Bachelor-Arbeit, Implementierung eines Routing Algorithmus

Fabian Halama, f.halama@fu-berlin.de *

April 2023

Inhaltsverzeichnis

Zusammenfassung

Diese Bachelor Arbeit ist ein Text, zu einem Computer-Programm. Außerdem sind Ergebnisse aus Testläufen enthalten. Alle Dateien zu dem Projekt existieren, in einem Gitlab Repository. Die Gitlab Instanz wird von dem Fachbereich gehostet. Das Repository ist nur mit Einladung sichtbar.

*Freie Universität Berlin, Informatik Institut

1 Einleitung

`Routing Algorithmen` arbeiten auf einem Netzwerk von Knoten. Sie beschreiben, wie Wege von einem Start-, zu einem Ziel-Knoten, gefunden werden. Der implementiere Algorithmus ist ein `Routing Scheme`. Routing Schemes sind verteilte, dezentrale Algorithmen. Bei dem implementierten Algorithmus, wird bei dem Preprocessing ein zentralisierter Prozess verwendet [SK85]. Wobei die genutzten Daten, auch aus einem dezentralen Netzwerk, zusammengetragen werden können.

Routing Schemes haben zwei Komponenten, eine Routing Funktion und Routing Tables. Die `Routing Funktion` wird in einem Knoten aufgerufen und berechnet immer nur einen Schritt. Das Routing Scheme führt die Funktion solange wiederholt aus, bis der Ziel-Knoten erreicht ist. Die `Routing Tables` enthalten alle Informationen, die der Funktion zur Verfügung stehen. Jeder Knoten speichert einen Routing Table. Konkrete Anwendung finden sie beim Internet: Die Daten sind auf Server, auf der ganzen Welt, verteilt und werden in Paketen versendet. Dabei durchlaufen sie unzählige Zwischenstationen, bevor sie auf einem PC ankommen. Jede Zwischenstation hat eine Routing-Tabelle, in der sie nachschaut, wohin sie das Paket weiterleitet.

In Routing Schemes gibt es immer ein Abwiegen, zwischen der Größe der Routing Tabellen und dem Stretch Faktor, der Abweichung von kürzesten Wegen. Es gilt, dass jedes Scheme mit einem Stretch Faktor < 5 einen Gesamtspeicherbedarf von $\Omega(n^{3/2})$ Bits hat, bei einem Netzwerk von n Knoten. Dabei gibt es mindestens einen Knoten, mit mindestens $\Omega(n^{1/2})$ Bits. Für einen Faktor <3 sind es $\Omega(n^2)$ Bits, mit mindestens $\Omega(n)$ Bits bei einem Knoten [TZ01]. Außerdem gibt es Unterscheidungen der Schemes anhand der Kommunikation. Einige Schemes nutzen Verfahren mit Handshake, wie bei TCP/IP.

Ein Netzwerk wird typischerweise als Graph dargestellt. Grundlegende Routing Algorithmen basieren nur auf den Kanten eines Graphen, darunter Dijkstra's single Source shortest Path [Dij59] und Floyd Warshall's all Pairs shortest Paths [Flo62] Algorithmus. Eine Spezialisierung von Dijkstra's Algorithmus ist der A* Such-Algorithmus. Bei diesem wird eine Heuristik genutzt. Es wird jeweils der Knoten, als nächster Zwischenschritt gewählt, der laut der Heuristik, den Weg zum Ziel, am weitesten verkürzt. Der implementierte Algorithmus hat Ähnlichkeit dazu. In diesem Fall, basiert die Heuristik auf geometrischen Eigenschaften, von einfachen Histogrammen. Der Unterschied, zwischen dem Routing Scheme und den genannten Algorithmen ist, dass diesem für jeden Schritt, keine globalen Informationen zur Verfügung stehen. Da Informationen fehlen, können einmal getroffene Entscheidungen, nicht rückgängig gemacht werden.

Graphen lassen sich in Graphklassen unterteilen, wobei die Kanten implizit mehr Informationen enthalten. Bei Histogrammen z.B. kann durch die Adjazenzen verschiedener Knoten, bestimmt werden, welcher Knoten einen höheren y-Wert hat. Ein Beispiel für eine Graphklasse sind Polygone mit Löchern. In dem Artikel [Ban+20], wird ein Routing Scheme, über diese Graphklasse vorgestellt. Seine Routing Tabellen haben eine Größe, von $\mathcal{O}((\varepsilon^{-1} + h) \log n)$ Bits. Dabei ist n, die Anzahl von Knoten und h die Anzahl von Löchern. Ein weiteres Beispiel, für eine Graphklasse, sind Unit Disk Graphen. Die Artikel [Kap+18] und [MW20] stellen Routing Schemes, in dieser Graphklasse, vor. Das Scheme, aus [Kap+18], hat Routing Tables, mit einer Größe von $\mathcal{O}(\varepsilon^{-5} \log^2 n \log^2 D)$ Bits, mit D dem euklidischen Durchmesser, des Unit Disk Graphen. Dabei ist die Header Größe $\mathcal{O}(\log n \log D)$ Bits. Das zweite Scheme, aus [MW20] hat keine Routing Tabellen. Die Informationen sind in den Labels der Knoten codiert. Die Größe der Labels ist $(1/\varepsilon)^{\mathcal{O}(\varepsilon^{-2})} \log D \log^3 n / \log \log n$. Alle zitierten Routing Schemes, in speziellen Graphklassen, haben einen Stretch Faktor von $< 1 + \varepsilon, \text{mit } \varepsilon > 0$. Sie sind also sehr effizient, mit einer minimalen Anzahl von Fehlern.

Ein einfaches Histogramm ist ein Polygon, bei dem eine Kante zwischen dem Knoten ganz links und dem ganz rechts, die obere Begrenzung darstellen. Jede Kante ist jeweils in einem rechten Winkel zu ihren benachbarten Kanten. Zwei Knoten sind voneinander aus r-visible, wenn ein Rechteck

zwischen den Knoten aufgespannt werden kann, ohne eine vorhandene Kante zu schneiden. Die Linien des Rechtecks sind dabei parallel zu denen des Histogramms, jeweils unterteilt in horizontal und vertikal [Chi+20].

