Fakultät für Mathematik
Lehrstuhl für Angewandte Geometrie und Diskrete Mathematik

Visualization of advanced graph algorithms

`push-relabel algorithm`
to solve the *maximum flow problem*
`label-setting algorithm`
to solve the *shortest path problem with resource constraints*

Final Report for an Interdisciplinary Project by Adrian Haarbach

Examiner:	Prof. Dr. Peter Gritzmann
Advisor:	Wolfgang F. Riedl
Submission Date:	December 30, 2016

I hereby confirm that this is my own work, and that I used only the cited sources and materials.

Munich, December 30, 2016

Adrian Haarbach

Abstract

This interdisciplinary project deals with the vivid visualization of advanced graph algorithms. In particular, algorithms to solve two distinctive problems in discrete math are considered. Namely the *maximum flow problem* as well as the *shortest path problem with resource constraints*. For efficiency reasons, one often employs advanced graph algorithms to solve above problems. The *maximum flow problem* is solved using the efficient `push-relabel algorithm`. To solve the *shortest path problem with resource constraints*, we employ a generic `label-setting algorithm` which follows the dynamic programming principle.

Both algorithms carry a lot of state variables and are thus not easy to understand intuitively. An additional visualization layer with an intuitive representation of all state variables and state transitions during algorithm execution was developed. It displays the height function of each node in case of the `push-relabel algorithm` or the pareto frontier of all labels resident in a certain node in case of the `label-setting algorithm`. To achieve the goal of a high interactivity, we replaced the previous Canvas based graph visualization code with a new implementation based on SVG, using D3.js, a JavaScript library for producing dynamic, interactive data visualizations in web browsers.

Zusammenfassung

Das vorliegende interdisziplinäre Projekt beschäftigt sich mit der anschaulichen Darstellung von fortgeschrittenen Graphalgorithmen. Betrachtet werden zwei Verfahren zur Lösung von Problemstellungen der diskreten Mathematik. Die zu visualisierenden Problemstellungen sind hierbei das *Max-Flow Problem* sowie das *Kürzeste-Wege Problem mit Ressourcenbeschränkungen*. Als Lösungsverfahren für die aufgeführten Problemstellungen werden aus Effizienzgründen häufig fortgeschrittene Graphalgorithmen herangezogen. Für die Lösung des *Max-Flow Problems* findet der bekannte `Push-Relabel Algorithmus` in der Praxis häufig Anwendung. Zur Lösung des *Kürzeste-Wege Problems mit Ressourcenbeschränkungen* wird mit einem `Label-Setting Algorithmus` ein bekanntes Verfahren der dynamischen Programmierung vorgestellt.

Beide Algorithmen führen eine Menge an Statusvariablen mit sich und sind deshalb nicht leicht intuitiv zu verstehen. Es wurde eine zusätzliche Visualisierungsebene mit intuitiver Repräsentation aller Statusvariablen und -übergänge entwickelt. Diese veranschaulicht die Höhenfunktion jedes Knotens im Falle des `Push-Relabel Algorithmus` oder dessen Pareto-Front im Falle des `Label-Setting Algorithmus`. Um hohe Interaktivität zu erreichen, ersetzten wir den bisherigen auf Canvas basierenden Graphen-Visualisierungscode mit einer neuen Implementierung basierend auf SVG mit D3.js, eine JavaScript Bibliothek zur Erzeugung dynamischer, interaktiver Datenvisualisierungen in Web Browsern.

Contents

Chapter 1

Introduction

All previous interdisciplinary projects [Sto13; Vel14; Sef15; BVZ15; Zön15; Fis16; Fei16] display the state of graph algorithms on top of a network visualization, e.g. by annotating vertices or edges with additional information.

For advanced graph algorithms [GT88; ID05], which are often employed for efficiency reasons, the state size to visualize may become quite large. It may be thus advantageous to visualize the state of such algorithms in an additional visualization layer.

Sketches of these visualizations exist in static form in textbooks [AMO93; Cor09; Jun13] trying to illustrate the idea of the algorithm. The interesting part is however the change of the state variables during algorithm execution, which is hard to print on paper. The motivation of this project was thus to develop two web applications with a highly dynamic and interactive visualization of these additional state variables. Since this interdisciplinary project report is limited in the same way as the textbooks, we encourage the readers to try out the web applications live.[1] All the code developed is made available as open source and can be used as basis for future projects.[2]

1.1 Related work

The primary sources of the algorithms of this work are [GT88; ID05]. Secondary source for the `push-relabel algorithm` is the review article [GT14] and for the `label-setting algorithm` three PhD theses, a diploma thesis and a journal article [Sol83; Zie01; Sch03; Fei+04; Gar09]. A deeper understanding of the problems at hand and a broader view of related algorithms was acquired using standard university textbooks [AMO93; Cor09; Jun13], where the last one comes from the math domain, the middle one from the computer science domain, while the first one lies somewhere in between. These allow to grasp the connection between *problem* and *algorithm*. Another important source of inspiration are the web resources such as lecture slides regarding maxflow [May13; Meh00; Wil07; Mat16] and SPPRC [Pet06]. The boost C++ library's documentation is a valuable source of information for both algorithms [Sie01; Dre06]. The SPPRC is additionally handled in an appealing website [NG13].

The implementation part of this interdisciplinary project is a large-scale refactoring of previous projects [Sto13; Vel14; Sef15; BVZ15; Zön15] over the duration of two years. The most drastical change is the usage of SVG and D3.js instead of a Canvas based

[1]web applications hosted at: `http://www.adrian-haarbach.de/idp-graph-algorithms`

[2]source code is available at: `https://github.com/adrelino/idp-graph-algorithms`

visualizations. A beta version of this project already forms the basis of the latest two interdisciplinary projects [Fis16; Fei16]. The needed JavaScript knowledge was acquired in part with the help of [Fla11; Cro08; Hav15; RB13; Her12; Ste10]. The first one is the definitive reference for javascript, the second one an advanced book about language features to use or to leave out, the third and fourth one a good introduction for beginners. The last two books cover important language aspects and design patterns, in particular scope and closure, prototypal-based inheritance, statics, singletons and code-reuse patterns. Concerning D3.js [BOH11], the crucial part one needs to understand is the data join and the enter, update and exit selection, which are nicely explained in two blog posts [Bos12; Bos16]. The introductory books [Mur13; Zhu13; Mee15] also give details on how to implement charts as used for the secondary visualization layer.

1.2 Contributions

The main contributions of this work are the following:

- New concepts for secondary visualization layers.
- Web apps implementing a push-relabel algorithm and a label-setting algorithm.

The contributions that will benefit future projects most directly are our improvements of the underlying implementation, structured according to MVC (Section 5.2):

Model A major refactoring of the basic Graph class with the extension to arbitrary resources, easier algorithm state handling and new upload/download functionalities.

View A complete rewrite of the abstract GraphDrawer class for network visualization using D3.js and SVG instead of Canvas with the possibility to download it in vector format at any time. A Logger utility which allows to log algorithm execution messages with up to three indentation levels.

Controller A new GraphEditor with support for modifying graphs with an arbitrary number of resources on nodes and edges. The new class Tab and a small refactoring improved future code reusability and simplified the reverse functionality and the synchronization between algorithm state and pseudocode lines.

1.3 Overview

The remaining chapters of this report are organized as follows: First (Chapter 2), we provide the necessary background knowledge from discrete math and algorithmic programming principles. Then (Chapter 3), we introduce the maximum flow problem before discussing previous work and providing important definitions, pseudocode and a visualization concept for the push-relabel algorithm that solves it. The next chapter (Chapter 4) follows the same structure for the shortest path problem with resource constraints and the label-setting algorithm that solves it. Subsequently (Chapter 5), we give an overview of our implementation with respect to used web technologies and the applied software design. We finally (Chapter 6) summarize this project and give hints for future work.

Chapter 2

Background

Problems defined on networks arise in many real-life applications, such as finding the fastest route between two cities or computing the maximum bandwidth of an internet connection. A network can mathematically be represented by a graph.

Definition 2.1 (simple directed graph)
A graph is an ordered pair $G = (V, E)$ consisting of a vertex (or node) set V and an edge (or arc) set E. The unqualified term "graph" usually refers to a *simple* graph which has no multiple edges or self-loop edges. In a *directed graph (digraph)*, E is a set of ordered pairs of vertices, a subset of the cartesian product of vertices, that is $E \subseteq V \times V$. If the graph is simple, each edge $e \in E$ can be uniquely identified by a pair of vertices from V, that is $e = (v, w)$ with $v, w \in V$ and $v \neq w$, where v is the start vertex and w is the end vertex of the directed edge or arc e [Jun13][1.6].

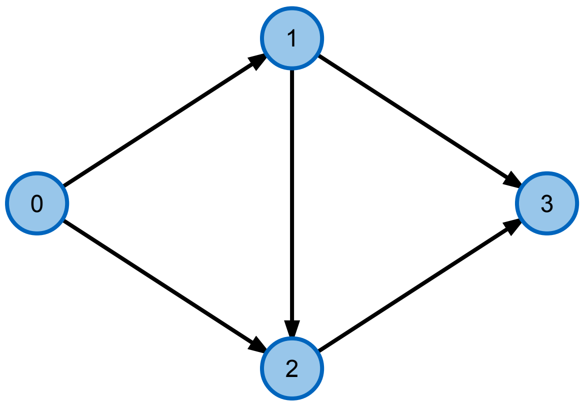


Figure 2.1: a digraph with $V = \{0, 1, 2, 3\}$, $E = \{(0, 1), (0, 2), (1, 2), (1, 3), (2, 3)\}$

For problems and algorithms defined on digraphs, it is often convenient to define the set of edges coming into a vertex v and zthe set of edges leaving it:

$$\delta^-(v) : \{e = (u, v) \in E \mid u, v \in V\} \qquad \text{incoming edges}$$
$$\delta^+(v) : \{e = (v, w) \in E \mid v, w \in V\} \qquad \text{outgoing edges}$$

In above example of a digraph (Fig. 2.1), the vertex $v = 2$ has an incoming edge set of $\delta^-(2) = \{(0, 2), (1, 2)\}$ and an outgoing edge set of $\delta^+(2) = \{(2, 3)\}$.

