FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Structural properties of sidewalk networks
Student:	Vojtěch Kopal
Supervisor:	Ing. Šimon Schierreich
Study program:	Informatics
Branch / specialization:	Computer Science 2021
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2025/2026

Instructions

1. Získejte data o chodníkových sítí pro různá města.
2. Převeďte data do vhodné grafové reprezentace.
3. Nastudujte alespoň 5 různých strukturálních parametrů, u kterých se dá očekávat, že hodnota těchto parametrů bude pro získané sítě malá (např. tree-width, feedback-vertex set number, feedback-edge set number, vertex cover number, distance to disjoint stars, a jiné [1]).
4. Pomocí existujících či vlastních solverů zjistěte hodnotu těchto parametrů na zpracovaných datech o chodníkových sítích a diskutujte získané výsledky.

[1] https://vaclavblazej.github.io/parameters/html/

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 7 March 2024 in Prague.

Bachelor's thesis

STRUCTURAL PROPERTIES OF SIDEWALK NETWORKS

Vojtěch Kopal

Faculty of Information Technology
Department of Theoretical Computer Science
Supervisor: Ing. Šimon Schierreich
May 16, 2024

Czech Technical University in Prague
Faculty of Information Technology
© 2024 Vojtěch Kopal. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Kopal Vojtěch. *Structural properties of sidewalk networks.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

Acknowledgments vi

Declaration vii

Abstract viii

List of Abbreviations ix

Introduction 1

1 Graph Theory 2
- 1.1 Graph . . . 2
- 1.2 Subgraph and Induced Subgraph . . . 3
- 1.3 Graph Families and Named Graphs . . . 3
  - 1.3.1 Path . . . 4
  - 1.3.2 Connected Graph . . . 4
  - 1.3.3 Cycle . . . 4
  - 1.3.4 Trees and Forests . . . 5
  - 1.3.5 Petersen Graph . . . 6
- 1.4 Graph Properties . . . 7
  - 1.4.1 Resemblance of a Tree/Forest . . . 7
  - 1.4.2 Graph Covering Numbers . . . 10

2 Computational Complexity 13
- 2.1 Input Encoding . . . 13
- 2.2 Decision Problem . . . 13
- 2.3 Computation . . . 14
- 2.4 Turing Machines . . . 14
  - 2.4.1 k-tape Turing Machine . . . 14
  - 2.4.2 Deterministic k-tape Turing Machine . . . 14
  - 2.4.3 Non-deterministic k-tape Turing Machine . . . 14
  - 2.4.4 Computation of Turing Machines . . . 15
- 2.5 Complexity Classes . . . 15
  - 2.5.1 P Complexity Class . . . 16
  - 2.5.2 NP Complexity Class . . . 16
  - 2.5.3 NP-hardness and NP-completeness . . . 17
  - 2.5.4 Relation of P and NP Complexity Classes . . . 17
- 2.6 Notable NP-hard Problems . . . 17

3 Linear Programming 19
- 3.1 Linear Program . . . 19
- 3.2 Algorithms . . . 19
- 3.3 Example . . . 20
  - 3.3.1 Graphical Representation . . . 20

3.3.2 Simplex Algorithm . . . 21
3.4 Integer Linear Programming . . . 21
3.5 Gurobi Optimization . . . 21

4 Previous Research of Sidewalk and Pedestrian Networks 22

5 Obtaining the Data 24
5.1 Data Formats . . . 24
5.1.1 XML . . . 24
5.1.2 JSON . . . 24
5.1.3 GR . . . 26
5.2 OpenStreetMap . . . 26
5.2.1 History . . . 27
5.2.2 Format . . . 27
5.2.3 APIs . . . 29
5.2.4 OSMnx . . . 31
5.3 Collecting and Serializing the Data . . . 31
5.3.1 Analysed Locations of the World . . . 31
5.3.2 Usage of OSMnx Package . . . 32

6 Graph Properties Measurement and Evaluation 35
6.1 Feedback Edge Set Number . . . 35
6.1.1 Motivation . . . 35
6.1.2 Measurement . . . 35
6.1.3 Results . . . 36
6.2 Feedback Vertex Set Number . . . 36
6.2.1 Motivation . . . 37
6.2.2 Measurment . . . 37
6.2.3 Results . . . 38
6.3 Treewidth . . . 38
6.3.1 Motivation . . . 38
6.3.2 Measurment . . . 39
6.3.3 Results . . . 39
6.4 Vertex Cover Number . . . 39
6.4.1 Motivation . . . 39
6.4.2 Measurment . . . 40
6.4.3 Results . . . 40
6.5 Edge Cover Number . . . 40
6.5.1 Motivation . . . 40
6.5.2 Measurement . . . 41
6.5.3 Results . . . 41

7 Conclusion 42

List of Figures

1.1 An example of graph visualization . . . 3
1.2 An example of a path subgraph and a path . . . 4
1.3 An example of a cycle as a subgraph of a graph . . . 5
1.4 An example of a tree . . . 6
1.5 Petersen graph . . . 6
1.6 Petersen graph with the spanning tree and the feedback edge set marked . . . 8
1.7 Petersen graph with the feedback vertex set marked and removed . . . 9
1.8 A tree decomposition of Petersen graph . . . 10
1.9 Petersen graph with the optimal vertex cover . . . 11
1.10 Petersen graph with the optimal edge cover . . . 12

3.1 A graphical representation of a linear program . . . 20
3.2 A visualization of the objective function . . . 21

5.1 A map from OpenStreetMap project . . . 27
5.2 A visualization of the data obtained from OSM for Dejvice, Prague, Czech Republic. 34

List of Tables

6.1 Measured values of feedback edge set number . . . 36
6.2 Measured values of feedback vertex set number . . . 38
6.3 Measured values of treewidth . . . 39
6.4 Measured values of vertex cover number . . . 40
6.5 Measured values of edge cover number . . . 41

7.1 Concluding table . . . 42

List of code listings

5.1 An example of XML document. . . . 25
5.2 An example of JSON representation. . . . 25
5.3 GR representation of Petersen graph . . . 26

5.4 An example of a single tag from OSM 28
5.5 An example of a single node feature from OpenStreetMap 28
5.6 An example of a way feature from OpenStreetMap 29
5.7 An example of a relation from OpenStreetMap 30
5.8 Python script exporting and visualizing the sidewalk data from OSM. 33

First of all, I want to thank my supervisor Ing. Šimon Schierreich, for helping me curating this thesis into its final appearance and patiently helping me understand crucial concepts of computer science.

I want to thank Jürgen Klopp, for teaching me, through the art of his football, the importance of hard work and resilience in life. I wish him all the best on his well-earned retirement.

I want to thank my boss Vláďa, for having faith in me in the last two years, being the greatest boss I could have ever wished for, and a great human being.

I want to thank my classmates and friends: Aleš, my sister Alice, Ferfa, Patrik and Setni, for helping me throughout my studies.

I also want to thank Pavel, his girlfriend Terka, and their friends, for helping me discover great people, fun activities and most importantly my own personality.

And finally, I want to thank my parents, for their never-ending care of my well-being.

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 the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act

In Prague on May 16, 2024

Abstract

This work analyses real world sidewalk networks from the perspective of graph theory. We start by obtaining the sidewalk data from OpenStreetMap project, serializing them into multiple data formats. We measure various complex structural graph properties of sidewalk networks from diverse places on the planet Earth, using modern solvers for integer linear programming and other NP-hard problems. Based on the results, we are providing estimations of relationships between graph properties that are computationally complex to measure. We are also giving numerous points towards designing an algorithm that would be able to create a realistic artificial sidewalk network for a given road network and building outlines.

Keywords sidewalk networks, graph theory, structural properties, feedback edge set, feedback vertex set, treewidth, vertex cover, edge cover

Abstrakt

V této práci analyzujeme reálné chodníkové sítě z pohledu grafové teorie. Začínáme získáním dat chodníkových sítí z OpenStreetMap a jejich serializací do různých datových formátů. Za pomocí řešičů celočíselného lineárního programování a řešičů jiných NP-těžkých problémů, měříme rozličné netriviální grafové vlastnosti chodníkových sítí z různých míst na planetě Zemi. Na základě našich výsledků, přinášíme odhady vztahů mezi parametry grafů, jejichž měření je výpočetně náročné. Zároveň přinášíme několik pozorování pro tvorbu algoritmu, jež by byl schopen vygenerovat chodníkovou síť ze silniční sítě a půdorysů budov.

Klíčová slova chodníkové sítě, grafová teorie, strukturální vlastnosti, feedback edge set, feedback vertex set, treewidth, vertex cover, edge cover

List of Abbreviations

3D	Three-dimensional
API	Application programming interface
CA	California
CZE	Czech Republic
DTIME	Deterministic time
EC	Edge cover
ECN	Edge cover number
ESP	Spain
FES	Feedback edge set
FESN	Feedback edge set number
FIN	Finland
FRA	France
FVS	Feedback vertex set
FVSN	Feedback vertex set number
GPS	Global positioning system
GR	Graph format
JSON	JavaScript Object Notation
LAT	Latvia
MD	Maryland
NC	North Carolina
NP	Non-deterministic polynomial time
NTIME	Non-deterministic time
OSM	OpenStreetMap
P	Deterministic polynomial time
PACE	Parameterized Algorithms and Computational Experiments
PRG	Prague
TW	Treewidth
VBS	Virtual Battlespaces
VC	Vertex cover
VCN	Vertex cover number
XML	eXtensible Markup Language

Introduction

At first glance, sidewalk networks and graph theory may seem as two very distant topics, which may make one wonder, how does an idea of such thesis even originate?
This work started as analysis for the newest expansion of VBS[1] Blue.
VBS Blue [1] is a large scale project developed by Bohemia Interactive Simulations k.s. creating a hyper-realistic 3D[2] model of the planet Earth, used as an environment for other products of the company. The project originates to year 2008. Since then, it has provided many interesting challenges for engineers developing and maintaining the software.
The state of visual outlook of VBS Blue highly differs around the globe. Many parts of the Earth could be surely described as well and realistically looking. We are mainly talking about vegetation and soil maps. However, the ever-lasting problem of the project are the residential zones. Automatically generated cities are still not perfect and there is a lot of work to be done. One of their limitations surely is the absence of sidewalks independent on roads. The expansion of virtual twin of the planet Earth with a plugin for sidewalks 3D model generation from network data is where the idea for this thesis originates.
One of the future goals for this plugin is to develop an algorithm that would be able to create a realistic sidewalk network to use for 3D model generation from given road network and map of building outlines. This algorithm could be used for places, where the sidewalk network data are not available, but there are available data of road network and building outlines.
Now, when this algorithm would yield its results, we would like to have a comparison with the real world data, to ensure that the generated artificial sidewalk network structurally resembles a real-world sidewalk network. Therefore the real-world sidewalk networks have to be first thoroughly analysed. Not only the algorithm may use the measured structural properties as the sample data to come close to. These properties may be also used for designing such algorithm.
In this work, we are going to discuss the process of obtaining the data of sidewalk networks from different parts of the world and measure many complex graph properties on them. Finishing with the evaluation and analysis of the measured values.

[1] Virtual Battle Spaces
[2] Three dimensional

Chapter 1
Graph Theory

Graph theory is a field of discrete mathematics studying the mathematical structures called *graphs*. Since these mathematical structures, mainly used for modeling of pair relationships between objects, have proven to be quintessential for various fields, including computer engineering and computer science, the theory grew vast and expansive. In the following chapter, we will go through a small selection of topics with direct connection to what we will discuss in our work. For much more concise and in-depth study of this topic, please refer to the monograph of Diestel [2].

1.1 Graph

▶ **Definition 1.1** (Simple undirected graph)**.** *A* simple undirected graph *is an ordered pair* $G = (V, E)$*, where* V *is an arbitrary non-empty set and* E *is a set of unordered pairs of elements contained in set* V*. More formally:* $E \subseteq \binom{V}{2}$*, this notation represents all subsets of* V *of size 2:* $\binom{V}{2} = \{\{x, y\} \mid x, y \in V \wedge x \neq y\}$.

The elements of the set V are called *vertices*, sometimes, they can be also referred to as *nodes* or *points*. The elements of the set E are called *edges*, or more rarely *lines*.
As we have previously declared, graphs are used for modeling of pair relationships between given entities. These given entities are objects contained in the set V. The set E contains the objects that are together in the modeled relationship. Note that since the pairs in E are unordered, the modeled relationships are taken as bilateral. If we are to introduce the concept of a non-symmetrical relationship, we can use the alternative of a *directed graph*.

▶ **Definition 1.2** (Simple directed graph)**.** *A* simple directed graph *is an ordered pair* $G = (V, E)$*, where* V *is an arbitrary non-empty set of vertices and* $E \subseteq V \times V \setminus \{(v, v) \mid v \in V\}$ *is a set of edges.*

Elements of the set V are typically called *vertices*, *nodes* or *points*, like in the case of undirected graph. In a simple directed graph the elements of set E are also typically called *edges*, sometimes, for better specification, they are called *directed edges* or *arcs*.
In this thesis, we will be referring to a simple undirected graph as a graph, as it will be the main article of our research. To simple directed graphs, we will be referring as directed graphs as they will be sporadically mentioned as well. Mainly when comparing some of their properties with their undirected counterparts.

For better understanding, we can visualize the vertices as points drawn on a plane and the edges as lines (or curves) connecting them. An example of this visualization can be seen in Figure 1.1. The positioning and shapes of drawn elements are irrelevant as it is not information usually contained in the graph.

