

INFORMATIcup 2022

ABFAHRT!

Team zügig zum Erfolg

*Berkehan Ünal, Han Yang, Ling-Hsuan Hsu und Lukas
Dreyer*

von der
Leibniz Universität Hannover

14. Januar 2022

Inhaltsverzeichnis

1	Glossar	4
2	Einleitung	5
2.1	Über uns	5
2.2	Vorwort	5
3	Installation	7
3.1	Linux	7
3.2	Windows	8
3.3	Dokumentation	10
4	Ausführung	11
4.1	Direkte Ausführung	11
4.1.1	Linux	11
4.1.2	Windows	11
4.2	Docker	12
5	Theoretischer Ansatz	13
5.1	Lösungsstrategie	14
5.1.1	Grundlegender Algorithmus	14
5.1.2	Einen Zug zu einem Bahnhof ordern bzw. rufen	18
5.1.3	Freigeben einer Station	19
5.1.4	Durchschnittliche Routenlänge	21
5.1.5	Verzögerung einer Fahrt	21
5.1.6	Splitten einer Passagiergruppe	21
6	Softwarearchitektur und -qualität	22
6.1	Architektur	22
6.1.1	Einheiten/Bausteine	23
6.1.2	Eingabe	25
6.1.3	Generator	25
6.1.4	Streckennetz	26
6.1.5	Algorithmus	27
6.1.6	Ausgabe/Fahrplan-Generierung	32
6.2	Ordnerstruktur	33
6.3	Testen	34
6.3.1	Teststrategien während der Entwicklung	34
6.3.1.1	Unittests	34
6.3.1.2	Automatische Systemtests	34
6.3.2	Testen der Software	35
6.4	Programmierstil	38
6.5	Wartbarkeit	39
7	Auswertung	41
7.1	Effizienz und Ausführungszeit	41
7.2	Gesamtverspätung der Passagiere	46

8 Diskussion	50
8.1 Berechnung der kürzesten Routen/Pfade	50
8.2 Priorisierung der Passagiere und Ankunftszeit	52
8.3 Freigabe nur in den Nachbar-Bahnhof	53
8.4 Dauer des Algorithmus	53
9 Ausblick	54
10 Fazit	55
11 Eigenständigkeitserklärung	56

Abbildungsverzeichnis

1	Beispielausschnitt aus der generierten Dokumentation	10
2	Flowdiagramm des Grundalgorithmus	14
3	Grundlegende Software-Architektur	22
4	Kommunikation zwischen Travel_Center, Stationlist und Linelist	29
5	Kommunikation zwischen Groups, Main und Travel_Center	31
6	Ausführungszeit	42
7	Ausführungszeit mit 5% Zügen	43
8	Ausführungszeit mit 40% Zügen	44
9	Ausführungszeit mit 90% Zügen	45
10	Gesamtverspätung generierter Eingaben	47
11	Gesamtverspätung manueller Eingaben	49

Tabellenverzeichnis

1	Dimensionen bezüglich dem Streckennetz	41
2	Dimensionen bezüglich dem Betrieb	41
3	Dimensionen bezüglich der Anzahl der Züge	41
4	Ausführungszeiten mit konstant 10 Zügen	42
5	Ausführungszeit mit 5%-Zügen	43
6	Ausführungszeit mit 40%-Zügen	44
7	Ausführungszeit mit 90%-Zügen	45
8	Dimensionen bezüglich dem Streckennetz	46
9	Dimensionen bezüglich dem Betrieb	46
10	Gesamtverspätungen nach Kapazitäten und Passagieren	47
11	Gesamtverspätungen manueller Eingaben	48

1 Glossar

Wort	Synonym	Bedeutung
Zug	Train	Ist ein Element in dem durch die Aufgabenstellung abstrahierten Modell und wird bei der Eingabe durch die vier Parameter ID, Startposition, Geschwindigkeit und Kapazität bestimmt. Dieser kann gemäß der Aufgabenstellung Passagiere transportieren.
Strecke	Line	Ist ein Element in dem durch die Aufgabenstellung abstrahierten Modell und wird bei der Eingabe durch die fünf Parameter ID, Erster-Bahnhof, Zweiter-Bahnhof, Länge und Kapazität bestimmt. Eine Strecke verbindet genau zwei Bahnhöfe miteinander.
Bahnhof	Station	Ist ein Element in dem durch die Aufgabenstellung abstrahierten Modell und wird durch die beiden Parameter ID und Kapazität bestimmt.
Passagier	Passenger	Ist ein Element in dem durch die Aufgabenstellung abstrahierten Modell und bedeutet in diesem Kontext eine Gruppe von Passagieren. Dieses Element wird bei der Eingabe durch die fünf Parameter ID, Startbahnhof, Zielbahnhof, Gruppengröße und gewünschte Ankunftszeit bestimmt.
Streckennetz	Plan	Wird durch die gesamten Bahnhöfe und Strecken beschrieben bzw. enthält diese. Entspricht der Verbindung/Zusammensetzung aller Strecken mit ihren jeweiligen Bahnhöfen in einem Verbund bzw. Graphen.

2 Einleitung

2.1 Über uns

Wir, also das Team Zügig-zum-Erfolg, besteht aus uns vier Studierenden der Leibniz Universität Hannover aus dem Bachelor-Studiengang Informatik.

2.2 Vorwort

Wer kennt es nicht? Der ICE ist mal wieder zu spät und der Anschluss-Zug ist verpasst. Oder die Züge sind zu voll und man sitzt auf den Fluren mit zahlreichen anderen Passagieren. Der Tag ist dann meistens gelaufen.

Die Bahn ist ein wichtiger Bestandteil des öffentlichen Verkehrs [12]. Im Jahr 2020 betreibt die Deutsche Bahn ein 33.399 km langes Schienennetz und 5.691 Personenbahnhöfe [3]. Im Jahr 2019 entschieden sich 2.603 Mio. Reisende für die Schiene in Deutschland und 232 Mio. Tonnen Güter wurden transportiert. Die Pünktlichkeit der Züge ist immer ein wichtiger Indikator für die Bewertung der Dienstleistungen. Der Dienst von DB Fernverkehr hat zum Beispiel eine Pünktlichkeitsquote von 81,8% im Jahr 2020, die Kundenzufriedenheit liegt bei 80,2%. Die Verbesserung der Pünktlichkeit der Züge als Maßnahme zur Steigerung der Kundenzufriedenheit hat sich in den letzten Jahren als eines der Top-Ziele der Strategie des Deutsche Bahn Konzerns entwickelt. Nach einer fünfjährigen Arbeit hat sich die Pünktlichkeit bei DB Fernverkehr von 78,9% im Jahr 2016 auf 81,8% und bei DB Cargo Deutschland von 73,4% im Jahr 2017 auf 77,6% verbessert. Insbesondere hat die Deutsche Bahn ein geschäftsfeldübergreifende Lagezentrum Pünktlichkeit im Jahr 2018 zur Sicherstellung der Pünktlichkeitsziele gegründet. Bundesweit haben die Eisenbahnen im Jahr 2020 eine Pünktlichkeit von 94,5% erreicht.

Obwohl die Pünktlichkeit der Züge seit Jahren steigt, wird die Pünktlichkeit der Deutschen Bahn immer wieder kritisiert. Im abgelaufenen Dezember 2021 hatten 91,5 Prozent der Züge eine Verspätung bis zu fünf Minuten, während 86,1 Prozent der Züge im DB Fernverkehr sich bis zu 15 Minuten verspätet haben [1]. Die Pünktlichkeit der Bahnen/Züge war im Verkehrsbereich schon immer eine große Herausforderung, die mit der (historischen) Entwicklung der Eisenbahn, sowie den Zügen überhaupt, eng verbunden war.

Theoretisch lassen sich die Gründe für Verspätungen im Schienenverkehr in planmäßige und unplanmäßige Verspätungen unterteilen [4]. Die planmäßigen Verspätungen werden als Pufferzeit in den Fahrplan eingebaut, um Konflikte mit anderem Verkehr oder anderen Zügen zu berücksichtigen. Die unplanmäßigen Verzögerungen sind hingegen stochastisch und können lediglich geschätzt werden, diese werden zum Beispiel durch die Wetterbedingungen usw. beeinflusst, welche bei extremen Ausprägungen eventuell zu mechanischen Ausfällen oder Schäden führen können. Eine Verspätung bei einem Zug kann zu Verspätungen bei anderen Zügen führen und Verspätungen bei diesen wiederum zu Verspätungen bei anderen. Das gesamte Schienennetz ist diesbezüglich also miteinander gekoppelt. Manchmal lassen sich Verspätungen jedoch nicht vermeiden, dann muss optimiert werden, also nach anderen Kriterien entschieden werden, welcher Zug pünktlich bzw. früher ankommen und welcher der Verspätung ausgesetzt sein soll. Von diesen Kriterien/Faktoren gibt es jedoch einige, soll ein Zug also nach seiner Geschwindigkeit oder der in diesem enthaltenden Passagiere bevorzugt werden? Oder nach möglichen anschließenden Fahrten mit neuen wartenden

Passagieren? Dies lässt sich nicht einfach beantworten oder berechnen. Das ist der Grund für die Schwierigkeit zur Verminderung und Beseitigung der Verspätungen für die gesamten Fahrten und Züge innerhalb einem Streckennetz.

Mit unserer Software sollen diese Probleme nun angegangen und nach einem geeigneten Verfahren/Algorithmus, zumindest ansatzweise, gelöst werden.

In dieser Ausarbeitung wird zunächst die Installation und Ausführung unserer Software erklärt. Zudem wird erklärt, wie man sich am bequemsten mit dieser auseinandersetzen kann, vor allem mit der entsprechenden Dokumentation. Danach wird unser Theoretischer Ansatz erläutert und begründet. Also wie und mit welcher Intention wir an die Problemstellung herangegangen sind und wie wir diese schlussendlich gelöst haben. Danach wird die Architektur der Software, die Qualität der Lösung und mögliche Weiterentwicklungen, auf Basis dieser, erläutert. Am Ende wird reflektiert/diskutiert, wie zufrieden wir mit unseren Ergebnissen sind und welche Verbesserungen man hätte vornehmen können.

3 Installation

Eine Entwicklungskopie des Repositories des Projekts bzw. der Software kann mit folgendem Befehl erstellt werden.

```
git clone https://github.com/Team-Zugig-zum-Erfolg/InformatiCup.git
```


Für die Nutzung unserer Software ist die Installation von der Programmiersprache Python ab der dritten Version (3.x.x) zwingend notwendig. Python-spezifische Erweiterungen oder Module muss hingegen nur für die Entwickler-Dokumentation installiert werden und werden von unserer Software auch nicht weiter benötigt. Für folgende Betriebssysteme ist für diese (recht einfache) Installation eine Anleitung gegeben.

3.1 Linux

In der Kommandozeile muss folgender Befehl ausgeführt werden:

```
sudo apt-get install python3
```

[



Achtung
Für diesen Befehl sind Administrator-Rechte notwendig!

]

Nachdem die Installation abgeschlossen wurde, kann mit folgendem Befehl die Version von Python überprüft werden:

```
python3 --version
```

Wenn die Ausgabe `Python 3.x.x` entspricht, ist die Installation erfolgreich abgeschlossen und die für die Software benötigte Version von Python wurde erfolgreich installiert.