1.1 Ziel der Arbeit

Das Ziel der Arbeit ist es, ein Programm in einer compilierbaren Programmiersprache zu schreiben. Das Programm basiert auf einem Algorithmus, aus der Doktorarbeit von Max Willert [Wil21]. Der Algorithmus ist ein Routing-Scheme in Histogrammen. Die Funktionsweise des Algorithmus wird vom Programm grafisch dargestellt. Die Daten von zufälligen Testläufen, werden in einer Datenbank gespeichert. Mit der Datenbank werden Abfragen gemacht. Das Ziel der Abfragen, auf den experimentellen Ergebnissen, mit der Implementierung, ist es, diese mit den theoretischen Ergebnissen zu vergleichen.

Das Dokument beginnt, mit den mathematischen Grundlagen. Der Rest sind die Erklärung der technischen Details und zum Schluss, die Ergebnisse der Datenbank-Abfragen.

2 Mathematische Grundlagen

In diesem Kapitel ist viel aus [Chi+20] übernommen. Ein `Graph` G besteht aus zwei Mengen V und E, G = (V,E). V enthält Knoten, E Kanten. Die Knoten haben einen Grad, $deg(v)$, mit $v \in V$. In einem ungerichteten Graphen, ist dieser, die Mächtigkeit der Menge von Kanten, die v als Endpunkt haben, $|\{\{v, w\} \in E | w \in V \setminus \{v\}\}|$.

Ein `Polygon` P ist eine geometrische Figur. Die betrachteten sind im 2D Raum. Sie bestehen aus einer Menge an Knoten V(P). Die Knoten werden entgegen dem Uhrzeigersinn nummeriert. Knoten, mit aufeinander folgenden Indizes, sind mit einer Linie verbunden, sowie der erste und letzte. Die Winkel zwischen benachbarten Linien betragen nie 180°. Als Notation wird verwendet: v_x, v_y für jeweils die x- bzw. y-Koordinate und v_{id} für die Knoten Nummer. Sei P ein zwei dimensionales orthogonales Polygon, wobei P an den Achsen des Koordinatensystems ausgerichtet ist. Dann ist die Menge der Knoten $V(P)$, und die Anzahl von Knoten $|V(P)| = n$. Die Punkte sind immer an den Endpunkten von Linien, es gibt also nicht 3 Knoten, in $V(P)$, auf derselben Linie. Die Knoten sind, entgegen dem Uhrzeigersinn, von 0 bis $n - 1$, nummeriert.

Die verwendete `r-visibility` bedeutet folgendes. Zwei Punkte $p, q \in P$ können einander sehen, was auch als `co-visible` bezeichnet wird, genau dann wenn das, ebenfalls an den Achsen des Koordinatensystems ausgerichtete, Rechteck, dass von p und q aufgespannt wird, innerhalb von P, liegt. Ein `Sichtbarkeits-Graph` Vis(P) = (V(P), E(P)), hat eine Kante zwischen zwei Knoten $v, w \in V(P)$, genau dann wenn v und w voneinander aus sichtbar sind.

`Histogramme` sind x-monotone orthogonale Polygone, bei denen die obere Grenze, aus genau einer horizontalen Kante besteht. Die Endpunkte, dieser Kante, haben die Nummern 0 und n-1. Sie werden Base-Vertices genannt. Ein randomisierter Histogramm Generator, für alle möglichen Sichtbarkeitsgraphen einer Knotenanzahl, ist in Section 4.2, in Listing 2.

Ein `Routing Scheme` enthält viele Bit-Strings. Jeder Knoten $v \in V$ erhält ein Label $lab(v) \in \{0, 1\}^*$, eine Routing Table $p(v) \in \{0, 1\}^*$ und eine Routing Funktion $f : (\{0, 1\}^*)^2 \to V \times \{0, 1\}^*$. Je nach Algorithmus lässt sich, die Größe und Anzahl, der Bitstrings reduzieren. Ob ein Routing-Scheme für eine Anwendung geeignet ist, lässt sich unter anderem an folgendem erkennen.

$$\mathrm{Lab}(n) = \max_{\substack{(V,E)\in\mathcal{G} \\ |V|=n}} \max_{v\in V} |\mathrm{lab}(v)|$$

$$\mathrm{Tab}(n) = \max_{\substack{(V,E)\in\mathcal{G} \\ |V|=n}} \max_{v\in V} |p(v)|$$

Mit $\mathcal{G}$ einer Graph-Klasse, für die das Routing Scheme geeignet ist. Für das implementierte Routing Scheme sind alle geeignet, die ein Sichtbarkeits-Graph, eines beliebigen Histogramms sind.

Das Routen funktioniert folgendermaßen. Seien $p_0, t \in V$ und $p_{i+1} = f(p(p_i), \mathrm{lab}(t))$, mit $i, k \geq 0$ und $p_k = t$. Dann ist der `Routing Pfad` $\pi : \langle p_0, ..., p_k \rangle$. Die `Routing Distanz` ist $d_p(p_0, t) = |\pi|$.

Der `Stretch` $\zeta(n)$ vergleicht die Länge des Routing Pfades zu der, des kürzesten Pfades.

$$\zeta(n) = \max_{\substack{(V,E)\in\mathcal{G} \\ |V|=n}} \max_{s\neq t\in V} \frac{d_p(s,t)}{d(s,t)}$$

3 Beschreibung und Eigenschaften des Algorithmus

Die Nachbarschaft $N(v)$, mit $v \in V$, sind alle Knoten, die von v aus sichtbar sind. Ein Knoten ist entweder konvex, die adjazenten Kanten haben einen 90° Winkel zueinander, oder reflex, mit einem 270° Winkel. Die Winkel liegen dabei innerhalb des Histogramms. Reflex Knoten sind unterteilt in l- und r-reflex, je nachdem ob sie auf der linken oder rechten Seite, ihrer horizontalen Kante sind.

In der Routing Tabelle $\mathrm{tab}(v)$ werden, zu jedem Knoten $w \in N(v)$, falls w convex ist, w_{id} und die Portnummer von w, p_w gespeichert. Ist w reflex, werden $(w_{id}, \mathrm{br}(w), p_w)$ in der Tabelle von v gespeichert. Der `Breakpoint` $\mathrm{br}(w)$, mit w, r-reflex oder dem linken Base-Vertex, ist der Knoten am linken Ende, der horizontalen Kante, mit der höchsten y-Koordinate, rechts von und unter w, der von w aus sichtbar ist [Wil21]. Für l-reflex-und den rechten Base-Vertex, ist es das gleiche, gespiegelt.

Zuletzt speichert $\mathrm{tab}(v)$ noch $\mathrm{lab}(v)$ und $\mathrm{lab}(v_{rb})$, mit v_{rb} dem rechten Base-Vertex. Für jedes Label werden $\lceil \log n \rceil$ viele Bits benötigt. Wenn man, von einer präfixfreien Kodierung, wie z.B. der von Huffman, ausgeht. Der Speicherbedarf pro Routing Table ist also, $\mathcal{O}(\log(n) \cdot (\deg(v) + 2))$.