Advanced algorithms to solve graph problems rely on common programming principles. The first one is so basic that it actually forms the basis of sorting algorithms such as `MergeSort`.

Definition 2.2 (divide-and-conquer)
In divide-and-conquer, we solve a problem recursively, applying three steps at each level of the recursion: ***Divide*** *the problem into a number of subproblems that are smaller instances of the same problem.* ***Conquer*** *the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.* ***Combine*** *the solutions to the subproblems into the solution for the original problem.* [Cor09, ch. 4]

Recursion is clear and appealing from a mathematical perspective, but for computational efficiency reasons, divide-and-conquer is not always the best choice. It is better to remember useful intermediate results.

Definition 2.3 (dynamic programming)
Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. [...] divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when the subproblems overlap - that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem. [Cor09, ch. 15]

Dynamic programming can be applied to a wide range of graph problems, as we will see later. It works because of Bellman's:

Definition 2.4 (Principle of Optimality)
An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision [Bel57, sec. 3.3].

For some kind of problems, we can get even more efficient with the greedy approach originating from matroid theory [AMO93, sec. 13.7, p. 528] [Jun13, ch. 5]. It is at the heart of the efficient and well known Dijkstra algorithm.

Definition 2.5 (greedy)
Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do. A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. [Cor09, ch. 16]

Chapter 3

Maximum flows

3.1 The *maximum flow problem*

An important problem in many applications is to find out the maximum amount of flow that can simultaneously be transferred over a network between two points. Depending on the context, flow can mean different things, e.g the amount of water in a water pipe system in your city or the bandwidth of a computer network. We call such a network a *flow network*:

Definition 3.1 (flow network)
A *flow network* N is a 4-tuple $N = (G, c, s, t)$ consisting of a *digraph* G, a positive real-valued capacity function $c : E \to \mathbb{R}_+, \forall e \in E : c(e) \geq 0$ defined on all edges of the graph and two designated vertices, the *source* $s \in V$ and the *sink* (or target) $t \in V$ [AMO93][1.2].

However, the individual links in the network can only handle flow up to their maximum capacity, e.g. they are limited by the diameter of the water pipe. Additionally, the total flow must be preserved at the intermediate joints, e.g. we don't want leaks in our pipe system. This is called a *feasible flow*:

Definition 3.2 (feasible flow)
A *feasible flow* f from s to t is a mapping $f : E \to \mathbb{R}$ satisfying two constraints: The capacity constraint (3.1) ensures that the flow over an edge is always positive and not exceeding the edge's maximum capacity, while the flow conservation (3.2) assures that the total flow into a vertex $v \notin s, t$ equals the total flow out of v:

$$0 \leq f(e) \leq c(e) \forall e \in E \tag{3.1}$$

capacity constraint

$$\sum_{e \in \delta^-(v)} f(e) = \sum_{e \in \delta^+(v)} f(e) \forall v \in V \setminus \{s, t\} \tag{3.2}$$

flow conservation

There might be many feasible flows (e.g the zero flow $f = 0\, \forall e \in E$), but we are especially interested in transferring as much as possible across the network, the *maximum flow*:

Definition 3.3 (maximum flow and flow value)
A *maximum flow* $\max |f|$ is a *feasible flow* that maximizes the *flow value* $|f|$, the amount

of flow which flows from s to t. This is the net flow into the sink t or out of the source s:

$$|f| = \sum_{e \in \delta^-(t)} f(e) = \sum_{e \in \delta^+(s)} f(e) \tag{3.3}$$

flow value

An important concept in the context of flow algorithms is the residual network capturing possible change to f, defined by the residual capacities c' and the *residual graph* G':

Definition 3.4 (residual graph)
For a flow f in $G = (V, E)$, we can construct the *residual graph* $G' = (V, E')$ by copying all the vertices $v \in V$ from G and for each $e \in E$ adding one or two edges e' to E' with *residual capacity* c' under the following rules (Fig. 3.1):

forward edge if $f(e) < c(e)$ for an edge $e = (a, b) \in E$, then add the forward edge $e' = (a, b)$ with residual capacity $c'(e') = c(e) - f(e)$ to E'.

backward edge if $f(e) > 0$ for an edge $e = (a, b) \in E$, then add the backward edge $e' = (b, a)$ with residual capacity $c'(e') = f(e)$ to E'.

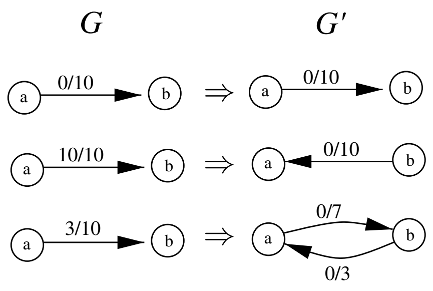


Figure 3.1: Construction of the residual graph G' [May13].

3.2 The push-relabel algorithm

An early way to compute a maximum flow on a directed graph was called the augmenting path method by Ford-Fulkerson [FF56].[1] A path with available capacity is an augmenting path, an s-t path in the residual graph. As long as such paths exist, one can increase the flow globally on these paths. If the method terminates, it computes a maximum flow. It is called a method and not an algorithm because the way to find augmenting paths in the residual graph is not fully specified. Furthermore, there is no guarantee on termination and runtime. About 15 years later, two algorithms based on the augmenting path method were developed. They ensure a polynomial time bound by augmenting flow along the shortest path first. The algorithm of Edmonds-Karp [EK72][2] ensures a runtime of $O(|V| \cdot |E|^2)$ while the one of Dinic [Din70][3] improves on that with a runtime of $O(|V|^2 \cdot |E|)$. This class of algorithms based on the Ford-Fulkerson method of augmenting paths has been visualized in a previous interdisciplinary project [Fis16].

Still about another 15 years later, an alternative and more efficient method which uses local operations based on the concept of a preflow and a height function was published: The push-relabel algorithm of Goldberg-Tarjan [GT88]. In contrast to the previous, less efficient algorithms based on augmenting paths, it changes the flow locally and only needs to construct the residual graph locally.

3.2.1 Excess and height

The increased efficiency comes at the cost of not maintaining a *feasible flow* during algorithm execution. Instead, a *preflow* is maintained:

Definition 3.5 (excess and preflow)
The *preflow* $\tilde{f}$ is a generalization of the flow f. The capacity constraint (3.1) of the edges is still maintained, but the flow conservation (3.2) at the vertices is not: The equality sign $=$ is replaced with $\geq$: $\sum_{e\in\delta^-(v)} f(e) \geq \sum_{e\in\delta^+(v)} f(e) \forall v \in V \setminus \{s\}$. One allows flow *excess*, that is, some vertices can have more incoming than outgoing flow at the intermediate stages of the algorithm [GT14]. The excess flow at a node v due to the preflow is non-negative for all nodes except for the start node and defined as:

$$e(v) = \sum_{e\in\delta^-(v)} f(e) - \sum_{e\in\delta^+(v)} f(e) \geq 0 \quad \forall v \in V \setminus \{s\} \tag{3.4}$$

excess

Definition 3.6 (active node)
As long as a vertex has positive (non-null) excess, it is called an *active node*.

Remark 3.7
An s-t preflow without active nodes is an s-t flow [Mat16].

[1]explained in textbooks [AMO93, sec. 6.4], [Cor09, sec. 26.2, p.724], [Jun13, sec. 6.1]
[2]explained in textbooks [Cor09, sec. 26.2, p. 727], [Jun13, sec. 6.2]
[3]explained in textbooks [AMO93, sec. 7.4], [Cor09, sec. 26.2], [Jun13, sec. 6.4]

Intuitively, we should try to push the excess at a node towards the sink, but what does that mean? Another important concept in the push-relabel algorithm is the height function, which is an approximation of a node's distance to the sink. The local *push* operations try to move *excess* at inner nodes 'downwards' towards the sink. If the current node is at a local minimum and still has excess, we *relabel* the node by increasing its *height* so that subsequent push operations can remove the excess.

Definition 3.8 (height, valid labeling)
A *height* function[4] $h(v) \geq 0 \quad \forall v \in V$ is defined for all vertices of the graph. It is a *valid labeling* of the nodes if it satisfies

$$h(t) = 0, h(s) = |V| \quad \text{and} \quad h(v) \leq h(w) + 1 \, \forall e' = (v, w) \in E' \tag{3.5}$$

height

Definition 3.9 (eligible edge)
An edge $e' = (v, w) \in E'$ of the residual graph G' is *eligible* if $h(v) = h(w) + 1$, meaning that the current node is one level above the one to where we wish to push excess to.

3.2.2 Termination and runtime

The important property of the push-relabel algorithm is that when the algorithm terminates, the computed preflow is actually a flow. *This is because there can be no augmenting path from s to t in the residual graph since any such path must contain a steep edge (since s is on level $|V|$, t is on level 0)* [Meh00].

The proof of the runtime involves counting the number of possible saturating and nonsaturating push operations as well as relabel operations. Depending on the way the vertices are selected we get different runtimes, proof sketches in [Meh00; Wil07; Mat16]:

- Generic (arbitrary selection rule) with runtime $O(|V|^2 \cdot |E|)$ explained and proved in [AMO93, sec. 7.6], [Cor09, sec. 26.4], [Jun13, alg. 6.6.1].
- Relabel-To-Front (FIFO selection rule) with runtime $O(|V|^3)$ explained and proved in [AMO93, sec. 7.7],[Cor09, sec. 26.5], [Jun13, alg. 6.6.14].
- Highest label selection rule with runtime $O(|V|^2\sqrt{|E|})$ explained and proved in [AMO93, sec. 7.8] [Jun13, alg. 6.6.16].