Optional:

Um die Dokumentation der Software generieren zu können, muss die Python-Erweiterung `pdoc` installiert sein. Für die Installation muss der folgende Befehl ausgeführt werden:

```
python3 -m pip install pdoc
```

Installation von Docker

Um die Software in einem Docker-Container auszuführen (siehe 4. Ausführung), muss Docker ebenfalls installiert werden, falls noch nicht geschehen.

Für die Installation von Docker gibt es, wie auf der offiziellen Internetseite unter <https://docs.docker.com/engine/install> beschrieben, mehrere Möglichkeiten. Wir empfehlen die Installation über ein DEB-/RPM-Package, denn bei dieser Möglichkeit sind am wenigsten Schritte durchzuführen.

1. Dazu muss zuerst im Browser zu der Internetseite unter <https://docs.docker.com/engine/install/> navigiert werden.
2. Dort muss dann im nächsten Schritt das Betriebssystem ausgewählt werden.
3. Im dritten Schritt muss dann zu dem Block `Install from a package` navigiert werden.

4. Dort befindet sich, entsprechend der Beschreibung im Block, ein Link zu den verfügbaren DEB-/RPM-Packages. Nach der System-Architektur und der gewünschten Version (also z.B. stable oder test) kann das benötigte DEB-/RPM-Package ausfindig gemacht und heruntergeladen werden. DEB-Packages haben die Dateiendung .deb und RPM-Packages die Dateiendung .rpm.

Danach muss in der Kommandozeile zu dem Verzeichnis des heruntergeladenen DEB-/RPM-Packages gewechselt werden und der folgende Befehl ausgeführt werden:

Ubuntu und Debian:

```
sudo dpkg -i [Name/Pfad des DEB-Packages]
```

CentOS:

```
sudo yum install [Name/Pfad des RPM-Packages]
```

```
sudo systemctl start docker
```

Liegt ein anderes Betriebssystem vor, muss die im oberen Abschnitt genannte Internetseite aufgerufen werden und die für das Betriebssystem entsprechende Anleitung befolgt werden. Für die Befehle oben sind außerdem Administrator-Rechte notwendig.


3.2 Windows

Eine herkömmliche Installation in Windows ist über die Kommandozeile eher weniger zu empfehlen. Stattdessen wird die Installation über die graphische Benutzeroberfläche und mithilfe eines Internet-Browsers empfohlen. Dazu muss lediglich die folgende Internetseite bzw. URL aufgerufen werden:

```
https://www.python.org/downloads/
```

Dort ist dann ein Link zum Herunterladen der aktuellen Version von Python (diese entspricht der benötigten Version) aufgeführt. Über diesen wird dann eine ausführbare Installationsdatei heruntergeladen. Diese muss dann ausgeführt und den Anweisungen der Installation gefolgt werden.

[



Achtung
Die Installation benötigt Administrator-Rechte!

]

Nachdem die Installation abgeschlossen wurde, kann mit folgendem Befehl in der Kommandozeile die installierte Version von Python überprüft werden:

```
python --version
```

Optional:

Um die Dokumentation der Software generieren zu können, muss die Python-Erweiterung pdoc installiert sein. Für die Installation muss der folgende Befehl ausgeführt werden:

```
python -m pip install pdoc
```

Installation von Docker

Um die Ausführung unserer Software auf einem Windows-System in einem Docker-Container durchzuführen, muss auf diesem Windows-System die Anwendung Docker bzw. Docker Desktop installiert sein.

Für die Installation sind die folgenden Schritte notwendig:

1. Zuerst muss zu der Internetseite unter <https://docs.docker.com/desktop/windows/install/> navigiert werden.
2. Danach muss auf der geladenen Seite über die (eindeutig sichtbare) Herunterladen-Schaltfläche, die ausführbare Installationsdatei heruntergeladen und dann ausgeführt werden.
3. Danach muss den Anweisungen des Installationsvorgangs Folge geleistet werden und zum Schluss der Rechner neu gestartet werden. Dabei muss (vermutlich) ein Update-Paket für den Linux-Kernel heruntergeladen werden, den Link zum diesem wird jedoch natürlich während der Installation mitgeteilt und ebenfalls darauf hingewiesen, dass dieses installiert werden muss. Ansonsten ist Docker auf dem Windows-System nicht vollständig einsatzbereit.

[**Achtung**

Die Installation benötigt Administrator-Rechte!

]

3.3 Dokumentation

Die Dokumentation von der Software kann in dem Hauptverzeichnis des geklonten Projekts mit dem folgenden Befehl generiert werden:

Windows:

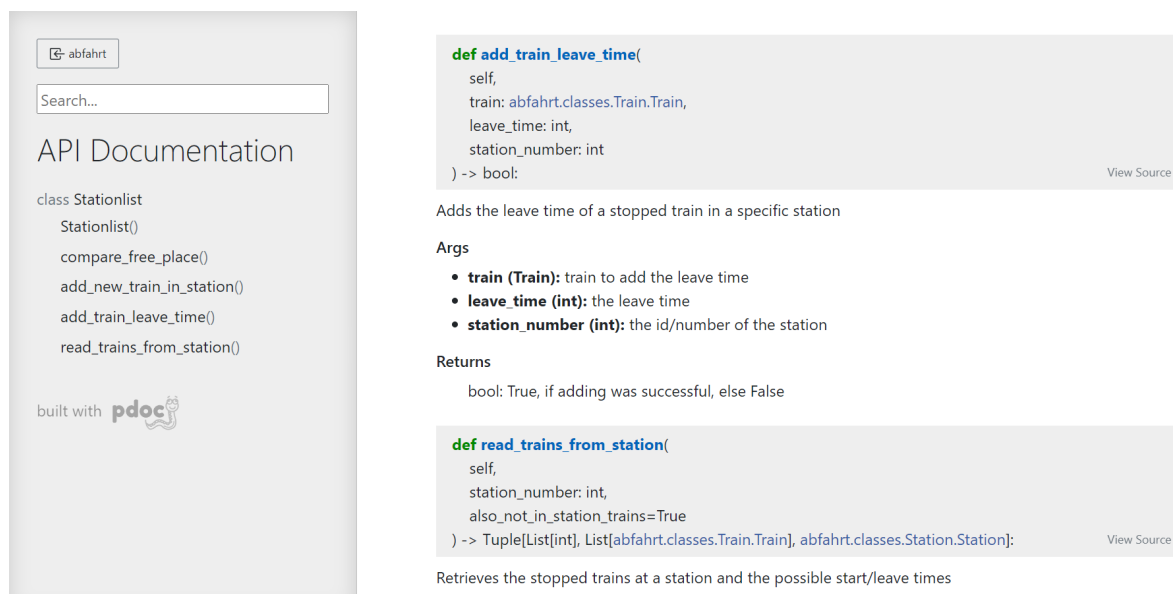
```
python -m pdoc -o ./doc -d google abfahrt
```

Linux:

```
python3 -m pdoc -o ./doc -d google abfahrt
```

In dem Unterverzeichnis `doc` befindet sich dann die generierte HTML-Dokumentation von unserer Software.

Beispielausschnitt aus der generierten Dokumentation:



The screenshot displays the generated API documentation for the 'abfahrt' project. On the left, a sidebar titled 'API Documentation' lists the methods of the 'Stationlist' class: `Stationlist()`, `compare_free_place()`, `add_new_train_in_station()`, `add_train_leave_time()`, and `read_trains_from_station()`. The main content area shows the source code for two methods: `add_train_leave_time()` and `read_trains_from_station()`. Each method is accompanied by a description, a list of arguments, and the return value. For `add_train_leave_time()`, the description is 'Adds the leave time of a stopped train in a specific station', the arguments are `train (Train)`, `leave_time (int)`, and `station_number (int)`, and the return value is `bool: True, if adding was successful, else False`. For `read_trains_from_station()`, the description is 'Retrieves the stopped trains at a station and the possible start/leave times', the arguments are `station_number: int` and `also_not_in_station_trains=True`, and the return value is `Tuple[List[int], List[abfahrt.classes.Train.Train], abfahrt.classes.Station.Station]`. The documentation is built with 'pdoc'.

Abbildung 1: Beispielausschnitt aus der generierten Dokumentation

4 Ausführung

Die Ausführung der Software kann auf zwei unterschiedliche Möglichkeiten erfolgen. Jedoch muss bei beiden die Kommandozeile (engl. Terminal) genutzt werden. Bei ersten Möglichkeit handelt es sich um die direkte Ausführung mit einem Befehl, indem unser Python-Modul einfach mit Python (der mind. dritten Version) gestartet wird. Die zweite Möglichkeit hingegen, beschreibt die Ausführung der Software in einem Docker-Container.

4.1 Direkte Ausführung

Falls noch nicht erfolgt, muss zuerst in das Hauptverzeichnis des Projekts navigiert werden. In der Kommandozeile muss dann einfach abhängig vom Betriebssystem folgender Befehl ausgeführt werden:

4.1.1 Linux

```
python3 -m abfahrt
```

Um den Inhalt einer Eingabedatei einlesen zu lassen:

```
python3 -m abfahrt < [Dateipfad]
```

4.1.2 Windows

```
python -m abfahrt
```

Um den Inhalt einer Eingabedatei einlesen zu lassen:

```
python -m abfahrt < [Dateipfad]
```

4.2 Docker

Zuerst muss mittels der Anwendung Docker ein Abbild (engl. Image) von der Software usw. erstellt werden. Dazu muss in dem Hauptverzeichnis des geklonten Projekts (engl. Repository) zuerst folgender Befehl ausgeführt werden:

```
docker build -t abfahrt .
```

Nun wurde ein Abbild (engl. Image) mit dem Namen `abfahrt` gebaut. Um nun die Software in einem Docker-Container auszuführen, muss nochmals anschließend der folgende Befehl ausgeführt werden:

```
docker run --rm -i --name container_abfahrt abfahrt [< input.txt]
```

Mit den Argumenten `--rm` und `-i` wird die korrekte Ausführung gewährleistet und deshalb sollten diese auch nicht fehlen. Mit `--rm` wird einerseits sichergestellt, dass der Docker-Container nach der Ausführung wieder gelöscht wird und der Name `container_abfahrt` wieder freigegeben wird, andererseits mit dem Argument `-i` dafür gesorgt, dass die Eingabe über die Standardeingabe eingelesen werden kann.



Achtung

Die Ausführung in einem Docker-Container erfordert zusätzlich die Installation von Docker, was standardmäßig nicht der Fall ist (siehe 3. Installation).