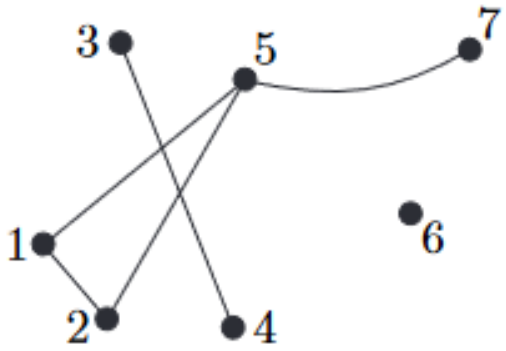

Figure 1.1 An example of graph visualization [2].

1.2 Subgraph and Induced Subgraph

When looking at Figure 1.1, we can notice that the *sub-parts* of the graph can be considered graphs as well. This leads us to an idea of a *subgraph*.

▶ **Definition 1.3** (Subgraph)**.** *Given two graphs* $G = (V, E)$ *and* $H = (U, F)$*. We say* H *is a* subgraph *of* G *if* $U \subseteq V$ *and* $F \subseteq E$*.*

If H is a subgraph of G, we can, slightly more informally, say that G *contains* H. Furthermore, if H is a subgraph of G, we symmetrically say that G is a *supergraph* of H.
Apart from the subgraph we also define an *induced subgraph*.

▶ **Definition 1.4** (Induced subgraph)**.** *Given two graphs* $G = (V, E)$ *and* $H = (U, F)$*, we say* H *is an* induced subgraph *of* H *if* $U \subseteq V$ *and* $F = E \cap \binom{U}{2}$*.*

The difference between the subgraph and the induced subgraph is that the subgraph can contain any edges from its supergraph. On the other hand, the induced subgraph must include all edges existing in its supergraph among the vertices it contains.

1.3 Graph Families and Named Graphs

In this section, we are going to look at some usual graphs and *graph patterns*. Typically, we are talking about a graph with a stand-out structure, or a graph that may represent a shape well-known and common in the real world. One of our main focuses will be to describe the relationships of graphs studied by us and these typical patterns.

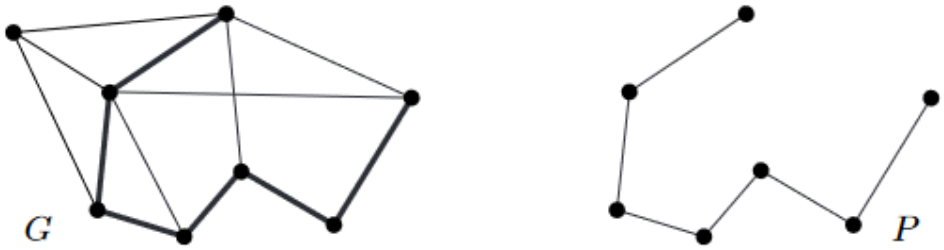

Figure 1.2 An example of a path subgraph (left) and a path (right) [2]

1.3.1 Path

▶ **Definition 1.5** (Path). *A* path *is a non-empty graph* $P = (V, E)$ *such that:*
$V = \{x_1, x_2, \ldots, x_n\}, E = \{\{x_1, x_2\}, \{x_2, x_3\}, \ldots, \{x_{n-1}, x_n\}\}$.

A path is a graph resembling a route or a way between two points. The vertices contained in only one edge (in definition named as x_1 and x_n) are typically referred to as the endpoints of the path. An important concept for us is whether two points in any graph are *connected* by a path. Meaning whether there exists a subgraph of such graph, which is a path with the two mentioned vertices being its endpoints. More formally, we define an *s,t-path*.

▶ **Definition 1.6** (*s,t*-path). *Let* $G = (V, E)$ *be a graph and* $s, t \in V$ *be two vertices (source, target). An* s,t-path *is a sequence* $P = (v_1, v_2, \ldots, v_l)$ *such that* $v_1 = s$, $v_l = t$*; each vertex* $v_i \in V$ *appears in* P *at most once, and* $\{v_i, v_{i+1}\} \in E$ *for every* $1 \leq i \leq (l - 1)$.

Both of these concepts can be seen visualized in Figure 1.2.

1.3.2 Connected Graph

A graph is called *connected*, if there exists a path between any pair of its vertices. More formally:

▶ **Definition 1.7** (Connected graph). *A graph* $G = (V, E)$ *is called* connected *if there exists an s,t-path for all* $s,t \in V$.

A graph which is not connected, is called *disconnected*. Such graph consists of multiple *connected components*.

▶ **Definition 1.8** (Connected component). *A* connected component *is a connected subgraph of graph that is not a part of any larger connected subgraph.*

All connected components of a graph are disjoint and together they add up to the whole graph.

1.3.3 Cycle

▶ **Definition 1.9** (Cycle). *A* cycle *is a graph* $C = (V, E)$ *of pattern:*
$V = \{x_1, x_2, \ldots, x_n\}, E = \{\{x_1, x_2\}, \{x_2, x_3\}, \ldots, \{x_{n-1}, x_n\}, \{x_n, x_1\}\}$ *where* $n \geq 3$.

A cycle is a typical graph resembling a closed shape. We can notice that cycle is very similar to path, just with an edge added between the endpoints. Visualization of a cycle and the existence of a cycle as a subgraph in a graph can be seen in Figure 1.3.

1.3.4 Trees and Forests

What is equally, if not more, important for us is the concept of a graph containing or not containing a cycle as its subgraph.

▶ **Definition 1.10** (Forest). *A graph is called a* forest *if none of its subgraphs is a cycle.*

We will often refer to forests as *acyclic graphs*, due to their characteristic of not containing a cycle. Furthermore, we define a tree.

▶ **Definition 1.11** (Tree). *A graph is called a* tree *if none of its subgraphs is a cycle and it is a connected graph.*

As we can see, every tree is a forest with an additional property of existence of a path between any pair of its vertices.

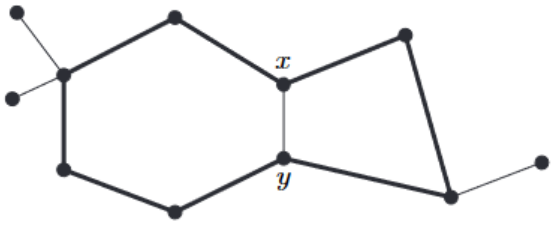

Figure 1.3 An example of a cycle (bold) as a subgraph of a graph [2].

The terminology of trees and forests becomes more clear and natural with a remark, that a forest consists of multiple connected components, in which the additional property holds. These components inherit the property of not containing cycles, making them trees. Therefore a forest consists of multiple trees and a tree is a forest of a single tree. Multiple definitions of a tree exist, all of them describing the same set of graphs. One of these alternative definitions is:

▶ **Definition 1.12** (Tree). *A graph $G = (V, E)$ is called a tree if it is a connected graph and $|V| = |E| - 1$.*

Both definitions and their equivalency are shown in the monograph of Diestel [2].
An example of visualization of a tree can be seen in Figure 1.4.

1.3.4.1 Spanning Trees and Forests

A *spanning* subgraph is a subgraph that contains all vertices of its supergraph. A spanning tree is a subgraph which is a tree.

▶ **Definition 1.13** (Spanning tree). *A* spanning tree *of graph $G = (V, E)$ is a subgraph $T = (V, E')$, where $E' \subseteq E$ and T is a tree.*

Existence of a spanning tree requires the existence of a path between any two vertices in the original graph. We can notice that a graph for which this property holds, can have more than one spanning tree.
Very similarly, we can define the spanning forest as a forest containing all vertices of its supergraph. Spanning forests do not have any special requirements for the original graph. We can find a spanning forest of any graph, and, once again, for a graph, multiple spanning forests may exist.

▶ **Definition 1.14** (Spanning forest). *A* spanning forest *of graph $G = (V, E)$ is defined as a subgraph $F = (V, E')$, where $E' \subseteq E$, F is a forest and for any two vertices holds, that if they were connected by a path in G, then they are also connected by a path in F.*

Figure 1.4 An example of a tree [2].

1.3.5 Petersen Graph

Petersen graph is an interesting example of a graph. It was constructed in [3] by Danish mathematician Julius Petersen, after whom it is named. Over the years, it has proven to be a typical counter-example for various propositions. In the next section, we are going to define many graph properties. For better explanation, we are going to demonstrate their value and the calculation procedure on this particular example.

▶ **Definition 1.15.** *Petersen graph is a graph*

$$G = (V = \{u_0, \ldots, u_4\} \cup \{v_0, \ldots, v_4\}, E = \{\{u_i, u_{i+1 \bmod 5}\}\{v_i, v_{i+2 \bmod 5}\}, \{u_i, v_i\} \mid 0 \leq i \leq 4\}$$

The visualization of this graph can be seen in Figure 1.5.

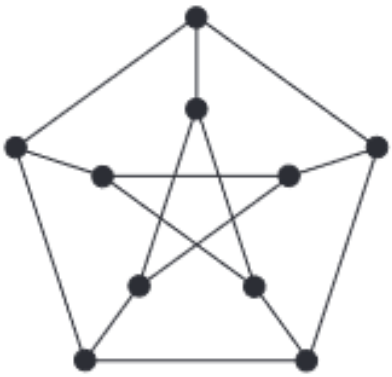

Figure 1.5 Petersen graph [2].

1.4 Graph Properties

We can define graph properties. Functions returning a value for any graph on which the property is defined.

1.4.1 Resemblance of a Tree/Forest

As we have declared previously, one of the focuses of our interest will be measuring the *distance* of a studied graphs to some of the particular graphs or members of graph families mentioned earlier. In this section, we define a few graph properties, stating how closely a graph resembles a tree or a forest.

1.4.1.1 Feedback Edge Set Number

Feedback edge set number is a property assigning any graph a numerical value equal to the minimum number of edges that have to be removed from a graph to create a forest (graph not containing any cycles). More formally, we define *the feedback edge set* as follows.

▶ **Definition 1.16** (Feedback edge set [4])**.** *For a given graph $G = (V, E)$, we define the* feedback edge set*, as $F \subseteq E$ where $G' = (V, E \setminus F)$ is a forest.*

Now, we can continue with defining the *feedback edge set number*.

▶ **Definition 1.17** (Feedback edge set number)**.** *We define the* feedback edge set number*, denoted FESN, of a graph G as the smallest possible cardinality of F across all possible feedback edge sets of a graph G.*

For better explanation, let us take the famous Petersen graph and demonstrate the value of this property on this example. The Petersen graph has the feedback edge set number equal to 6. As there are 6 edges that can be removed in order to create an acyclic graph, and there are no such 5 (or less) edges that would make the graph acyclic after their removal. To discover this value, we first identify any spanning tree (or forest for a disconnected graph) of the graph, which is the maximal acyclic subgraph, and then take the remaining edges as the feedback edge set. The calculation procedure can be better seen in Figure 1.6. The black edges are the 6 edges that have to be removed, the remaining red edges form a spanning tree. We can see that if we would add any of the black edges to the graph, then the graph would contain a cycle. The black edge connects two vertices between which s, t-path surely exists (because the spanning tree is connected). This s, t-path, together with the added black edge would form a cycle. Therefore, the feedback edge set number can not be 5 or lower, as removing 5 edges will surely leave a cycle in the graph. For better description of the computation of feedback edge set number for Petersen graph and other graphs, please refer to [4].

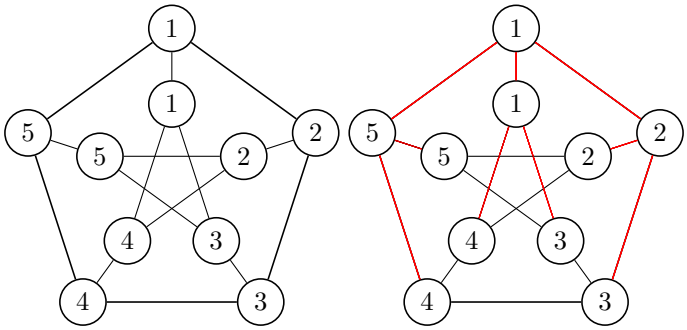

Figure 1.6 Petersen graph with the spanning tree (red) and the feedback edge set (black) marked, taken from [5], modified.

1.4.1.2 Feedback Vertex Set Number

Very similar in terms of definition, but, as we will discuss later, very different in terms of complexity of computation, is the *feedback vertex set number*. Feedback vertex set number is a property assigning any graph a numerical value equal to the minimum number of vertices that have to be removed from a graph to create an acyclic graph (a forest).
First, we define the *feedback vertex set*.

▶ **Definition 1.18** (Feedback vertex set [4]). *For a given graph $G = (V, E)$, we define the* feedback vertex set*, as $F \subseteq V$ where $G' = (V \setminus F, E)$ is a forest.*

Or equivalently:

▶ **Definition 1.19** (Feedback vertex set). *We can also define the* feedback vertex set *for a given graph $G = (V, E)$, as $F \subseteq V$, where vertices of every cycle contained in G have a non-empty intersection with F.*

These two definitions are equivalent. If we hold a set of vertices that includes at least one vertex from every cycle, then removing these vertices will break all the cycles, as the cycles will disconnect on the removed vertex.
If we would have a set of vertices that does not include any vertex from at least one of the cycles in the graph, then such set will not make the graph acyclic after removal, as the cycle not *hit* by the chosen vertices subset, will remain present after the removal of this subset.
Now, we proceed to defining the *feedback vertex set number*.

