 3.1.

Requirement F2. Algorithm for obtaining a nice tree decomposition from general tree decomposition was successfully implemented. General decomposition is supplied to the function as an input (see Section 3.2). File `nice_src/-tree_decomposition.hpp` contains implementation of the algorithm.

Requirement F3. Algorithm for solving the maximum weighted independent set was chosen as a demonstration of usage of dynamic programming over a tree decomposition. The implementation of the algorithm is located in `src/max_weighted_independent_set.hpp`. For more details see Section 3.4.

Requirement F4. Every function defined in our library extension uses template parameters, so that user is not bound to certain type of graph. This also applies to other data structures, e.g., *Property Map* representing bags of the decomposition.

Requirement F5. Exceptions are thrown only if the user provides an input, that would cause an infinite loop—e.g., a decomposition that is not a tree in the function `src/max_weighted_independent_set.hpp`. Incorrect template arguments cause a compilation error thanks to usage of concept checks.

Requirement F6. Algorithm `nice_tree_decomposition` is not bound to any algorithm for obtaining a tree decomposition. Instead, tree decomposition is supplied as an input (therefore avoiding the decision to use specific decomposition algorithm) as opposed to retrieving the decomposition as a part of the algorithm.

Requirement N1. Our library extension follows most of the conventions used by BGL, e.g., code structure, naming conventions (words separated by „_“ instead of using camel-case) or indentation. As it is customary for BGL, each of the algorithms is implemented and defined in a header-file.

Requirement N2. Each of the algorithms used in our library extension uses only C++ Standard Library or Boost libraries—no other third-party libraries were used.

Requirement N3. Documentation of the interface is provided in form of HTML pages generated by Doxygen (see Section 4.2).

Requirement N4. Algorithms and helper-functions used in those algorithms are tested by BTL (note that not every helper-function is tested, because they are either a wrapper or they cannot be tested). Tests are located in directory `test/` and they are initiated by `test/main_test.cpp`. These tests are also a part of continuous integration (see Subsection 4.1.4).

4.2 Documentation

Documentation of this library is supposed to serve as a reference guide for users, that will be using the functions provided by our extension. This is the main reason why only the interfaces of those functions are described and not the helper-functions, since they are irrelevant to the user. Documentation is generated by Doxygen—a tool frequently used for documentation of C++ source codes.

It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual from a set of documented source files. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code [30].

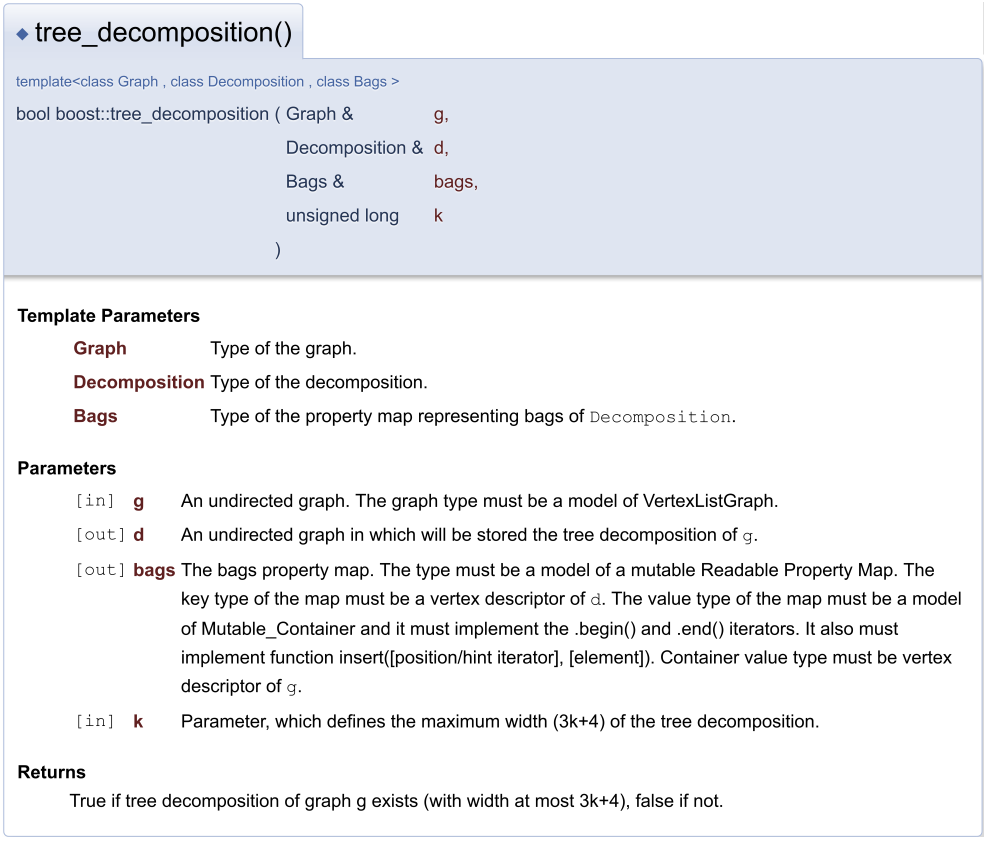
Source files have to be described with specific annotations[17] in order to be documented (see Code snippet 4.4 for an example).

```
/**
 * @tparam Graph Type of the graph.
 * @tparam Decomposition Type of the decomposition.
 * @tparam Bags Type of the property map representing bags ...
 * @param[in] g An undirected graph. The graph type must be ...
 * @param[out] d An undirected graph in which will be ...
 * @param[out] bags The bags property map. The type must be ...
 * @param[in] k Parameter, which defines the maximum width ...
 * @return True if tree decomposition of graph g exists ...
 */
template <class Graph, class Decomposition, class Bags>
bool tree_decomposition(Graph & g, Decomposition & d, Bags &
↪  bags, unsigned long k)
```