5 Theoretischer Ansatz

Um bei Beginn der Analyse des Problems bzw. der Aufgabenstellung überhaupt erst einmal einen ersten Ansatz für einen Algorithmus zu entwickeln, welcher Passagiere über ein Streckennetz verteilt unter Auswahl mehrerer Züge optimal transportieren kann, haben wir zuerst versucht, das Problem hinsichtlich der Optimierung theoretisch einzugrenzen. Dabei haben wir zuerst die Parameter bzw. Eigenschaften ermittelt, welche die Optimalität von einem Fahrplan bzw. die Gesamtverspätung der Passagiere beeinflussen und für diese relevant sind. Die Geschwindigkeit oder Kapazität von einem Zug hat da z.B. einen gewissen Einfluss. Danach haben wir versucht das Problem mit anderen (klassischen) bekannten Problemen der theoretischen Informatik [11] zu vergleichen und so einzuordnen. Dabei kam dann das (umgekehrte) Rucksack-Problem in Betracht, da die Passagiere mit einer Gruppengröße wie auch einer gewünschten Ankunftszeit bestückt sind, also zwei Parameter, wobei die Gruppengröße analog zu dem Rucksack-Problem als Gewicht und die gewünschte Ankunftszeit als Wert aufgefasst werden kann, welcher im Gegensatz zum richtigen Rucksack-Problem hier minimiert statt maximiert werden muss. So könnte man die Passagiere an jedem Bahnhof entsprechend der Kapazität der dort vorhandenen Züge einteilen. Jedoch wurden wir dann mit weiteren Abhängigkeiten konfrontiert, denn die Züge haben zudem z.B. eine unterschiedliche Geschwindigkeit. Zudem musste andere Aspekte abgewogen werden, wie z.B. einen (optimaleren) Zug, welcher nicht bereits bei einem Bahnhof steht, dort zu platzieren oder nicht, denn dadurch würde man die freie Kapazität bei dem Bahnhof eventuell unnötig vermindern, sofern die Passagiere auch mit einem anderen Zug pünktlich an ihr Ziel kommen. Zudem musste bei den Ideen auch die Effizienz des Algorithmus stets beachtet werden, denn ein Algorithmus mit einer Komplexität von $O(N^N)$ könnte niemals für eine reale Software in Betracht gezogen werden, egal wie optimal oder effektiv dieser auch ist. Deshalb mussten bzw. haben wir bei unseren Ansätzen des öfteren Einschränkungen gemacht, welche eine realistische Effizienz garantieren konnten. Zudem stellte sich uns während der Entwicklung immer wieder die Frage in den Weg, worauf wir bei unserem Algorithmus-Ansatz unseren Fokus legen sollen, entweder eher auf eine zuverlässige Berechnung eines vor allem validen Fahrplans oder auf einen eher besonders optimierten Fahrplan. Wir haben uns dabei auf ersteres geeignet. Wir legten den Fokus also zuerst auf die Entwicklung von einem vor allem stabilen und zuverlässigen Algorithmus. Dieser sollte einen validen Fahrplan, zuerst unabhängig von den Ankunftszeiten, auch bei den kompliziertesten und unvorhergesehensten Eingaben mit größtmöglicher Zuverlässigkeit fehlerfrei berechnen können, denn das Ziel war ja nicht nur die Optimierung von einem Fahrplan, sondern auch die Berechnung von einem validen Fahrplan. Nachdem wir diesen weitgehend entworfen/entwickelt hatten, haben wir diesen dann, auch neben der Implementierung, optimiert.

Wir haben uns bei dem Entwurf von einem Algorithmus natürlich auch mit spezifischen wissenschaftlichen Quellen [8] [18] für ein ähnliches/gleiches Problem auseinandergesetzt und diese versucht eventuell, sofern sich dies als möglich bzw. geeignet erwies, auch in unseren Algorithmus-Entwurf miteinzubringen. Jedoch erwiesen sich diese für unseren Lösungsumfang bzw. -tiefe als entweder zu komplex oder nicht geeignet, da bei diesen nicht die gleichen Einflussfaktoren gegeben waren wie bei uns, da z.B. die Bahnhöfe des öfteren auch Kapazitäten für Passagiere hatten oder die Strecken nur von einer Seite befahren werden konnten, also gerichtet waren und nicht, wie bei uns, ungerichtet und damit von beiden Seiten befahren werden konnten.

5.1 Lösungsstrategie

5.1.1 Grundlegender Algorithmus

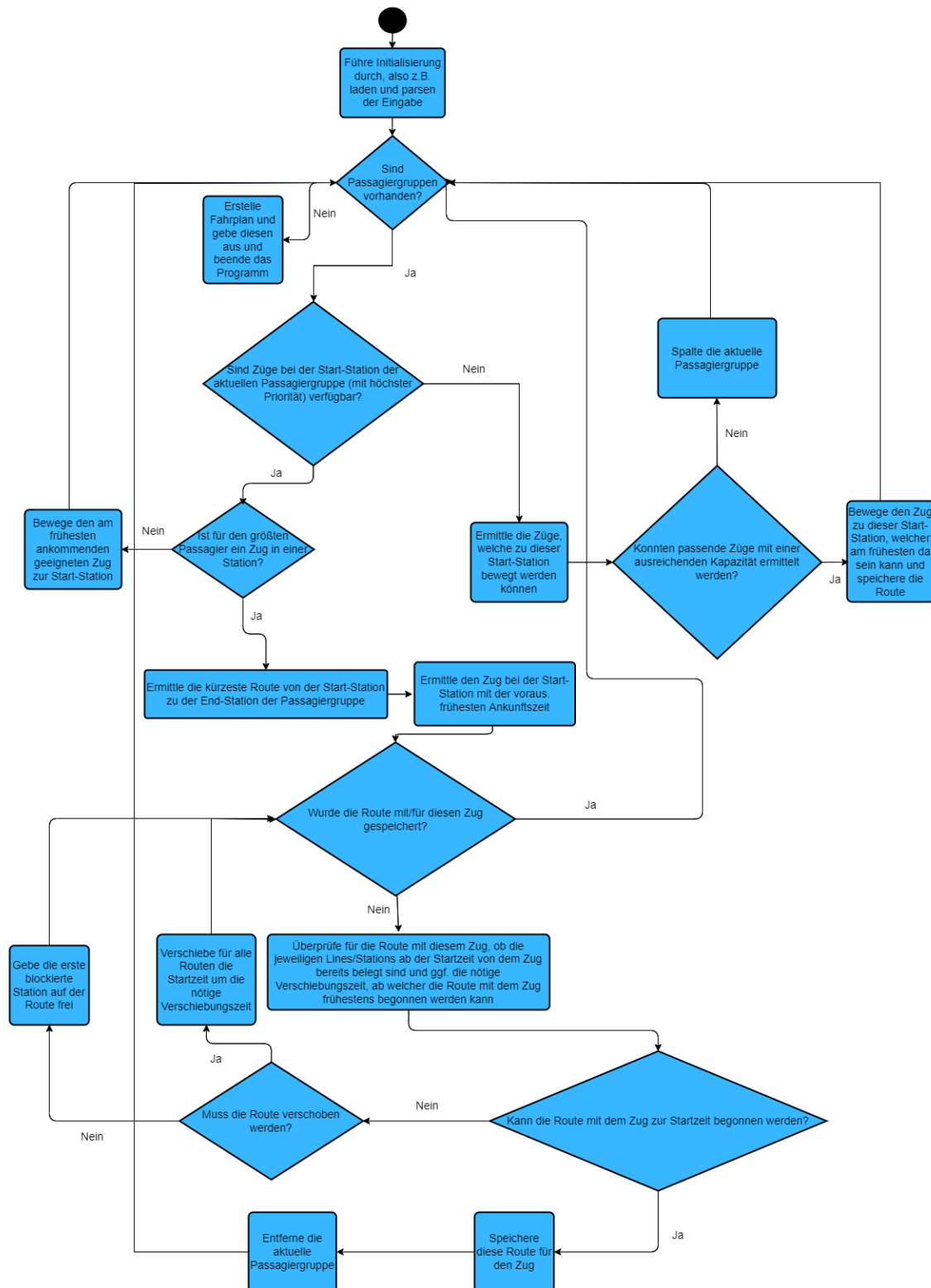


Abbildung 2: Flowdiagramm des Grundalgorithmus

Unser Algorithmusansatz für die Lösung wird durch das Fluss-Diagramm oben illustriert und hat im Detail folgenden Verlauf bzw. folgende Strategie:

Alle Passagiere werden zunächst nach ihrem Start- und Zielbahnhof gruppiert. Diese Gruppen von Passagieren enthalten dann also nur Passagiere mit dem gleichen Start- und Zielbahnhof. Diese Gruppen werden dann nach der frühesten bzw. kleinsten gewünschten Ankunftszeit bzw. (-runde) sortiert. Die kleinste gewünschte Ankunftszeit einer Gruppe ergibt sich dabei aus der kleinsten gewünschten Ankunftszeit von allen Passagieren innerhalb der Gruppe. Die Gruppe mit der kleinsten gewünschten Ankunftszeit hat dann die höchste Priorität, da die Passagiere von dieser Gruppe am dringendsten zu ihrem gemeinsamen Ziel transportiert werden müssen. Das Gruppieren der Passagiere nach gleichem Start- und Zielbahnhof hat dabei den Vorteil, dass ein Zug (mit ausreichender Kapazität für alle Passagiere der Gruppe) nicht unnötig zweimal oder öfter dieselbe Route fahren muss, wenn dieser doch auch dazu in der Lage ist, gleich alle oder zumindestens mehrere Passagiere von dem Startbahnhof mitzunehmen (vor allem dann, wenn diese dasselbe Ziel haben). Für jede Passagiergruppe wird dann entsprechend der Sortierung (in einer Schleife nacheinander, also nicht gleichzeitig) folgende Behandlung durchgeführt:

Ermittlung von geeigneten Zügen:

Es werden zuerst mögliche Züge für den Transport der Passagiergruppe mit höchster Priorität ermittelt. Dazu werden zuerst die Züge herangezogen, welche sich bereits in dem Startbahnhof befinden bzw. dort stehen geblieben bzw. dort gestoppt sind und die Züge, welche keinen definierten Startbahnhof haben und natürlich ab der initialen Runde (Runde 0) bei diesem Bahnhof platziert werden können, also der Bahnhof von Beginn an durchgehend mindestens eine freie Kapazität hat.

Gegebenenfalls Zug rufen oder Passagiergruppe splitten:

Gibt es diese Züge nicht bzw. ist die Nutzung keiner solchen Züge möglich, wird ein geeigneter Zug zu diesem Startbahnhof "gerufen" bzw. "geordert". Damit wird dieser Zug dann also zu dem entsprechenden Startbahnhof gefahren und danach die Behandlung (also aktuelle Schleife) der aktuellen Passagiergruppe von vorne begonnen, denn dann befindet sich ja nun ein geeigneter Zug mit genügend Kapazität in dem Startbahnhof. Ist jedoch auch dann (also insgesamt) kein Zug mit genügend Kapazität für die aktuelle Passagiergruppe vorhanden, muss diese Passagiergruppe gesplittet werden (siehe 5.1.6 Splitten einer Passagiergruppe).