▶ **Definition 1.20** (Feedback vertex set number). *We define the* feedback vertex set number*, denoted FVSN, of a graph G as the smallest possible cardinality of F across all possible feedback vertex sets of graph G.*

Let us demonstrate the property once again at the notorious Petersen graph. First, we show that Petersen graph has a feedback vertex set of size 3. That can be seen in Figure 1.7. In the second picture, the red vertices are the ones marked for removal. In the third picture, we can see that the graph is acyclic, after the marked vertices have been removed. Note that there are multiple possibilities how to choose such 3 vertices. For example, choosing an outer 2, instead of outer 5, would work as well.
Now, we show that there can not exist a feedback vertex set of size 2. We surely have to break the inner and the outer cycle, so the 2 selected vertices must be from the different parts of the graph. Let us consider without loss of generality that we select vertex number 1 from the outer cycle (for any other vertex, the situation would be identical, the numbers would only slightly shift). Removing vertex 1, 3, or 4 from the inner cycle, would leave us with cycle 5-5-2-2-3-4 and removing vertex 2 or 5, would leave us with cycle 1-1-3-3-2 [4].

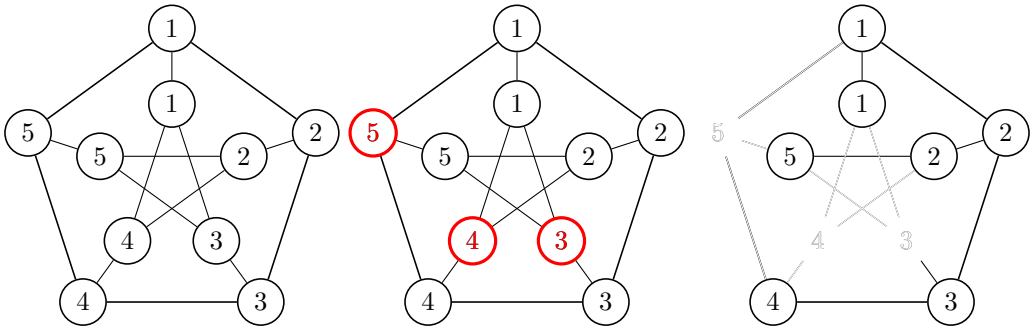

Figure 1.7 Petersen graph (left) with the feedback vertex set marked in red (middle) and removed (right), taken from [5], modified.

1.4.1.3 Treewidth

The treewidth is another numerical property describing how distant is the graph in question to a tree. The distance in this case is not so straight-forward to see like in the case of feedback set numbers. The treewidth describes rather how close is the graph to being a tree when processing the graph algorithmically. This makes the treewidth a very useful parameter for further processing of the graph, as many computations can be simplified, when the treewidth of a graph is known, especially, when the treewidth is low. [2]
Let us start with the definition of a *tree decomposition*, which is a crucial concept for the definition of the treewidth.

▶ **Definition 1.21** (Tree decomposition)**.** *Given a graph $G = (V, E)$, we define its* tree decomposition *as a triple $T = (V_T, E_T, \beta)$, where $\beta : V_T \to 2^V \setminus \{\emptyset\}$. For better clarity, we will refer to elements of V as vertices and elements of V_T as nodes. β is a map mapping nodes of V_T onto subsets of vertices of V. These subsets must satisfy three conditions.*

- $\bigcup_{x \in V_T} \beta(x) = V$;
- *for every edge $\{u, v\} \in E$ there exists a node $n \in V_T$, such that $u \in \beta(n) \wedge v \in \beta(n)$;*
- *for every vertex $v \in V$, a subgraph T' of T, such that $T' = (\{x \mid x \in V_T \wedge v \in \beta(x)\}, \{\{x, y\} \mid v \in \beta(x) \wedge v \in \beta(y) \wedge \{x, y\} \in E_T\})$ is a tree.*

The graph defined as (V_T, E_T) must form a tree.

To put this rather complicated definition into a natural language. We construct a tree with nodes being the subsets of vertices of the original graph. There are three conditions on these subsets. The first item of the three tells us that every vertex from V has to be included in at least one subset from V_T.
The second condition tells us, that if there are two vertices directly connected by an edge in graph G, then there necessarily must exist a subset in V_T containing both of them.
The last required property of the tree decomposition is the tree-ness of subgraphs of T induced by vertices from V. If we take a vertex from V, and then look at subgraph of V_T that includes only nodes containing the selected vertex and edges between these nodes, we must get a tree. This must hold for all vertices from V. Now, we can continue with the definition of width of a tree decomposition and treewidth of a graph.

▶ **Definition 1.22** (Treewidth)**.** *A width of a tree decomposition $T = (V_T, E_T, \beta)$ is equal to* $\max_{v \in V_T} |\beta(v)| - 1$.
A treewidth *of a graph is equal to the smallest possible width across all possible tree decompositions of this graph.*

Let us demonstrate the treewidth on the Petersen graph. The minimal tree decomposition (in terms of its treewidth) can be seen in.
This decomposition is a valid tree decomposition, as the graph is clearly a tree (does not contain any cycle and is connected) and all three required conditions are met. The first property holds, as every number is present in at least one node of the decomposition. The second condition also holds. The vertex labeled by 0 in the original graph is connected by an edge with vertices 1, 4 and 5. Therefore in the tree decomposition, we can identify a node containing 0 and 1, and another node (could be also the same node) containing 0 and 4, and another node containing 0 and 5. This holds for any vertex in the original graph. The last required property also holds. If we take the subgraph of the tree decomposition induced by nodes containing vertex labeled with 2 in the original graph, we get the two bottom-right vertices. These vertices are connected and acyclic, thus they form a tree. This once again holds for all vertices from the original graph. Therefore the shown graph is a valid tree decomposition. The proof that we can not find a tree decomposition with smaller width is shown in [6]. Therefore the treewidth of Petersen graph is equal to 4.

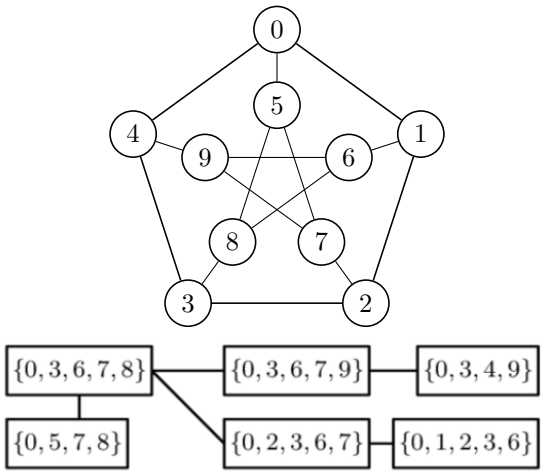

Figure 1.8 A tree decomposition of Petersen graph, [6].

1.4.2 Graph Covering Numbers

Coverings problems typically refer to taking a part of a graph, which, in a certain way, *covers* the whole graph.
The graph properties of graph covering numbers typically refer to numerically expressing the minimal covering of the graph.

1.4.2.1 Vertex Cover Number

Vertex cover is a set of vertices, such that every edge contains at least one vertex of this subset.

▶ **Definition 1.23** (Vertex cover [7]). *Given a graph $G = (V, E)$, we say that a subset of vertices $F \subseteq V$ is a* vertex cover *if* $\forall \{u, v\} \in E : u \in F \vee v \in F$

Then we define the *vertex cover number* as the smallest possible vertex cover for a graph.

▶ **Definition 1.24** (Vertex Cover Number). *We define* vertex cover number *of a graph G as a smallest possible cardinality across all vertex covers of the graph G.*

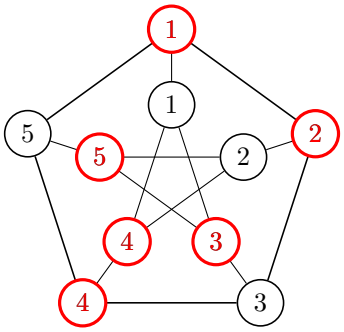

Figure 1.9 Petersen graph with the optimal vertex cover highlighted in red, taken from [5], modified.

For better clarity, we are going to once again demonstrate this property on Petersen graph. Petersen graph has the vertex cover number equal to 6. First, we show that there exists a set of 6 vertices that covers all edges of the graph. Such cover can be seen in Figure 1.9. Next we show that we can not identify a set of size 5 (or smaller) that would cover the whole graph. We can see that every vertex in the Petersen graph is incident with exactly 3 edges. Therefore, any vertex added to the cover, can add at most 3 covered edges. Since the graph has 15 edges, there can not exist a vertex cover of size 4 or smaller, because 4 vertices could only cover at most 12 edges. Should there exist a vertex cover of size 5, than necessarily, each vertex must add all of its 3 incident edges to the set of covered edges. Meaning that each incident edge must not have been added previously to the set by any other vertex in the cover. This implies that the 5 selected vertices must not share a common edge[1]. Now we show that such 5 vertices can not be found in this graph. The graph consists of two cycles of length 5. Once we select 2 vertices from a cycle of length 5, we can not find any other vertex in the cycle that would be connected to neither of selected. Once we select two vertices from the outer cycle and two vertices from the inner cycle, we can not find any other vertex in the graph that would not be incident with the selected vertices. Thus we can not find 5 vertices not sharing a common edge, which means that there can not exist a vertex cover of size 5. The calculation of the vertex cover number of Petersen graph is better explained in [8].

1.4.2.2 Edge Cover Number

Edge cover number also refers to minimal covering of a graph, but this time using its edges. We once again start with defining the covering set. *Edge cover* is a subset of edges, such that for every vertex, there is an edge in the subset, containing it.

▶ **Definition 1.25** (Edge cover [7])**.** *Given a graph $G = (V, E)$ we say that a subset of edges $F \subseteq E$ is an* edge cover *if:* $\forall v \in V : \exists e \in E : e \in F \wedge v \in e$

We then proceed to define the *edge cover number* very similarly as the vertex cover number.

▶ **Definition 1.26** (Edge cover number)**.** *We define the* edge cover number *of a graph G as the smallest possible cardinality across all edge covers of the graph G.*

▶ Remark 1.27. Smallest possible edge cover number for a graph of n vertices is $\lceil \frac{n}{2} \rceil$, as every edge selected into cover is incident with exactly 2 vertices. Therefore a single edge can add at most 2 covered vertices. Assuming there exists an edge cover of size $\lceil \frac{n}{2} \rceil - 1$ or smaller, would necessarily mean that such edge cover covers at most $n - 1$ vertices, which is a contradiction with the edge cover covering all n vertices in the graph.

[1] In other words, they must form an independent set.

For better clarity, we are going to once again demonstrate this property on Petersen graph.
An optimal edge cover for Petersen graph is not hard to find. We can simply take the edges connecting the outside cycle and the inner cycle. Such edge cover has cardinality 5. Since the Petersen graph has 10 vertices, we can not find a smaller edge cover (according to the previous remark); therefore the edge cover number of Petersen graph is 5. The cover can be seen in Figure 1.10

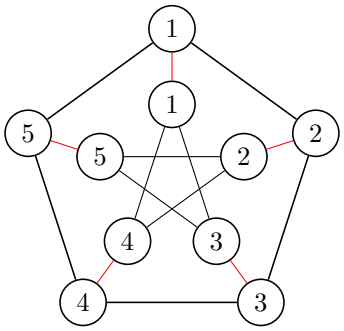

Figure 1.10 Petersen graph with the optimal edge cover highlighted in red, taken from [5], modified.

Chapter 2
Computational Complexity

In this chapter, we are going to discuss theory of computational complexity. Throughout this thesis, we are going to stumble upon many *NP-hard* problems, this chapter gives an explanation, why we have not tried to design an efficient algorithm for these problems, and rather opted to use existing solvers. Although designing an efficient algorithm for these problems is not deemed impossible the question whether such algorithm can exist, is the central undisclosed topic of computer science unsolved for long years.
To keep this chapter brief, we are going to limit ourselves only on the P and NP complexity classes and the related sets of problems. For more information about this topic please refer to the monograph of Arora and Barak [9].

2.1 Input Encoding

In this section, we briefly mention unification of input formats by encoding inputs into a string. First we start by defining few basic concepts of language theory.

▶ **Definition 2.1** (Alphabet)**.** *An* alphabet *is a finite non-empty set of characters.*

▶ **Definition 2.2** (String)**.** *A* string *over an alphabet is a finite sequence of characters of the alphabet.*

▶ **Definition 2.3** (Language)**.** Language L *over the given alphabet* A *is a subset of all possible strings of* A*.*

Our typical alphabet will be binary $\{0, 1\}$, as all the assumed inputs can be encoded as binary strings. When we say that a certain problem has two numbers on the input, we actually mean that on input it has a binary string with these numbers encoded. We are not going to let distract ourselves with the low level details of encoding, as they are not the main object of our study, and focus on the bigger picture.

2.2 Decision Problem

In a *decision problem*, our goal is to decide whether given input satisfies a certain condition. More formally:

▶ **Definition 2.4.** *A* decision problem *is a language* $L = \{x \mid f(x) = 1\}$*, where* f *is a function mapping a string onto 0 or 1, depending on whether the string satisfies a specified condition.*

2.3 Computation

Computation is the process of determining the value of the function for given input. The computation process generally consists of following steps:

- read a character from input;
- read a character from inner memory;
- write a character into inner memory;
- either stop outputting a character, or pick a new rule that will be applied next in the computation process.

2.4 Turing Machines

We are going to use the computation model of *Turing machine* as described by Alan Turing [10].

2.4.1 k-tape Turing Machine

The inner memory of a k-tape Turing machine consists of k tapes, with the tape being an infinite line of memory cells, which can hold a symbol of working alphabet of the machine.
All tapes have their own *head*. In every step of the computation, the head can read symbol from its current cell, write a symbol into the current cell, and/or change the current cell by moving left or right.
The first tape is considered the *input tape*, containing the input of the problem. It is *read-only*, meaning that its head is not capable of writing characters into memory cells.
The last tape is considered the *output tape*. The output of computation is present on the output tape, when the Turing machine *halts*.
The remaining tapes are considered the *working tapes*.