Die exakte Routing Funktion ist in Section 4.3, in Listing 5.

4 Implementierung

4.1 Werkzeuge

Das Programm ist in C++ geschrieben [cppa] [cppb] [Str] [Gre21]. Bei der Wahl der Programmiersprache war, aufgrund der Größe des Projekts, eine IDE unterstützte Sprache entscheidend. Da C++ nicht im Rahmenplan des Studiums enthalten ist und eine wichtige Sprache ist, wurde sie ausgewählt. Die grafische Oberfläche wird, mit dem QT-Framework erstellt [Qt-] [Coe21]. Als IDE wird QT-Creator(Community Edition) verwendet. Die Build Umgebung ist Cmake [Kit]. Es werden die Boost-Graph und Boost-String Bibliothek verwendet [Boo]. Die Datenbankabfragen werden mit Pythons Pandas, Numpy, Jupyter-Notebook und Matplotlib Bibliotheken gemacht [Pan]. Die Dokumentation wird mit Doxygen erstellt [Hee], die LaTex Dokumente mit Overleaf [ent].

```
1 cmake_minimum_required(VERSION 3.16)
2 project(cmake_import VERSION 1.0.0 LANGUAGES CXX)
3 set(CMAKE_INCLUDE_CURRENT_DIR ON)
4 set(CMAKE_CXX_STANDARD 17)
5 set(CMAKE_CXX_STANDARD_REQUIRED ON)
6 find_package(Qt6 REQUIRED COMPONENTS Widgets Core Gui)
7 qt_standard_project_setup()
8 set(BOOST_INCLUDE_DIRS E:/boost_1_81_0)
9 add_subdirectory(src)
```

Listing 1: CMake-File im Root Directory

Die CMake Konfiguration ist auf zwei Dateien verteilt. Eine Datei ist im Root Verzeichnis des Projektes, die zweite im `src`-Verzeichnis. Das Ausführen von CMake übernimmt die IDE. Sie stellt die Verknüpfung zum QT-Framework her.

Zeile 8 in Listing 1 muss, je nach Speicherort der `Boost-Library`, angepasst werden.

4.2 C++ Klassen

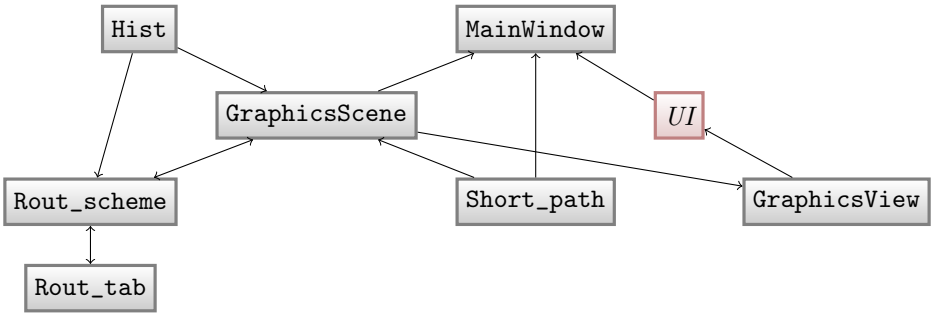

Abbildung 1: Klassen-Abhängigkeiten

In der Abbildung 1 ist das Zusammenspiel der Klassen. In der IDE gibt es eine drag und drop Schablone. Aus der Schablone wird der Quelltext zu `UI` generiert. Ein Pfeil von Klasse A nach Klasse B bedeuted, A ist eine Komponente von B, oder dass A eine Informationen, zum Initialisieren von B, zur Verfügung stellt.

Die `Hist` Klasse speichert eine Menge von Punkten, mit x- und y-Koordinaten. Die Punkte sind die Knoten des Histogramms.

In Listing 2 ist ein Zufallsgenerator für Histogramme. Die Funktion hat einen Parameter `size`. Es wird size-mal, ein Wert von 1 bis einschließlich size gewürfelt. In der anschließenden Schleife, werden die 2 Rand-Knoten, einer horizontalen Kante, des Histogramms, in das Return-Array gespeichert.

Die Hilfsfunktion `visibility_point_helper`, in Listing 3, speichert eine untere Grenze `y_lim`. Die Hilfsfunktion geht die Histogramm Punkte, in eine Richtung durch. Dabei bestimmt sie alle sichtbaren Punkte, durch einen Vergleich mit y_lim.

Die Funktion, `Hist::visibility_points`, ruft die Hilfsfunktion auf. Pro Knoten, ruft sie sie jeweils, mit einem reverse- und normalen-Iterator, auf.

```
11 std::vector<QPoint> create_random_vertex_list(int size)
12 {
13     std::list<QPoint> vertex_list;
14     vertex_list.push_back(QPoint(0,0));
15     std::random_device rd;
16     std::uniform_int_distribution<int> y_distribution(1, size);
17     std::vector<int> y_vals(size);
18     for (int var = 0; var < size; ++var) {
19         do {
20             y_vals.at(var) = y_distribution(rd) * (-5);
21             } while ((var > 0) && (y_vals.at(var) == y_vals.at(var-1)));
22     }
23     for (int var = 0; var < (size); ++var) {
24         vertex_list.push_back(QPoint(vertex_list.back().x(),
25                                      y_vals.at(var)));
26         vertex_list.push_back(QPoint(vertex_list.back().x()+5,
27                                      vertex_list.back().y()));
28     }
29     vertex_list.push_back(QPoint(vertex_list.back().x(), 0));
```

Listing 2: hist.cpp Ausschnitt 1

```
54 template <typename T_it>
55 std::list<int> visibility_point_helper(T_it ngbrhd_of_id_it,
56                                        T_it last, T_it begin)
57 {
58     std::list<int> list_acc;
59     if (std::distance(ngbrhd_of_id_it,last) > 1) {
60         T_it cur_v_it = std::next(ngbrhd_of_id_it);
61         int y_lim = std::min(ngbrhd_of_id_it->y(), cur_v_it->y());
62         for (;cur_v_it != last; std::advance(cur_v_it,1)){
63             if (cur_v_it->y() < y_lim){}
64             else {
65                 list_acc.push_back(std::abs(std::distance(begin, cur_v_it)));
66                 y_lim = cur_v_it->y();
67                 if (cur_v_it->y() > ngbrhd_of_id_it->y()) {
68                     break;
```