Code snippet 4.4: Annotation of the function `tree_decomposition`.

Generated documentation is located in a directory `doc/html/` and available from `doc/index.html`, which redirects to a list of functions provided by our library extension. For each function there is a description of its purpose, used libraries, (template) parameters, and the return value. Segment of documented function `tree_decomposition` is shown in Figure 4.3.

[17] Annotation is a special type of comment block that contains additional or meta-information in the source code.



◆ tree_decomposition()

template<class Graph , class Decomposition , class Bags >

bool boost::tree_decomposition (Graph & g,
Decomposition & d,
Bags & bags,
unsigned long k
)

Template Parameters

Graph	Type of the graph.
Decomposition	Type of the decomposition.
Bags	Type of the property map representing bags of `Decomposition`.

Parameters

`[in]`	**g**	An undirected graph. The graph type must be a model of VertexListGraph.
`[out]`	**d**	An undirected graph in which will be stored the tree decomposition of `g`.
`[out]`	**bags**	The bags property map. The type must be a model of a mutable Readable Property Map. The key type of the map must be a vertex descriptor of `d`. The value type of the map must be a model of Mutable_Container and it must implement the .begin() and .end() iterators. It also must implement function insert([position/hint iterator], [element]). Container value type must be vertex descriptor of `g`.
`[in]`	**k**	Parameter, which defines the maximum width (3k+4) of the tree decomposition.

Returns

True if tree decomposition of graph g exists (with width at most 3k+4), false if not.

Figure 4.3: Example of the documentation generated from annotation presented in Code snippet 4.4.

Conclusion

The goal of this thesis was to extend the C++ Boost Graph Library with algorithms for obtaining a general/nice tree decomposition along with algorithm, that demonstrates the usage of dynamic programming over a tree decomposition, which was successfully fulfilled along with all of the functional and non-functional requirements (as we discussed in Subsection 4.1.5).

Heavy emphasis was put on genericity of the functions provided by our library extension, which was achieved using the C++ templates and concepts defined by BGL—functions can be used with various types of graphs and data structures.

There is lot of space for further improvement of our library extension:

- Optimization of used algorithms.
- Provide more algorithms that use a tree decomposition to improve their performance.
- Make the parameter k of the function `tree_decomposition` optional.
- Implement exact FPT algorithm for obtaining a tree decomposition, allowing the user to choose, whether exact or approximate FPT algorithm should be applied.