2.4.2 Deterministic k-tape Turing Machine

Let us move to defining the computation model of a deterministic variant of the Turing machine.

▶ **Definition 2.5** (Deterministic k-tape Turing machine)**.** *A* deterministic k-tape Turing machine *is a tuple* $TS = (A, Q, \delta)$*, where*

- A *is the alphabet of the machine containing two special symbols. One of them being a blank symbol* $\square \in A$ *and the other being the start symbol* $\triangleright \in A$*;*
- Q *is the set of states of the machine, with one state* $q_0 \in Q$ *being the start state and one state* $q_{halt} \in Q$ *being the halting state;*
- δ *is a function* $Q \times A^k \to Q \times A^{k-1} \times \{L, S, R\}^k$ *called the transition function.*

2.4.3 Non-deterministic k-tape Turing Machine

Now we will define the non-deterministic counterpart of this computation model.

▶ **Definition 2.6** (Non-deterministic k-tape Turing machine)**.** *A* non-deterministic k-tape Turing machine *is a tuple* $TS = (A, Q, \delta)$*, where*

- A *is the alphabet of the machine containing two special symbols. One of them being a blank symbol* $\square \in A$ *and the other being the start symbol* $\triangleright \in A$*;*

- *Q is the set of states of the machine, with two special states. One of them being again the start state $q_0 \in Q$, and the other being the accept state $q_{accept} \in Q$.*
- *δ is a function $Q \times A^k \to 2^{Q \times A^{k-1} \times \{L,S,R\}^k}$ called the transition function.*

2.4.4 Computation of Turing Machines

The transition function has on its input a state the machine is currently in, and k characters, read by all k heads. For such input, it returns a new state, in which the machine will be now; $k-1$ characters to write by all heads (except for the input tape one), and a command *Left*, *Stay*, or *Right* for all k heads to change their current cell.
At the beginning of the computation, all tapes are filled with the blank symbol $\square$, except for the input tape. Input tape contains a start symbol $\triangleright$ followed by the finite input string. The rest of the input tape is initialized with the blank symbol, just like the other tapes.
The initial state of the machine is the start state q_0. The machine then applies the transition function as explained above, until it reaches the halting state. After the Turing machine reaches its halting state, it halts and the contents of the tapes are not modified further. It is also possible for a certain input that the Turing machine may never reach the halting state, in that case the Turing machine never halts, therefore it does not accept the input.

2.4.4.1 Computation of Non-deterministic Turing Machines

In the non-deterministic variant of the Turing machine the transition function returns the same instructions as the deterministic variant (new state, characters to write, commands for heads), however the function returns a set of possibilities. In every step the non-deterministic Turing machine makes an additional decision which of these possibilities to use. If there exists a sequence of these decisions that gets the Turing machine into the state of q_{accept}, then the result of the computation is 1. If none of the decision sequences gets the machine into the state of q_{accept}, the result of the computation is 0.

2.5 Complexity Classes

Now, with the computational models defined, we can define the concept of *complexity class* and few particular complexity classes.

▶ **Definition 2.7** (Complexity class)**.** Complexity class *is a set of decision problems that can be computed with a given complexity resource.*

We are going to focus on complexity resource of *running time*. Let us firstly formalize and define this complexity resource.

▶ **Definition 2.8** (Running time)**.** *Let f be a decision problem, T be a function $\mathbb{N} \to \mathbb{N}$, and M be a Turing machine. We say that M computes f in $T(n)$ time, if for every string x it halts with $f(x)$ on output tape after at most $T(|x|)$ steps*[1]*.*

[1] Applications of transition function.

2.5.1 P Complexity Class

Let us start with the definition of the complexity classes of $DTIME$[2].

▶ **Definition 2.9** (DTIME). *For a function $f : \mathbb{N} \to \mathbb{N}$, we define DTIME(f(n)) as a set of all decision problems computable on a deterministic Turing machine in $c \cdot f(n)$ time, for some constant $c > 0$.*

Now, we can define the P[3] *complexity class.*

▶ **Definition 2.10** (P complexity class). $P = \bigcup_{c \geq 1} DTIME(n^c)$

P class consists of all decision problems computable in polynomial time on a deterministic Turing machine. This can be vaguely translated as *problems that are efficiently computable in our world*. We will often address problems contained in P as problems computable in polynomial time.

2.5.2 NP Complexity Class

Very similarly, we are going to define the NP[4] *complexity class.* We will start with the definition of $NTIME$[5], a non-deterministic variation of DTIME defined earlier.

▶ **Definition 2.11** (NTIME). *For a function $f : \mathbb{N} \to \mathbb{N}$, we define NTIME(f(n)) as a set of all decision problems computable on a non-deterministic Turing machine in $c \cdot f(n)$ time, for some constant $c > 0$.*

Now we can define the NP complexity class as follows:

▶ **Definition 2.12** (NP complexity class). $NP = \bigcup_{c \geq 1} NTIME(n^c)$

An equivalent definition of NP class is the following:

▶ **Definition 2.13** (NP complexity class). *A decision problem f is a member of NP class, if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a deterministic Turing machine TM, such that for every binary string x: $f(x) = 1 \iff \exists\, u \in \{0,1\}^{p(|x|)}$, such that $TM(x, u) = 1$, which can be computed in polynomial time.*

Proof of equivalency of these definitions can be seen in [9].
This alternative definition tells us that for every input string x exists a *certificate* u of polynomial length that *certifies* the fact that x is an element of the language.
To see an example of such certificate, let us take the decision problem of whether a given graph has a feedback vertex set (as defined in Subsubection 1.4.1.2) of size k. This problem is a member of NP class. The certificate in this case is a list of k vertices that form a feedback vertex set. Given such certificate, we could detect whether the graph contains any cycle after removing the vertices included in the list. Such problem is known to be solvable in polynomial time using depth-first search [2].
The NP class consists of decision problems that can be *verified* in polynomial time on a deterministic Turing machine. Given an instance of a problem and a solution (certificate), Turing machine can verify that the solution solves the problem.
The non-deterministic Turing machine can try all possible solutions in its computational branches and if one of them is a solution that certifies the presence of input in language, then the Turing machine will reach q_{accept} in that computational branch, yielding the output of 1.
If we let a non-deterministic Turing machine compute a problem from NP class, then the non-deterministic decisions used for transition into q_{accept}, made by such Turing machine, are also a valid certificate for the input.

[2] DTIME stands for deterministic time.
[3] P stands for polynomial
[4] NP stands for non-deterministic polynomial
[5] NTIME stands for non-deterministic time

2.5.3 NP-hardness and NP-completeness

In this subsection, we are going to define the concepts of *NP-hardness* and *NP-completeness*. We start by defining the *polynomial reducibility* among decision problems. [11, 12]

▶ **Definition 2.14** (Polynomial-time reudcibility)**.** *We say that a language $A \subseteq L$ is* polynomial-time reducible *to a language $B \subseteq L$, where L is set of all binary strings; if there exists a polynomial-time computable function $f : L \to L$, such that $\forall x \in L : x \in A \iff f(x) \in B$.*

We can see that if the problem A is polynomial reducible to B, and we would be able to compute B in polynomial time, then we would be able to compute A in polynomial time.
Now we can move onto definition of an *NP-hard problem*.

▶ **Definition 2.15** (NP-hardness)**.** *A decision problem K is said to be* NP-hard *when for every problem L in NP there exists a polynomial time reduction from L to K*

Subsequently, we define the *NP-complete* problems.

▶ **Definition 2.16** (NP-completeness)**.** *A decision problem is said to be* NP-complete*, if it is NP-hard and it is a member of NP class.*

2.5.4 Relation of P and NP Complexity Classes

Trivially, we can see that: P $\subset$ NP. However, probably the biggest undisclosed question of computer science is whether P *equals* NP. Finding an answer to this question might be one of the biggest advances in the modern science. Thanks to its importance, *P versus NP problem* has been placed on the list of Millenium Prize Problems [13].
If P would equal NP, then all NP problems would most probably be efficiently solvable in our world. This equality could be proved by finding a polynomial algorithm for any of the NP-hard problems. As it would mean that all NP problems can be converted in polynomial time to a problem, solvable in polynomial time. Therefore, solving NP-hard problems in polynomial time is a very ambitious task. In this thesis we rather used existing solvers, which are able to solve some NP-hard problems in bearable time for smaller inputs using heuristics or other techniques.
Solving NP-hard problems using heuristics is an interesting discipline of computer science. For example PACE[6] holds every year an annual competition, where a particular NP-hard problem is selected and the competitors are trying to design the fastest program for computation of the problem [14]. In this thesis, we are going to use for practical usage some of the algorithms submitted into this competition.

2.6 Notable NP-hard Problems

In this section we mention few NP-hard problems that we will discuss further in our work. Definitions of these problems and a proof of NP-hardness, can be seen in [2, 9, 11]. Following problems are included in the set of NP-hard problems:

- the problem of whether graph contains a feedback vertex set of size k,
- the problem of whether there exists a tree decomposition of the given graph of width k,
- the problem of whether the given graph can be covered using k vertices,
- the problem of finding the hitting set of size k,
- the problem of 0-1 integer programming.

[6] Parameterized Algorithms and Computational Experiments

The first three problems were introduced in the first chapter. Let us introduce the other two.

▶ **Definition 2.17** (The problem of finding the hitting set of size k)**.** *Given a family of sets $F = \{S_1, S_2, \ldots, S_n\}$, find a subset of size k of the universum $A \subseteq \bigcup_{S \in F} S$, such that $\forall S \in F : A \cap S \neq \emptyset$.*

▶ **Definition 2.18** (The problem of 0-1 integer programming)**.** *Given a matrix $A \in \mathbb{Z}^{m,n}$ and a vector $b \in \mathbb{Z}^m$, determine, whether there exists a vector $x \in \{0,1\}^n$, such that $Ax \leq b$[7].*

[7] Using elementwise comparison.

Chapter 3
Linear Programming

Linear programming revolves around solving optimization problems in a given mathematical model. Its origins date to the times of Fourier and it has been described long before invention of the modern computers; therefore it has been well studied by mathematicians, economists, and other scientists, and grew to giant dimensions. We will only go quickly through few topics related to our work. For a better overview of the whole topic, please refer to [15].

3.1 Linear Program

The *linear programming* is focused around optimizing a linear objective function while respecting given linear equalities or inequalities.
Linear program consists of

- variables, typically denoted $x_1, \ldots, x_n \in \mathbb{R}$;
- an objective function $c_1x_1 + \cdots + c_nx_n$, where $c_1, \ldots, c_n \in \mathbb{R}$, to be maximized or minimized;
- a set of constraints in form $a_1x_1 + \cdots + a_nx_n \,(\geq \mid = \mid \leq)\, b$.

A vector $x \in \mathbb{R}^n$ satisfying all constraints, is called a *feasible point* or a *feasible solution*. The set of all feasible points is called the *feasible region* of the model. If the model has an empty feasible region, then the model is called *infeasible*. An optimal solution is the element of the feasible region that has the maximal or minimal value of objective function out of all points in the feasible region.

3.2 Algorithms

Many algorithms for solving a general linear program have been introduced over the last decades. First described algorithm for solving linear problems was the *Simplex algorithm*, invented by George B. Dantzig in 1947 [16]. We are going to better introduce and demonstrate the run of Simplex algorithm on an example later in this chapter in Subsection 3.3.2.
In the late 1970's Leonid Khachiyan discovered a polynomial algorithm for solving linear programs. This algorithm is nowadays referred to as *ellipsoid method* [17].

3.3 Example

Let us consider a following linear program with two variables and six constraints:

$$\begin{gathered}\text{maximize } 3x_1 + 2x_2 \\ \text{such that:} \\ x_1 + x_2 \leq 9 \\ 3x_1 + x_2 \leq 18 \\ x_1 \leq 7 \\ x_2 \leq 6 \\ x_1, x_2 \geq 0\end{gathered}$$

3.3.1 Graphical Representation

We have chosen only two variables to simplify the visualization of the problem. With two variables we can visualize the value of variables using Cartesian coordinate system with x_1 using the x axis and x_2 using the y axis.
We can easily see that a single inequality represents a line that splits the plane to two sub-planes. One of these sub-planes contains points that satisfy the inequality. Therefore, if we take all lines and mark the intersection of all sub-planes satisfying the constraints, we will get the feasible region.
In case of our example, we get the feasible region as in Figure 3.1:

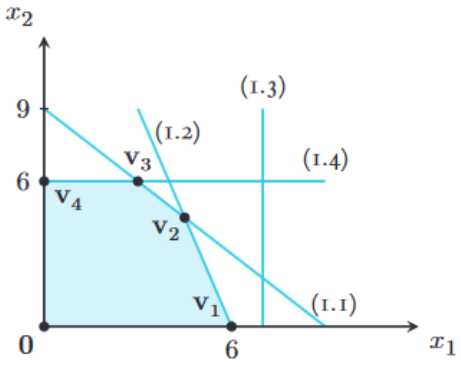

Figure 3.1 A graphical representation of a linear program [15].

Although for more than two variables it would get harder to visualize. The property of any constraint splitting a general hyperspace to two sub-hyperspaces still holds. Therefore, a feasible region, if it exists, always has a shape of convex polytope.
To visualize the value of objective function over the feasible region, we would need a third dimension. We can either project three dimensional graph onto two dimensions, or we can draw level lines over the two dimensional graph as contours, connecting points in which the objective function has identical value. Both of these approaches can be seen in Figure 3.2
As we can see from these visualizations, the maximal value of the objective function is achieved in point marked in the previous figures as v_2, which is the intersection of lines
$3x_1 + x_2 = 18$ and $x_1 + x_2 = 9$. This intersection can be evaluated as $[4.5, 4.5]$, meaning that the optimal solution of this linear problem is $x_1 = 4.5$, $x_2 = 4.5$, with value of 22.5.