We implemented the Relabel-To-Front variant with the first-in-first-out (FIFO) selection rule. In this variant, a node with excess flow stays active either until a non-saturating push or a relabel occurred.

[4]also called distance labeling or level function

3.2.3 Pseudocode

Algorithm 1: Goldberg-Tarjan Push-Relabel algorithm with FIFO selection rule

```
   Input: digraph $G = (V, E)$ with nodes $s, t \in V$ and edge capacities $c(e) \forall e \in E$
   Output: A feasible maximum s-t flow f(e)
 1 (* Initialize the preflow *)
 2 forall $e = (u, w) \in E$ do
 3     $f(e) \leftarrow (u == s)\ ?\ c(e) : 0$
 4     if $u == s$ AND $w \neq t$ then
 5         $Q$.add(w)
 6 (* Initialize the height function *)
 7 $h(s) \leftarrow |V|$
 8 forall $v \in V \setminus \{s\}$ do
 9     $h(v) \leftarrow$ number of arcs on shortest v-t path
10 (* Main Loop *)
11 while $Q \neq \emptyset$ do
12     $v \leftarrow Q$.pop()
13     while $e(v) > 0$ AND $\exists\, e' = (v, w) \in E' \mid h(v) == h(w) + 1$ do
14         (* Push *)
15         push min$(e(v), c'(e'))$ flow from v to w
16         if $w \neq s, t$ AND $w \notin Q$ then
17             $Q$.add($w$)
18     if $e(v) > 0$ then
19         (* Relabel *)
20         $h(v) \leftarrow 1 + \min(\{h(w) | e^* = (v, w) \in E'\})$
21         $Q$.add($v$)
```

The pseudocode is split into blocks each consisting of a few lines to form different states of the algorithm that can be visualized. This basically transforms the algorithm into a finite state machine, which was first observed in another recent interdisciplinary project [Fei16]. The different states are: 1. INITPREFLOW (lines 1-5), 2. INITHEIGHT (lines 6-9), 3. MAINLOOP (lines 10-12), 4. ADMISSIBLEPUSH (line 13), 5. PUSH (lines 14-14), 6. ADMISSIBLERELABEL (line 18) and 7. RELABEL (lines 19-21). For the detailed description of these states we refer to our web application.

3.2.4 Visualization concept

The crucial requirement is [...] $h(v) = h(w) + 1$. Thus we are only allowed to push along [eligible] residual edges $e' = (v, w)$ for which $h(v)$ is exactly one unit larger than $h(w)$ [...]. We may visualize this rule by thinking of water cascading down a series of terraces of different height, with the height corresponding to the labels. Obviously, water will flow down, and [the eligible edge] condition has the effect of restricting the layout of the terraces so that the water may flow down only one level in each step [Jun13, sec. 6.6].

In our visualization concept (Fig. 3.2), we show how excess flow $e(v)$ is pushed downwards the terraces of different height $h(v)$. The primary visualization layer displays the graph network, the capacity of an edge and its current flow value. The secondary visualizaiton layer allows to arrange the graph nodes in a 2-dimensional chart, where the axis can be chosen to be y/x (the usual graph), height/id (so that no nodes overlap) or height/excess. The algorithm switches between the different axes depending on the current state.[5]

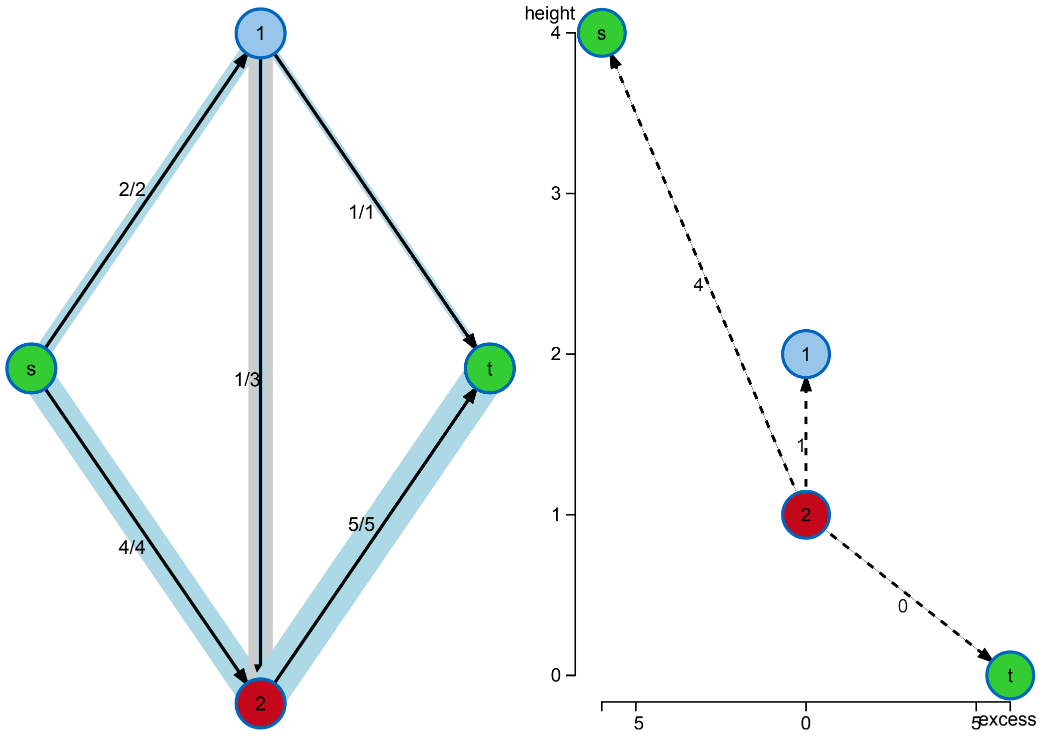


Figure 3.2: Maxflow concept : The primary visualization layer (left) shows the graph network, with vertices as circles and edges as lines connecting them. Source and target node are coloured in green, the current node in red. The labels of the edges denote the current flow and the maximum capacity in the form flow/cap. The capacity is furthermore drawn as a thick gray line with a width corresponding to its capacity, and the flow is drawn on top of it as a thick blue line. The secondary visualization layer (right) shows the outgoing edges e' of the current node in the residual graph with dashed lines. The nodes are currently arranged according to the height/excess coordinate system axes.

[5]The axes can also be kept fixed based on a user request from TU Ilmenau `https://github.com/adrelino/idp-graph-algorithms/commit/4f145861dfba5f8305a24c0f9cc4263cf2b17dcf`

Chapter 4

Shortest paths

4.1 The *shortest path problem with resource constraints*

Another basic problem defined on networks is how to traverse a network to get from one point to another one as cheaply as possible. For this problem we wish to find a shortest path between two points.

Definition 4.1 (path)
A *path* $P = (e_1, e_2, ...e_p)$ is a finite sequence of arcs (some arcs may occur more than once) where the end vertex of $e_i \in E$ is identical to the start vertex of $e_{i+1} \in E$ for all $i = 1, \ldots, p-1$. For simple graphs, a path can be also be written as $P = (v_0, v_1, \ldots, v_p)$ since the edges $e_i = (v_{i-1}, v_i)$ can be uniquely identified by the start and end vertex. The length of a path is p [ID05].

Definition 4.2 (SPP)
The ordinary shortest path problem (SPP) is perhaps the simplest of all network problems. It seeks an (unconstrained) *s*-*t* path of minimal cost (or length) between two points. A real-valued cost function $c : E \to \mathbb{R}$ is defined on all edges of the graph. The cost of a path is defined as the sum of the costs of all the edges along the path, that is $c(P) = \sum_{i=1}^{p} c(e_i)$. The problem exists in two variants: shortest paths from a single source (APSP) or between all pairs (APSP) of vertices.

Definition 4.3 (SPPRC)
A possible generalization of the SPP is the shortest path problem with resource constraints (SPPRC), where each edge additionally carries a secondary (possibly higher-dimensional) resource vector or function. A path P is now constrained at the intermediate vertices v_i with lower and upper bounds on the accumulated resource consumptions along the (partial) paths.

Definition 4.4 (SPPTW)
An illustrative example is the two-resource SPPRC, the shortest path problem with time windows (SPPTW). In addition to *cost* c, each edge additionally bears the resource *time* t. Thus each edge is associated with the two-dimensional resource vector $(t, c)\, \forall e \in E$. The secondary resource *time* is constrained, while the primary resource *cost* is unconstrained, but seeks to be minimized. The accumulated consumptions of the resource *time* along a path are constrained at the intermediate vertices along that path by lower and upper limits, that is the earliest arrival time t_a and the latest departure time t_b and called the *resource window*, denoted as tuples $[t_a, t_b]\, \forall v \in V$. The objective of the SPPTW is to find a resource-feasible s-t path $P^* = (v_0 = s, v_1, v_2, \ldots, v_n = t)$ of minimal cost:

- $\forall v_j \in P^* : t_a(v_j) \leq \sum_{i=1}^{j} time(v_{i-1}, v_i) \leq t_b(v_j)$
- $cost(P^*) = \min\{cost(P)\} \quad \forall$ feasible s-t paths P

4.2 The `label-setting algorithm`

We start with the distinction between label-setting [AMO93, ch. 4] and label-correcting [AMO93, ch. 5] algorithms that solve the ordinary shortest path problem (SPP). The first label-setting algorithm was suggested by Dijkstra [Dij59].[1] It solves the SSSP with non-negative edge weights and is visualized in [Vel14]. Its runtime is $O(|V|^2)$ which can further be improved to $O(|E| + |V| \log |V|)$ when using a Fibonacci heap for the queue of active nodes. After a node has been processed, the distance label of the path ending in that node is permanent and will thus not change in subsequent iterations. The algorithm is a greedy method and is thus very efficient. On the other hand, label-correcting algorithms allow negative edge weights and can detect negative cycles. Labels stay temporary until the very end of the algorithm execution. An example which solves the SSSP with negative edge weights is the Bellman-Ford algorithm.[Bel58; FF62][2] It has a runtime of $O(|V| \cdot |E|)$ and is visualized in [Sto13]. An example which solves the APSP is the Floyd-Warshall algorithm [Flo62; War62][3] with a runtime of $O(|V|^3)$ and it is visualized in [BVZ15]. These two algorithms are dynamic programming approaches.

