PAD Projekt

Dokumenation

Roman Rezvan, Branko Ribic, Ines Petrusic

# Spezifikation

## Zielsetzung

Im Rahmen des Themenkreises Nummer 4 wollen wir einen Virus-Tracker implementieren, indem wir zuerst eine Testdatenerstellung kreieren. Diese Testdatenerstellung besitzt eine Personendatenbank, die Personen mit ID, Namen, Kontaktzeit und Kontaktpersonen generiert. Dafür wird eine vorhandene Namensdatenbank verwendet. Dazu kriegt man durch eine Benutzereingabe die Möglichkeit, dass man die erste Person (die infizierte Person, Patient Null) mit Namen eingibt.

Die Auswertung der Testdaten erfolgt durch einen Algorithmus, der die Kontaktpersonen untereinander auf ID und Zeit prüfen wird. Wenn beispielsweise Person A infiziert ist und mehr als 20 Minuten mit Person B Kontakt hatte, speichert das Programm Person B als markiert ab und muss für den Benutzer deutlich machen, dass Person B auf einen Virus getestet werden sollte. Nachdem die Person B markiert wurde, wird geprüft mit wem Person B noch über 20 Minuten Kontakt hatte, um die nächste Person zu markieren.

Dieser Vorgang führt sich so lange aus, bis die letzte Verdachtsperson gefunden wurde. Man sollte am Ende den Weg, den der Virus von der ersten bis zur letzten Person (über andere Personen) zurückgelegt hat, nachvollziehen können. Der Weg zwischen den zwei Personen zeigt also alle Personen, die sehr viel Kontakt zueinander hatten (große Minutenanzahl).

Zum Ende hin möchten wir eine Visualisierung einbringen, damit der Benutzer einen visuellen Überblick über die Gesamtlage haben kann. Die Auswertung des Programms wird durch Blasen, welche Namen beinhalten, bildlich dargestellt. Damit wird gesehen, wie sich der Virus über welche Personen von Patient Null bis zum letzten Infizierten übertragen konnte.

## Risiken

* Es besteht die Möglichkeit, dass es keine verdächtigen Personen finden kann, falls die Personen nur unter der festgelegten Zeit (20 Minuten) Kontakt hatten.
* Die Algorithmen, sowohl A\*-Algorithmus als auch Dijkstra-Algorithmus sind ursprünglich dafür gedacht, dass man den schnellstmöglichen Weg zwischen A und B findet. In unserer Idee muss man diese Prinzipien etwas umschreiben, dass man, zum einen alle Verdachtspersonen mit 20+ Minuten Kontakt zum Infizierten aufnimmt und abspeichert und in der Visualisierung anzeigen kann. Zum anderen, den Weg nachverfolgen kann, bis zur letzten Person, bei der Verdacht besteht. (Start bis Ende)
* Es besteht die Gefahr, dass die Visualisierung bei einer zu hohen Anzahl an Personen unübersichtlich wird.
* Es besteht die Möglichkeit, dass durch fehlendes Fachwissen, Ziele nicht so umgesetzt werden können, wie zuvor vorgestellt. Auch das Beheben von eventuellen Bugs könnte sich als schwierig erweisen.

## Alternativen

Sollten wir bei der Visualisierung nicht mit Graphviz arbeiten können, würden wir ein anderes Tool aufsuchen, um damit die Aufgabe zu bewältigen.

# Team

An sich konnte im Team ganz gut gearbeitet werden. Da die Mitglieder des Teams nicht unbedingt die erfahrensten im Programmieren sind, stellte sich die Aufgabenstellung in manchen Bereichen doch als etwas herausfordernder heraus als Anfangs gedacht. Niemand der Teammitglieder hat vor dem Studium Erfahrungen im Programmieren sammeln können, was sich während der Umsetzung der Aufgabenstellung manchmal auch gezeigt hat. Allerdings konnte sich durch die Materie durchgearbeitet werden und so konnte man mit Zusammenarbeit auch zu einem erfolgreichen Ergebnis kommen.