Verklemmung vermeiden (größter Passagier braucht mind. einen Zug):

Jetzt, wo mindestens ein Zug für den Transport der aktuellen Passagiergruppe genutzt werden kann, wird vor dem Ermitteln und Speichern einer Route mit einem dieser Züge, geprüft, ob auch für den Passagier mit der größten bzw. maximalen Gruppengröße, mindestens ein Zug mit einer Kapazität größer, gleich der Gruppengröße von diesem Passagier in mindestens einem Bahnhof auf dem Streckennetz vorhanden ist. Ist dies der Fall, wird einfach fortgefahren, ist dies nicht der Fall, wird ein Zug frühestmöglich zu dem Start-Bahnhof der aktuellen Passagiergruppe gerufen oder dort platziert, welcher eine entsprechende Kapazität besitzt und von allen anderen Zügen, welche ebenfalls eine solche Kapazität besitzen, am frühesten bei dem Bahnhof ankommen würde oder dort direkt platziert werden kann. Danach wird die aktuelle Behandlung (also aktuelle Schleife) erneut von vorne begonnen.

Route ermitteln/prüfen/freigeben:

Nun wird (mit dem Dijkstra-Algorithmus) die bezüglich der Länge kürzeste Route von dem Startbahnhof zu dem Zielbahnhof ermittelt. Daraufhin wird für alle möglichen Zü-

gen unter Einbeziehung der frühestmöglichen Startzeit und der Geschwindigkeit von dem Zug (in Bezug zur Länge der Route) ermittelt, welcher Zug am ehesten am Ziel ankommen würde. Durch andere Züge blockierte Bahnhöfe oder Strecken (Lines) werden hier noch nicht miteinbezogen.

Der voraussichtlich am frühesten ankommende Zug wird dann für den Transport der aktuellen Passagiergruppe gewählt. Für diesen Zug wird dann unter anderem die voraussichtliche Abfahrtszeit ab dem Startbahnhof sowie die voraussichtlichen Belegungen der Strecken (Lines) und der Bahnhöfe auf der Route vorerst gespeichert bzw. ermittelt.

Daraufhin wird dann für diese Belegungen geprüft, ob diese erfüllt sind bzw. die Strecken (Lines) und Bahnhöfe zu diesen Zeiten bzw. Runden frei sind oder durch andere Züge bereits belegt.

Ist eine Strecke (Line) oder ein Bahnhof auf bestimmte Zeit vollständig belegt, dann wird die nötige Zeit (Verögerungszeit) ermittelt, ab welcher die Strecke (Line) bzw. der Bahnhof wieder frei bzw. wieder befahrbar ist. Diese Verzögerungszeiten werden im weiteren Verlauf alle verglichen. Ist ein Bahnhof hingegen auf unbestimmte Zeit vollständig von anderen Zügen belegt, so ist dieser blockiert und muss freigegeben werden, damit es jedoch nicht zu Problemen kommt, nachdem ein Bahnhof freigegeben wurde, da die Freigabe ja als Fahrt von einem Zug aus dem Bahnhof fest gespeichert wird, wird zuerst einmal die Zeit/Runde (wie oben, bei auf bestimmte Zeit vollständig belegte Strecken oder Bahnhöfen) ermittelt, ab welcher die Freigabe aller blockierten Bahnhöfe auf der Route (unter Einbezug gegebenenfalls blockierter Vorgänger-Bahnhöfe) erfolgen kann. (Erst bevor die Freigabe aller blockierten Bahnhöfe auf der Route reibungslos ohne Verzögerung erfolgen kann, wird der erste bzw. überhaupt irgendein Bahnhof auf der Route freigegeben). Die dafür nötigen Verzögerungszeiten (also für jeweils jeden blockierten Bahnhof) werden wie die Verögerungszeiten oben bei den auf bestimmte Zeit vollständig belegten Strecken oder Bahnhöfen behandelt.

Am Ende wird die voraussichtlich ermittelte Fahrt von dem Zug um die höchste von allen diesen Verzögerungszeiten der belegten Strecken (Lines) und Bahnhöfen verlängert bzw. verzögert (siehe 5.1.5 Verzögerung einer Fahrt).

Damit wird garantiert, dass die Strecke (Line) oder der Bahnhof mit der längsten Verzögerungszeit, auf jeden Fall befahrbar ist, denn eine geringere Verlängerung der Fahrt würde keinen Sinn machen, da genau diese Strecke (Line) oder Bahnhof dann ja immer noch vollständig belegt oder blockiert wäre. Dies wird nun solange wiederholt, bis keine Strecke (Line) oder kein Bahnhof zu der jeweiligen voraussichtlichen Zeit mehr vollständig belegt ist oder die blockierten Bahnhöfe auf der Route alle ohne Verzögerung freigegeben werden können.

Blockierte Bahnhöfe auf der Route werden von der Abfahrt an beginnend in dieser Reihenfolge freigegeben (siehe 5.1.3 Freigeben einer Station). Dies liegt daran, dass es (sollte es keine anderen freien Nachbar-Bahnhöfe geben) auch zu einem Tausch/-Wechsel von einem Zug mit dem eigentlichen Zug der Route zwischen zwei Bahnhöfen auf der Route kommen kann. Also ein blockierter Bahnhof auf der Route wird dann durch einen solchen Tausch/Wechsel mit dem eigentlichen Zug der Fahrt, kommend aus dem Vorgänger-Bahnhof auf der Route, freigegeben.

Speicherung der Fahrt:

Ist die Route einer Fahrt durch den Zug nun ohne Probleme und Einschränkungen befahrbar, wird diese Fahrt auf dieser Route fest gespeichert und ist im Nachhinein nicht

mehr änderbar. Eine Fahrt wird durch das Speichern der Belegungen der Strecken und Bahnhöfen (zu den entsprechenden Zeiten/Runden), welches sich auf der Route befinden, gespeichert. Alle nachfolgenden Fahrten müssen sich dann nach diesen fest gespeicherten Belegungen, dieser Strecken und Bahnhöfe, richten bzw. sich entsprechend ausrichten und dürfen diese natürlich nicht verändern oder ignorieren. Nun wird die gleiche Behandlung/Prozedur von Anfang an mit der nächsten Passagiergruppe durchgeführt.

5.1.2 Einen Zug zu einem Bahnhof ordern bzw. rufen

Da nicht immer ein Zug mit passender Kapazität an dem Startbahnhof von einem oder mehrerer Passagiere vorhanden ist, müssen Züge manchmal erst zu diesen hingefahren bzw. „geordert“ oder „gerufen“ werden.

Dazu werden dann zuerst von allen Zügen die Züge ermittelt, welche genügend Kapazität haben um die aktuelle Passagiergruppe zu transportieren. Dann wird mit der Anzahl der Bahnhöfe, die maximale Anzahl (das Limit) an Zügen ermittelt, für welche überhaupt eine Route berechnet werden soll bzw. welche für die weitere Nutzung in Betracht gezogen werden sollen. Ab 200 Bahnhöfen sollte z.B. nur ein Zug für die weitere Behandlung gewählt werden. Dies kann sich natürlich negativ auf die Optimierung des Fahrplans auswirken, ist jedoch für die Gewährleistung der Dauer bzw. der zeitlichen Effizienz der Software notwendig. Das Berechnen einer Route mit dem Dijkstra-Algorithmus hat eine Komplexität von $O(N^2)$, wobei N die Anzahl der Bahnhöfe darstellt. Bei 10 Bahnhöfen gibt es da noch keine (in den Dimensionen unseres Projekts) spürbare Dauer bzw. Verzögerung, welche zum Nachteil wäre. Bei 200 Bahnhöfen hingegen schon (da hat eine Berechnung bei 30 Zügen und 200 Bahnhöfen komplett ohne ein solches Limit z.B. 30-40 Minuten in Anspruch genommen). Wenn nun also für z.B. 30 geeignete Züge jedes Mal eine Route berechnet wird, dann würde sich die Dauer der Software im schlimmsten Fall (Worst-Case) verdreifachen. Bei einer durchschnittlichen Dauer von z.B. drei Minuten, mit einem Limit von einem Zug, würde die Zeit ohne ein solches Limit somit circa 90 Minuten dauern.

Wenn dann von allen möglichen Zügen also nur eine bestimmte Anzahl (= dem Limit) ausgewählt werden müssen bzw. mehr mögliche Züge vorhanden sind als das Limit vorgibt, wird wie folgt vorgegangen:

Mit der frühestmöglichen Startzeit und der Geschwindigkeit von jeweils jedem Zug, wird unter Zunahme der durchschnittlichen Routenlänge, für jeden Zug die wahrscheinliche, durchschnittliche, voraussichtliche Ankunftszeit bei dem Startbahnhof der Passagiergruppe berechnet. Die ersten unter dem Limit stehenden Züge mit dieser geringsten „Ankunftszeit“ werden dann für die weitere Behandlung genutzt, die anderen nicht weiter beachtet. Für nun jeden dieser unter diesem Limit ausgewählten Züge, wird dann (mit dem Dijkstra-Algorithmus) die schnellste Route zu dem Startbahnhof der aktuellen Passagiergruppe ermittelt. Unter Einbeziehung der Geschwindigkeit der Züge, wird dann die voraussichtliche Ankunftszeit bei dem Startbahnhof für jeden dieser Züge einzeln ermittelt. Der Zug, welcher dann am frühesten bei dem Startbahnhof ankommen würde, wird dann zu diesem gefahren. Davor wird geprüft, ob sich Passagiere bei der Start-Station von diesem Zug befinden, eine Gruppengröße kleiner oder gleich der Kapazität von diesem Zug besitzen und die gleiche Ziel-Station haben wie dieser Zug, also zu welcher der Zug gerufen werden soll. Ist dies der Fall, werden diese direkt von diesem Zug mitgenommen, da diese ja wieder aussteigen werden, bevor die eigentlichen Passagiere in diesen dann einsteigen werden. Dann wird die Route von diesem Zug gespeichert.

Davor werden natürlich, genau wie beim Grundalgorithmus oben, die Belegungen der Strecken (Lines) und Bahnhöfe auf der Route, von anderen Zügen, geprüft. Bei möglichen vollständigen Belegungen auf bestimmte Zeit, kann die Fahrt also insgesamt verzögert werden. Bei blockierten Bahnhöfen hingegen, müssen diese ebenfalls, nach dem gleichen Verfahren wie oben, erst freigegeben werden.

5.1.3 Freigeben einer Station

Aufgrund der begrenzten Kapazität von Stationen/Bahnhöfen und der Tatsache, dass ein Train/Zug (eigentlich) immer irgendwann zum Stillstand kommt und dann in einer Station/Bahnhof auf unbestimmte Zeit verbleibt, kommt es nicht selten zu blockierten Stationen/Bahnhöfen. Bei einer solchen Station kann ein Zug ab einer bestimmten Runde weder durch diese durchfahren, noch bei dieser endgültig halten, da vor bzw. ab dieser Runde die Station voll belegt ist und zusätzlich ausschließlich Züge/Trains enthält, welche dort auf unbestimmte Zeit verbleiben bzw. halten. Liegt eine solche Station auf der zuvor ermittelten Route, die ein Zug/Train befahren soll, muss diese freigegeben werden. Dazu wird einer der Züge frühestmöglich ab dem Zeitpunkt, zu dem der wartende Zug (für welchen die Route ermittelt wurde und auf welcher diese blockierte Station liegt) bei dieser Station ankommen würde, aus der Station in eine benachbarte Station gefahren. Dabei gibt es dann für die Zielstation zwei Möglichkeiten (wobei die beiden oberen Möglichkeiten der unteren vorgezogen werden):

Nachbarstation außerhalb der Route

Diese Möglichkeit wird vorrangig genutzt bzw. nur dann nicht, wenn alle benachbarten (und nicht auf der Route liegenden) Stationen blockiert sind oder die zweite Möglichkeit unten bereits eingetreten ist, also bereits eine irgendeine Nachbarstation mit mindestens zwei freien Kapazitäten gefunden wurde. In diesem Fall wird der blockierende Zug in diese Nachbarstation gefahren, welcher dort am frühesten ankommen würde, also somit nur am kürzesten zu fahren hat. Dies hat den Grund, dass dieser Zug dann somit wieder am ehesten anderen Passagieren zur Verfügung stehen kann.