This work, including the thesis, source-codes, and documentation is uploaded on the attached SD card. The work is also available from faculty GitLab in repository:
`https://gitlab.fit.cvut.cz/kralva10/treewidth-support-for-bgl`

Bibliography

1. KARP, Richard M. Reducibility among Combinatorial Problems. In: ed. by MILLER, Raymond E.; THATCHER, James W.; BOHLINGER, Jean D. Boston, MA: Springer US, 1972, pp. 85–103. ISBN 978-1-4684-2001-2. Available from DOI: 10.1007/978-1-4684-2001-2_9.
2. CHEN, G.; KUO, M.; SHEU, Jang-Ping. An optimal time algorithm for finding a maximum weight independent set in a tree. Vol. 28, pp. 353–356. Available from DOI: 10.1007/BF01934098.
3. ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2011. Available also from: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.
4. ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2017. Available also from: https://www.iso.org/standard/68564.html.
5. Genericity in the Boost Graph Library. In: *Boost* [online] [visited on 2020-03-01]. Available from: https://www.boost.org/doc/libs/1_72_0/libs/graph/doc/index.html.
6. Graph Concepts. In: *Boost* [online] [visited on 2020-03-01]. Available from: https://www.boost.org/doc/libs/1_72_0/libs/graph/doc/graph_concepts.html.
7. Subgraph class. In: *Boost* [online] [visited on 2020-03-26]. Available from: https://www.boost.org/doc/libs/1_72_0/libs/graph/doc/subgraph.html.
8. Subgraph. In: *Boost* [online] [visited on 2020-05-02]. Available from: https://www.boost.org/doc/libs/1_72_0/libs/graph/doc/subgraph.html.

9. CYGAN, Marek; FOMIN, Fedor V.; KOWALIK, Lukasz; LOKSHTANOV, Daniel; MARX, Dániel; PILIPCZUK, Marcin; PILIPCZUK, Michal; SAURABH, Saket. *Parameterized Algorithms*. Springer, 2015. ISBN 978-3-319-21274-6. Available from DOI: 10.1007/978-3-319-21275-3.

10. ARNBORG, Stefan; CORNEIL, Derek G.; PROSKUROWSKI, Andrzej. Complexity of Finding Embeddings in a k-Tree. *SIAM Journal on Algebraic Discrete Methods*. 1987, vol. 8, no. 2, pp. 277–284. Available from DOI: 10.1137/0608024.

11. REED, B. A. Algorithmic Aspects of Tree Width. In: *Recent Advances in Algorithms and Combinatorics*. Ed. by REED, Bruce A.; SALES, Cláudia L. New York, NY: Springer New York, 2003, pp. 85–107. ISBN 978-0-387-22444-2. Available from DOI: 10.1007/0-387-22444-0_4.

12. Container Concept Checking Classes. In: *Boost* [online] [visited on 2020-04-04]. Available from: https://www.boost.org/doc/libs/1_72_0/libs/concept_check/reference.htm#container-concepts.

13. LAMM, Sebastian; SCHULZ, Christian; STRASH, Darren; WILLIGER, Robert; ZHANG, Huashuo. Exactly Solving the Maximum Weight Independent Set Problem on Large Real-World Graphs. In: *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*. SIAM, 2019, pp. 144–158. Available from DOI: 10.1137/1.9781611975499.12.

14. VALLA, T.; MATOUŠEK, J. Kombinatorika a grafy. In: *Skripta, KAM MFF UK* [online]. Praha, 2008, pp. 21–22 [visited on 2020-03-29]. Available from: https://iuuk.mff.cuni.cz/~valla/kg.pdf.

15. Creating Concept Checking Classes. In: *Boost* [online] [visited on 2020-04-04]. Available from: https://www.boost.org/doc/libs/1_72_0/libs/concept_check/creating_concepts.htm.

16. std::vector::insert. In: *cppreference* [online] [visited on 2020-04-04]. Available from: https://en.cppreference.com/w/cpp/container/vector/insert.

17. std::deque::insert. In: *cppreference* [online] [visited on 2020-04-04]. Available from: https://en.cppreference.com/w/cpp/container/deque/insert.

18. std::list::insert. In: *cppreference* [online] [visited on 2020-04-04]. Available from: https://en.cppreference.com/w/cpp/container/list/insert.

19. std::set::insert. In: *cppreference* [online] [visited on 2020-04-04]. Available from: https://en.cppreference.com/w/cpp/container/set/insert.

20. std::map::insert. In: *cppreference* [online] [visited on 2020-04-04]. Available from: https://en.cppreference.com/w/cpp/container/map/insert.

21. multiset::insert. In: *cppreference* [online] [visited on 2020-04-04]. Available from: https://en.cppreference.com/w/cpp/container/multiset/insert.

22. std::multimap::insert. In: *cppreference* [online] [visited on 2020-04-04]. Available from: https://en.cppreference.com/w/cpp/container/multimap/insert.

23. std::unordered_set::insert. In: *cppreference* [online] [visited on 2020-04-04]. Available from: https://en.cppreference.com/w/cpp/container/unordered_set/insert.