## Aufteilung

* Testdatenerstellung: Ines Petrusic
* Auswertung (Dijkstra-Algorithmus): Roman Rezvan
* Graphische Darstellung (Graphviz): Branko Ribic

## Teamtreffen

**12.5.**

* erste Besprechung
* Ideen gesammelt (Virus-Tracker)

**13.5.**

* ausführlichere Besprechung über Zielsetzung
* Schreiben der Spezifikation

**14.5.**

* Spezifikation weiter ausgeführt
* Details hinzugefügt

**20.5.**

* Algorithmen-Wahl (Dijkstra)
* Besprechung über mögliche Umsetzungen der Zielsetzung
* Aufteilung der Aufgabenstellung

**29.5.**

* weitere Details über Umsetzung / Schnittstellen besprochen

**30.5.**

* erste Umsetzungen besprochen

**9.6.**

* Besprechung der bisherigen Implementierungen
* Besprechung über Schnittstellen
* weitere Implementierungen besprochen

**10.6.**

* Implementierungen besprochen

**11.6.**

* Implementierungen besprochen
* Testing
* Dokumentation

**12.6.**

* Behebung von Bugs

# Probleme und Lösungen

Während der Umsetzung unserer Zielsetzung und der Implementierung von Funktionen tauchten folgende Probleme auf:

## Testdatenerstellung

Anfangs stellte sich bei der uns zur Verfügung gestellten Namensdatenbank die Frage, wie man die Namen am Besten entnimmt und dabei Namen, die mit diversen Sonderzeichen (oder Namen, die mit unserem Alphabet nicht richtig dargestellt werden können) weglässt. Schnell stellte sich heraus, dass die Namensdatenbank eine Funktion besitzt, mit der man die Namen eines bestimmten Landes in einer Datei ausgeben kann. Für unsere Implementierungen haben wir uns also nur österreichische Namen in eine csv-Datei (*names.csv*) ausgeben lassen.

Bei der Erstellung von zufälligen Interkationen zwischen zwei Personen-ID war anfangs geplant, dass statt einer Minutenanzahl eine Start- und Endzeit (time() - seconds since epoch) generiert wird. Allerdings wurde dann doch dagegen entschieden, da wir uns für die weitere Verwendung sowieso die Minutenanzahl zwischen zwei Personen ausrechnen hätten müssen. Außerdem hätte dann noch in Bedacht gezogen müssen, dass eine Person nicht mit mehreren Kontakten gleichzeitig in Kontakt stehen hätte können. Man hätte also weiters implementieren müssen, dass sich die Zeitfenster einer Person sich nicht überlappen. Dieser Aufwand erschien dann doch als eher unnötig, da wie schon erwähnt, für die weitere Verwendung nur die Minutenanzahl zwischen zwei Personen notwendig war. (Die Variante mit Start- und Endzeit wurde zunächst quasi als Stretch Goal beiseitegelegt, dann schließlich aus zeitlichen Gründen verworfen. Man hätte am Ende, wenn man Zeit übriggehabt hätte, weitere Implementierungen einbauen können, wie z.B. von wann bis wann 2 bestimmte Personen Kontakt zueinander hatten.)

## Auswertung (Dijkstra-Algorithmus)

Die Idee hinter dem Dijkstra-Algorithmus ist, dass man den kürzesten Weg zwischen zwei Nodes, über mehrere Nodes hinweg ermittelt und sich Stück für Stück fortbewegt. Allerdings ist in unserem Themenkreis die Aufgabe gewesen, eine Kontaktanalyse (in unserem Fall einen Virustracker) zu implementieren. Es gibt im Internet viele vorgegebene Codes und Erklärungen, wie solch ein Algorithmus funktioniert. An manchen Stellen sogar ein gesamter Code. Für unseren Fall war dies eine große Herausforderung. Einen Algorithmus zu verstehen ist der erste Schritt dieser Aufgabe, jedoch bei weitem nicht ausreichend. Es ist dringend notwendig ein gewisses Mindset zu entwickeln und einen Algorithmus auf seine eigene Struktur des gesamten Codes zu adaptieren. Deswegen musste man mit Kreativität und Umstellungen eine Herangehensweise finden, um das Ziel zu erreichen.

Der erste Grundgedanke am Anfang der Aufgabe, bevor überhaupt etwas geschrieben und aktiv in CodeBlocks programmiert wurde, war dass wir die Minutenanzahl der Pärchen mit einem Exponenten von -1 versehen, damit man es einfacher hat den ‚„kürzesten Weg“ auf seine Minuten zu adaptieren. (Beispiel 50 Minuten oder 20 Minuten? 1/50 < 1/20 also geht man den Weg von 1/50 usw.)