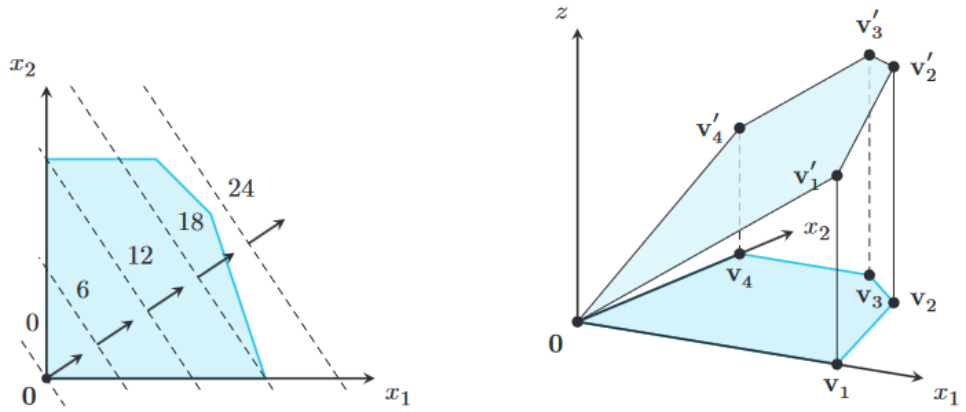

Figure 3.2 A visualization of the objective function [15].

3.3.2 Simplex Algorithm

Description of the simplex algorithm can be simplified as follows: Start in one of the vertices (points, in which the constraints intersect, marked as v_i in previous figures) and then travel by edges to a neighbouring vertex, in which the value of objective function increases (or decreases). More detailed description is available in the cited monography [15]. It is good to note that over the years the algorithm itself had many modifications, with this being the general idea that stood the same.

In this example, we might for example start in vertex marked as v_1, which has a neighbouring vertex v_2, in which the value of objective function is higher. Therefore we travel to vertex v_2. With all neighbours of v_2 having a lower value, we finish the run of algorithm. Since the object is convex, we can safely say that we have found the global maximum.

3.4 Integer Linear Programming

A special case of linear programming is the *integer linear programming*. In this special case, we require some (or all) of the decision variables to be integers, instead of real numbers. As we have discussed, the linear programming is solvable in polynomial time. On the other hand, the integer linear programming is known to be NP-hard already for instances where the domain is $\{0, 1\}$ as shown in [11]. On a positive note, integer linear programming is one of the most efficiently solvable NP-hard problems with existing software. In our work, we are going to reduce many problems onto problem of integer linear programming and use the help of existing solver introduced in the next section.

3.5 Gurobi Optimization

Gurobi optimization engine [18] is one of the best available solvers for linear programming, integer linear programming and other mathematical optimization problems. Gurobi is internally switching used algorithm, based on the input, and it is also able to run the computation in parallel on multiple threads.

It offers an interface for many programming languages, including Python, C++, and many more. We have used this solver for solving the problem of integer linear programming, as the solver is greatly optimized and provides a good way of solving NP-hard problems.

Chapter 4

Previous Research of Sidewalk and Pedestrian Networks

The sidewalk networks have not been studied as exhaustively as the other types of real-world networks (like road networks or infrastructural networks). Mainly since they do not suffer from congestion and overcrowding so much, compared to the other types of networks. However, in the last years, the attention given to sidewalks and generally pedestrians has risen. This is a reflection of promoting walking as a mean of transport. With the growing attention to sustainability and protection of the planet Earth, walking is (and the trend is most likely going to continue) becoming a promoted mean of transport worldwide. Therefore, modern science pays closer attention, whether cities are designed so that walking is a sufficient alternative for other, more pollution-heavy, means of transport, especially personal cars.

However, typical main article of research, of most works concerned by this topic, are not sidewalks, but rather generally *pedestrian networks*. To simplify the difference between these two terms, pedestrian networks include sidewalks, but apart from that, they also include pedestrian crossings, residential roads or patches of green, which pedestrians typically use for walking. Additionally, some sidewalks might also not be a suitable part of pedestrian network, if the sidewalk is damaged or inaccessible.

The main research topics of sidewalk networks or pedestrian networks vary, however there has been no such work, measuring graph properties of sidewalk networks purely for theoretical research. On the other hand, there have been few studies concerned with the similar topic, we will mention a few of them in this chapter.

In [19], Rhodas et al. are measuring betweenness centrality and general efficiency of pedestrian networks. These parameters are typically measured for other types of networks (as discussed earlier, road networks or infrastructure networks), as it is an important parameter for identifying bottlenecks in networks, preventing congestion. Apart from that, the authors are giving an attention to sidewalk coverage and availability. Studying, whether a pedestrian can use walking as a mean of transport, to fulfill their needs, with a special attention to the length of tour they have to take to do so. As it is mentioned in this work, a pedestrian walking on a sidewalk is in constant danger of getting involved in an accident. The danger walking brings is another topic of this work, with a final takeaway that future cities should minimize the distance a pedestrian needs to travel.

The dangers that pedestrians face while walking are also analysed in [20]. In this work, Osama and Sayed are evaluating the impacts of pedestrian network structure on safety of its users. Authors are studying the probability of a pedestrian being involved in a crash, subject to factors, such as continuity, linearity, coverage and slope of the pedestrian network.

Our work varies in two main points. First of all, we are interested solely in the sidewalk networks, rather than the pedestrian networks, as our motivation is to create a realistic sidewalk network generator for 3D model generation[1]. Secondly, we are studying the sidewealk networks in a much more theoretical way, as we want to give an insight to this topic purely from the perspective of discrete mathematics and graph theory.
Many works studying sidewalk or pedestrian networks remark that obtaining datasets for studying these networks is a considerably harder task, compared to obtaining similar data for other types of real-world networks. Stating that this is once again due to the fact that sidewalk network analysis has not been deemed as important as analysis of the other types of networks. We certainly can concur with this notion. In fact, availability of road networks being significantly higher than the availability of sidewalk networks, is the main motivation for designing an algorithm that could generate a realistic sidewalk network from road network and building outlines.

[1]Therefore, for example pedestrian crossings can not be generated together with the sidewalks, as they are visually completely different.

Chapter 5
Obtaining the Data

In this chapter, we discuss the process of obtaining the data of sidewalk networks. We start by describing data formats. Then we introduce the source of our data. And we finish this chapter with the description of the process of collecting and serializing the data.

5.1 Data Formats

With growing number of applications operating with data, the need for unification of *data formats* used has risen. In this section we are going to introduce few modern data formats used by computational programs, web services, desktop applications and other software.

5.1.1 XML

XML[1] is a markup language defining a structure of documents for storing generally any type of data.
XML document consists of elements. Start of an element is marked with the *opening tag* and the end is marked with the *closing tag*. The opening tag consists of the opening angular bracket, name of the element, additional non-mandatory attributes and the closing angular bracket. The closing tag contains the opening angular bracket, backslash, the name of the element and the closing angular bracket. The names in the opening and closing tags must match. An element can contain other elements in its body (space between the opening and closing tags), thus creating the structure of the document. An XML document starts with the version and encoding specification, followed by root element, which contains all other elements in the document. An example of XML document can be seen in Code listing 5.1. XML aims to be a human-readable, self-descriptive format to be used for data exchange, configuration files and other use cases. [21]

5.1.2 JSON

JSON[2] is another data format for representing data of any kind. It was designed to be more light-weight, more human-readable and easier for parsing and generating by computers than other data formats.
The JSON format consists of objects. An object is enclosed by curly brackets. The object

[1] eXtensible Markup Language
[2] JavaScript Object Notation

```xml
<?xml version="1.0" encoding ="UTF-8" ?>
<bookstore>
    <book category="fiction">
        <title>Harry Potter and the Sorcerer's Stone</title>
        <author>J.K. Rowling</author>
        <year>1997</year>
        <price>20.00</price>
    </book>
    <book category="non-fiction">
        <title>On the Origin of Species</title>
        <author>Charles Darwin</author>
        <year>1859</year>
        <price>15.00</price>
    </book>
</bookstore>
```

Listing 5.1 An example of XML document.

contains key-value pairs, with key being the name of an attribute, enclosed by quotation marks. The value may be in a string format (enclosed by quotation marks), array format (enclosed by sharp brackets, containing any number of objects or primitive values), number format or boolean format. The individual key-value pairs are separated by a comma and between the key and the value in a pair, there is a colon. An example of JSON representing the same entities as the XML document from the previous section can be seen in Code listing 5.2 [22]

```json
{
  "bookstore": {
    "books": [
      {
        "category": "fiction",
        "title": "Harry Potter and the Sorcerer's Stone",
        "author": "J.K. Rowling",
        "year": 1997,
        "price": 20.00
      },
      {
        "category": "non-fiction",
        "title": "On the Origin of Species",
        "author": "Charles Darwin",
        "year": 1859,
        "price": 15.00
      }
    ]
  }
}
```

Listing 5.2 An example of JSON representation.

```
p tw 10 15
1 2
1 5
1 6
2 3
2 7
3 4
3 8
4 5
4 9
5 10
6 8
6 9
7 9
7 10
8 10
```

Listing 5.3 GR representation of Petersen graph

5.1.3 GR

GR[3] is a format used by PACE [14] for encoding of undirected graphs, as defined in Section 1.1. This format is used for annual competitions focused on implementing the most efficient algorithms for computationally complex problems. This format is designed mainly with efficiency in mind and it is supposed to be very quickly and easily readable by a computer.
The description of the graph starts with a *p-line*. This line starts with a letter *p* followed by an abbreviation of the problem, for which the dataset was defined[4]. Additionally p-line includes two numbers. The first number represents the number of vertices, and the second one represents the number of edges. p-line is followed by a list of edges, each line represents one edge given by a pair of vertices split by a single space. The vertices are being implicitly represented as natural numbers from 1 to number of vertices. An example of this graph encoding can be seen in Code listing 5.3, where we can see Petersen graph (as defined in Subsection 1.3.5) encoded into GR.

5.2 OpenStreetMap

OpenStreetMap[5] is a free, collaborative and open map of planet Earth built by volunteers. Apart from the map, OSM makes publicly available data it uses for creation of the map. An example of the map can be seen in Figure 5.1.
The publicly available data from OpenStreetMap are widely used for building other projects[6], since the OSM provides a rich API[7] for retrieving of this data. [23, 24]

[3] Graph format
[4] For example *tw* as treewidth.
[5] Abbreviated as OSM, also incorrectly called Open Street Map or Open Street Maps.
[6] One of them being the VBS Blue.
[7] Application Programming Interface

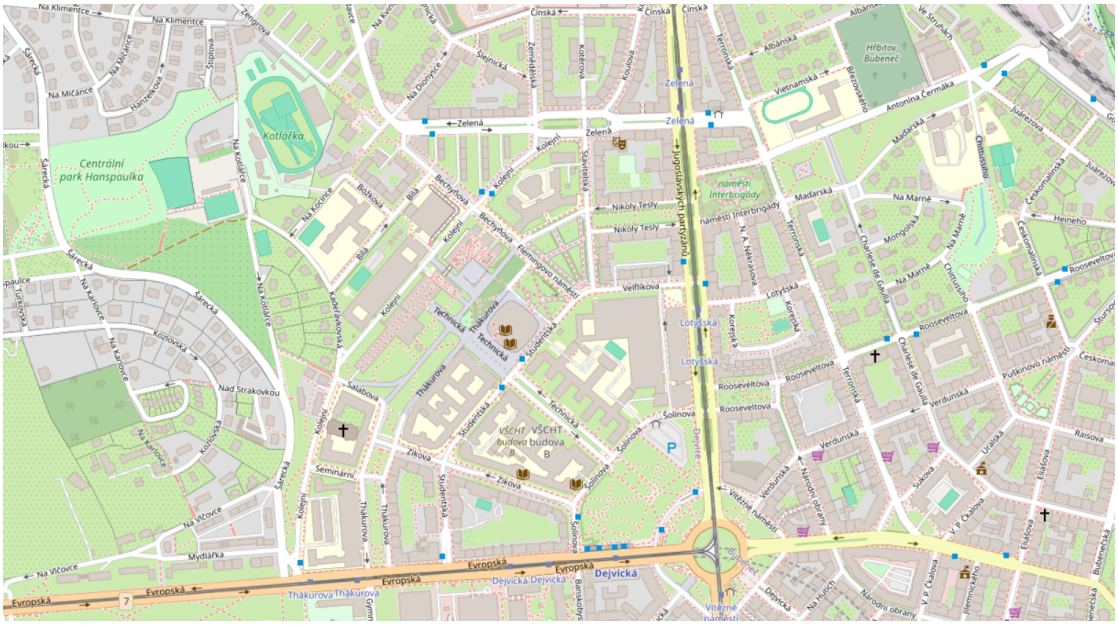

Figure 5.1 A map from OpenStreetMap project [25].

5.2.1 History

The OpenStreetMap project was founded in 2004 by Steve Coast, initially focusing on mapping of the United Kingdom. In 2006, *OpenStreetMap Foundation* was established to promote, support and protect the project. However OpenStreetMap Foundation does not own the data, as nobody owning the data is the main idea behind this project. In the same year, Yahoo let its aerial photography to be a base for OpenStreetMap, which enabled more contributors to get involved, benefiting the project massively.
The ways for importing and exporting the data only continued to grow. For example, in year 2008, the data became exportable to portable GPS[8] devices. Nowadays, OSM is a very unique project, which distributes quality geographical data for free and various usage. However, it is good to note that the amount of detail varies significantly depending on the region.