Nachbarstation auf der Route mit mind. zwei freien Kapazitäten

Diese Möglichkeit wird analog zu der ersten Möglichkeit dann gewählt, wenn die erste Möglichkeit nicht schon bereits eingetreten ist oder bei allen benachbarten Stationen keine oder nur maximal eine Kapazität frei ist.

In diesem Fall kann eine solche Station gewählt werden, obwohl diese auf der Route liegt, da diese durch die Belegung einer weiteren Kapazität immer noch frei bleibt, bzw. nicht blockiert, der wartende Zug kann diese Station dann also weiterhin problemlos passieren. Es wird dann (wie in der ersten Möglichkeit) der Zug mit der frühesten Ankunftszeit, bei dieser Zielstation, gewählt.

Nachbarstation zuvor auf der Route (also von welcher der neue Zug kommt)

Diese Möglichkeit bedient sich der Tatsache, dass die Kapazität einer Station (Bahnhof) oder Line (Strecke), auch gemäß der Aufgabenstellung, während einer Runde kurzzeitig überschritten werden darf, solange am Ende der Runde alle Kapazitäten eingehalten werden bzw. nicht überschritten sind. Falls ein beliebiger Zug also in einer bestimmten Runde bei einer Station ankommt und in genau derselben Runde ein anderer Zug diese Station über dieselbe Strecke (Line) verlässt, über welche der ankommende Zug in die Station fährt, dann ist dies also (unabhängig von den Kapazitäten der Station und der Line) stets eine valide Befahrung.

Ist dieser Nachbar-Bahnhof jedoch ab bzw. bei der Ankunft von dem Zug (welcher aus dem blockierten Bahnhof gefahren wird) ebenfalls durch wiederum einen ganz an-

deren Zug blockiert, dann wird dieser Nachbarbahnhof ebenfalls freigegeben. Somit erfolgt die Freigabe von den Bahnhöfen, sofern in den Vorgänger-Bahnhof, also rekursiv. Unendlich freigegeben auf diese Weise jedoch niemals, da der Start-Bahnhof der Route ja auf jeden Fall mindestens eine freie Kapazität mindestens ab der Start-Zeit der Fahrt von dem Zug (dieser Route) haben muss, da dieser ja dort vor der Fahrt auf unbestimmte Zeit steht bzw. hält und vor dem Prüfen/Freigeben der Bahnhöfe auf der Route usw. ja natürlich auch nicht aus dieser gefahren wird. Somit ist garantiert, dass, ab dieser Zug diesen Start-Bahnhof verlassen wird, auch eine freie Kapazität zu Verfügung steht. Ist also z.B. der erste Bahnhof nach dem Start-Bahnhof auf der Route blockiert, dann wird (sofern keine anderen Nachbar-Bahnhöfe zur Verfügung stehen) in den Start-Bahnhof freigegeben und damit gleichzeitig eine neue freie Kapazität bei diesem blockierten Bahnhof geschaffen. Ist der Bahnhof danach auf der Route ebenfalls blockiert, kann also wieder in diesen (zuvor blockierten) Bahnhof freigegeben werden und die geschaffene freie Kapazität genutzt werden. So kann die Freigabe also auch quasi als Kette von aufeinander folgendem Tauschen der Züge erfolgen.

5.1.4 Durchschnittliche Routenlänge

Die durchschnittliche Routenlänge entspricht bei uns der durchschnittlichen Länge der kürzesten Routen (kürzesten Pfade) von insgesamt jedem Bahnhof (Knoten) zu jeweils jedem anderen Bahnhof (Knoten). Wir ermitteln/berechnen diese also wie folgt:

$$\text{durchschnRoutenlaenge} = \text{durchschnStreckenlaenge} * (\text{AnzBahnhoeefe} - 1) / 2$$

Diese grobe, aber dafür sehr effiziente Berechnung der durchschnittlichen Routenlänge, welche ein Zug für eine Fahrt zurücklegen muss, ergibt sich nach folgender Überlegung:

Wir nehmen für den besten Fall an, dass jeder Bahnhof (Knoten) mit jedem anderen Bahnhof (Knoten) verbunden ist, denn dann müsste jeder Zug für jede Fahrt nur eine Strecke (Kante) befahren. Die durchschnittliche Routenlänge würde dann der durchschnittlichen Streckenlänge entsprechen.

Für den schlechtesten Fall nehmen wir an, dass es genau $\text{AnzBahnhoeefe} - 1$ Strecken gibt, also jeder Bahnhof (Knoten) nur über jeweils zwei Strecken mit zwei anderen Bahnhöfen (Knoten) verbunden ist (ausgenommen von den zwei Bahnhöfen an den beiden Enden natürlich) und die Züge zusätzlich nur Passagiere von dem Bahnhof bei einem Ende zu dem Bahnhof bei dem jeweils anderen Ende transportieren müssen. Die Züge müssen dann also ausschließlich die längste Route fahren. Damit würde die durchschnittliche Routenlänge dann dem Produkt der durchschnittlichen Streckenlänge mit der um eins dekrementierten Anzahl an Bahnhöfen entsprechen.

Von diesen beiden Werten (MIN und MAX) wird dann grob der Durchschnitt gebildet bzw. ermittelt. Dies geschieht in diesem Fall durch das Halbieren, da wir annehmen, dass alle Fälle zwischen dem MIN- und MAX-Wert mit der gleichen Häufigkeit auftreten.

5.1.5 Verzögerung einer Fahrt

Soll eine Fahrt (bzw. eine Route mit einem Zug) verzögert werden, weil z.B. eine Strecke (Line) für eine bestimmte Zeit/Dauer blockiert ist, dann wird lediglich zu allen Zeiten dieser Fahrt (bzw. Route mit einem Zug) die gegebene Verzögerungszeit hinzuaddiert. Zu diesen Zeiten zählen also z.B. die Abfahrtszeiten bei den Bahnhöfen oder die Zeiten für die Belegungen der Strecken (Lines). Danach ist dann, nach dem Beispiel zuvor mit der belegten Strecke, zumindest diese Strecke (Line) für diese Fahrt frei.

5.1.6 Splitten einer Passagiergruppe

Ist eine Passagiergruppe zu groß, also hat keiner der verfügbaren Züge eine passende Kapazität für diese, wird diese gesplittet. Dabei wird zuerst geprüft, ob der kleinste Passagier in den größten Zug bei der Start-Station, der zu fahrenden Route, passt. Ist dies der Fall, werden die Passagiere der gewünschten Ankunftszeit nach aufsteigend zu einer neuen Gruppe hinzugefügt, bis dieser Zug vollständig/maximal genutzt wird. Die anderen Passagiere werden einer anderen Gruppe hinzugefügt. Ist auch der kleinste Passagier für den größten Zug bei der Start-Station zu klein, wird die Gruppe nach dem gleichen Verfahren für den überhaupt größtmöglichen Zug gesplittet. Somit wird gewährleistet, dass die Passagiere der ersten neuen Gruppe nach dem Splitten die maximal passende Gesamt-Gruppengröße für einen verfügbaren Zug besitzen.

gene Klasse. Ein Objekt der Klasse Station repräsentiert z.B. einen Bahnhof oder ein Objekt der Klasse TrainInLine die Belegung von einem bestimmten Zug in einer bestimmten Strecke (Line) zu einer bestimmten Zeit. Solche Klassen enthalten natürlich keine weiteren komplexen Funktionen oder Logik, außer gegebenenfalls Getter/Setter-Funktionen, da die Objekte dieser Klassen nur zur Repräsentierung entsprechender Einheiten (wie z.B. von einem Passagier usw.) dienen. Als Schnittstellen zwischen diesen dienen dann bestimmte Funktionen, welche natürlich in diesen auch definiert sind. Gehandhabt werden die Klassen dann hingegen nicht statisch (also ohne die Nutzung von ausschließlich statischen Funktionen in den Klassen) oder global (bzw. mit global definierten Variablen), sondern indem jeweils eine Instanz (Objekt) einer Klasse erstellt wird (dieses gegebenenfalls vorher entsprechend initialisiert wird) und danach nur dieses zur Erledigung der Aufgabe genutzt wird. Benötigt also eine andere/neue Funktion oder Klasse den Zugriff auf die Verwaltung der Bahnhöfe, kann dann einfach das Objekt der Klasse Stationlist übergeben werden. Somit könnte man theoretisch (praktisch natürlich nicht zu empfehlen) auch zwei Objekte von Stationlist miteinander zwischendurch austauschen, also z.B. wie das Rad in einem Getriebe. So können auch einzelne Bestandteile der Software einzeln genutzt oder getestet werden und die Unabhängigkeit dieser voneinander ist weitgehend gewährleistet (siehe 6.5 Wartbarkeit). Unabhängigkeit hatte bei der Entwicklung der Software sowieso einen gehobenen Stellenwert, damit auch eine besonders einfache Weiterentwicklung/Verbesserung der Software im Nachhinein möglich ist.

Bezüglich der Datenstrukturen haben wir in Python vor allem auf Listen, anstatt z.B. Dictionaries, gesetzt. Der Zugriff auf ein Element in Listen in Python, über den Index zumindestens, hat eine Komplexität von nur $O(1)$ und dies sowohl im Worst- als auch Best-Case. Bei Dictionaries hingegen beträgt der Zugriff auf ein Element nach dem Schlüssel bzw. Index im Average-Case auch $O(1)$, jedoch im Worst-Case $O(N)$ [13]. Auch bei den kleineren, einfacheren Implementierungen wurde also stets auf die Optimierung der Komplexität geachtet.