Dies stellte sich als einen großen Fehler heraus, denn beim Schreiben des Programms wurde es sehr kompliziert mit den Datentypen von float zu arbeiten. Außerdem gab es große Rätsel bei einigen Angaben. (Es gab Werte, die ungefähr so gewertet wurden, 1/999 = 1/1000)

Wir haben uns dafür entschieden, dass wir mit jeder Verbindung die Größte herausfinden, anhand der *interactions.*csv-Datei und diese direkt in die Funktion unseres adaptierten und umgekehrten Dijkstra-Algorithmus einlesen. Dafür machten wir Matrizen und befüllten die Matrix-Koordinaten mit den jeweils, durch die Schleifen ausgewählten Personen IDs. Diese IDs der Personen gaben in Kombination in den Matrizen die richtigen Minutenwerte heraus. Alle Personen-IDs, die nicht miteinander zu tun hatten, wurden mit 0 initialisiert. In jedem der Dijkstra-Algorithmen im Internet wurden diese mit „unendlich“ initialisiert. Das wäre für uns absolut irreführend, denn wir suchen die größte Minutenanzahl heraus. Das bedeutet, dass die Personen, die sich nie getroffen haben, immer angezeigt werden als hätten sie sich unendlich viele Minuten getroffen. Aus dem Grund ließen wir die Kontaktbeziehungen, die sich nie getroffen haben, weiterhin auf 0 initialisiert.

Als nächstes gab es die Schwierigkeit, dass wir nicht zurückkehren möchten, wenn wir einst einmal von Person A nach Person B die größte Kontaktzeit ermittelt haben. Durch ein Array haben wir alle Personen zunächst mit einer 0 initialisiert. Diese bedeutet für uns, dass die Person mit dem Wert 0 dieses Arrays noch nie vom Programm abgecheckt wurde. Sollte eine Person abgecheckt werden, springt das Array an der Stelle der Personen-ID auf 1 und lässt nicht zu, dass wir jemals wieder dorthin zurückkehren werden.

Grundsätzlich gab es in jedem gegebenen Tutorial mehrere Eingaben des Users. Das wollten wir anders machen. Schließlich entscheidet man nicht wohin ein Virus als nächstes ausbreiten wird. Alles was wir brauchten ist eine Eingabe, welche uns die erste infizierte Person festlegt. Der User gibt diese Eingabe nach Abfrage des Programms ein und durch den adaptierten Algorithmus springt der Virus von Person zu Person mit der höchsten Kontaktzeit über. Das wird so lange gemacht, bis es nicht mehr weiter im Programm geht. Am Ende des ausgeführten Programms wird die letzte infizierte Person angegeben.

# Implementierung und Funktionen

## testdaten-library

### int count\_lines(FILE \* input)

Die Funktion bekommt eine Textdatei übergeben und gibt die Zeilenanzahl dieser Datei zurück. (Dabei zählt sie die Zeilenumbrüche ‚\n‘.)

### int id\_name(char \* input, char \* output, int n)

Die Funktion bekommt die Datei (*names.csv*), welche die österreichischen Namen aus der Namensdatenbank enthält, übergeben. Ebenfalls bekommt sie den Namen der Ausgabedatei (*id\_name.csv*) und die Anzahl an Personen, welche generiert werden sollen, übergeben. Die Funktion kopiert also zufällige Namen aus der Namensliste in eine neue Datei und gibt jedem Namen eine ID.

Format: *Person-ID | Name*

Falls alles funktioniert hat, liefert sie die Zahl „0“ zurück, bei einem Fehler beim Öffnen der *names.csv* liefert sie „-1“ zurück und bei einem Fehler beim Erstellen der neuen Datei *id\_name.csv* liefert sie „-2“ zurück.

### int interactions(char \* output, int n, int max\_i, int t)

Diese Funktion generiert Interaktionen in eine neue Ausgabedatei (*interactions.csv*). Sie bekommt also den Namen der neuen Datei, sowie die Personenanzahl, Anzahl der Interaktionen und der Zeitraum, in dem die Interaktionen stattfinden sollen (bzw. die maximale Zeit, die 2 Personen miteinander verbringen können) übergeben. Dabei wird jeder Interaktion auch eine Interaktions-ID zugeteilt.

Format: *Interaktions-ID | Person1-ID | Person2-ID | Minutenanzahl*

Falls alles funktioniert hat, liefert sie die Zahl „0“ zurück, bei einem Fehler beim Erstellen der Datei liefert sie „-1“ zurück.