5.2.2 Format

OpenStreetMap defines its own data format for data available via its API. This format is called *OSM XML* and it is based on XML.
An OSM XML file begins with the version of OSM API, followed by bounding box (described in the geographic coordinate system). The main part of the document consists of list of *nodes*, list of *ways* and list of *relations*. These terms are defined in following subsections.

5.2.2.1 Tag

A *tag* is a singular property of a real-world object represented by a key word pair. An example of a tag can be seen in Code listing 5.4.

[8] Global Positioning System

```
<tag k="colour" v="brown"/>
```

Listing 5.4 An example of a single tag from OSM, response from OSM API to GET /api/0.6/node/2905214181, modified.

5.2.2.2 Node

A *node* is a representation of a single point feature, or more commonly it is a part of a *way*, another OSM entity, defined in the next section.
Every node contains its ID[9], latitude and longitude. Additionally, a node can include any number of tags (with no tags also being a possibility), describing better what kind of a real-world feature the node represents.
An example of a node from OSM representing a bench can be seen in Code listing 5.5.

```
<osm
    version="0.6"
    generator="CGImap 0.9.2 (1123753 spike-07.openstreetmap.org)"
    copyright="OpenStreetMap and contributors"
    attribution="http://www.openstreetmap.org/copyright"
    license="http://opendatacommons.org/licenses/odbl/1-0/">
<node
    id="2905214181"
    visible="true"
    version="3"
    changeset="95334039"
    timestamp="2020-12-05T13:28:55Z"
    user="koldas"
    uid="6383771"
    lat="50.1034230"
    lon="14.3903959">
<tag k="amenity" v="bench"/>
<tag k="backrest" v="no"/>
<tag k="colour" v="brown"/>
<tag k="material" v="wood"/>
</node>
</osm>
```

Listing 5.5 An example of a single node feature from OpenStreetMap, a response from OSM API to GET /api/0.6/node/2905214181, reformatted.

5.2.2.3 Way

A *way* is an ordered sequence of nodes. A way is used for representation of linear features (like roads or sidewalks) or outlines of areal features (like building outlines or state borders). Apart from nodes, it can contain any number of tags better describing the real-world feature. An example of a way from OSM representing a residential road can be seen in Code listing 5.6

[9] ID stands for identificator

```
<osm
    version="0.6"
    generator="CGImap 0.9.2 (1214374 spike-07.openstreetmap.org)"
    copyright="OpenStreetMap and contributors"
    attribution="http://www.openstreetmap.org/copyright"
    license="http://opendatacommons.org/licenses/odbl/1-0/">
<way
    id="8588965"
    visible="true"
    version="21"
    changeset="143633267"
    timestamp="2023-11-04T23:21:02Z"
    user="Martin2035"
    uid="6588887">
    <nd ref="683826"/>
    <nd ref="1244162315"/>
    <nd ref="60953383"/>
    <nd ref="60953384"/>
    <nd ref="331441551"/>
    <nd ref="683828"/>
    <tag k="bicycle" v="yes"/>
    <tag k="covered" v="no"/>
    <tag k="highway" v="residential"/>
    <tag k="lanes" v="1"/>
    <tag k="lit" v="yes"/>
</way>
</osm>
```

Listing 5.6 An example of a way feature from OpenStreetMap, a response from OSM API to GET /api/0.6/way/8588965, reformatted, modified.

5.2.2.4 Relation

A *relation* is an ordered sequence of nodes and ways which are connected together, typically in a more abstract way. Relations are typically used for modeling of cities, states and other regions. Relations can optionally include tags, just like the other entities. An example of a relation from OSM can be seen in Code listing 5.7

5.2.3 APIs

OpenStreetMap specifies two different APIs, both with different use-cases. The first one is the classical modern API enabling both reading and writing of raw OSM data. Apart from that, OSM also provides a read-only API called *Overpass API*. Overpass API does not enable modification of data, its purpose is only to fetch the data. Overpass API is ready for handling big chunks of data and defines its own querying language. This language more resembles a scripting language, rather than a classical query language. It provides concepts better known from programming and scripting languages, like cycles or conditional jumps. The main purpose of this language is to give user a simple way for obtaining and filtering the data, as the amount of OSM data in certain parts of the world is humongous. Therefore, if user is interested in specific data for their application, they can easily filter unwanted data on OSM server side.

```
<osm
    version="0.6"
    generator="CGImap 0.9.2 (3191307 spike-07.openstreetmap.org)"
    copyright="OpenStreetMap and contributors"
    attribution="http://www.openstreetmap.org/copyright"
    license="http://opendatacommons.org/licenses/odbl/1-0/">
<relation
    id="428868"
    visible="true"
    version="16"
    changeset="143023364"
    timestamp="2023-10-23T15:16:14Z"
    user="StenSoft" uid="255936">
<member type="node" ref="297896636" role="admin_centre"/>
<member type="way" ref="181639516" role="outer"/>
<member type="way" ref="181639512" role="outer"/>
<member type="way" ref="181639509" role="outer"/>
<member type="way" ref="181639506" role="outer"/>
<member type="way" ref="512311736" role="outer"/>
<member type="way" ref="181639502" role="outer"/>
<member type="way" ref="181639496" role="outer"/>
<member type="way" ref="180740868" role="outer"/>
<member type="way" ref="577535890" role="outer"/>
<tag k="admin_level" v="10"/>
<tag k="boundary" v="administrative"/>
<tag k="name" v="Dejvice"/>
</relation>
</osm>
```

Listing 5.7 An example of a relation from OpenStreetMap, response from OSM API to GET /api/0.6/relation/428868, reformatted,

5.2.4 OSMnx

Most modern programming languages have multiple dedicated libraries or packages for communication with OSM Overpass API. One of them being the *OSMnx* [26], which is a Python package specialized for obtaining network data from OSM. OSMnx uses data representations compatible with another Python package *NetworkX* [27] (Python package for studying of networks, mostly from the perspective of graph theory).

5.3 Collecting and Serializing the Data

As a source of the data we have used the OpenStreetMap. Choosing OpenStreetMap was one of the most straight-forward decisions we have made, as the whole existing project of VBS Blue uses OSM as its source data for various features (including road networks or building outlines, which would be the input data of the future potential algorithm).
It should be noted that there are not many better alternatives. As we have discussed in previous chapter, obtaining sidewalk data is a fairly more challenging task than obtaining, for example, road network data.

5.3.1 Analysed Locations of the World

As we have mentioned in the previous section covering the OpenStreetMap project, the level of detail in OpenStreetMap highly depends on the location we choose to study. Non-surprisingly, the level of detail is at its highest in the most developed or the most culturally significant parts of the world.
We have used the assistance of a community managed website [28], which summarizes the most well-mapped placed in the OSM project. Although, we had to ensure ourselves that the destinations include the sidewalk data, as they are often not present, even in places, marked by this site as the ”Best of OSM”.
We have tried to make the datasets more diverse, as the city design naturally varies over different nations and cultures. Unfortunately, there is only a very few data available outside of Europe and the United States. On the other hand, the city design in Europe is very different compared to the American one.
In the end, we have settled for the following destinations of the world:

- Černý Most, Prague, Czech Republic;
- Cēsis, Latvia;
- College Park, Maryland, United States of America;
- Dejvice, Prague, Czech Republic;
- Grenoble, France;
- Helsinki, Finland;
- Raleigh, North Carolina, United States of America;
- Donostia-San Sebastián, Spain[10];
- Santa Cruz, California, United States of America.

[10] Official name of Donostia-San Sebastián is a bilingual combination of Basque and Spanish name of the city (both meaning Saint Sebastian), in this work we will refer to this city as San Sebastián, as it is a name more commonly used outside of the Basque area.

5.3.2 Usage of OSMnx Package

When we selected the locations, obtaining their sidewalk data from OSM was not a difficult task. We have taken all features having a value of tag `highway` set to `footway`, `path` or `steps`; or a value of tag `foot` set to `designated` or `yes`.
With the help of Python package OSMnx [26], we have been able to acquire the data using just a few lines of code, which can be seen listed in Code listing 5.8.
This code snippet generates an outcome as in Figure 5.2. It can be seen,that the data obtained from OSM include more information than we need, such as the location of the vertices or shapes of the edges. Since we were interested solely in vertices and edges connecting them, we have simplified the data before serializing, forgetting the unnecessary information.
We have serialized the obtained data into JSON and GR formats.
The JSON format is a good general representation of any data. Therefore exporting the data to a JSON was the first thing we have done. We have decided to use the following JSON representation. The object of graph includes two lists, called ”vertices” and ”edges”. The list of vertices is an unordered list of unique numbers. These numbers are OSM IDs of OSM nodes represented by these vertices. The list of edges is a list of lists of length two (for simplicity, we will call these lists *pairs*). Each pair represents an edge in the graph. Naturally, such pair represents an edge between the two vertices it contains.
For exporting data into JSON, Python is already providing a library without any additional package, therefore the export was a simple task.
Additionally, we have decided to export the data into the GR format, as many solvers used for identifying the structural properties of the networks accept input in this format. We have done the conversion to GR format ourselves. The main difference between the chosen JSON representation and the GR format is that GR format implicitly represents the vertices as natural numbers between 1 and the number of vertices, whereas our JSON format uses the OSM IDs for vertex representation.

```python
#!/bin/env python3

from osmnx import graph_from_place, plot_graph as ox_plot_graph, project_graph
import networkx as nx
import matplotlib.pyplot as plt

def get_osm_graph(place_name : str) -> nx.MultiDiGraph:
    """
        Fetches data of sidewalks from place given by place_name
    """
    custom_filters = ['["highway"~"footway|path|steps"]','["foot"~"yes|designated"]']
    graph_parts = [graph_from_place(place_name, network_type='all', retain_all=True, simplify=Tru
                   for custom_filter in custom_filters]

    return nx.compose_all(graph_parts)

def visualize_osm_data(graph : nx.MultiDiGraph):
    """
        Visualization of OSM data
    """
    ox_plot_graph(project_graph(graph))
    plt.show()

def main():
    place_name = "Dejvice, Prague, Czechia"
    osm_graph : nx.MultiDiGraph = get_osm_graph(place_name)
    visualize_osm_data(osm_graph)

if __name__ == "__main__":
    main()
```

Listing 5.8 Python script exporting and visualizing the sidewalk data from OSM.

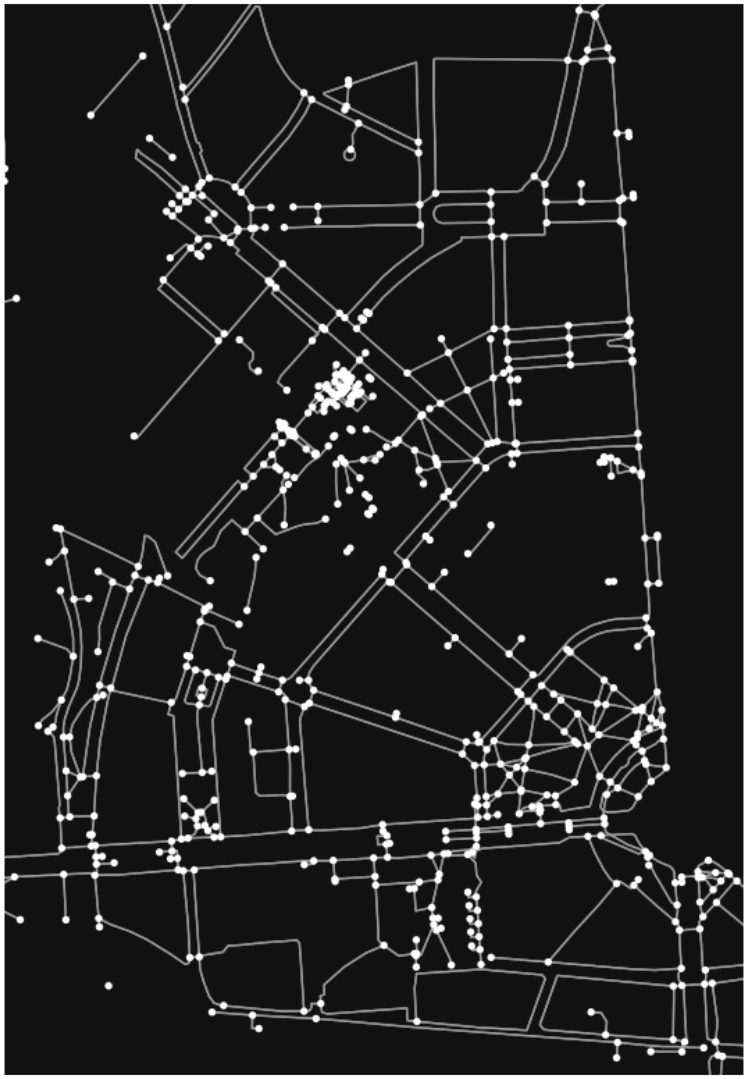

Figure 5.2 A visualization of the data obtained from OSM for Dejvice, Prague, Czech Republic.

Chapter 6

Graph Properties Measurement and Evaluation

The main part of our research was the analysis of graph properties of the real-world sidewalk networks. In this chapter, we will go through the process of measuring properties, describing challenges it brought, and how we coped with them. We will also describe the reason, why we selected these particular properties for measuring, and discuss our results, stating the main takeaways for the development of the sidewalk network generating algorithm.

6.1 Feedback Edge Set Number

The first property we measured was the *feedback edge set number* of the graphs, as defined in Subsubection 1.4.1.1.

6.1.1 Motivation

The feedback edge set number is a property that could help us massively in creation of the algorithm for sidewalk generation. Our first idea for designing such algorithm is to identify vertices (how should the vertices be identified is yet undisclosed) and continue by finding a minimal spanning tree[1]. At that point we would start adding edges to make the network look more realistic. How many edges we should add can be answered by this parameter.