24. std::unordered_map::insert. In: *cppreference* [online] [visited on 2020-04-04]. Available from: https://en.cppreference.com/w/cpp/container/unordered_map/insert.

25. std::unordered_multiset::insert. In: *cppreference* [online] [visited on 2020-04-04]. Available from: https://en.cppreference.com/w/cpp/container/unordered_multiset/insert.

26. unordered_multimap::insert. In: *cppreference* [online] [visited on 2020-04-04]. Available from: https://en.cppreference.com/w/cpp/container/unordered_multimap/insert.

27. Usage variants. In: *Boost* [online] [visited on 2020-03-12]. Available from: https://www.boost.org/doc/libs/1_72_0/libs/test/doc/html/boost_test/usage_variants.html.

28. Fixtures. In: *Boost* [online] [visited on 2020-03-12]. Available from: https://www.boost.org/doc/libs/1_72_0/libs/test/doc/html/boost_test/tests_organization/fixtures.html.

29. Security on protected branches. In: *gitlab* [online] [visited on 2020-04-06]. Available from: https://docs.gitlab.com/ee/ci/pipelines/#security-on-protected-branches.

30. Doxygen main page. In: *Doxygen* [online] [visited on 2020-03-13]. Available from: http://www.doxygen.nl/index.html.

Appendix A

Acronyms

2D Two-Dimensional

BCCL Boost Concept Check Library

BGL Boost Graph Library

BTL Boost Test Library

DRY Don't Repeat Yourself

FPT Fixed-Parameter Tractable

GCC GNU Compiler Collection

GNU GNU's Not Unix

HTML HyperText Markup Language

OS Operating System

PNG Portable Network Graphics

PRNG Pseudo-Random Number Generator

SD Secure Digital

SDLC Software Development Life Cycle

STL Standard Template Library

TV Television

UML Unified Modelling Language

Appendix B

Installation instructions

In this appendix we will describe contents, requirements, and usage of our library extension.

B.1 Contents

Our library extension consists of three header files located in directory `src/`:

- `tree_decomposition.hpp` containing a function for retrieving a tree decomposition.
- `nice_tree_decomposition.hpp` containing a function for retrieving a nice tree decomposition.
- `max_weighted_independent_set.hpp` containing a function that computes a maximum weighted independent set.

Documentation of those three header files is located in `doc/index.html`. Examples can be found in directory `example/`. Lastly, `Makefile` which is used to build example and tests is in the root of the project.

B.2 Requirements

Our library extension was developed on Linux OS using the GCC compiler (version C++14) and GNU Make. Successful compilation with older versions of GCC or different OS is not guaranteed.

Provided header files require *Boost 1.72.0*[18] library (older versions might not be compatible). Library can be installed using the command:

```
sudo apt-get install libboost-dev
```

[18] `https://www.boost.org/users/history/version_1_72_0.html`

B.3 Usage example

In order to help the users to get more familiar with our library extension we also provide examples of usage. The file with the examples is located in `example/example.cpp` and contains a sample graph on which are applied functions that we presented earlier in this chapter.

The form of the output of the example depends on the boolean variable `SAVE_GRAPHS`. If it is set to *false*, the graphs (e.g., constructed tree decompositions) will be printed in text form into `stdout`. Otherwise, the graphs will be saved into *.dot*[19] files in the `example/` directory. In addition, if the user has installed the *Graphviz* library[20], PNG files will be created from *.dot* files using the `dot` command. *Graphviz* library can be installed using the command:

```
sudo apt-get install graphviz
```

B.4 Make commands

As we mentioned, to compile and run our library extension together with example or tests we are using GNU Make. See Table B.1 for available `make` commands that can be run from the root of the project.

Command	Usage
`make all`	Compile both example and tests.
`make compile`	Same as `make all`
`make doc`	Generate documentation [21]
`make run`	Compile and run example.
`make test`	Compile and run tests.
`make help`	Show help.
`make run`	Clean binaries and `obj/` directory.

Table B.1: Available `make` commands used to build example and tests.

[19] DOT is a graph description language used to represent graphs in a simple text form.
[20] Available from `https://www.graphviz.org/download/`
[21] Doxygen version 1.8.17 or newer required.

Appendix C

Contents of enclosed SD card

- `README.md` ..... the file with SD card content description
- `project` ..... the directory with the source files
  - `doc` ..... the directory with documentation
  - `example` ..... the directory with examples of usage
  - `src` ..... the directory of source codes
  - `test` ..... the directory with tests
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format