The recent survey [ID05] gives an overview of algorithms for the solution of shortest path problems with resource constraints (SPPRC). They provide a generic SPPRC algorithm based on dynamic programming. Depending on the path selection strategy, the generic algorithm results in a label setting or label correcting algorithm. We choose the simplest variant of the SPPRC, the shortest path problem with time windows (SPPTW). The underlying network must either be acyclic or the resource consumptions for at least one resource must be strictly positive along a path to allow for a label-setting algorithm.

4.2.1 Dynamic programming solution

An optimal solution of the full problem consists of optimal solutions of partial problems, that is, the shortest s-t path with time window constrains contains optimal shortest partial paths s-v for all v on the shortest s-t path. This optimality principle of Bellman [Bel57] can be written as a dynamic programming (Chapter 2) recursion for time-windows $[t_a, t_b] \, \forall v \in V$ and resource vectors $(t, c) \, \forall e \in E$:

$$\begin{aligned} T(P_0) &= t_a^0 \\ T(P_i) &= \max\{t_a^i, T(P_{i-1}) + t_i\} \end{aligned}$$

Here, T stands for the accumulated consumption of the resource time t along the edges of a path P. In the recursion case, we take the maximum of earliest arrival time t_a and accumulated time consumption of the prefix path P_{i-1} extended along the current edge e, $T(P_{i-1}) + t_i$, because waiting at a node is allowed.

[1]explained in textbooks [AMO93, sec. 4.5],[Cor09, sec. 24.3],[Jun13, sec. 3.7, p. 83]
[2]explained in textbooks [AMO93, sec. 5.4],[Cor09, sec. 24.1],[Jun13, sec. 3.7, p. 87]
[3]explained in textbooks [AMO93, sec. 5.6],[Cor09, sec. 25.2],[Jun13, sec. 3.9]

4.2.2 Feasibility and domination

Fortunately, not all of the dynamic programming partial results are kept. First of all, it can be the case that $T(P_{i-1}) + t_i > t_b^i$. This means that the time window at the node is exceeded. The path or label obtained from above extension is *infeasible* and thus discarded. Secondly, path or labels who are feasible, but equal or worse in both time and cost than other feasible labels can be discarded. A label *dominates* all labels in its right upper cone in the chart of cost/time. The set of all such labels is *pareto-optimal.* Even though we will finally be interested in feasible, minimum cost labels ending in t, we cannot discard any of them at intermediate nodes since extensions of one label with lower cost but higher time might have unfeasible extensions at subsequent vertices along the paths while others with higher cost and lower time don't. Discarding labels makes the algorithm faster, but it is still not as efficient as a greedy (Chapter 2) approach.

4.2.3 Pseudocode

Algorithm 2: Generic Dynamic Programming SPPTW Label Setting Algorithm

```
Input: digraph $G = (V, E)$ with start node $s \in V$, target node $t \in V$, resource
       windows for all nodes and resource vectors for all edges
Output: feasible, pareto-optimal $s$-$t$ path $l^*$ with minimal cost

1  (* Initialize *)
2  $U \leftarrow \{(\epsilon, s)\}$ and $P \leftarrow \emptyset$
3  (* Main Loop *)
4  while $\exists l = (\sim, v) \in U$ do
5      $U \leftarrow U \setminus \{l\}$
6      (* Path extension step *)
7      forall $e = (v, w) \in E$ do
8          $l' = (l, w) \leftarrow$ EXTEND$(l, e)$
9          if $l' \in$ FEASIBLE$(w)$ then
10             $U \leftarrow U \cup \{l'\}$
11     $P \leftarrow P \cup \{l\}$
12     (* Dominance step *)
13     forall $v \in V \setminus \{s\}$ do
14         $U, P \leftarrow$ REMOVE-DOMINATED$(U, P)$
15 (* Filtering step *)
16 $l^* \in P \mid cost(l^*) == \min(\{cost(l = (\sim, t) \in P)\})$
```

The different states of the algorithm are: 1. INIT (lines 1-2), 2. MAINLOOP (lines 3-5), 3. PATHEXTEND (lines 6-8), 4. PATHEXTEND_FEASIBLE (lines 9-10), 5. LABEL_PROCESSED (line 11), 6. DOMINANCE_STEP (lines 12-13), 7. DOMINANCE_RESIDENTNODE (line 14) and 8. FILTERING_STEP (lines 15-16). For the detailed description of these states we refer to our web application.

4.2.4 Visualization concept

In our visualization concept (Fig. 4.1), we show how paths are extended and discarded. The primary visualization layer displays the graph network, the resource vector of an edge and the time-window at the vertices. Paths can be highlighted. The secondary visualization layer displays the corresponding labels in a 2-dimensional coordinate system, where the axes are cost/time. It also displays the time window of the current node as a rectangle. The labels to be displayed can be filtered by their resident nodes by selecting them, which allows to view the pareto-frontier of a node. By clicking on a label, the corresponding path is highlighted. These filters change dynamically during algorithm execution to highlight the paths and labels of the label currently extended, all labels resident in a node for dominance, or the minimum cost path for the filtering step.

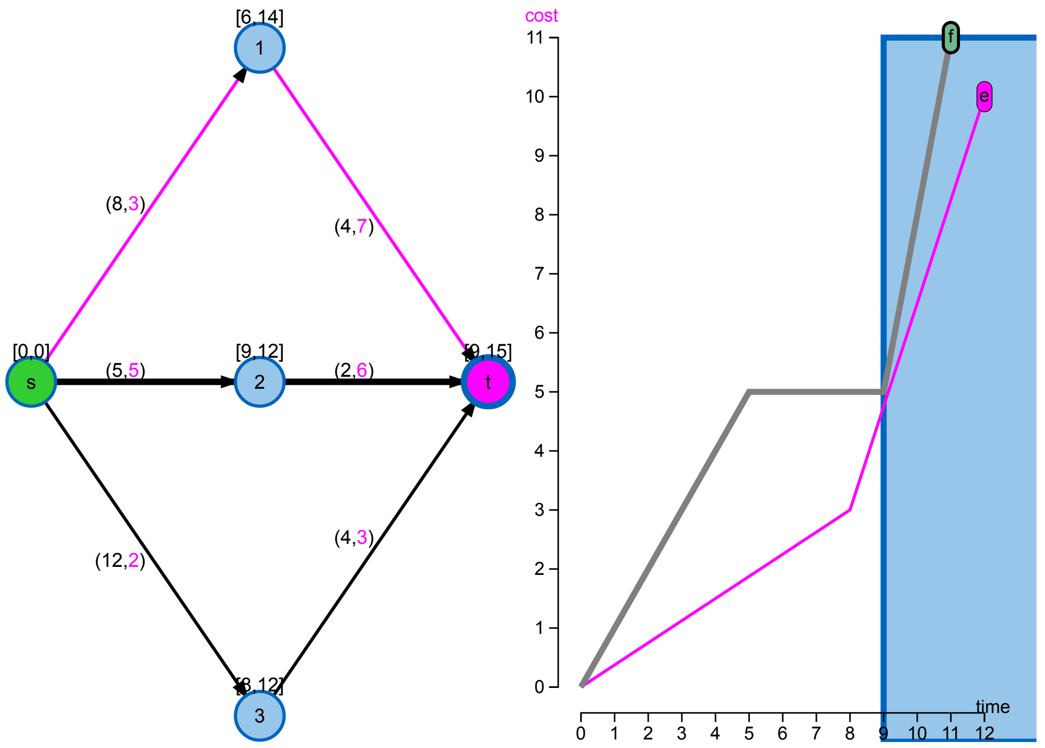


Figure 4.1: SPPTW concept: The primary visualization layer (left) shows the graph network, with vertices as circles and edges as lines connecting them. The labels of the edges denote the resource vector in the form (time,cost) and the labels above the nodes the time-window in the form [arrival,departure]. The secondary visualization layer (right) shows the paths of the labels resident in the currently selected node in a cost/time coordinate system. The time-window of the currently selected node t is drawn as a blue rectangle, whose corresponding circle is drawn with a thick blue border in the primary layer. The currently selected label f is drawn with a thicker path in both primary and secondary visualiztion layer. The pink color is only used in the very end of the algorithm execution, the above is a snapshot during the filtering step. It corresponds to the solution e of the algorithm, a time-feasible s-t path with time 12 and minimal cost 10.

Chapter 5

Implementation

The implementation is an evolution of previous web applications for the visualization of graph algorithms [Sto13; Vel14; Sef15; BVZ15; Zön15]. However, the requirements for graphs with arbitrary resources, a secondary visualization layer and high interactivity urged us to reimplement large parts of the existing codebase using different technologies (Section 5.1) while still maintaining the same look and feel. In the process, a complete understanding of the interplay between the different components was acquired, which allowed us to refactor them systematically to achieve a better software design (Section 5.2).

5.1 Web technologies

Web technologies form the basis of our implementation. The Mozilla Developer Network (MDN)[1] is an excellent reference, which subdivides the technologies into basics (Section 5.1.1), scripting (Section 5.1.2) and graphics (Section 5.1.3). Furthermore, we use two JavaScript Software libraries (Section 5.1.4) to facilitate certain tasks.