Listing 3: hist.cpp Ausschnitt 2

```
118 const std::map<int, std::list<int>>::const_iterator
119         Rout_tab::find_nd(int t) const
120 {
121     if (mSelf_id < t) {
122         return std::next(mEntries.lower_bound(t), -1);
123     } else {
124         return mEntries.upper_bound(t);
125     }
126 }
```

Listing 4: rout_tab.cpp Ausschnitt 1

4.3 `Rout_tab` und `Rout_Scheme` Klasse

Die `Rout_tab` Klasse ist eine Einheit des Routing-Netzwerks. Jede Einheit kann unabhängig einen Routing-Schritt berechnen, wenn der Zielknoten gegeben ist. In einem balancierten Suchbaum werden Einträge gespeichert. Jeder benachbarte Knoten hat einen Eintrag, mit ID, Port-Nummer und falls vorhanden, seinem Breakpoint. Die Initialisierung der Objekte wird von `Rout_Scheme` gemacht. Der Konstruktor von `Rout_tab` erstellt keine funktionsfähigen Objekte.

In Listing 4 wird der near-dominator, mit Binärsuche berechnet. Die Funktion für den far-dominator, ist das gleiche, gespiegelt. Der `lower_bound` ist hierbei, das gleiche wie `upper_bound`, da der Knoten `t`, keinen Eintrag haben sollte. Das ist so, weil die Bedingung, vor dem Aufruf der Funktion, in der Routing Funktion, überprüft wird.

In Listing 5 ist der eigentliche Routing Algorithmus. Es wird zu der ID des Zielknotens, noch ein Integer returned. Dieser wird nicht für den Routing-Algorithmus benötigt. Die Menge an Funktionen, `find_`$[l, r, br]$ geben Integer zurück, `find_`$[nd, fd]$ einen Iterator auf einen Map-Eintrag.

Die `Rout_scheme` Klasse ist das Routing-Netzwerk. Es enthält eine Menge von `Rout_tabs`. Die weiteren Member-Variablen sind für die Datenbank.

In Listing 6 werden die Breakpoints bestimmt. Das Verfahren wurde aus dem Pseudocode, der Dissertation, übernommen [Wil21].

Das Verfahren, zum Bestimmen von `l(v)` und `r(v)`, wurde nicht übernommen. Die Routing-Tables sind RB-Bäume, mit der Knoten ID als Index. Die Minimum und Maximum Knoten sind l(v) und r(v), sie können in $\mathcal{O}(1)$ bestimmt werden. Diese Datenstruktur wurde gewählt, weil sie in der C++ Standard Library ist und um die Laufzeit von $\mathcal{O}(\log(\deg(v))$, für die Routingfunktion zu erreichen.

Die Preprocessing Zeit berechnet sich damit wie folgt. Sei m = Anzahl von Kanten in Vis(P), die Laufzeit vom Erstellen der Routing Table ist in $\mathcal{O}(m)$, die vom Berechnen der Breakpoints in $\mathcal{O}(n)$ [Wil21] und die vom Einfügen in den Suchbaum in $\mathcal{O}(n \log n)$. Die Obere Grenze wird von dem Maximum, aus dem Erstellen der Tabellen oder dem Einfügen in den Suchbaum bestimmt.

4.4 Weitere C++ Klassen

Die `MainWindow` Klasse ist das gesamte Fenster der GUI. Wenn ein Knopf gedrückt wird, werden die Funktionen hier gestartet.

In Listing 7 werden die Header der CSV-Dateien geschrieben. Die `hist_no` und `rout_no` Spalten, enthalten Indizes, über die die Tabellen gejoint werden. Die erste Tabelle enthält Daten über die Histogramme, die zweite über die Routen und die dritte über die Ergebnisse der Routing Funkti-

```
42  const std::tuple<int, int> Rout_tab::rout_step(int t) const
43  {
44      if (t == this->mSelf_id) {
45          return std::tuple{t, 0};
46      }
47      auto search_it = mEntries.find(t);
48      if ( search_it != mEntries.end()) {
49              return std::tuple{search_it->first, 1};
50      }
51      if (t < this->find_l_v() || this->find_r_v() < t)
52      {
53          if (this->find_l_v() == 0) {
54              return std::tuple{this->find_l_v(), 2};
55          }
56          if (this->find_r_v() == mRight_base_id) {
57              return std::tuple{this->find_r_v(), 3};
58          }
59          if (this->find_l_v() < this->mEntries.at(this->find_r_v()).front()) {
60              return std::tuple{this->find_l_v(), 4};
61          } else {
62              return std::tuple{this->find_r_v(), 5};
63          }
64      }
65      auto nd_entry = this->find_nd(t);
66      auto nd_br = nd_entry->second.front();
67      if (t < this->mSelf_id) {
68          if (nd_br <= t && t <= nd_entry->first) {
69              return std::tuple{nd_entry->first, 6};
70          } else {
71              return std::tuple{this->find_fd(t)->first, 7};
72          }
73      } else {
74          if (nd_entry->first <= t && t <= nd_br) {
75              return std::tuple{nd_entry->first, 8};
76          } else {
77              return std::tuple{this->find_fd(t)->first, 9};
```

Listing 5: rout_tab.cpp Ausschnitt 2

```
3   const std::map<int, int> Rout_scheme::find_brs() const
4   {
5       std::map<int,int> ret;
6       for (size_t var_id = 0; var_id < (mTables.size()-1); ++var_id) {
7           if (var_id % 2 == 1 ) {
8               ret.insert_or_assign(mTables.at(var_id).find_l_v(), var_id);
9           }
10      }
11      for (int var_id = (mTables.size() - 1); var_id > 0 ; --var_id) {
12          if (var_id % 2 == 0) {
13              ret.insert_or_assign(mTables.at(var_id).find_r_v(), var_id);
14          }
15      }
16      return ret;
17  }
```

Listing 6: rout_scheme.cpp Ausschnitt 1

```
90  void write_streams_headers(std::vector<std::ofstream>& streams)
91  {
92      streams.at(0)
93              << "hist_no,hist_size,entries_no,entries_max_len,hist_vs"
94              << std::endl;
95      streams.at(1)
96              << "hist_no,rout_no,rout_from,rout_to,"
97              << "dist_t,rout_to_t"
98              << std::endl;
99      streams.at(2)
100             << "hist_no,rout_no,step_no,v_id,dec_id,ngbr_count,time"
101             << std::endl;
102 }
```