### int search\_id(char \* input, char search\_name[])

Die Funktion bekommt eine Datei (die Datei, in der IDs und Namen stehen, *id\_name.csv*) und einen Namen (string) übergeben. Sie sucht in der Datei nach dem Namen, und falls gefunden, liefert sie die ID der Person zurück. Sonst liefert sie, wenn es einen Fehler beim Öffnen der Datei gab, „-1“ zurück oder, falls der Name nicht gefunden wurde, „-2“.

### char \*search\_name(char \* input, int search\_id)

Die Funktion bekommt eine Datei (die Datei, in der IDs und Namen stehen, *id\_name.csv*) und eine ID übergeben. Sie sucht in der Datei nach der ID, und falls gefunden, liefert sie den Namen der Person zurück. (Gegenstück zur int *search\_id* Funktion.)

## Auswertung (Dijkstra-Algorithmus)

**int dijkstra\_algo(int PPL, int contact[PPL][PPL], int patient\_null)**

Die Funktion bekommt eine Personenanzahl, ein Array und die ID der, vom User eingegebenen Person übergeben. Sie geht die *interactions.csv* Datei durch und sucht sich alle Personen raus, die am längsten Kontakt mit der vorherigen Person hatten.   
(Patient 0 -> Person A -> Person B -> Person C)

Sie printet alle Wege in eine neue Datei, *infektionsweg.csv* und liefert die Anzahl an Personen zurück.

## Graphische Darstellung (Graphviz)

**void graphviz(int node\_count, char \* input\_weg)**

Die Funktion bekommt die Anzahl der darzustellenden Nodes (Personenanzahl) und eine Datei (*interactions.csv*) übergeben. Sie entnimmt der Datei den Infektionsweg und erstellt einen Code, welcher in der Konsole ausgegeben wird. Diesen Code kann man sich auf Graphviz reinkopieren und dann bekommt man eine Abbildung des Infektionswegs zu sehen.

# Ablauf der main-Funktion

1. Personenanzahl wird abgefragt

* läuft so lange bis man eine gültige Zahl (zwischen 0 und 500) eingegeben hat
* Personen (Name & ID) werden in eine Datei gespeichert

1. die ID der Person wird gesucht (*id\_name*-Funktion)

* wird sie nicht gefunden, wird das Programm beendet

1. Anzahl an Interaktionen wird abgefragt

* läuft so lange bis man eine gültige Zahl (größer als 0) eingegeben hat
* Interaktionen werden in einer Datei gespeichert

1. Zeitraum (maximale Anzahl an Tagen), welche 2 Personen miteinander verbringen können, wird abgefragt

* läuft so lange bis man eine gültige Zahl (größer als 0) eingegeben hat

1. Name einer Person wird abgefragt (muss eine generierte Person sein)
2. Name wird der *dijkstra\_algo*-Funktion übergeben

* der Weg zwischen den Personen wird in einer Datei gespeichert
* falls die Person keine Kontakte hat, wird das Programm beendet

1. Datei wird der *graphviz*-Funktion übergeben

* Code wird auf Konsole ausgegeben, kann kopiert werden und auf Graphviz eingegeben werden

# Testing

Beim Testen der *dijkstra\_algo*-Funktion haben wir zunächst die Personenzahl immer als 100 angenommen. Da hatte allerdings die Funktion nur bei der Person mit der ID 100 Probleme gezeigt. Und zwar hat sie als Kontakte dieser Person IDs ausgegeben, obwohl sie laut *interactions.csv* gar nicht Kontakt mit diesen Personen hat. Als Minutenanzahl zwischen den Personen wurde dementsprechend auch eine Speicheradresse stattdessen ausgegeben. Nach dem Zusammenfügen aller Funktionen und libraries wurde das Ganze nochmal getestet mit maximal Zahlen über 100 und hat sich manchmal als erfolgreich, und manchmal wieder als weniger erfolgreich erwiesen.

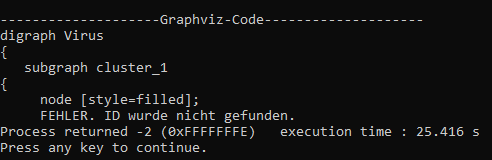


Abb. 1 fehlerhafte Ausgabe (#define people = 500)

Schließlich sind wir allerdings auf das „Problem“ gestoßen. Nach rumprobieren wurde klar, dass dieser Fehler oft passiert, wenn die Personenzahl hoch, doch die Interaktionszahl eher niedrig ist. Schließlich haben wir die ID der infizierten Person (Usereingabe -> Name) in der *interactions.csv* gesucht und sind draufgekommen, dass die Person gar keine Interaktionen zu Kontakten hat. Das Problem war also, dass wir nicht den Fall bedacht haben, dass eine Person ja auch keine Kontakte haben kann. (Schließlich sind die Kontakte ja ebenfalls zufällig generiert und es nicht gewährleistet, dass jede Person in der Liste aufscheint.) Somit haben wir also die main erweitert, mit der Möglichkeit, dass eine Person keine Kontakte hat.