5.1.1 Basics: HTML, CSS, HTTP and AJAX

The HyperText Markup Language (HTML) is used to define the static *content* of the webpage. Each webpage contains just one HTML file, `maxflow-push-relabel/index_en.html` or `spp-rc-label-setting/index_en.html`, for all its static content. These serve as the entry point for our single-page web application and contain all language-specific features. For localization purposes one needs to modify only this file. Cascading Style Sheets (CSS) are used to describe the *appearance or presentation* of the content on the webpage. They can be used both for HTML (`library-d3-svg/css/style.css`) and for SVG Graphics such as our graph (`library-d3-svg/css/graph-style.css`). The Hypertext Transfer Protocol (HTTP) is used to deliver HTML and other hypermedia documents on the Web. The basic files we need everywhere are statically linked from within our HTML page with `<link href="...">` for CSS and `<script src="...">` for JavaScript files. By using asynchronous JavaScript and XML (AJAX) we can issue HTTP requests dynamically, for example when selecting another sample graph, without the need to completely reload the entire page.

[1] `https://developer.mozilla.org/docs/Web`

5.1.2 Scripting: JavaScript, DOM, Web APIs, HTML5 and Node.js

JavaScript is the scripting language that runs natively in a browser. It was originally developed to add interactivity and other dynamic features to a webpage. This is achieved by manipulating the Document Object Model (DOM), a programming interface for HTML, XML and SVG documents. It provides a structured representation of the document as a tree which can be modified and extended using JavaScript and other languages. The JavaScript language contains standard built-in global objects, e.g. the JSON[2] object with methods for parsing JavaScript Object Notation (JSON), used for serialization of algorithm state to provide the replay functionality. Web Application Programming Interfaces (Web APIs) complement the standard built-in global objects to provide a way to access the browser's advanced functionality programmatically. We make use of this functionality to serialize and download dynamically generated SVG using `XMLSerializer.serializeToString()`[3] and `WindowBase64.btoa()`[4] and we make use of the `FileReader.readAsText()`[5] for the local graph upload functionality. Because of their importance, the APIs and the DOM are now fundamental parts of the new HTML5 specification, which extends the HTML markup with new syntactic features such as `<video>`, `<audio>`, `<canvas>` and `<svg>` tags and the support for mathematical formulas with MathML markup.

Server-side scripting using Node.js

However, the JavaScript language itself is not restricted to client-side scripting in the browser, it can also be used for server-side scripting using the Node.js host environment. Nowadays there is a huge JavaScript ecosytem consisting of different open-source packages that allow to use JavaScript for all kinds of programming. We used NPM as package manager and Grunt as build tool to implement a simple web-server for development purposes. This circumvents some browsers' security settings of disallowing AJAX requests when files are served locally, which would prohibit us from loading different graphs.[6]

JavaScript - a fully featured programming language

JavaScript is not just a lightweight scripting language, it is actually a fully featured programming language. It has matured a lot during its standardization process (ECMAScript), a complete reference is given by [Fla11]. It is an interpreted, prototype-based, multi-paradigm dynamic scripting language with first-class functions. It supports imperative, object-oriented, and declarative or functional programming styles. As such, it is actually superior to more traditional languages such as Java or C++, which only made a functional programming style possible in their latest editions of Java 7 and C++ 11 through *lambdas*. The object-oriented part of JavaScript works a little different than in the traditional

[2]`https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/JSON`

[3]`https://developer.mozilla.org/docs/Web/API/XMLSerializer`

[4]`https://developer.mozilla.org/docs/Web/API/WindowBase64`

[5]`https://developer.mozilla.org/docs/Web/API/FileReader`

[6]In fact, this restriction is only present in Google Chrome and can be circumvented when starting it with the `--allow-file-access-from-files` flag. The latest versions of Firefox and Safari don't seem to have this restriction anymore as it was already observed by [Fei16]. Another alternative would be to install a production-quality web-server such as Apache, but this is overkill during development.

languages, because it relies on prototype object based inheritance instead of class based inheritance. A good introduction to the language suitable for beginners covering the important concepts of scope and closure of the functional style and prototypal-based inheritance of the object-oriented style is given by [Hav15; RB13]. While being an incredibly expressive and flexible language, the original creators of JavaScript initially made a few bad design decisions, but it is too late to remove them completely from the language nowadays. This is why it has become the *world's most misunderstood programming language* according to Douglas Crockford, whose advice is to stick to *JavaScript: The Good Parts* [Cro08], which is the title of his advanced book about the language features one should use and the ones to be avoided. Since we partly follow the object-oriented programming style to create reusable software components, we had to deal with good software design, which will be covered later (Section 5.2). To harness the full power of JavaScript effectively, e.g. with statics, singletons and different code-reuse patterns, we recommend [Her12].

5.1.3 Graphics: SVG vs. Canvas

SVG and Canvas are part of HTML5 and both are used to display interactive graphics on a webpage. The previous interdisciplinary projects were based on **Canvas**, which allows for dynamic, scriptable rendering of *raster-based* 2D graphics using JavaScript. It is a *low-level*, procedural model that updates a bitmap pixel by pixel using drawing routines. Once an object is drawn, it is forgotten by the browser. The complete scene thus has to be redrawn after any changes. The quality of the resulting bitmap is resolution dependent, leading to poor text rendering and scaling capabilities. On the other hand, Scalable Vector Graphics (**SVG**) is a *high-level* language for describing *vector-based* 2D graphics using XML notation, in complete analogy to HTML which describes page layout using XML notation. Similar to the HTML DOM which allows to access individual nodes of the page tree, individual shapes of the vector graphic can be accessed and modified via an SVG DOM, the scene graph. This allows to modify and re-render only a subset of the scene. Furthermore, it allows to attach JavaScript event handlers to individual shapes, easing interaction capabilities like clicks on nodes and edges or resources in our graph editor. Because of its advantages over Canvas, the previous interdisciplinary projects have lately been migrated to SVG [Fei16] and all current projects [Fei16; Fis16] already use our beta SVG implementation, profiting from its new features.

Arrowhead markers for directed edges

SVG primitives such as `circle` and `line` are used to represent a graph. For digraphs, we need directed arrows. These can be added to a line by setting its `marker-end` style to a previously defined marker url, e.g. `url(#arrowhead2)`. The marker definition is:

```
<marker id="arrowhead2" refX="24" refY="4" markerUnits="userSpaceOnUse" markerWidth="24"
markerHeight="8" orient="auto"><path d="M 0,0 V 8 L12,4 Z"></path></marker>
```

At its core is the path element defining a filled triangle using SVG's mini plotting language: `M 0,0` (move to the origin), `V 8` (draw a vertical line up to 8,0), `L12,4` (then a line to 12,4, which is the tip of the arrow/triangle), `Z` (go back to the origin and fill the resulting polygon). By using `orient="auto-start-reverse"` we can get arrows in the reverse direction, which we use to display backward edges of the residual graph using just the original edge. The concurrent project [Fei16] used an early version of our implementation which had the limitation that the marker definition was defined in the HTML page for each combination of arrow size and colour. We now removed most of the shortcomings. By using `markerUnits="userSpaceOnUse"` the size of the marker is independent of the line thickness, which previously led to misplaced and oversized markers when a line was highlighted by drawing it thicker. Above code is added dynamically via JavaScript to an invisible SVG element on the webpage so that we don't need to define it in HTML, easing reuse.[7]

Edge label positioning and anchor point

Edge labels should be positioned at the middle of a line [Fei16]. If the anchor point of a text element is in its middle, the line passes directly through the label, rendering it unreadable. The text should have a small offset from the line. We came up with a simple way that always gives readable edge labels by changing the anchor or origin of the text element to one of its four corners depending on the line orientation (Fig. 4.1). For this we used the `text-anchor` style and the `dominant-baseline` property of svg's `text` element. In previous Canvas-based projects, offsets had to be computed explicitly depending on the the text length, which was very error-prone.

Export functionality for vector graphics

The small file size and sharp images of this report are possible because all figures are vector graphics. In our web applications, we implemented a functionality allowing to download all SVG graphics. This export functionality is an important contribution for future scientific publications in discrete math, because it allows to easily include high-quality graph drawings at minimal file size into any documentation. Exporting the dynamically generated SVG from our webpage into this PDF document required some tricks. All styles regarding the appearance of nodes are defined in CSS files to be shared among different SVG graphics of the page. These need to be inlined into the `<defs>` section of the SVG DOM of each graphic before the download. In the same section, we also have to copy our shared marker definitions for directed edges, since otherwise our exported graph would look like an undirected graph. The modified SVG DOM root node is serialized to string and base64 encoded using above Web API's so that it can be downloaded by a simple click on a link in the browser. The downloaded, standalone SVG file can be opened by browsers and image processing or vector graphics software. One can use free software such as `inkscape.org` to convert the SVG vector graphic into a PDF vector graphic which is recognized by the LaTeX command `\includegraphics` used throughout this report.[8]

[7]It is important that the same definition exists only once per web page and not once per SVG element, because otherwise Firefox does not render the arrows correctly. Chrome however is unaffected by this.

[8]However, there is a bug `https://bugs.launchpad.net/inkscape/+bug/811862` in Inkscape, it does not support CSS' `dominant-baseline` property and `text-anchor` style for vertical and horizontal text alignment. Thus, the labels for nodes and edges are misplaced directly on top of the edges. A workaround is to open the SVG with a browser and using the print dialog to save it to PDF. This PDF can then be opened with Inkscape and cropped, the previously mentioned problems are no longer present.

5.1.4 Libraries: MathJax, jQuery (UI) and D3.js

JavaScript libraries were used to facilitate certain tasks. MathJax allows to write LaTeX math equations in HTML and render them nicely on a webpage via HTML5's MathML. The library jQuery (adding the global symbol $ to the JavaScript runtime) was already used in previous projects for easy HTML DOM element selection and modification using the concise syntax `$("#tabs")` instead of built-in lengthier code. Its plugin jQuery UI is used for the Graphical User Interface (GUI) of the application.