Listing 7: mainwindow.cpp Ausschnitt 1

on. Die Spalte `entries_no` sind die Anzahl von Einträgen in den Routing Tabellen kombiniert und `entries_max_len` sind die Anzahl von Einträgen in der Routing Tabelle, mit den meisten Einträgen. In `hist_vs` sind die Positionen aller Knoten. In `dist_t` und `rout_to_t` sind jeweils die Ergebnisse laut Dijkstra's Algorithmus. In `v_id` sind die Ergebnisse der Routingfunktion, in `ngbr_count` die Anzahl von Nachbarn, des Knotens auf dem die Funktion aufgerufen wurde. Eigentlich sollte `ngbr_count` noch mit `time` in Relation gesetzt werden, um die erwartete Laufzeit zu bestätigen. Aufgrund von Unklarheiten, bei dem Ergebnis der Zeitmessung, wurde darauf verzichtet. Es ist auch schwierig, die Ergebnisse darzustellen, da es so viele verschiedene Returns mit unterschiedlichen Laufzeiten gibt.

In der `GraphicsScene` Klasse werden Grafik-Items gespeichert. Je nach User Input werden neue Items hinzugefügt oder entfernt. Sie hat 10 Member Variablen.

Die `Short_path` Klasse ist ein anderer Routing Algorithmus. Sie wird zum Vergleichen der Pfad-Länge verwendet. Sie benutzt die Boost Graph Bibliothek.
Der Konstruktor erstellt eine boost::adjacency_list.

4.5 Datei-Verzeichnis Struktur

Das Programm wird mit einer Ordner-Struktur versendet. Die Ordner-Namen und deren Inhalt sind wie folgt:

1. `algorithm_describtion`, Doktorarbeit zu dem Thema.
2. `build-cmake_import-Desktop_Qt_6_3_1_MinGW_64_bit-Debug`, DLL- und compilierte Dateien.
3. `data`, Datenbank in CSV-Format und das Python Query-Programm.
4. `doc`, von Doxygen generierte Dokumentation.
5. `latex`, Dateien der Ausarbeitung der Bachelor-Arbeit.
6. `licence`, GPLv3.
7. `src`, Header und .cpp Dateien.

Das Programm kann unter Windows durch Ausführen von `cmake_import.exe`, in `build-cmake_import-Desktop_Qt_6_3_1_MinGW_64_bit-Debug` gestartet werden.
Die Dokumentation kann in einem Browser, mit `./doc/html/index.html` geöffnet werden.

`Doxygen` verarbeitet Header- und cpp-Dateien, zu interaktiven html-Dateien, Beispiel Abbildung 2. Eine Liste aller Klassen wird über `Classes -> Class List` geöffnet. Die Header Dateien enthalten besonders formatierte Kommentare, die von Doxygen, in die Dokumentation integriert werden.

4.6 Gebrauchsanleitung

In der Abbildung 3, ist die grafische Nutzer Schnittstelle zu sehen.
Die Knöpfe sind jeweils den Eingabefeldern, in derselben Zeile, zugeordnet. Bei Drücken der Knöpfe:

- `rout(s,t)`, wird die erzeugte Route gezeigt. Sie ist pro Schritt, mit der Begründung beschriftet.