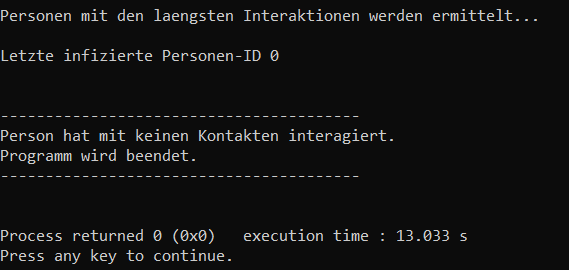


Abb. 2 Ausgabe bei Person ohne Kontakte

Weiters hatten wir eigentlich ganz am Anfang des Projekts vorgesehen, dass Kontakte, die weniger als 20 Minuten miteinander interagiert haben, nicht beachtet werden. Allerdings wurde diese Implementierung von uns doch wegelassen, da sie als eher unnötig erschien. Wir arbeiten nämlich bei der Testdatenerstellung mit Tagen und die Wahrscheinlichkeit, dass Kontakte weniger als 20 Minuten miteinander agieren, ist sehr gering. Außerdem würden solch geringe Minutenzahlen sowieso nicht beachtet werden, da unsere Funktion ja die größten Minutenzahlen raussucht.

Nachdem die einzelnen libraries erstellt wurden, musste das Programm noch das gesamte Programm getestet werden. Dabei sind allerdings Fehler aufgetaucht.

Nicht selten wurde das Programm frühzeitig abgebrochen, vor allem wenn größere Zahlen vom User eingegeben werden.

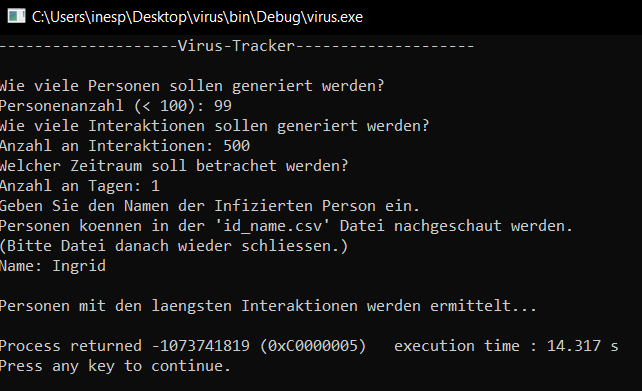


Abb. 3 Ausgabe bei frühzeitig abgebrochenem Programm

Der Abbruch schien in den meisten Fällen (wenn nicht in allen) während der Ausführung der *dijkstra\_algo*-Funktion zu geschehen. „0xC0000005“ ist ein Zugriffsverletzungsfehler, was bedeutet, dass das Programm wahrscheinlich versucht, in einem Speicherbereich zu lesen oder zu schreiben, auf den es keinen Zugriff hat. Das Programm schaffte es sich eher bei kleineren Zahlen erfolgreich auszuführen.

Bei erfolgreicher Ausführung sieht die Ausgabe folgendermaßen aus:

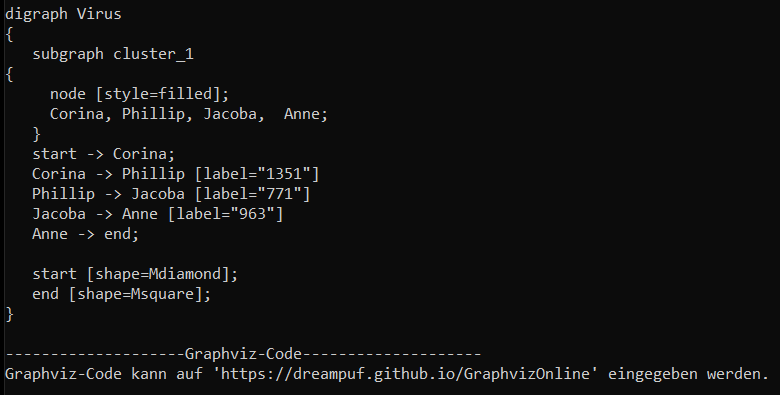


Abb. 4 Ausgabe bei erfolgreicher Ausführung

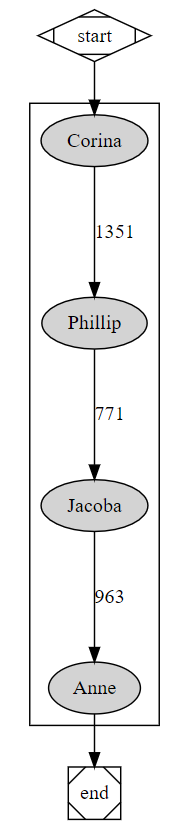


Abb. 5 Graphviz Visualisierung

Da allerdings noch keine Probleme mit einer älteren Version der *dijkstra\_algo*-Funktion auftauchten, wurde nochmal die alte Funktion hergenommen und die notwendigen Änderungen vorgenommen. Das Programm funktionierte schließlich einwandfrei. Die Vermutung, wieso die Funktion sich vorher nicht erfolgreich ausführen ließ, liegt nahe, dass beim Abändern der Funktion und löschen von Kommentaren eventuell etwas unabsichtlich verändert wurde, dass nicht hätte verändern werden sollen.

Beim Testen erwies es sich auch als nicht sonderlich einfach, einen Namen bei der Eingabe des Infizierten zu erwischen, welcher auch wirklich der Name einer generierten Person ist. Somit wurde die Funktion *id\_name* etwas abgeändert, damit diese auch alle Namen der generierten Personen ausgibt. So kann man sich schließlich bei der Usereingabe einen dieser Namen aussuchen.

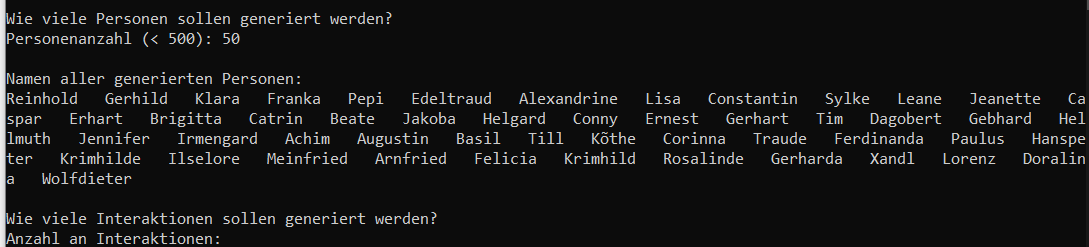


Abb. 6 Ausgabe aller Namen der generierten Personen

# Fazit

Abschließend kann man sagen, dass dieses Projekt uns gezeigt, wie wichtig eine gute Zeiteinteilung bei solch aufwendigen Projekten ist. Man darf so etwas auf keinen Fall unterschätzen, da einem sonst schnell auch die Zeit davonrennt. Es wäre also von Vorteil, mit solchen Projekten frühzeitig anzufangen, damit man sich genügend vorbereiten kann, verschiedene Implementierungsmöglichkeiten austesten kann und am Ende ausreichend Zeit zum Testen und Beheben von Fehlern und Bugs hat. Eventuell hätte man mit mehr übriger Zeit auch zusätzliche Implementierungen einbringen können, welche über die Mindestaufgabenstellung hinaus gehen.

Dieses Projekt hat uns ebenfalls gezeigt, wie man selbstständig sich Wissen aneignet (z.B. Dijkstra-Algorithmus) und eben auch wie zeitintensiv so etwas sein kann. Allein um Verständnis für zum Beispiel solche Algorithmen zu gewinnen, sollte man sich genügend Zeit einplanen, um später bei der eigenen Implementierung nicht in Stress zu verfallen.

Ein Weiterer wichtiger Aspekt, den wir mit implementieren dieses Programm gelernt haben, ist das Aufsuchen von Fehlern und Bugs, und wie man diese behebt. Bei einfachen und kurzen Codes ist es oft keine Schwierigkeit, Fehler zu finden. Aber bei solchen etwas größeren Projekte, die sich aus einzelnen Komponenten zusammensetzen, ist das Herausfinden, woher ein Fehler kommt, eine größere Herausforderung.

Auch im Thema Teamarbeit wurde natürlich wieder Neues gelernt. Wie man sich gegenseitig helfen und unterstützen kann, wurde durch dieses Projekt gezeigt.