Aus folgenden Bestandteilen/Komponenten besteht die Software nun:

6.1.1 Einheiten/Bausteine

Zu den Einheiten bzw. Bausteinen gehören die folgenden Klassen:

Station

Repräsentiert einen Bahnhof und hat folgende Attribute:

- id: ID des Bahnhofes
- capacity: Kapazität des Bahnhofes

Line

Repräsentiert eine Strecke zwischen zwei Bahnhöfen und hat folgende Attribute:

- id: ID der Strecke
- start: Start-Bahnhof
- end: End-Bahnhof
- length: Länge der Line
- capacity: Kapazität der Line

Train

Repräsentiert einen Zug und hat folgende Attribute:

- id: ID des Zuges
- start_station: Start-Bahnhof
- speed: Geschwindigkeit des Zuges
- capacity: Kapazität des Zuges

Passenger

Repräsentiert einen Passagier und hat folgende Attribute:

- id: ID von dem Passagier
- start_station: Start-Bahnhof
- end_station: Ziel-Bahnhof
- group_size: Größe der Gruppe
- target_time: Gewünschte Ankunftszeit der Passagiere

TrainInStation

Repräsentiert die Belegung von einem Zug in einem Bahnhof:

- arrive_train_time: Ankunftszeit des Zuges in der Station
- passenger_in_train_time: Ankunfts-/Ausstiegszeit der Passagiere
- train: Zug von der Belegung
- leave_time: Abfahrtszeit des Zuges (die letzte Runde in dem Bahnhof)
- station_id: ID der Station/Bahnhof

In der Klasse Stationlist (siehe Stationlist) erhält jede Kapazität von einem Bahnhof eine eigene Liste mit Instanzen der Klasse TrainInStation. Möchte ein Zug einen Bahnhof belegen, werden die Listen dieser Kapazitäten von dem Bahnhof geprüft. Ist eine Belegung möglich, wird eine Instanz dieser Klasse mit den entsprechenden Attributen erstellt und in die geeignete Liste der Kapazitäten hinzugefügt. Auch sonst, wo auch immer es um die Belegung von Bahnhöfen geht, wird fast immer mit Instanzen dieser Klasse hantiert.

TrainInLine

Repräsentiert die Belegung von einem Zug auf einer Strecke:

- train: Zug von der Belegung
- start: Runde in welcher die Strecke/Line befahren wird
- end: Letzte Runde auf der Strecke/Line
- line_id: ID der Strecke/Line

In der Klasse Linelist (siehe Linelist) erhält jede Kapazität von einem Bahnhof eine eigene Liste mit Instanzen der Klasse TrainInLine. Möchte ein Zug eine Strecke belegen, werden die Listen dieser Kapazitäten von der Strecke geprüft. Ist eine Belegung möglich, wird eine Instanz dieser Klasse mit den entsprechenden Attributen erstellt und in die geeignete Liste der Kapazitäten hinzugefügt. Auch sonst, wo auch immer es um die Belegung von Strecken geht, wird fast immer mit Instanzen dieser Klasse hantiert.

Travel

Repräsentiert die Fahrt eines Zuges von einem Bahnhof zu einem anderen Bahnhof:

- start_time: Start-Runde der Fahrt
- on_board: Einstiegszeit/-runde der Passagiere
- line_time: Belegungen der Strecken
- station_time: Belegung vom Zug bei dem Ziel-Bahnhof
- start_station: Start-Bahnhof
- end_station: Ziel-Bahnhof
- train: Zug der Fahrt
- station_times: Belegungen in allen Bahnhöfen
- length: Länge der Strecke

Unabhängig davon, ob für einen Zug nur eine Route geprüft werden soll oder diese fest gespeichert werden soll, wird direkt bei bzw. nach der Ermittlung einer Route eine Fahrt (Instanz der Klasse Travel) angelegt bzw. erstellt. Diese enthält alle wichtigen bzw. notwendigen Daten für die weitere Verarbeitung der Fahrt und gegebenenfalls um diese zu speichern.

6.1.2 Eingabe

Bei der Eingabe handelt es sich um das Einlesen und Verarbeiten der Eingabe über entweder die Standard-Eingabe oder einer Eingabedatei (z.B. eine einfache TXT-Datei) mit der Eingabe als Inhalt. Diese Aufgaben übernimmt die Klasse Input. Diese stellt Funktionen bereit, mit welchen sich die Eingabe über die oben genannten Methoden einlesen, verarbeiten und dann zurückgeben lässt.

Die Klasse Input bietet unter anderem die folgenden Schnittstellen:

Einlesen der Eingabe über die Standardeingabe:

```
from_stdin()
```

Einlesen der Eingabe über eine Datei:

```
from_file(path)
```

Einlesen von einer zufälligen Eingabe über den Generator:

```
from_generator()
```

6.1.3 Generator

Um die Software automatisch Testen bzw. Nutzen zu können, dient die Klasse Generator. Diese bietet eine Schnittstelle für die Eingabe (siehe oben), um automatisch zufällig generierte Eingaben einlesen/erhalten zu können. Der Bestandteil Generator ist für die funktionalen Anforderungen unserer Software also nicht notwendig, es kann also auch ohne dem Generator ein Fahrplan generiert werden. Außerdem wird der Programmcode auch nicht standardmäßig ausgeführt, sondern nur wenn die Software entsprechend geändert wird bzw. die entsprechenden Schnittstellen zusätzlich genutzt werden.

Die Klasse Generator bietet unter anderem die folgenden Schnittstellen:

Erzeugen von einer zufälligen (aber dennoch validen) Eingabe:

```
random_input_generate(self, size_station, size_lines, size_trains,  
size_pa, sc_max, lc_max, ll_max, tc_max, pgs_max, ptr_max, max_speed_train)
```

Erzeugen von einer zufälligen (aber dennoch validen) Eingabe, wobei die Einheiten (z.B. Züge) nicht als Liste, sondern als Instanz der Klassen zurückgegeben werden:

```
random_input_generate_as_classes(self, size_station, size_lines, size_trains,  
size_pa, sc_max, lc_max, ll_max, tc_max, pgs_max, ptr_max, max_speed_train)
```

Erzeugen von einer zufälligen (aber dennoch validen) Eingabe, wobei diese direkt in eine Datei mit dem entsprechenden Namen geschrieben wird:

```
random_input_generate_file(self, file_name, size_station, size_lines,  
size_trains, size_pa, sc_max, lc_max, ll_max, tc_max, pgs_max, ptr_max,  
max_speed_train)
```

6.1.4 Streckennetz

Um möglichst effizient Routen ermitteln zu können, müssen auf dem Streckennetz eventuell Suchen oder Algorithmen durchgeführt werden. Ein Beispiel ist der Dijkstra-Algorithmus oder die Tiefen-Suche. Um diese jedoch einfach implementieren, ändern oder weiterentwickeln zu können, haben wir diese von der anderen Logik getrennt gehalten und in der Klasse Plan implementiert. Eine Instanz von dieser Klasse enthält eine Liste von Knoten, eine Liste von Kanten und eine Liste von Gewichten für jede Kante. Diese stellen dann somit einen Graphen dar. Dieser bildet dann das Streckennetz der Eingabe ab und auf diesem werden dann auch Suchen bzw. Algorithmen durchgeführt.

Die Klasse Plan bietet unter anderem die folgenden Schnittstellen:

Einen Knoten (bildet einen Bahnhof ab) hinzufügen:

```
add_node(node_id)
```

Eine Kante (bildet eine Strecke ab) hinzufügen:

```
add_edge(from_node_id, to_node_id, length)
```

Auf Basis des Dijkstra-Algorithmus den kürzesten Pfad von einem Knoten zu allen anderen Knoten berechnen bzw. die Vorgängerliste ermitteln:

```
dijkstra(start_id)
```

Mit der Vorgängerliste den kürzesten Pfad zu einem bestimmten Knoten ermitteln:

```
get_shortest_path(prev_list, target_id)
```

6.1.5 Algorithmus

Der Bereich/Bestandteil Algorithmus ist sozusagen das Herzstück bzw. der Kern unserer Software, in diesem wird die Software zentral gesteuert bzw. die eigentliche Berechnung des Fahrplans durchgeführt. Zu diesem Bestandteil gehören die folgenden Klassen:

main

Dieses Modul gibt den zentralen Algorithmus vor und stellt wiederum den Kern der gesamten Software dar und bedient sich bei den wichtigsten Klassen. Dieses beinhaltet ausschließlich die main-Funktion und bietet keine eigene Logik an. Die main-Funktion stellt also sozusagen die Steuerungseinheit dar. In dieser werden die für die gesamte Software nötigen Instanzen der Klassen Stationlist, Linelist, Input, Travel_Center, Result und Groups erstellt und initialisiert. Da wir bei unserer Implementierung auf eine weitgehende programmiertechnische Unabhängigkeit geachtet haben, ist es möglich, diese Klasse bzw. Funktion ohne das Ändern des Programmcodes auszutauschen bzw. selbst zu implementieren. Dazu muss lediglich das Python-Modul abfahrt importiert werden:

```
from abfahrt import *
```

Danach sind alle anderen benötigten Abhängigkeiten, Klassen usw. genau wie in der Klasse main verfügbar und man kann basierend auf unserer Software (wie in der main-Funktion) einen eigenen Algorithmus schreiben (siehe 6.5 Wartbarkeit).

Travel_Center

Diese Klasse beinhaltet den größten Teil der Logik der gesamten Software. Diese dient zum Verwalten/Prüfen/Erweitern des aktuellen Fahrplans, also z.B. das Ermitteln/Prüfen/Speichern einer Route für einen Zug oder das Ermitteln von geeigneten Zügen an bestimmten Bahnhöfen für eine bestimmte Passagiergruppe. Diese Klasse wird deshalb eigentlich nur von dem Teil der Software verwendet, welcher für die zentrale Steuerung der Software bzw. dem Algorithmus zuständig ist, also von der main-Funktion. Dazu greift die Klasse Travel_Center auf die ihr bei der Initialisierung übergebenen Instanzen der Klassen Stationlist, Linelist, Result und Plan zurück (diese werden also von Travel_Center zwingend benötigt). Mit den Instanzen Stationlist und Linelist werden die Strecken und Bahnhöfe verwaltet. Mit der Instanz von Plan wird bei der Initialisierung das komplette Streckennetz abgebildet bzw. ein Graph erstellt, welcher dieses abbildet. Die Längen der Strecken dienen dann als Distanz und die Bahnhöfe als Knoten und die Strecken als Kanten. Über Plan wird dann auch z.B. die kürzeste bzw. eine geeignete Route gesucht. Plan bestimmt dann also, welcher Algorithmus verwendet wird. Dies ist bei uns Dijkstra, aber möchte z.B. ein anderer Algorithmus genutzt werden, dann muss lediglich eine neue Plan-Klasse erstellt werden, natürlich mit den entsprechenden Schnittstellen. Somit wird auch hier Unabhängigkeit zwischen der Berechnung der kürzesten (geeignetsten) Route und Travel_Center bzw. dem zentralen Management geboten. Mit der Instanz von Result wird mitgeteilt, wann z.B. ein Zug abfährt oder ein Passagier aussteigt.

Die Klasse Travel_Center bietet die Logik jedoch nur als Schnittstelle und dient nicht dazu bzw. bietet keine Funktion, den Fahrplan selbstständig zu berechnen. Dies wird in der Hauptfunktion main() durchgeführt.

Die Klasse Travel_Center bietet nun unter anderem folgende wichtige Schnittstellen:

Ermitteln von Zügen in einem bestimmten Bahnhof:

```
check_train_in_station(start_station , group_size ,  
[include_not_in_station_trains])
```

Ermitteln von Zügen in allen Bahnhöfen auf dem Streckennetz:

```
check_train_not_in_station(group_size ,  
[include_not_in_station_trains])
```

Freigeben von einem bestimmten Bahnhof ab einer bestimmten Zeit/Runde:

```
clear_station(target_station , prev_station , arrive_time , stations_to_ignore ,  
[train_to_replace , original_travel])
```

Rufen von einem Zug bzw. zu einem Bahnhof ordern:

```
train_move_to_start_station(start_station , trains , start_times ,  
start_stations , groups)
```

Die kürzeste Route von mehreren Routen auswählen/prüfen/speichern:

```
determine_and_save_shortest_travel(travels , groups , passengers)
```

Eine berechnete und geprüfte Route von einem Zug speichern:

```
save_travel(travel , groups , passengers , [train_to_replace])
```

Die wohl wichtigste Kommunikation bzw. der wohl wichtigste Austausch von Travel_Center mit anderen Klassen, findet dabei mit Stationlist und Linelist statt.