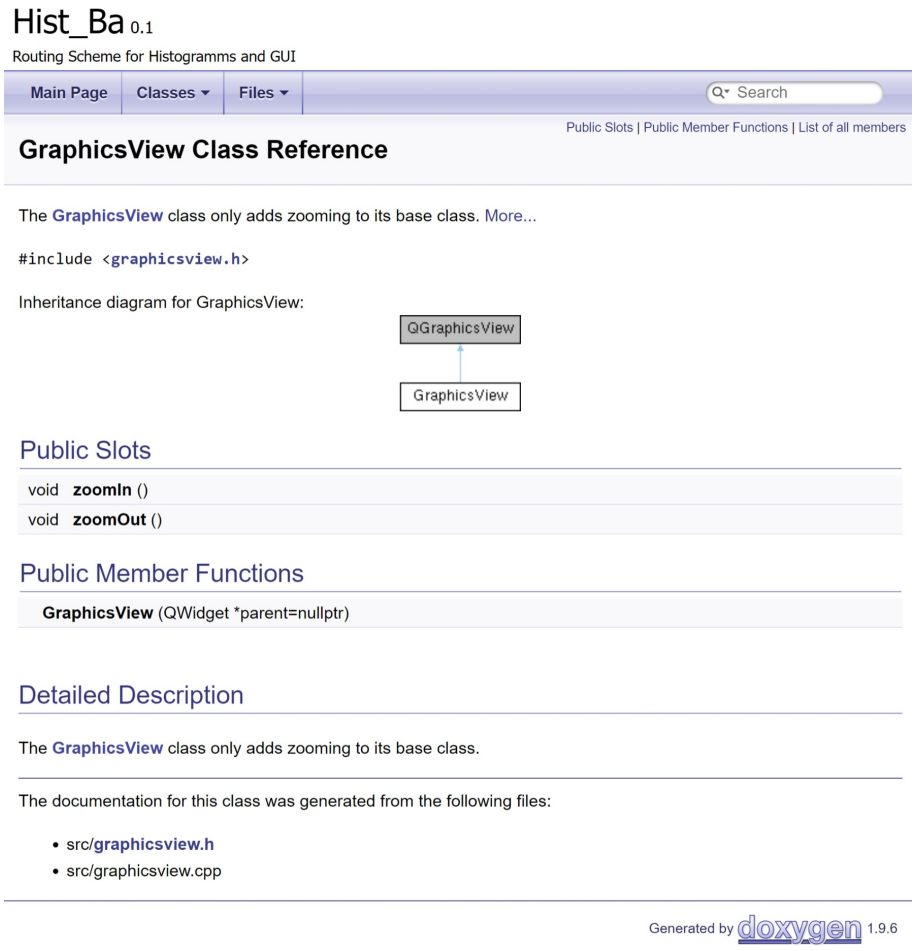

Abbildung 2: Dokumentation Beispiel Klasse

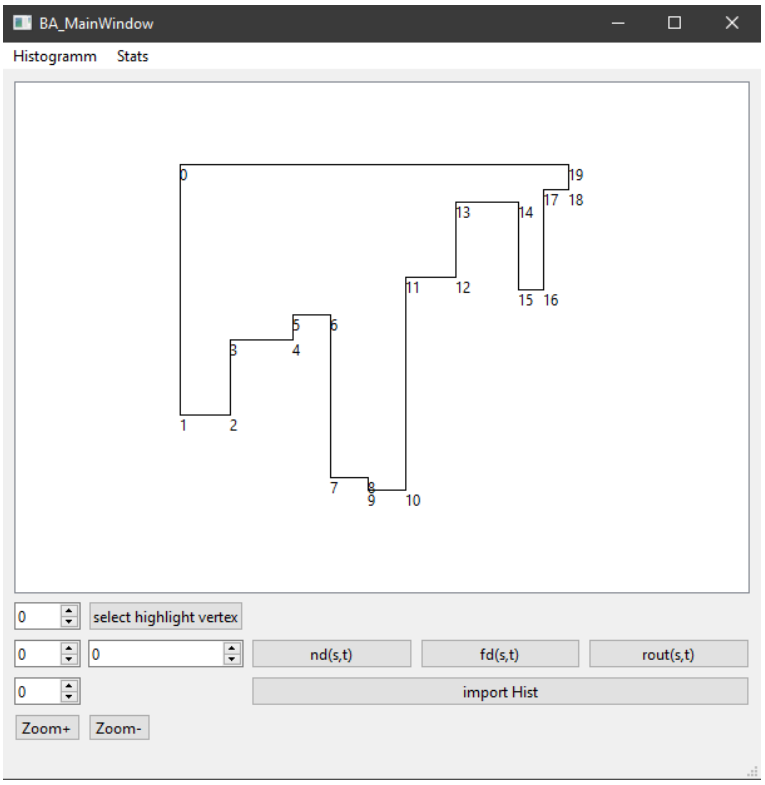

Abbildung 3: Ansicht bei Programmstart

- `import Route`, nutzt die gleichen Felder für Start- und Ziel-Knoten, wie `rout(s,t)`. Es zeigt die gleiche Route wie `rout(s,t)`, nur dass das Histogramm entfernt wird. Auch wird die Größe, der Grafik reduziert. Es lässt sich am besten, mit der `import Hist` Funktion nutzen, da dass Löschen sonst unwiderruflich ist.

- `import Hist`, bezieht sich auf die Histogramm-ID, in der Datenbank. In der Datenbank sind die Knoten Koordinaten gespeichert.

- `Zoom-/-+`, den Knopf länger gedrückt halten, zum schnellen Heran- oder Heraus-Zoomen.

Es erscheinen Schiebe-Balken am Rand, wenn nur ein Teil des Histogramms sichtbar ist. Mit den Balken kann, der sichtbare Ausschnitt verändert werden.

Durch beliebige `import Route` Aufrufe, ist erkennbar, dass Routen aus 3 Segmenten bestehen. In Segment 1 wird versucht einen Knoten zu erreichen, mit dem sich möglichst viele Knoten überspringen lassen. Es hat ein geringes Intervall entlang der x-Achse und nähert sich dem y-Wert null an. Die Distanz zwischen den y-Werten sinkt mit jedem Schritt. Segment 2 hat ein großes Intervall entlang der x-Achse und die enthaltenen Knoten haben einen ähnlichen y-Wert. Dieses Segment enthält deutlich weniger Knoten als Segment 1 oder 3. Segment 3 ist wie Segment 1, nur dass y-Werte kleiner, statt größer werden. Die Segmente kombiniert haben die Form, des kleinen Buchstaben "n".

In der oberen Leiste unter `Histogramm` gibt es die Optionen:

- `reset highlight vertex`, löscht einige Grafik Items. Es entfernt `nd, fd` und `highlight` Markierungen.

- `generate new Histogramm`, ersetzt das Histogramm. Das neue Histogramm hat die Größe, die im Eingabefeld links vom `import Hist` Knopf, eingetragen ist.

- `reset Route`, löscht auch einige Grafik Items. Es entfernt `rout(s,t)` und `optimal Route` Markierungen.

- `optimal Route` markiert die kürzeste Route. Die Route wird mit Dijkstra's SSSP Algorithmus berechnet. Die Start und Ziel Knoten werden, aus den s- und t-Feldern, von `route(s,t)` entnommen.

Auch in der oberen Leiste, unter `Stats` gibt es die Option, `gen Data`. Sie überschreibt die Datenbank. Die Parameter, der Größe der Datenbank sind in der mainwindow.cpp-Datei festgelegt.

5 Statistische Ergebnisse

5.1 Knotengröße

Die `Datenbank` wurde mit zufällig generierten Daten erstellt. Es sind jeweils 100 Histogramme, in 8 verschiedenen Größen. Die geringste Knoten Anzahl sind 5000. Die Unterschiede, zwischen den Größen, sind jeweils 5000 Knoten. In jedem Histogramm wurden 100 Routen berechnet.

Das Routing-Scheme verteilt Daten auf den Knoten. In Abbildung 4 sind zwei Kurven. Die orange zeigt den Logarithmus scaliert mit Faktor 8. Die blaue Kurve, ist die maximale Anzahl von Einträgen, in einem Knoten. Die Anzahl von Einträgen in einem Knoten ist: zwei Einträge, für jeden Knoten der Nachbarschaft und noch einen mehr, für jeden Knoten, der Nachbarschaft, der einen Breakpoint hat, $\text{num\_entries}(v) = 2 \cdot \deg(v) + |\{w \in V | \{v, w\} \in E \wedge \text{isReflex}(w)\}|$.

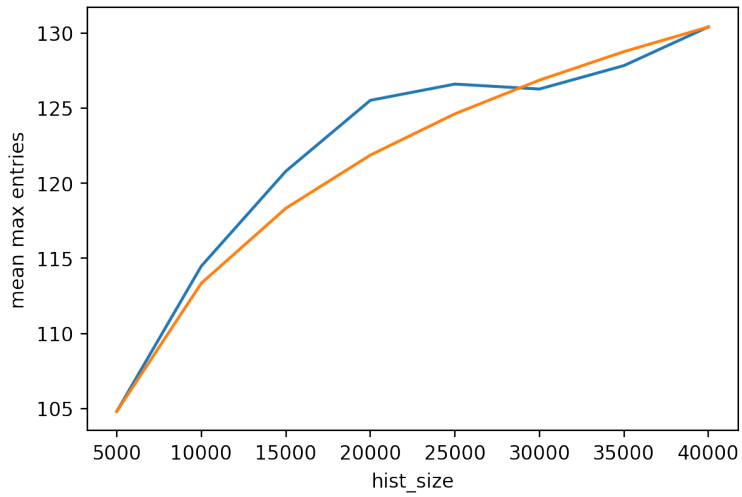

Abbildung 4: Knoten Speicherbedarf

Aus den Daten, jeder Histogramm Größe, ist der Durchschnitt berechnet. Die y-Werte der Kurve sind also: mean $_{\substack{|V|=\text{hist\_size} \\ (V,E)\in\text{Database}}}$ $\max_{v\in V}$ num_entries(v). Die Labelgröße wurde nicht integriert. Um den Speicherbedarf in Bits zu berechnen, muss noch mit log(hist_size) multipliziert werden. Bei zufälligen Histogrammen, scheint der Speicherbedarf eine logarithmische Funktion von der Anzahl an Knoten, zu sein.