6.1.2 Measurement

Measuring value of this property was a surmountable task. Finding feedback edge set number is a problem known to be solvable in polynomial time. The idea of the polynomial algorithm is fairly simple. Spanning trees (or forests) are maximal acyclic subgraphs. Therefore to find the feedback edge set number of a given graph, we take the difference between the number of its edges and the number of edges in any of its spanning trees (or forests). [4]
For any connected graph $G = (V, E)$ and its spanning tree $G_T = (V, E_T)$, the feedback edge set number is equal to $|E| - |E_T|$. The problem can be simplified further, when we realize that the number of edges in a spanning tree, can be identified just from the number of vertices of the graph. Using one of the alternative definitions of a tree: Graph $G = (V, E)$ is a tree $\iff$ it is connected and $|E| = |V| - 1$, we can conclude that we can replace $|E_T|$ in the expression and get a simple

[1] Minimal in its length.

way to calculate feedback edge set number for a connected graph as $|E| - |V| + 1$.
Furthermore, we can handle disconnected graphs in a similar way, using this expression for all of its connected components. For any graph G and its maximal connected components: $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, ..., $G_k = (V_k, E_k)$ such $\bigcup_{i=1}^{k} G_i = G$, we can calculate the feedback edge set number as $\sum_{i=1}^{k} |E_i| - |V_i| + 1$. We can split this sum into separate sums as follows $\sum_{i=1}^{k} |E_i| - \sum_{i=1}^{k} |V_i| + \sum_{i=1}^{k} 1$. And since all connected components are disjoint and add up to the whole graph G, we can rewrite this as follows: $|E| - |V| + k$, where k is the number of connected components of G. Since a connected graph will have k equal to 1, this expression can be actually applied on all graphs, regardless of whether they are connected or not. This is the expression we used to identify the feedback edge set number of the sidewalk networks.

6.1.3 Results

First, we show our results for all datasets in Table 6.1. In this table, we abbreviate feedback edge set number as FESN and feedback edge set as FES.

Table 6.1 Measured values of feedback edge set number.

Dataset	‖V‖	‖E‖	FESN	Relative size of FES
Černý Most, PRG, CZE	1 553	1 893	377	24.87 %
Cēsis, LAT	1 122	1 244	245	24.52 %
College Park, MD, USA	5 110	6 392	1 534	31.58 %
Dejvice, PRG, CZE	1 675	1 907	372	24.23 %
Grenoble, FRA	9 647	11 688	2 711	30.20 %
Helsinki, FIN	115 318	123 291	18 483	17.64 %
Karlín, PRG, CZE	1 070	1 364	345	33.86 %
Raleigh, NC, USA	36 385	39 930	5 667	16.54 %
San Sebastián, ESP	4 782	4 961	813	19.60 %
Santa Cruz, CA, USA	13 060	14 494	2 710	23.00 %

We can see that the feedback edge set number differs across all datasets considerably. It came as no surprise; rather, it aligned with our anticipatory projections, that with the increasing number of edges, the size of minimal feedback edge set will also grow.
Therefore, apart from feedback edge set number, we have also measured the relative size of the minimal feedback edge set. The relative size of feedback edge set expresses, how big is the feedback edge set compared to the rest of edges. To be more specific, the exact formula used for calculation is $\frac{f}{\|E\| - f}$, where f is the value of the feedback edge set number. We can see that this number does not differ too dramatically.
We are interested in this ratio, as it gives us a relative number of edges we should add, after we have constructed the spanning tree for vertices, in the generation algorithm. We can see that this number typically stays around 20 % and for bigger datasets (which are generally more trustworthy, as they do not contain that many false endpoints, where the network is ended by the end of the dataset, rather than the actual end of the sidewalk), it gets even lower.

6.2 Feedback Vertex Set Number

After we have successfully measured the feedback edge set number, we continued with the *feedback vertex set number*, a property, which we have defined in Subsubection 1.4.1.2.

6.2.1 Motivation

We were interested in the feedback vertex set number, as it is a parameter that could tell us whether the selection of vertices in the algorithm is structurally correct. Additionally, since deciding the feedback vertex set number is definitely a non-trivial task, with the problem being NP-hard, our results can provide a rough estimation for any sidewalk network, without the need of the computationally intense calculation of the parameter.

6.2.2 Measurment

As we have mentioned, deciding the value of the feedback vertex set number is a problem belonging to the NP-hard class. Therefore, for calculation of this property, we have used existing solvers. Despite that, for the biggest datasets, we can still only provide boundaries of the value, rather than the exact value, which proved to be too difficult to measure in some cases.

6.2.2.1 Exact Value and Upper Bound

For calculation of this property, we have used a solver developed by Iwata and Imanishi [29, 30], as a part of PACE challenge 2016.
This solver gives an upper bound and incrementally improves it, until it reaches an optimum, at which point it stops.
We were able to obtain the exact value of the feedback vertex set number for few datasets. We were also able to get an upper bound for almost all the datasets, apart from the biggest ones.
The specification of the challenge mentions that the solvers were tested against graphs with the number of vertices ranging from 30 to 20 000, therefore it does not come as a surprise that the dataset of Helsinki (with more than 100 000 vertices), does not get any upper bound calculated.
What was more surprising, was definitely the solver achieving the best performance on the dataset representing the city of San Sebastián. This dataset belongs among our datasets to the ones with the medium size[2], so it was rather an unexpected outcome. At this point, we were not able to tell the reason, why the solver computed the feedback vertex set number of this dataset so swiftly. An answer for this question we got, after we have measured the treewidth of the datasets. We will discuss this topic in-detail in the following section.

6.2.2.2 Lower Bound

We have also measured the lower bound, although it does not carry as much value as the upper bound. On the other hand, we were able to measure the lower bound for all of the datasets, including the biggest ones.
For the calculation of the lower bound, we have used the integer linear programming. First, we used the NetworkX library [27] to identify all cycles in a graph, and then we used the Gurobi solver [18] to find a minimal hitting set of the cycles. Formal description of the mathematical model used for the calculation of this property is:
Let F be a set of cycles contained in $G = (V, E)$, let there be a variable x_v for every vertex $v \in V$, set to 1 if the vertex v is present in the feedback vertex set; else set to 0. Minimize $\sum_{v \in V} x_v$ subject to $\forall\, C \in F : \sum_{v \in C} x_v \geq 1$.
This description corresponds to the second definition given in Subsubection 1.4.1.2.
However, finding all cycles in a graph, is also a computationally very complex task (with all cycles found we would be easily able to decide, whether one of them is a *Hamiltonian cycle*[3], this decision problem is known to be NP-hard [11]). However we would only need that for the exact

[2] In terms of number of vertices and edges.
[3] Hamiltonian cycle is a cycle containing all vertices of a graph.

computation. Since we wanted to provide a lower bound, we have used a length boundary for cycles (namely 9), giving us an incomplete set of cycles in acceptable time. We have then found a minimal subset of vertices that covers this subset of cycles. Such subset of vertices will surely be smaller than the minimal feedback vertex set, therefore we got a lower bound for the feedback vertex set number.

6.2.3 Results

Let us start once again with the outcome of our measurements, which can be seen in Table 6.2. In this table, we abbreviate feedback vertex set number as FVSN and feedback vertex set as FVS.

Table 6.2 Measured values of feedback vertex set number.

Dataset	‖V‖	‖E‖	FVSN	Relative size of FVS
Černý Most, PRG, CZE	1 553	1 893	146–167	10.38–12.05 %
Cēsis, LAT	1 122	1 244	110–113	10.87–11.20 %
College Park, MD, USA	5 110	6 392	571–595	12.58–13.18 %
Dejvice, PRG, CZE	1 675	1 907	173	11.52 %
Grenoble, FRA	9 647	11 688	1 017–	11.78– %
Helsinki, FIN	115 318	123 291	7 842–	7.30– %
Karlín, PRG, CZE	1 070	1 364	138–153	14.81–16.68 %
Raleigh, NC, USA	36 385	39 930	2 472–	7.29– %
San Sebastián, ESP	4 782	4 961	407	9.30 %
Santa Cruz, CA, USA	13 060	14 494	1 168–	9.82– %

Analogously to the feedback edge set number, the feedback vertex set number also grows with the size of the graph. Therefore we have once again introduced a relative parameter expressing the ratio of the feedback vertex set size and the number of vertices.
This calculation uses a slightly different formula as opposed to the feedback edge set. We have used the formula: $\frac{f}{\|V\|}$, where f represent the feedback vertex set number. We have switched to this formula, since at feedback edge set number, we were interested in how many edges we should add to a constructed spanning tree (in the potential algorithm for a sidewalk network generation). This time, however, we are rather trying to give out a way to estimate feedback vertex set number for any sidewalk network, from the number of its vertices.
Judging from the exact measurements and boundaries, we can estimate this ratio to be somewhere between 8 and 15 percent.

6.3 Treewidth

Our third measured parameter was the parameter of *treewidth* as defined in Subsubection 1.4.1.3.

6.3.1 Motivation

The treewidth is a very interesting graph property. Many graph problems are simpler for computation, when the input graph has a small treewidth [2]. However, deciding the treewidth itself is an NP-hard problem. Our measurements can provide an interesting estimation of the treewidth of sidewalk networks.
Subsequently, we can use this property for analysis of the outcomes of the future potential algorithm, to see whether yielded outputs are correct.

6.3.2 Measurment

Measuring the treewidth of a graph is a very extensively studied topic, since it is a parameter of a serious value. Therefore we have used for measurement an existing solver developed by Tamaki [31, 32, 33], as a part of PACE challenge 2017.
This solver gives a lower bound and an upper bound, incrementally improving both, until they meet, at which point it stops. For the smaller datasets, we were able to get the exact value of the treewidth. For the more sizable data sets, we were able to obtain a lower bound and an upper bound. However, for the biggest datasets, we were able to measure neither of the boundaries.

6.3.3 Results

The measured values can be seen in Table 6.3. The table does not include the datasets of Helsinki and Raleigh, as we have failed to measure any values for these datasets.

Table 6.3 Measured values of treewidth.

Dataset	$\|\mathbf{V}\|$	$\|\mathbf{E}\|$	**Treewidth**	$\frac{\|E\|}{\|V\|}$
Černý Most, PRG, CZE	1 553	1 893	10	1.22
Cēsis, LAT	1 122	1 244	10	1.11
College Park, MD, USA	5 110	6 392	15–22	1.25
Dejvice, PRG, CZE	1 675	1 907	9	1.14
Grenoble, FRA	9 647	11 688	14–25	1.21
Karlín, PRG, CZE	1 070	1 364	8	1.27
San Sebastián, ESP	4 782	4 961	7	1.04
Santa Cruz, CA, USA	13 060	14 494	13–16	1.11

We can see that for the smaller datasets the value of the treewidth stays around 10. For the larger datasets, representing more populous cities, the value rises to 20.
Very interesting is the case of San Sebastián. This dataset has considerably lower treewidth, than we would expect from its size. To explain this occurrence, we can notice that this graph is the most *sparse* among all the datasets. To enumerate the "sparsity" of the datasets, we have calculated a simple parameter of *ratio of edges and vertices*, expressed in the additional column. Seemingly, apart from the size of the graph, the treewidth is considerably influenced by this ratio.
The low treewidth of San Sebastián sidewalk network had a serious impact on its measurements. Both feedback vertex set number computation and treewidth computation were significantly faster for this dataset, compared to the other datasets.

6.4 Vertex Cover Number

Another parameter we have measured was the *vertex cover number* as defined in Subsubection 1.4.2.1.

6.4.1 Motivation

The vertex cover number is another non-trivial graph parameter we have measured, as it could be used for confirmation of the results yielded by the algorithm. We believed that we would be able to measure this property exactly for all datasets, since it is known to be easily represented via the integer linear programming.

6.4.2 Measurment

We have used the integer linear programming solver Gurobi [18], as the vertex cover number is known to be solvable with the integer linear programming as described in [34].
Let there be a binary variable x_v for each vertex $v \in V$, set to 1, if the vertex v is present in the vertex cover; else set to 0. Minimize $\sum_{v \in V} x_v$ subject to: $\forall \{u, v\} \in E : x_u + x_v \geq 1$.

6.4.3 Results

The results of the measurements can be seen in Table 6.4. In this table, we abbreviate vertex cover number as VCN and the vertex cover as VC. We have been able to exactly measure the value of the vertex cover number for all datasets, as we expected and hoped.

Table 6.4 Measured values of vertex cover number.

Dataset	‖V‖	‖E‖	VCN	Relative size of VC
Černý Most, PRG, CZE	1 553	1 893	764	49.20 %
Cēsis, LAT	1 122	1 244	521	46.43 %
College Park, MD, USA	5 110	6 392	2 440	47.75 %
Dejvice, PRG, CZE	1 675	1 907	790	47.16 %
Grenoble, FRA	9 647	11 688	4 672	48.43 %
Helsinki, FIN	115 318	123 291	52 824	45.81 %
Karlín, PRG, CZE	1 070	1 364	535	50.00 %
Raleigh, NC, USA	36 385	39 930	16 576	45.56 %
San Sebastián, ESP	4 782	4 961	2 173	45.44 %
Santa Cruz, CA, USA	13 060	14 494	6 059	46.39 %

The vertex cover number is another graph property heavily dependent on the graph size. However, if we once again take the relative size, defined as $\frac{v}{\|V\|}$, where v is the vertex cover number, we get a property that is very stable among all the used datasets. The relative size of the vertex cover stays between 45 and 50 percent at all studied graphs.