Travel_Center selbst hat keine Informationen über die Bahnhöfe (Stationen) und Strecken (Lines) vorliegen, sondern ruft diese jedesmal von Stationlist und Linelist ab. Falls ein Bahnhof (Station) oder eine Strecke (Line) belegt werden sollen, wird dies ebenfalls nur über Stationlist oder Linelist getätigt. Die Trennung von der Verwaltung von den Bahnhöfen (Stationen) und Strecken (Lines) hatte bei uns das Ziel, möglichst sinnvolle und geeignete Unabhängigkeit zwischen diesen Bereichen zu schaffen, damit z.B. Stationlist bei Bedarf auch einfach neu entworfen oder ausgetauscht werden kann. Die neue Stationlist müsste dann einfach nur dieselben Schnittstellen anbieten und mehr ist dann auch nicht nötig. Die entsprechenden Funktionen für die Kommunikation bzw. das Abrufen sind unten oder in der Code-Dokumentation (siehe 3.3 Dokumentation) beschrieben.

Folgendes Diagramm veranschaulicht die Kommunikation zwischen diesen drei Klassen:

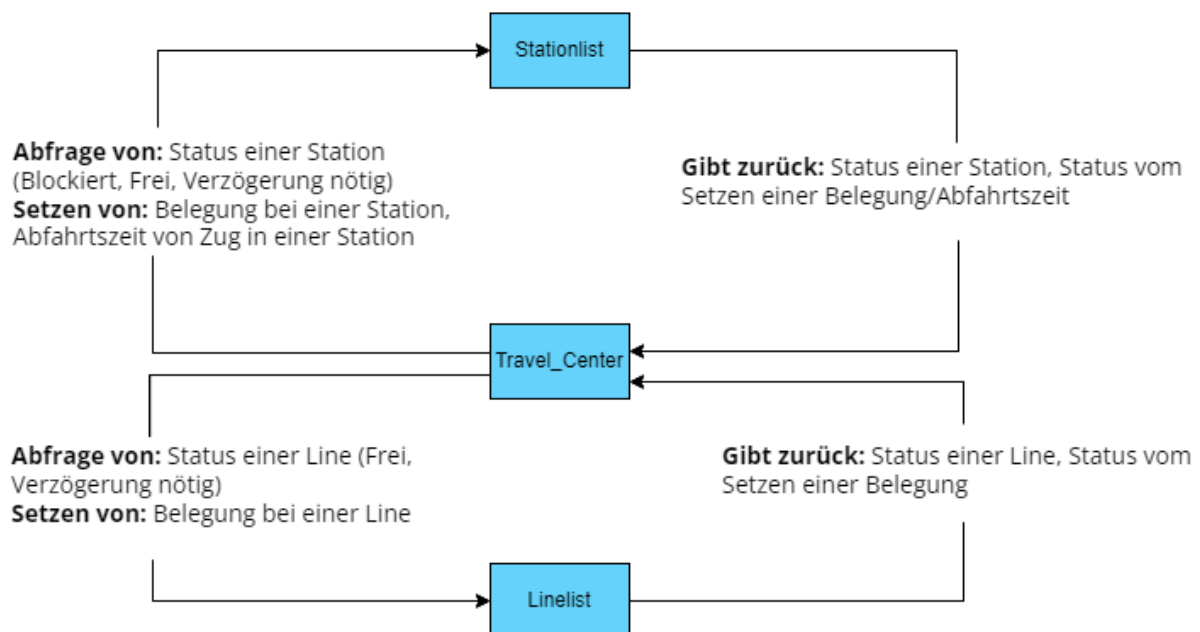


Abbildung 4: Kommunikation zwischen Travel_Center, Stationlist und Linelist

Stationlist

Diese Klasse dient dem Verwalten und Prüfen der Belegungen der Bahnhöfe. Diese besitzt eine Liste als Attribut, welche als Elemente für jeden Bahnhof wiederum eine Liste besitzt, welche dann der Kapazität des Bahnhofs entsprechend viele Listen mit TrainInStation-Instanzen besitzt. Jede Kapazität von einem Bahnhof besitzt dann also eine Liste mit Belegungen von Zügen in diesem Bahnhof.

Diese Klasse bietet unter anderem die folgenden wichtigen Schnittstellen:

Einen bestimmten Bahnhof mit einem Zug belegen:

```
add_new_train_in_station(train_in_station , [train_to_replace])
```

Eine gewünschte Belegung von einem Bahnhof mit einem Zug prüfen:

```
compare_free_place(train_in_station)
```

Einen gestoppten/gehaltenen Zug aus einem Bahnhof abfahren lassen:

```
add_train_leave_time(train , leave_time , station_number)
```

Linelist

Diese Klasse dient dem Verwalten und Prüfen der Belegungen der (einzelnen) Strecken. Diese besitzt eine Liste als Attribut, welche als Elemente für jede Strecke wiederum eine Liste besitzt, welche dann der Kapazität der Strecke entsprechend viele Listen mit TrainInLine-Instanzen besitzt. Jede Kapazität von einer Strecke besitzt dann also eine Liste mit Belegungen von Zügen in dieser Strecke. Die Klasse Linelist erfüllt also den gleichen Zweck wie die Klasse Stationlist, jedoch für die Strecken und nicht die Bahnhöfe.

Diese Klasse bietet unter anderem die folgenden wichtigen Schnittstellen:

Eine bestimmte Strecke mit einem Zug belegen:

```
add_new_train_in_line(train_in_line)
```

Eine gewünschte Belegung von einer Strecke mit einem Zug prüfen:

```
compare_free(train_in_line)
```

Groups

Diese Klasse dient zum Verwalten/Sortieren/Ordnen der Passagiere und dem Gruppieren von Passagieren mit dem gleichen Start-Bahnhof und dem gleichen Ziel-Bahnhof. Zum Speichern/Aufbewahren der Passagiergruppen wird als Datenstruktur in Python eine Liste vom Typ `list` verwendet, welche als Elemente wiederum Listen mit Passagieren besitzt. Die Sortierung der Liste mit den Passagiergruppen insgesamt, sowie der einzelnen Passagiere in den Passagiergruppen, erfolgt mit der in Python eingebauten `sort()` Funktion. Diese nutzt den Timsort-Algorithmus um die Passagiere bzw. Passagiergruppen nach der frühesten Ankunftszeit zu sortieren [7]. Dieser Algorithmus hat im Worst-Case eine Zeitkomplexität von $O(n * \log(n))$ und im Best-Case von $O(n)$ und im Average-Case von $O(n * \log(n))$ und ist damit schneller als der Merge- und Quicksort-Algorithmus [9]. Die Funktion `sort()` wird übrigens auch in `Stationlist` und `Linelist` zum sortieren der Kapazitäten verwendet.

Diese Klasse bietet unter anderem die folgenden Schnittstellen:

Die Passagiergruppe mit der höchsten Priorität abfragen/erhalten:

```
get_priority()
```

Eine Passagiergruppe entfernen, da diese beim Ziel angekommen ist:

```
passengers_arrive(group)
```

Eine Passagiergruppe splitten, da diese zu groß ist:

```
split_group(group, train_capacity_in_station, max_train_capacity)
```

Der Datenfluss und die Kommunikation mit anderen Klassen erfolgt bei `Groups` hauptsächlich mit `Main` und `Travel_Center`. Folgendes Diagramm veranschaulicht diese Kommunikation und den entsprechenden Datenfluss:

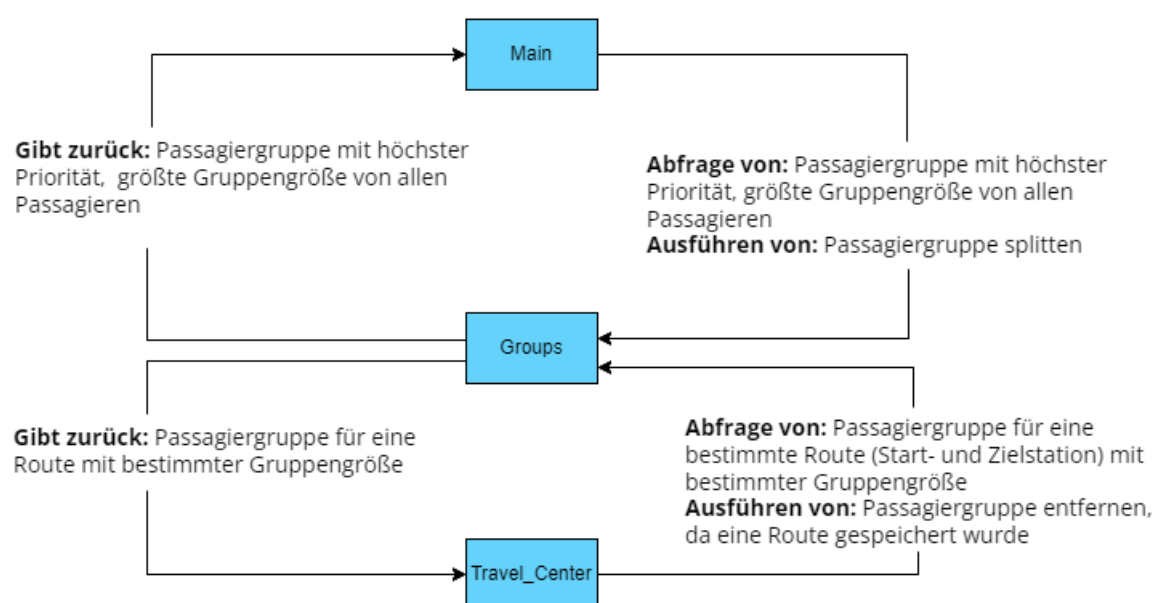


Abbildung 5: Kommunikation zwischen `Groups`, `Main` und `Travel_Center`

6.1.6 Ausgabe/Fahrplan-Generierung

Alles was mit der direkten Ausgabe des Fahrplans oder dem Konvertieren in das entsprechende Ausgabe-Format zutun hat, in eine Datei oder (standardmäßig) in die Standard-Ausgabe, wird von der Klasse Result durchgeführt.