5.2 Stretch

Bei der Berechnung eines Routing-Schrittes, stehen nicht alle Daten zur Verfügung. Daher ist es möglich, dass die Route nicht optimal ist. Zum Vergleichen, sind in der Datenbank auch die Ergebnisse, des Dijkstra-SSSP Algorithmus. In den überprüften Routen, wurde immer ein kürzester Weg gefunden. Der Beweis aus [Wil21], zu einem Stretch-Faktor von eins, wurde mit diesem Experiment bekräftigt.

Die durchschnittliche Distanz beträgt 10 Schritte, vom Start- zum Ziel-Knoten. Sie ist im Vergleich, zu der Anzahl von Knoten, sehr gering. Das bedeutet, dass Sichtbarkeitsgraphen von Histogrammen, nur eine kleine Graphklasse sind. Auch die maximale Pfadlänge von 23 Schritten, schließt alle Graphen, die längere kürzeste Wege enthalten aus.

Der Algorithmus findet also immer einen kürzesten Weg, jedoch sind Graphen mit längeren Wegen ausgeschlossen.

5.3 Return-ID

In dem Routing-Algorithmus gibt es 10 verschiedene Return-Möglichkeiten. In Abbildung 5 sind die Häufigkeiten, der gewählten Returns. Der Quelltext dazu, wird in Section 4.3 gezeigt, in Listing 5.

Die Abkürzung `dec_id` steht für decision ID, und ist der Name der Spalte in der Datenbank. ID eins ist der Code dafür, dass der Zielknoten Teil der Nachbarschaft ist. Es wird daher in jeder Route einmal ausgewählt.

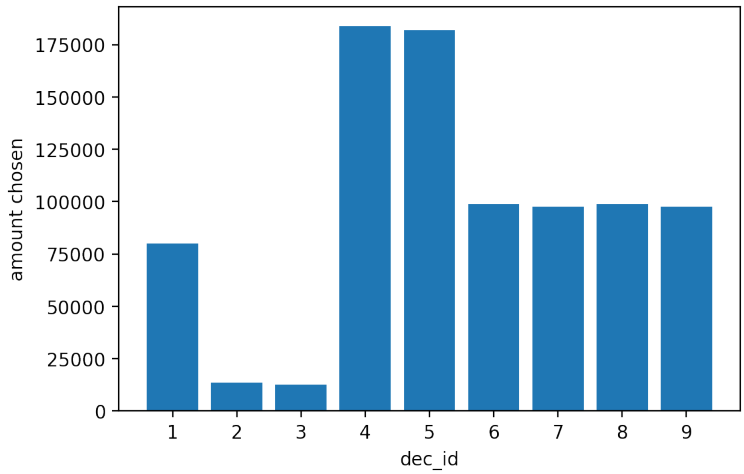

Abbildung 5: Häufigkeit von Routing Entscheidungen

IDs 2,3,4 und 5 können gewählt werden, falls der Zielknoten weiter entfernt ist, als die am weitesten entfernten Knoten der Nachbarschaft, nach links oder rechts. Dann ist der Knoten außerhalb des `Intervalls` des Knotens. IDs 2 und 3 werden ausgewählt, falls zusätzlich einer der Base-Vertices in der Nachbarschaft des Knotens sind. IDs 4 und 5 sind jeweils Sprünge zum linken oder rechten Rand der Nachbarschaft.

IDs 6 und 7 können gewählt werden, falls der Zielknoten links vom aktuellen Knoten ist, IDs 8 und 9 bei rechter Lage.

Niedrigere IDs haben eine höhere Priorität, bei der Überprüfung ob ihr Auswahl-Kriterium zutrifft. Aus der Grafik lässt sich ablesen, dass die Prioritäten besser, anders angeordnet, werden sollten. Die gesamte Nachbarschaft, auf einen Knoten, als erstes zu Überprüfen, macht keinen Sinn. Zuerst sollte überprüft werden, ob der Zielknoten im Intervall des Knotens ist. Falls das zutrifft, sollte überprüft werden ob der Zielknoten ein Benachbarter ist. Da dies ausgeschlossen werden muss, für eine Auswahl von IDs 6 bis 9.

Durch das Entfernen von IDs 2 und 3, ändert sich das Resultat nicht. Werden ihre Bedingungen nicht abgefragt, so wird die gleiche Entscheidung getroffen. ID 2 ist ein Spezialfall von ID 4, so wie 3 von 5. Die Berechnung der Bedingung von 2 und 3, ist ein bisschen weniger komplex. Sie trifft jedoch so selten zu, dass es nicht sinnvoll ist, sie extra zu überprüfen.

Die Tablle 1 enthält die Laufzeiten nach dem Pseudocode in [Wil21]. Dabei wird davon ausgegangen, dass unabhängige Berechnungen parallel stattfinden. Die Konstante c ist die Laufzeit, von Funktionen, wie std::map.begin oder der Auswertung von Vergleichsoperationen. Durch Reordering kann bei IDs 4,5 der logarithmische Summand entfallen.

6 Zusammenfassung und Ausblick

Im Rahmen dieser Bachelor-Arbeit, wurden zuerst die relevanten mathematischen Grundlagen, für das Routing Scheme in Histogrammen, erarbeitet. Anschließend wurden die Werkzeuge für das Programm gewählt. Mit diesen wurden schrittweise Klassen geschrieben und in die GUI integriert. Erst nachdem, die manuellen Tests, mit der GUI erfolgreich waren, wurde mit dem nächsten Schritt be-

Return IDs	Laufzeit
1	$\mathcal{O}(log(\deg(v))$
2,3	$\mathcal{O}(log(\deg(v)) + 4c)$
4,5	$\mathcal{O}(log(\deg(v)) + 6c)$
6,8	$\mathcal{O}(2 \cdot log(\deg(v)) + 3c)$
7,9	$\mathcal{O}(3 \cdot log(\deg(v)) + 3c)$

Tabelle 1: orginal Laufzeiten pro Return

gonnen. Ebenfalls wechselweise wurden, größer werdende, Datenbanken erstellt und diese mit Queries abgefragt. Hier gibt es auch eine Integration, der Datenbank, in die GUI, um Tests auszuführen. Die Queries, auf der Datenbank, hatten folgende statistische Ergebnisse: 1. Es wird immer ein kürzester Weg gefunden, 2. Die häufigsten Routing Entscheidungen, haben die geringste Laufzeit und 3. Die Anzahl von sichtbaren Knoten nimmt, im Vergleich zu einer steigenden Histogrammgröße, nur wenig zu.