D3.js' core concept

DOM element selection and modification can also be done with another library, Data Driven Documents (D3 or D3.js) [BOH11], with the syntax `d3.select("#tabs")`. The real power of the library lies in the fact that it can select multiple elements at once using `selectAll()` and bind them to data using `data()`, which is called a *data join* (Fig. 5.1). The result of the join are enter, update and exit selections. New elements are added to the

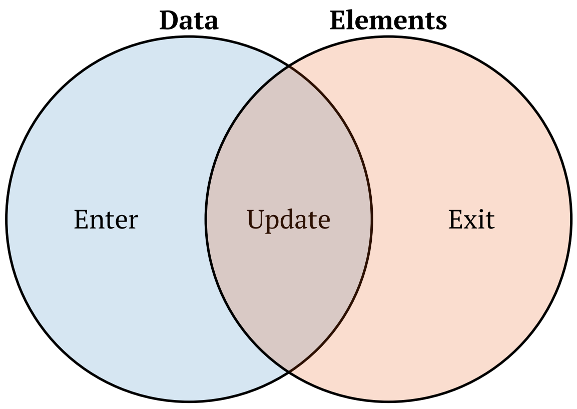


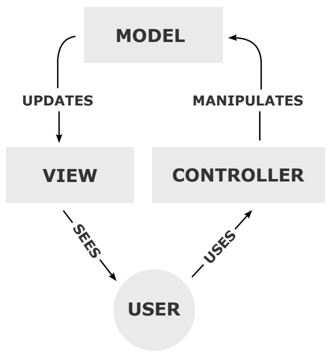
Figure 5.1: D3's *data join*: Data points joined to existing elements produce the update (inner) selection. Leftover unbound data produce the enter selection (left), which represents missing elements. Likewise, any remaining unbound elements produce the exit selection (right), which represents elements to be removed. From *Thinking with Joins* [Bos12].

enter selection using `append()`, existing elements of the update selection modified using `style()` or `attr()`, and old elements of the exit selection removed with `remove()`.[9] This is the *General Update Pattern, III* [Bos16], which is nicely illustrated using another of D3's strengths, transitions. These are used in our applicaiton to improve the understanding of the transitions between states of our algorithms. For the secondary visualization layer, we needed even more of D3's concepts, in particular axes and scales, because the numbers we want to visualize need to be converted to pixel coordinates that fit onto the screen. We recommend the books [Mur13; Zhu13; Mee15] as an introduction to these techniques.

[9]To get familiar with D3's basics I suggest running and actively modifying small code snippets by yourself, which can be done directly in the web-based presentation of this interdisciplinary project: `http://www.adrian-haarbach.de/idp-graph-algorithms/presentation/slides.html#/2/9`

5.2 Software design

We improved code quality and maintainability by adhering to object oriented programming best practices of *high cohesion and low coupling* (Fig. 5.2b). The separation of the components now fits the *Model-View-Controller (MVC)* design pattern (Fig. 5.2a), which was already applied partially in previous work, even better. A good reference explaining how to apply these patterns using JavaScript is [Ste10].



(a) Model-View-Controller(MVC)[10]

- Cohesion refers to the degree to which the elements of a class belong together, suggestion is all the related code should be close to each other, so we should strive for **high cohesion** and bind all related code together as far as possible. It has to do with the elements *within* the class.
- Coupling refers to the degree to which the different classes depend on each other, suggestion is all modules should be independent as far as possible, that's why **low coupling**. It has to do with the elements *among* different classes.

(b) high cohesion, low coupling[11]

Figure 5.2: Important design pattern and best practices in object oriented software engineering

The improvements make it easier to reuse and extend the code in other projects. Thus, an early prototype of our implementation already served as a base for concurrent interdisciplinary projects [Fis16; Fei16]. These described some details of our implementation and its benefits over previous approaches in German. Here, we want to describe the overall concept and all improvements made in a unified manner and in English to serve as a good starting point for future projects. The chapter is organized into sections according to the MVC principle.

5.2.1 Model

The graph Model of previous work was not very clean and thus substantially refactored and simplified while new features were added. It contained code concerning the naming, colouring and layout of nodes and edges and methods to draw them on Canvas and to detect clicks using coordinate comparisons. However, the Model should be oblivious to the actual graph drawing and interactions, since these belong to the View or the Controller. Moreover, the previous graph Model only allowed for a single scalar weight to be defined on edges. We extended it so that an arbitrary number of resources can be defined on both the edges and the nodes. This was especially needed for the SPPTW. Furthermore we added an associative array to nodes and edges to store the changing algorithm state, easing the replay functionality.

[10] https://en.wikipedia.org/wiki/Model-view-controller
[11] http://stackoverflow.com/questions/14000762/low-in-coupling-and-high-in-cohesion

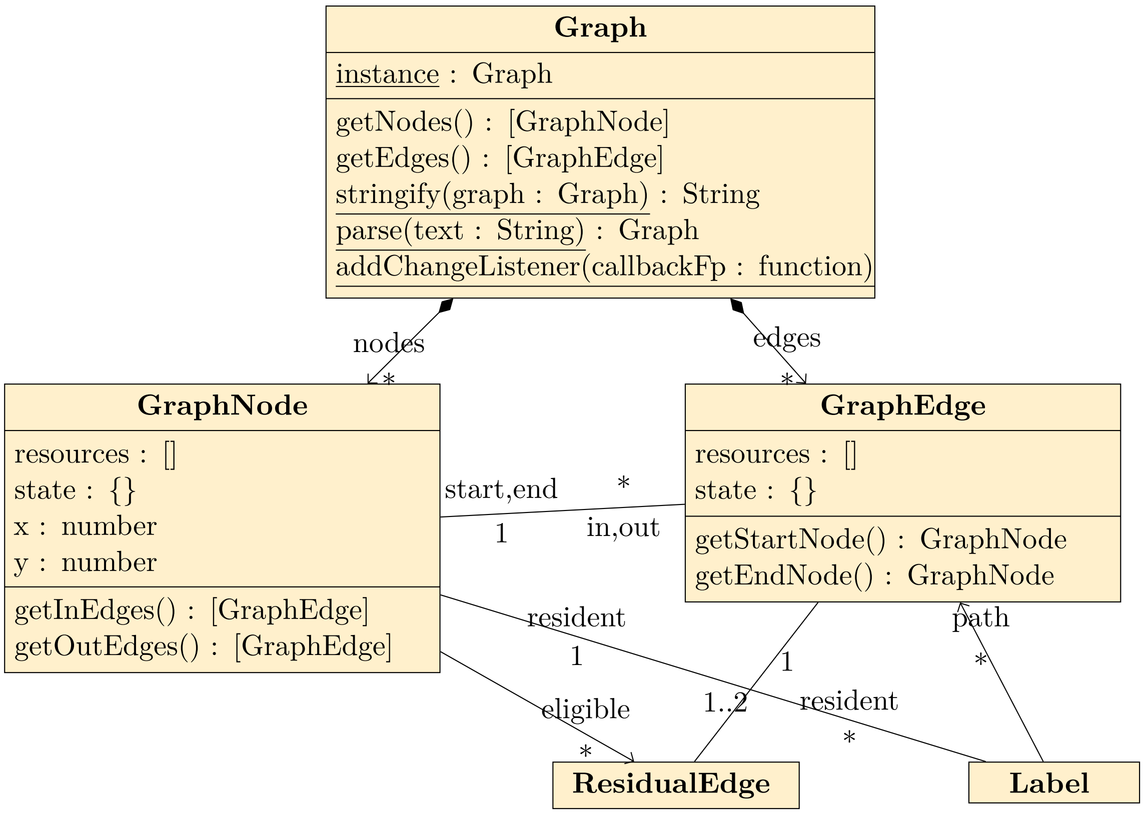


Figure 5.3: UML class diagram for the Model. Static attributes and methods are underlined. From a high-level perspective, we model a Graph as a composition of nodes and edges with associations between these two entities reflecting the network structure: Each GraphEdge has a start and an end node, while each GraphNode has an arbitrary number of incoming and outgoing edges. Arrays for resources denoted by [] and associate arrays for state variables denoted by {} are attributes of both nodes and edges. ResidualEdge and Label are two concepts that we needed for the implementation of our algorithms. These are primarily associated with GraphEdge, either 1:1 or 2:1 for the first or 1:n for the latter. For performance reason, we also established associations with GraphNode. A GraphNode can be queried for all its current outgoing eligible ResidualEdges, which is needed for applying a push operation. Label and GraphNode are associated bidirectionally: a Label needs to know its resident vertex so it can check path extensions easily for feasibility using the time-window of the vertex, while a GraphNode can be queried for all the Labels ending in it so we can apply dominance rules to them.

The new Graph class, composed by its subclasses GraphNode and GraphEdge, achieves high cohesion as can be seen in the UML diagram (Fig. 5.3). The static serialization and deserialization methods parse and stringify were extended to support arbitrary resource vectors. In previous work, the serialization capabilities were unaccessible to the end user. We provide a link to download a graph in its textual representation, which is backwards compatible to previous work. Additionally, a user can now locally upload a previously saved graph right from the browser using HTML5's FileReader (Section 5.1.2) capabilities.

The previous raw AJAX (Section 5.1.1) calls to load a saved sample graph from a server are now nicely wrapped in d3.text calls.

The loading of a graph, either from a local file or a remote server, is an asynchronous operation. According to *MVC* (Fig. 5.2a), the Model should update the View (Section 5.2.2) after any modifications to it, e.g. when another sample graph was selected or uploaded. We achieve *low coupling* (Fig. 5.2b) between the different components through the use of a static callback function registration method addChangeListener (Fig. 5.3). All functions of the View that have been registered will be called after the graph has been loaded asynchronically without errors. The current graph is made available to the View and the Controller (Section 5.2.3) via the static instance object of Graph. This singleton pattern allows the Controller to easily access and manipulate the Model and the View to easily update its visual representation after any changes.