6.5 Edge Cover Number

The last studied property was the *edge cover number* as defined in Subsubection 1.4.2.2

6.5.1 Motivation

The edge cover number is another structural parameter known to be computable in polynomial time [7]. We have decided to measure this parameter, so that we have another parameter to test the results of the algorithm against. However, unlike the previous three parameters, this parameter is efficiently computable. Therefore, the potential algorithm can calculate the value of its currently constructed network, while it is running, and alter its result and flow accordingly. That is not possible for the previous properties, as the designed algorithm has to be reasonably efficient (therefore it can not solve NP-hard problems while running).

6.5.2 Measurement

Even though this is by far not the optimal approach[4]. We have described this problem with integer linear programming and used the Gurobi solver [18]. Although we are technically converting a simple problem solvable in polynomial time, to an NP-hard problem, which may seem like an irrational move; it was the fastest way to create a solver in terms of the human time spent developing it. Since the Gurobi solver is greatly optimized, we were able to get the results for all datasets in few seconds nevertheless.
We have described the problem in integer linear programming as follows: Let there be a binary variable x_e for every edge $e \in E$, set to 1, if the edge e is present in the edge cover; else set to 0. Minimize $\sum_{e \in E} x_e$ subject to: $\forall v \in V : \sum_{e \in E, v \in e} x_e \geq 1$.

6.5.3 Results

The results of our measurements can be seen in Table 6.5. In this table, ECN stands for edge cover number, r. size of EC stands for relative size of edge cover and r. dist. to opt. stands for relative distance to optimum.

Table 6.5 Measured values of edge cover number.

Dataset	‖V‖	‖E‖	ECN	R. Size of EC	R. Dist. to Opt.
Černý Most, PRG, CZE	1 553	1 893	806	42.58 %	1.56 %
Cēsis, LAT	1 122	1 244	606	48.71 %	3.62 %
College Park, MD, USA	5 110	6 392	2 725	42.63 %	2.66 %
Dejvice, PRG, CZE	1 675	1 907	900	47.19 %	3.28 %
Grenoble, FRA	9 647	11 688	5 093	43.57 %	2.31 %
Helsinki, FIN	115 318	123 291	63 343	51.38 %	4.61 %
Karlín, PRG, CZE	1 070	1 364	549	40.25 %	1.03 %
Raleigh, NC, USA	36 385	39 930	19 942	49.94 %	4.38 %
San Sebastián, ESP	4 782	4 961	2 639	53.19 %	5.00 %
Santa Cruz, CA, USA	13 060	14 494	7 135	49.23 %	4.17 %

We have introduced two additional columns in the table. First of them being the relative size of the edge cover taken against the set of all edges. This column is calculated using formula $\frac{e}{\|E\|}$, where e is the edge cover number. The second additional column is a relative distance to optimum. The optimum we are referring to is the theoretical best edge cover of size $\frac{\|V\|}{2}$. The relative distance to the optimum is calculated as $\frac{e - \frac{\|V\|}{2}}{\|E\|}$, where e once again represents the edge cover number.
The relative size differs highly, depending on the *density* of graphs. We can notice that San Sebastián, which had low density and therefore a low treewidth, has one of the highest relative size of the edge cover.
More interesting is the second parameter, we can see that this parameter does not exceed 5 percent for any of the datasets. Therefore, we can assume that the edge cover number of a sidewalk network should typically not exceed 50 % of its vertices + 5 % of its edges.

[4] In terms of computational complexity.

Chapter 7
Conclusion

We conclude this thesis with a final table containing all the measured data. In this table FESN stands for feedback edge set number, FVSN stands for feedback vertex set number, TW stands for treewidth, VCN stands for vertex cover number, ECN stands for edge cover number.

Table 7.1 Measured values of all parameters.

Dataset	‖V‖	‖E‖	FESN	FVSN	TW	VCN	ECN
Černý Most, PRG, CZE	1 553	1 893	377	146–167	10	764	806
Cēsis, LAT	1 122	1 244	245	110–113	10	521	606
College Park, MD, USA	5 110	6 392	1 534	571–595	15–22	2 440	2 725
Dejvice, PRG, CZE	1 675	1 907	372	173	9	790	900
Grenoble, FRA	9 647	11 688	2 711	1 017–	14–25	4 672	5 093
Helsinki, FIN	115 318	123 291	18 483	7 842–		52 824	63 343
Karlín, PRG, CZE	1 070	1 364	345	138–153	8	535	549
Raleigh, NC, USA	36 385	39 930	5 667	2 472–		16 576	19 942
San Sebastián, ESP	4 782	4 961	813	407	7	2 173	2 639
Santa Cruz, CA, USA	13 060	14 494	2 710	1 168–	13–16	6 059	7 135

The main takeaways of our research can be summarized as follows:

- when holding a minimum spanning tree or a minimum spanning forest of a graph, we should add 15–25 % of edges to create a realistic sidewalk network;
- the edge cover number of a realistic sidewalk network should not exceed 50 % of its vertices and 5 % of its edges;
- the minimal vertex cover of a realistic sidewalk network should contain between 45 and 50 percent of its vertices;
- the treewidth of a realistic sidewalk network typically stays around 10 for smaller cities or neighbourhoods of larger cities, and it stays around 20 for medium sized cities;
- the minimal feedback vertex set of a realistic sidewalk network should contain between 8 and 15 percent of its vertices.

The most natural continuation of this thesis should be the development of the algorithm for sidewalk generation, towards which this thesis has made its research.
Additionally, it would be very interesting to obtain the sidewalk network data for other parts

of the world (Africa, Asia, ...) and measure the values of the properties we have measured. We unfortunately could not have included these parts of the world because of the lack of data available. It would be fascinating to discover, whether there exists a significant difference in the city design, based on the cultural diversity of people around the world.
Should there be an improvement in computation speed of modern computers, or discoveries in computer science of better algorithms for calculating graph properties, it would be tempting to remeasure the values of feedback vertex set number or treewidth, we failed to measure exactly.
Also there are many more alluring graph properties to analyse like distance to disjoint stars, twinwidth, domination numbers or maximum independent set size; to name a few.

Bibliography

1. MANUKJAN, Marek. *Effective terrain data transformation using GPU acceleration*. Prague, Czech Republic, 2016. Available at https://dspace.cvut.cz/handle/10467/62758.
2. DIESTEL, Reinhard. *Graph Theory*. Vol. 173. 5th ed. Heidelberg: Springer, 2018. Graduate Texts in Mathematics. ISBN 978-3-662-53622-3.
3. PETERSEN, Julius. Sur le théorème de Tait. *L'Intermédiaire des Mathématiciens*. 1898, vol. 5, pp. 225–227.
4. BEINEKE, Lowell W.; VANDELL, Robert C. Decycling graphs. *Journal of Graph Theory*. 1997, vol. 25, no. 1, pp. 59–77. Available from DOI: https://doi.org/10.1002/(SICI)1097-0118(199705)25:1<59::AID-JGT4>3.0.CO;2-H.
5. USER30471 [online]. 2014. [visited on 2024-04-25]. Available from: https://tex.stackexchange.com/questions/207953/petersen-graph-with-new-tikz-graph-library.
6. HUSZÁR, Kristóf; SPREER, Jonathan; WAGNER, Uli. On the Treewidth of Triangulated 3-Manifolds. In: SPECKMANN, Bettina; TÓTH, Csaba D. (eds.). *34th International Symposium on Computational Geometry (SoCG 2018)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, vol. 99, 46:1–46:15. Leibniz International Proceedings in Informatics (LIPIcs). ISBN 978-3-95977-066-8. ISSN 1868-8969. Available from DOI: 10.4230/LIPIcs.SoCG.2018.46.
7. GAREY, Michael R.; JOHNSON, David S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990. ISBN 0716710455.
8. BEHSAZ, Babak; HATAMI, Pooya; MAHMOODIAN, Ebadollah. On minimum vertex cover of generalized Petersen graphs. *Australasian Journal of Combinatorics*. 2008, vol. 40, pp. 253–264.
9. ARORA, Sanjeev; BARAK, Boaz. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009. ISBN 9781139477369.
10. TURING, Alan M. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*. 1950, vol. LIX, no. 236, pp. 433–460. ISSN 0026-4423. Available from DOI: 10.1093/mind/LIX.236.433.
11. KARP, Richard M. Reducibility among Combinatorial Problems. In: *Proceedings of a symposium on the Complexity of Computer Computations*. 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.

12. COOK, Stephen A. The complexity of theorem-proving procedures. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. STOC '71. ISBN 9781450374644. Available from DOI: 10.1145/800157.805047.

13. JAFFE, Arthur M. The Millennium Grand Challenge in Mathematics. *Notices of the American Mathematical Society*. 2006, vol. 53, no. 6, pp. 652–660.

14. PACE CHALLENGE. *About PACE Challenge*. 2024. Available also from: https://pacechallenge.org/about/. Accessed on: 2024-05-05.

15. SIERKSMA, Gerard; ZWOLS, Yori. *Linear and Integer Optimization: Theory and Practice*. 3rd ed. CRC Press, 2015. ISBN 978-1498710169.

16. DANTZIG, George B. Maximization of a Linear Function of Variables Subject to Linear Inequalities. In: KOOPMANS, Tjalling C. (ed.). *Activity Analysis of Production and Allocation*. Wiley, 1951, pp. 339–347.

17. KHACHIYAN, L.G. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*. 1980, vol. 20, no. 1, pp. 53–72. ISSN 0041-5553. Available from DOI: https://doi.org/10.1016/0041-5553(80)90061-0.

18. GUROBI OPTIMIZATION, LLC. *Gurobi Optimizer Reference Manual*. 2023. Available also from: https://www.gurobi.com.

19. RHOADS, Daniel; RAMES, Clément; SOLÉ-RIBALTA, Albert; GONZÁLEZ, Marta C.; SZELL, Michael; BORGE-HOLTHOEFER, Javier. Sidewalk networks: Review and outlook. *Computers, Environment and Urban Systems*. 2023, vol. 106, p. 102031. ISSN 0198-9715. Available from DOI: https://doi.org/10.1016/j.compenvurbsys.2023.102031.

20. OSAMA, Ahmed; SAYED, Tarek. Evaluating the impact of connectivity, continuity, and topography of sidewalk network on pedestrian safety. *Accident Analysis & Prevention*. 2017, vol. 107, pp. 117–125. ISSN 0001-4575. Available from DOI: https://doi.org/10.1016/j.aap.2017.08.001.

21. POKORNÝ, Jaroslav. *XML Technologie – Principy a aplikace v praxi*. Praha: Grada Publishing, a.s., 2008. ISBN 978-80-247-2725-7.

22. ECMA INTERNATIONAL. *ECMA-404: The JSON Data Interchange Syntax* [online]. 2nd ed. 2017-12. [visited on 2024-05-15]. Available from: https://ecma-international.org/publications-and-standards/standards/ecma-404/.

23. OPENSTREETMAP FOUNDATION, OpenStreetMap contributors. *About OpenStreetMap - OpenStreetMap Wiki*. 2024. Available also from: https://wiki.openstreetmap.org/wiki/About_OpenStreetMap. Accessed on: 2024-05-15.

24. RAMM, Frederik; TOPF, Jochen; CHILTON, Steve. *OpenStreetMap: Using and Enhancing the Free Map of the World*. Cambridge, England: UIT Cambridge, 2010. ISBN 978-1-906860-11-0.

25. OPENSTREETMAP CONTRIBUTORS. *Planet dump retrieved from https://planet.osm.org* [https://www.openstreetmap.org]. 2024.

26. BOEING, Geoff. *Modeling and Analyzing Urban Networks and Amenities with OSMnx* [Working paper]. 2024. https://geoffboeing.com/publications/osmnx-paper/.

27. *NetworkX*. NetworkX Developers, 2024. Available also from: https://networkx.org/. Accessed on: 2024-05-05.

28. GEOFABRIK GMBH, OPENSTREETMAP CONTRIBUTORS [online]. 2013. [visited on 2024-05-14]. Available from: https://bestofosm.org/about.html.

29. IWATA, Yoichi; WAHLSTRÖM, Magnus; YOSHIDA, Yuichi. Half-integrality, LP-branching, and FPT Algorithms. *SIAM Journal on Computing*. 2016, vol. 45, no. 4, pp. 1377–1411. Available from DOI: 10.1137/140962838.

30. IWATA, Yoichi. Linear-Time Kernelization for Feedback Vertex Set. In: CHATZIGIANNAKIS, Ioannis; INDYK, Piotr; KUHN, Fabian; MUSCHOLL, Anca (eds.). *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, vol. 80, 68:1–68:14. Leibniz International Proceedings in Informatics (LIPIcs). ISBN 978-3-95977-041-5. ISSN 1868-8969. Available from DOI: 10.4230/LIPIcs.ICALP.2017.68.

31. TAMAKI, Hisao. A heuristic for listing almost-clique minimal separators of a graph. *CoRR*. 2021, vol. abs/2108.07551. Available from arXiv: 2108.07551.

32. TAMAKI, Hisao. Heuristic Computation of Exact Treewidth. In: SCHULZ, Christian; UÇAR, Bora (eds.). *20th International Symposium on Experimental Algorithms (SEA 2022)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, vol. 233, 17:1–17:16. Leibniz International Proceedings in Informatics (LIPIcs). ISBN 978-3-95977-251-8. ISSN 1868-8969. Available from DOI: 10.4230/LIPIcs.SEA.2022.17.

33. TAMAKI, Hisao. Computing treewidth via exact and heuristic lists of minimal separators. In: *International Symposium on Experimental Algorithms*. Springer, 2019, pp. 219–236.

34. VAZIRANI, Vijay V. *Approximation Algorithms*. Springer-Verlag, 2003. ISBN 978-3-662-04565-7.