Fragestellung, die künftig andere Menschen betrachten könnten sind: Was ist der Erwartungswert für den Speicherbedarf der Routing Tabellen, dieses Routingscheme, bei zufälligen Histogrammen, in Abhängigkeit zur Knotenanzahl? Was ist der Erwartungswert, vom Graphen-Durchmesser dieser Graphklasse? Welche Routingmuster dieser Graphklasse, lassen sich auf die Graphklasse von allgemeinen Polygonen übertragen? Die Graphklasse zu Histogrammen ist eine Teilmenge, von der der allgemeinen Polygone. Kann in allgemeinen Polygonen ebenfalls, ein kürzester Weg mit r-Sichtbarkeit gefunden werden? Bei welchen real existierenden Netzwerken lässt sich das Routing Scheme anwenden?

Beim Lesen der Arbeit, könnte der Eindruck entstehen, die Anwendung wird auf gegebenen Histogrammen beruhen. Realistischer ist es, dass ein Graph existiert und geprüft werden muss, ob dieser ein Sichtbarkeitsgraph, eines Histogramms sein kann.

Bei einem der Experimente haben sich Probleme ergeben. Eine Zeitmessung von Aufrufen der Routing Funktion, hat Ergebnisse geliefert, die mit dem Quelltext in Widerspruch standen. Das Experiment hat wahrscheinlich als Voraussetzung, dass die Optimierungstufe, des Compilers, so weit wie möglich herunter gesetzt wird. Das ist mit dem Flag -O möglich, doch dann werden alle Funktionen, mit dieser niedrigen Stufe, compiliert. Eigentlich soll nur eine Funktion genau so, wie sie geschrieben ist, übersetzt werden und nicht alle. Den ganzen Prozess zu verlangsamen ist keine gute Lösung.

Auch das Experiment zu dem Speicherbedarf sollte noch einmal, mit einer Datenbank ohne Routen und dafür mit mehr Histogrammen, wiederholt werden. Dann kann die Übereinstimmung mit einem größeren Ausschnitt der Logarithmus Kurve geprüft werden.

Bibliography

[Ban+20] Bahareh Banyassady u. a. „Routing in polygonal domains“. In: *Computational Geometry* 87 (2020). Special Issue on the 33rd European Workshop on Computational Geometry, S. 101593. ISSN: 0925-7721. DOI: https://doi.org/10.1016/j.comgeo.2019.101593. URL: https://www.sciencedirect.com/science/article/pii/S0925772119301348.

[Chi+20] Man-Kwun Chiu u. a. „Routing in Histograms“. In: *WALCOM: Algorithms and Computation*. Hrsg. von M. Sohel Rahman, Kunihiko Sadakane und Wing-Kin Sung. Cham: Springer International Publishing, 2020, S. 43–54. ISBN: 978-3-030-39881-1.

[Dij59] E.W Dijkstra. „A note on two problems in connexion with graphs“. eng. In: *Numerische Mathematik* 1.1 (1959), S. 269–271. ISSN: 0029-599X.

[Flo62] Robert Floyd. „Algorithm 97: Shortest path“. eng. In: *Communications of the ACM* 5.6 (1962), S. 345–. ISSN: 0001-0782.

[Kap+18] Haim Kaplan u. a. „Routing in Unit Disk Graphs“. In: *Algorithmica* 80 (2018), S. 830–848. DOI: https://doi.org/10.1007/s00453-017-0308-2.

[MW20] Wolfgang Mulzer und Max Willert. „Routing in Unit Disk Graphs without Dynamic Headers“. In: *CoRR* abs/2002.10841 (2020). arXiv: 2002.10841. URL: https://arxiv.org/abs/2002.10841.

[SK85] N SENTORO und R KHATIB. „Labelling and implicit routing in networks“. eng. In: *Computer journal* 28.1 (1985), S. 5–8. ISSN: 0010-4620.

[TZ01] Mikkel Thorup und Uri Zwick. „Compact Routing Schemes“. In: *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '01. Crete Island, Greece: Association for Computing Machinery, 2001, S. 1–10. ISBN: 1581134096. DOI: 10.1145/378580.378581. URL: https://doi.org/10.1145/378580.378581.

[Wil21] Max Willert. „Routing and Stabbing“. Dissertation. 2021. URL: http://dx.doi.org/10.17169/refubium-31032.

Books

[Coe21] Ben Coepp. *Introducing Qt 6: Learn to Build Fun Apps and Games for Mobile and Desktop in C++*. eng. Berkeley, CA: Apress L. P, 2021. ISBN: 9781484274897.

[Gre21] Marc Gregoire. *Professional C++ / Marc Gregoire.* eng. Fifth edition. Indianapolis, Indiana: Wrox, a Wiley brand, 2021. ISBN: 9781119695547.

Websites

[Boo] Boost-community. *Boost-Homepage*. URL: https://www.boost.org/. (accessed: 16.04.2023).

[cppa] cpp-community. *c++-reference*. URL: https://en.cppreference.com/w/. (accessed: 16.04.2023).

[cppb] cpp-community. *Tutorials C++ Language*. URL: https://cplusplus.com/doc/tutorial/. (accessed: 16.04.2023).

[ent] Overleaf enterprise. *overleaf-Homepage*. URL: https://www.overleaf.com/. (accessed: 16.04.2023).

[Hee] Dimitri van Heesch. *doxygen-Homepage*. URL: https://www.doxygen.nl/. (accessed: 16.04.2023).

[Kit] Kitware. *CMake-Homepage*. URL: https://cmake.org/overview/. (accessed: 16.04.2023).

[Pan] Pandas-community. *Pandas-Homepage*. URL: https://pandas.pydata.org/. (accessed: 16.04.2023).

[Qt-] Qt-Company/Nokia. *QT-Framework-Homepage*. URL: https://www.qt.io/. (accessed: 16.04.2023).

[Str] Bjarne Stroustrup. *Tour Standard C++*. URL: https://isocpp.org/tour. (accessed: 16.04.2023).