5.2.2 View

A complete rewrite of the graph visualization code was necessary because of the different drawing concepts of SVG vs. Canvas (Section 5.1.3). D3.js is used extensively to achieve minimal code size with high expressiveness. The UML diagram (Fig. 5.4) shows all View classes in blue. The main result is the abstract GraphDrawer class, which should be used as a base class in all future projects. Full customization is easily possible by overwriting methods that will be called from inside of D3's *data join* (Section 5.1.4), in particular `onNodesEntered` and `onEdgesEntered` for the enter selections and `onNodesUpdated` and `onEdgesUpdated` for the update selections. For simple casess, it is sufficient to overwrite the methods `edgeText` and `nodeText` to customize the text that is drawn next to an edge or on top of a node. Per default these are the resource vector of the edge wrapped in `()` and possible constraints specified on the node wrapped in `[]`. The method `nodeLabel` returns the text displayed inside a node, typically a node's id. This method is typically overwritten to label start and end vertex with s and t. Any SVG based visualization can be saved to disk in vector format, for which styles and marker definitions are automatically inlined (Section 5.1.3). A Logger utility which allows to log algorithm execution messages with up to 3 indentation levels was furthermore developed. The log message data is joined with HTML list elements `<ol>` and `<ul>` using D3, which is not only limited to modifying SVG DOM, it can also handle HTML DOM. This class is very useful for development, but also for the final algorithm to display dynamic messages for each state of the algorithm and a complete trace of algorithm execution. The secondary visualization layers of each algorithm, namely the LabelDrawer and the ResidualGraphDrawer are also considered part of the View. The LabelDrawer (Fig. 4.1) is used to visualize the Model Label by plotting its associated path with accumulated resource consumptions in a 2-dimensional height/cost chart together with the resident vertex' time window resource constraints as a rectangle. The ResidualGraphDrawer (Fig. 3.2) is used to display all outgoing residual edges of the currently active vertex which are considered for push or relabel operations. By limiting the display in such a way, we avoid the need for multiedges, because either only the forward or only the backward residual edge e' of the original edge e is shown at any time. The ResidualGraphDrawer allows to change the arrangement of nodes by changing the coordinate axes. While the original graph layout is obtained with y/x axes,

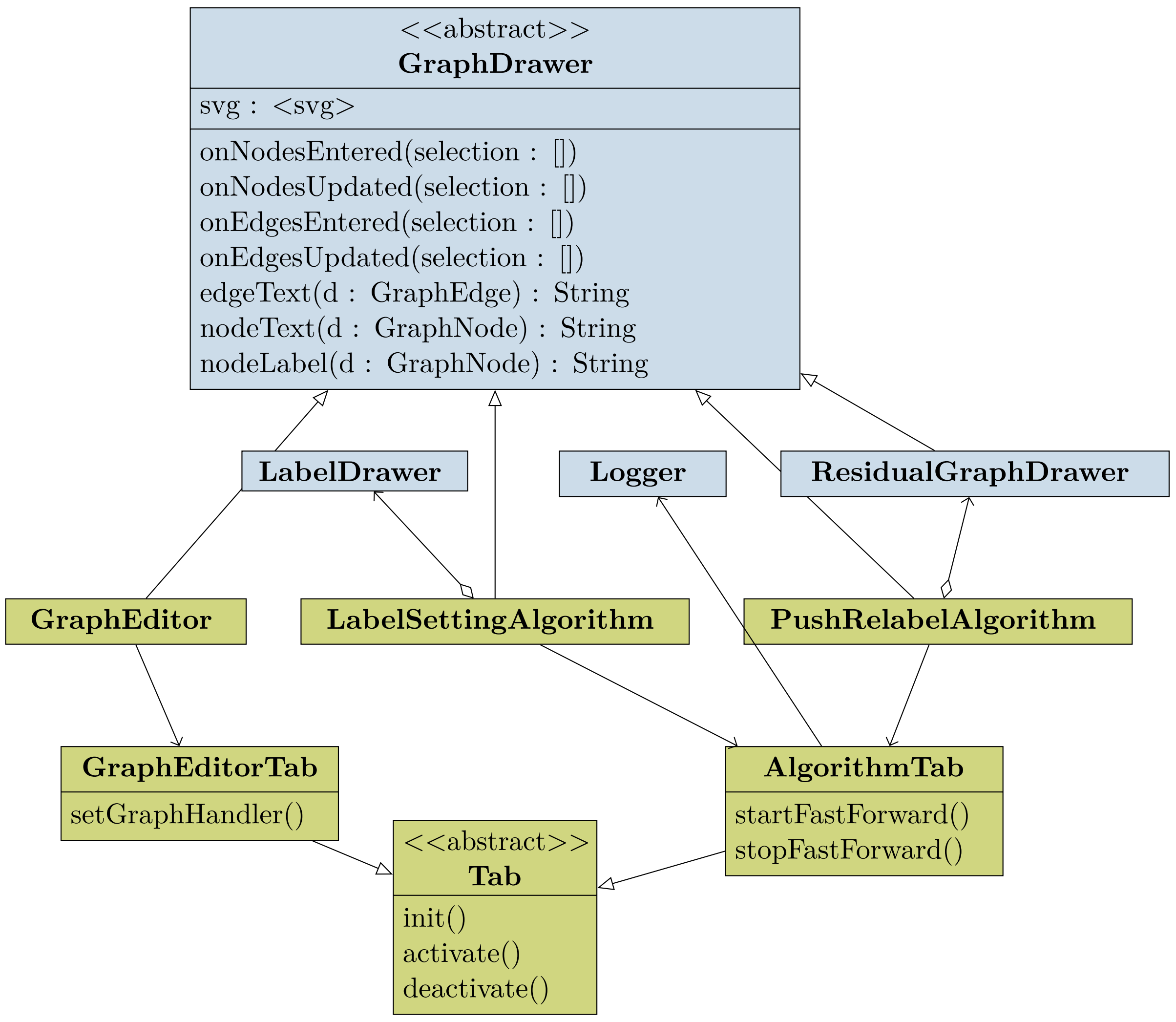


Figure 5.4: UML class diagram for View (blue) and Controller (green). The multiplicities are all 1:1 and thus not drawn. The abstract base class GraphDrawer of the View forms the basis for all graph-based visualizations. The Logger and the secondary visualization layers LabelDrawer and ResidualGraphDrawer are part of the View, but only the latter one displays a graph and thus extends the GraphDrawer. The abstract base class Tab of the Controller is extended to form a GraphEditorTab and an AlgorithmTab. Finally, the actual GraphEditor and the two implemented algorithms are associated with a Tab while inheriting from the GraphDrawer. Each algorithm has a secondary visualization layer, we model this relationship with an aggregation.

we can rearrange the nodes with height/id or height/excess axes in order to visualize the concepts of height function and excess. This layered view with the height on the y axis allows to easily see which residual edges are eligible by checking if the difference of their node's heights is exactly one.

5.2.3 Controller

The UML diagram (Fig. 5.4) shows all Controller classes in green. The basis of the Controller is the new abstract class Tab, which listens to changes of a Tab's visibility. The View of a Tab is defined in our HTML page with jQuery UI. The methods `init, activate` and `deactivate` should be overwritten to react to these visibility changes. A Tab is extended to form a GraphEditorTab and an AlgorithmTab with more specific functionality. The GraphEditorTab is used to wire together the events when another sample graph was selected or uploaded. These events trigger a call to the `setGraphHandler` method. This tab is used as an attribute of our new GraphEditor, which extends the GraphDrawer but is considered part of the Controller since it wires together different functionality. It comes with support for modifying graphs with an arbitrary number of resources on nodes and edges. This is achieved by rendering a number spinner HTML DOM element for each resource on top of the SVG network visualization, which allows to change the graph's underlying data Model conveniently. The AlgorithmTab connects the events of the fast forward jQuery UI buttons defined in the HTML page. Calls to the `startFastForward` and `stopFastForward` methods result from these events. It furthermore has a Logger instance of the View as an attribute, which allows to easily log and visualize algorithm execution. The AlgorithmTab is an attribute of the two main classes we developed for our algorithms, the LabelSettingAlgorithm and the PushRelabelAlgorithm. These extend the GraphDrawer and have an aggregation to their secondary visualization layers of LabelDrawer and ResidualGraphDrawer. The above design made it possible to separate most framework-related code from our main classes. If another algorithm is implemented in future work, one just has to copy and modify the LabelSettingAlgorithm or PushRelabelAlgorithm classes.

Chapter 6

Conclusion

This report first summarized the related work and the background needed for the graph problems and algorithms visualized in this project. We provided mathematical definitions for the maxflow and the SPPRC problems. Important concepts of the push-relabel and the label-setting algorithms were presented and their pseudocode provided. We motivated the need for a secondary visualization layer concept and implemented it. Two interactive web applications for advanced graph algorithms were developed. These can be accessed freely and the source code is made available as open source to ease further extensions. New technologies such as D3.js and SVG have been introduced to achieve high interactivity. In the process, we refactored or reimplemented large parts of the core code basis resulting in an improved software design, which is thoroughly documented to serve as a reference for future web applications building upon our implementation.

Future work

The following are possible directions for further extensions:

- Implement the highest-label selection rule for the push-relabel algorithm so it can be compared to the current FIFO selection rule.
- Allow edges of negative weight in the SPPRC and extend our label-setting algorithm into a label-correcting algorithm that solves this problem.
- Over the years, a lot of web applications have been developed independently by different students. A lot of code was copied and partially modified, which resulted in a huge amount of duplication. The duplication of JavaScript files was minimized through the improved software design of this project. However, the static HTML files are still copied and adapted for each new project, even though some parts don't change at all. To overcome this issue, one could generate the HTML sites dynamically on a server using PHP, which is unattractive because our apps are client-side only. The nicest solution would be the W3C working draft of *HTML Imports*, which would allow to split HTML into parts and load the parts just like CSS or JavaScript. However, it is only implemented in Chrome so far.[1] The most practical way in my opinion is to split up the HTML file into parts and then use the JavaScript ecosystem to concatenate the pieces to turn it into a complete HTML page before deployment.

[1] https://www.w3.org/TR/html-imports/ and http://caniuse.com/#search=imports

Bibliography

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice hall, 1993. ISBN: 0-13-617549-X.

[BVZ15] M.-J. Becker, A. Voroncovs, and R. Zabrodin. "Adaption eines didaktischen Konzepts zur Darstellung weiterführender Graphalgorithmen in einer Web-Applikation". *The Floyd-Warshall Algorithm*. Interdisciplinary Project. Technische Universität München, June 2015. URL: https://www-m9.ma.tum.de/graph-algorithms/spp-floyd-warshall/.

[Bel57] R. Bellman. *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 1957.

[Bel58] R. Bellman. "On a routing problem". In: *Quarterly of applied mathematics* (1958), pp. 87–90.

[Bos16] M. Bostock. *General Update Pattern, III*. Nov. 2016. URL: http://bl.ocks.org/mbostock/3808234.

[Bos12] M. Bostock. *Thinking with Joins*. Feb. 2012. URL: https://bost.ocks.org/mike/join/.

[BOH11] M. Bostock, V. Ogievetsky, and J. Heer. "D^3 data-driven documents". In: *IEEE transactions on visualization and computer graphics* 17.12 (2011), pp. 2301–2309.

[Cor09] T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.

[Cro08] D. Crockford. *JavaScript: The Good Parts*. " O'Reilly Media, Inc.", 2008.

[Dij59] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1.1 (1959), pp. 269–271.

[Din70] E. A. Dinic. "Algorithm for solution of a problem of maximum flow in a network with power estimation". In: *Soviet Math Doklady* 11 (1970), pp. 1277–1280.

[Dre06] M. Drexl. *r-c-shortest-paths*. The boost C++ library documentation, version 1.62. 2006. URL: http://www.boost.org/doc/libs/1_62_0/libs/graph/doc/r_c_shortest_paths.html.

[EK72] J. Edmonds and R. M. Karp. "Theoretical improvements in algorithmic efficiency for network flow problems". In: *Journal of the ACM (JACM)* 19.2 (1972), pp. 248–264.

[Fei16] J. Feil. "Visualisierung fortgeschrittener Graphalgorithmen mit D3.js am Beispiel des Blossom Algorithmus". *Edmonds's Blossom Algorithm*. Interdisciplinary Project. Technische Universität München, Nov. 2016. URL: https://www-m9.ma.tum.de/graph-algorithms/matchings-blossom-algorithm/.

[Fei+04] D. Feillet, P. Dejax, M. Gendreau, and C. Gueguen. “An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems”. In: *Networks* 44.3 (2004), pp. 216–229.

[Fis16] Q. Fischer. “Visualisierung von Flussalgorithmen”. *Ford-Fulkerson Algorithm (IDP Flow Algorithms)*. Interdisciplinary Project. Technische Universität München, Sept. 2016. URL: `https://github.com/fischerq/idp-flow-algorithms`.

[Fla11] D. Flanagan. *JavaScript: The Definitive Guide, 6th Edition*. 6th. O’Reilly Media, 2011. ISBN: 0596805527,9780596805524. DOI: `doi/10.1002/cbdv.200490137`.

[Flo62] R. W. Floyd. “Algorithm 97: shortest path”. In: *Communications of the ACM* 5.6 (1962), p. 345.

[FF56] L. R. Ford and D. R. Fulkerson. “Maximal flow through a network”. In: *Canadian journal of Mathematics* 8.3 (1956), pp. 399–404.

[FF62] L. R. Ford and D. Fulkerson. “Flows in Networks”. In: (1962).

[Gar09] R. Garcia. “Resource constrained shortest paths and extensions”. PhD thesis. Georgia Institute of Technology, 2009.

[GT88] A. V. Goldberg and R. E. Tarjan. “A new approach to the maximum-flow problem”. In: *Journal of the ACM (JACM)* 35.4 (1988), pp. 921–940. DOI: `10.1145/48014.61051`.

[GT14] A. V. Goldberg and R. E. Tarjan. “Efficient Maximum Flow Algorithms”. In: *Commun. ACM* 57.8 (Aug. 2014), pp. 82–89. ISSN: 0001-0782. DOI: `10.1145/2628036`. URL: `http://doi.acm.org/10.1145/2628036`.

[Hav15] M. Haverbeke. *Eloquent JavaScript: A Modern Introduction to Programming*. 2nd ed. No Starch Press, 2015. URL: `http://eloquentjavascript.net/`.

[Her12] D. Herman. *Effective JavaScript: 68 Specific ways to harness the power of JavaScript*. Addison-Wesley, 2012.

[ID05] S. Irnich and G. Desaulniers. “Shortest path problems with resource constraints”. In: *Column generation*. Springer, 2005, pp. 33–65. ISBN: 978-0-387-25485-2. DOI: `10.1007/0-387-25486-2_2`.

[Jun13] D. Jungnickel. *Graphs, networks and algorithms*. 4th ed. Springer, 2013. ISBN: 978-3-642-32278-5. DOI: `10.1007/978-3-642-32278-5`.

[Mat16] J. Matuschke. *Network Flows (MA5518), Lecture 2*. Lecture notes. Technische Universität München. Apr. 2016. URL: `https://www-m9.ma.tum.de/foswiki/pub/SS2016/NetworkFlows/nf160426.pdf`.

[May13] E. W. Mayr. *Praktikum Algorithmenentwurf (Teil 6)*. Lecture notes. Technische Universität München. Nov. 2013. URL: `http://wwwmayr.in.tum.de/lehre/2013WS/algoprak/uebung/tutorial6.english.pdf`.

[Mee15] E. Meeks. *D3.js in Action*. Manning, 2015.

[Meh00] K. Mehlhorn. *The Maximum Flow Problem*. Lecture notes. Max Planck Institut Informatik. Nov. 2000. URL: `http://people.mpi-inf.mpg.de/~mehlhorn/DatAlg/Maxflow.pdf`.

[Mur13] S. Murray. *Interactive data visualization for the Web.* " O'Reilly Media, Inc.", 2013.

[NG13] Networking and E. O. R. Group. *Vehicle Routing Problem with Time Windows (VRPTW).* Website, Universidad de Malaga, Spain. Jan. 2013. URL: http://neo.lcc.uma.es/vrp/vrp-flavors/vrp-with-time-windows/.

[Pet06] B. Petersen. *Label-Setting Algorithm for Shortest Path Problems, Recent Research Results.* Lecture notes for lecture 3, DIKU, University of Copenhagen. Nov. 2006. URL: http://www.diku.dk/OLD/undervisning/2006-2007/2006-2007_b2_426/slides3.pdf.

[RB13] J. Resig and B. Bibeault. *Secrets of the JavaScript Ninja.* 1st ed. Manning, 2013. ISBN: 193398869X,9781933988696.

[Sch03] T. Schlechte. "Das Resource-Constrained-Shortest-Path-Problem und seine Anwendung in der ÖPNV-Dienstplanung". Diplomarbeit. Technische Universität Berlin, 2003.

[Sef15] S. R. Sefidgar. "Enhancement and Adaption of a didactic Concept to the Presentation of Spanning Tree Algorithms in a Web Application". *Prim's Algorithm.* Interdisciplinary Project. Technische Universität München, Apr. 2015. URL: https://www-m9.ma.tum.de/graph-algorithms/mst-prim/.

[Sie01] J. Siek. *push-relabel-max-flow.* The boost C++ library documentation, version 1.62. 2001. URL: http://www.boost.org/doc/libs/1_62_0/libs/graph/doc/push_relabel_max_flow.html.

[Sol83] M. M. Solomon. "Vehicle routing and scheduling with time window constraints: Models and algorithms". PhD thesis. 1983.

[Ste10] S. Stefanov. *JavaScript patterns.* " O'Reilly Media, Inc.", 2010.

[Sto13] R. Storz. "Entwicklung und Implementierung eines didaktischen Konzepts zur Veranschaulichung verschiedener Graphalgorithmen zum Einsatz in der gymnasialen Oberstufe". *The Bellman-Ford Algorithm.* Interdisciplinary Project. Technische Universität München, Apr. 2013. URL: https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford/.

[Vel14] L. Velden. "Entwicklung und Implementierung eines didaktischen Konzepts für die Wissenskontrolle zu verschiedenen Graphenalgorithmen zum Einsatz in der gymnasialen Oberstufe". *Der Dijkstra - Algorithmus.* Interdisciplinary Project. Technische Universität München, May 2014. URL: https://www-m9.ma.tum.de/graph-algorithms/spp-dijkstra/.

[War62] S. Warshall. "A theorem on boolean matrices". In: *Journal of the ACM (JACM)* 9.1 (1962), pp. 11–12.

[Wil07] D. P. Williamson. *ORIE 633: Network Flows.* Lecture notes for lectures 3,4,5. Cornell University. Sept. 2007. URL: https://people.orie.cornell.edu/dpw/orie633/.

[Zhu13] N. Q. Zhu. *Data visualization with D3.js cookbook.* Packt Publishing Ltd, 2013.

[Zie01] M. Ziegelmann. “Constrained shortest paths and related problems”. PhD thesis. Universität des Saarlandes, 2001.

[Zön15] B. Zönnchen. “Darstellung des k-Center Problems in einer Web-Anwendung”. Interdisciplinary Project. Technische Universität München, July 2015.