Die Klasse Result bietet unter anderem die folgenden Schnittstellen:

Den ermittelten/erstellen Fahrplan als Zeichenkette zurückgeben:

`to_output_text()`

Den ermittelten/erstellen Fahrplan in eine Datei mit dem Namen output.txt schreiben:

`to_file_same()`

Die Abfahrt von einem Zug auf eine Strecke speichern:

`save_train_depart(id_train, time, id_line)`

Den Start-Bahnhof von einem Zug speichern:

`save_train_start(id_train, time, id_station)`

Die Einstiegsrunde/-zeit von einem Passagier in einen Zug speichern:

`save_passenger_board(self, id_passenger, time, id_train)`

Die Ausstiegsrunde/-zeit von einem Passagier speichern:

`save_passenger_detrain(self, id_passenger, time)`

6.2 Ordnerstruktur

Unsere Software hat die folgende Ordnerstruktur:

```
team-zug-zum-erfolg
├── doc
├── abfahrt
│   ├── classes
│   │   ├── Line.py
│   │   ├── Passenger.py
│   │   ├── Station.py
│   │   ├── Train.py
│   │   ├── TrainLine.py
│   │   ├── TrainInStation.py
│   │   └── Travel.py
│   ├── testfiles
│   ├── testutils
│   │   ├── test_generator
│   │   │   └── test_generator.py
│   │   ├── test_simulator
│   │   │   └── test_simulator.py
│   └── unittest
│       ├── unittest_linelist.py
│       ├── unittest_stationlist.py
│       └── unittest_travelcenter.py
├── Groups.py
├── Generator.py
├── Input.py
├── Linelist.py
├── main.py
├── Plan.py
├── Result.py
├── Stationlist.py
├── Travel_Center.py
├── CODE_OF_CONDUCT.md
├── dockerfile
├── Dokumentation.pdf
├── README.md
├── LICENSE.md
├── bonus_eingabe_datei.txt
└── CONTRIBUTING.md
```

6.3 Testen

Sowohl bei der Entwicklung als auch der Weiterentwicklung von vor allem Software dieser Art, sind richtiges Testen und das strategische Finden von Fehlern unvermeidbar. Neben der Nutzung von herkömmlichen Tools/Modulen zum Testen, haben wir uns während der Entwicklung zum großen Teil auch eigene Werkzeuge geschaffen, um dies gründlich zu tun. Diese stellen wir natürlich auch Dritten für die Weiterentwicklung der Software zur Verfügung.

6.3.1 Teststrategien während der Entwicklung

Um möglichst früh und schnell Fehler erkennen und direkt beheben zu können, nutzen wir bereits während der Entwicklung unserer Software sowohl einfache Tests (Unit-tests) zum Testen und Überprüfen unserer Funktionen, welche die kleinsten Einheiten (Units) unserer Software darstellen, sowie auch vollständige, automatische Systemtests zum testen bzw. überprüfen unserer gesamten Software, also vom Einlesen des Eingabe bis hin zum Ausgeben des ermittelten Fahrplans.

6.3.1.1 Unittests

Um die Funktionen, vor allem welche sich während der Entwicklung als am fehleranfälligsten und kompliziertesten herausgestellt haben, automatisch und mit Einbeziehung aller Randfälle testen zu können, haben wir Unittests genutzt bzw. erstellt. Vor allem nach der Bearbeitung dieser Funktionen haben sich die Unittests als besonders hilfreich erwiesen. Dazu haben wir die Python-Erweiterung `unittest` verwendet.

Folgender Code-Ausschnitt illustriert zwei Testfälle unserer Unittests:

```
self.assertEqual(test_linelist.compare_free(
    TrainInLine(trains[2], 2, 2, 1)), [True, -1])

self.assertEqual(test_linelist.compare_free(
    TrainInLine(trains[3], 4, 5, 1)), [False, 5])
```

6.3.1.2 Automatische Systemtests

Um die Software nun vollständig Testen zu können, haben wir mögliche (vor allem besonders kritische) Eingaben in einfachen TXT-Dateien festgehalten. Damit dann vollständige Systemtests durchgeführt werden können, haben wir ein Python-Skript erstellt, welches automatisch diese Eingabedateien erfasst und dann den Inhalt dieser Dateien, beim Starten der Software, in die Standard-Eingabe schreibt und den dann ausgegebenen Fahrplan mit dem Bahn-Simulator validiert.

6.3.2 Testen der Software

Um auch bei der Weiterentwicklung unserer Software (von dritten) einfach und unkompliziert Tests durchführen zu können, haben wir in unserem Repository zwei Python-Module in `abfahrt/testutils` zur Verfügung gestellt, welche wir selbst bei unserer Entwicklung verwendet haben.

Manuell erstellte Eingaben automatisch Testen

Mit dem Modul `test_simulator` können die (manuell erstellten) Eingabedateien in `abfahrt/testfiles` automatisch getestet werden. Da können natürlich beliebig viele Eingabedateien hinzugefügt werden (das Format oder die Dateiendung spielt dabei keine Rolle).

Dazu wird in der Kommandozeile folgender Befehl ausgeführt:

```
python3 -m abfahrt.testutils.test_simulator
```

Windows:

```
python -m abfahrt.testutils.test_simulator
```

Dieses Modul bietet auch hilfreiche erweiterte Funktionen.

Ausschnitt des Hilfe-Menüs von diesem Modul:

Simulator zum automatischen Testen der Software mit den Eingabedateien aus `abfahrt/testfiles`

optional arguments:

<code>-h, --help</code>	Zeigt dieses Hilfemenü an
<code>-verbose</code>	Zeigt die vollständige Ausgabe an
<code>-singletest [file]</code>	Führt nur einen Test mit dieser Eingabedatei aus
<code>-soft</code>	Stoppt das Testen bei einem Fehler

Der `test_simulator` kann auch in einem anderen Programmcode als Modul importiert und ausgeführt werden (siehe 6.5 Wartbarkeit).

Dies ist ein Ausschnitt aus einer Beispielausgabe nach einem durchgeführten Test mit diesem Modul:

```
=====test_one_capacity.txt=====
Score: 0
Time: 0.0029 min
=====test_multiple_stations.txt=====
Score: 48
Time: 0.0016 min
=====test_train_low_capacity.txt=====
Score: 0
Time: 0.0016 min
=====test_two_trains_one_line.txt=====
Score: 0
Time: 0.0016 min
```

Zufällig generierte Eingaben automatisch Testen

Mit dem Modul `test_generator` können Eingaben bzw. Eingabedateien zufällig generiert werden und dann getestet werden. Die Dimension der generierten Eingaben können dabei durch sämtliche Parameter bestimmt bzw. beeinflusst werden. Zum Beispiel kann die maximale Anzahl an Stationen festgelegt werden, welche generiert werden sollen. Folgende Parameter kann dieses Modul dafür entgegennehmen:

```
-test_amount [Anzahl der auszuführenden Tests]

-number_stations [Anzahl an Bahnhöfen]

-number_lines [Anzahl an Strecken]

-number_trains [Anzahl an Zügen]

-number_passengers [Anzahl an Passagieren]

-max_capacity_station [Maximale Kapazität von einem Bahnhof]

-max_capacity_line [Maximale Kapazität von einer Strecke]

-max_length_line [Maximale Länge von einer Strecke]

-max_capacity_train [Maximale Kapazität von einem Zug]

-max_groupsize_passenger [Maximale Gruppengröße bei einem Passagier]

-max_targettime_passenger [Maximale Zielzeit bzw. -runde bei einem Passagier]

-max_speed_train [Maximale Geschwindigkeit von einem Zug]
```

Um dieses Modul dann zu nutzen/starten, wird in der Kommandozeile folgender Befehl ausgeführt:

```
python3 -m abfahrt.testutils.test_generator -number_stations 20 ....
```

Windows:

```
python -m abfahrt.testutils.test_generator -number_stations 20 ....
```

Der `test_generator` kann auch in einem anderen Programmcode als Modul importiert und ausgeführt werden (siehe 6.5 Wartbarkeit).

Unittests automatisch durchführen

Mit dem Modul `unittest` können automatisch die von uns vordefinierten Unittest-Testfälle ausgeführt werden. Diese überdecken/testen vor allem die wichtigsten bzw. fehleranfälligen Funktionen und Module mit ausgereiften und sinnvollen Eingaben.

Um dieses Modul dann zum Testen zu starten, wird in der Kommandozeile folgender Befehl ausgeführt:

```
python3 -m abfahrt.unittest
```

Windows:

```
python -m abfahrt.unittest
```

6.4 Programmierstil

Bei unserem Programmierstil bzw. unseren Coding conventions (engl.) handelt es sich um Snake-Case. Wir haben diesen Programmierstil bzw. diese Namenskonvention anstatt z.B. Camel-Case gewählt, da diese für uns mehr Übersichtlichkeit bietet. Zudem wird durch das strikte Trennen durch einen Unterstrich zwischen jedem Wort von dem Namen einer Funktion oder Variable, die Unterscheidbarkeit der Wörter gestärkt und somit die Erkennbarkeit/Lesbarkeit der Nutzen/Anwendungsfall einer Funktion bzw. Variable schneller erkannt werden. Bei der Benennung der Variablen und Funktionen wurden ausschließlich die Sprache Englisch verwendet, da mit den meisten englischen Begriffen die Beschreibung des Nutzen einer Variable oder Funktion eindeutiger gelingt, da diese im Programmier-Spektrum bzw. -Szene häufiger anzutreffen sind und genutzt werden. Zum Beispiel der Name/Alias `clear_station()` wird von einem durchschnittlichen Programmierer im Kontext von Fahrplänen und Streckennetzen und dem Bezug zu ähnlichen Funktionen aus (ganz) anderem Software-Code (welche ebenfalls `clear` im Namen haben) wohl schneller verstanden als freigeben, da der Begriff freigeben eigentlich nur in unserem Kontext hier verwendet wird und nur für diesen entsprechenden Vorgang. Der Begriff `clear` hingegen wird in den Funktionsnamen von Funktionen in auch sehr bekannter Software und Bibliotheken mit ähnlichen Vorgängen verwendet. Eine Zuordnung gelingt da schneller. Außerdem ist die englische Sprache aus einem anderen wichtigen Grund hier für den Programmcode besser geeignet, denn die bereits eingebauten Funktionen der Erweiterungen usw. von Python sind ebenfalls in der englischen Sprache benannt. Nun eine andere Sprache zu verwenden, könnte die Code-Qualität verringern bzw. weniger angenehm machen. Abgesehen von der genutzten Sprache für die Benennung der Variablen und Funktionen, haben wir diese weitgehend ohne (schlecht verständliche) Abkürzungen benannt, also vor allem bei den wichtigeren Funktionen und Variablen nicht abgekürzt, wenn es auch anders möglich war oder die Abkürzung nicht sofort verständlich gewesen wäre. Die Dokumentation von dem Programmcode wurde zudem auch in Englisch verfasst, da so an besser an den Kontext und die Variablen und andere Funktionen, welche ebenfalls in Englisch benannt sind, besser angeknüpft werden kann. Da wir die Software in Python entwickelt haben, war uns neben herkömmlichen Kommentaren, auch möglich, DocStrings zu verwenden um die Funktionen und ihre Parameter zu beschreiben. Außerdem konnten wir, da wir bei diesen ein bestimmtes Format (google) verwendet haben, auch eine Dokumentation mit dem Python-Modul `pdoc` generieren lassen bzw. Dritten anbieten, mit diesem Werkzeug die Dokumentation für unsere Software selbst zu generieren. Diese wird dann z.B. in HTML generiert und lässt sich sehr angenehm im Browser begutachten. Im Gegensatz zu dem (älteren) Werkzeug `pydoc`, welches die gleiche Funktion erfüllt, bietet `pdoc` ein wesentlich qualitativeres Layout der Dokumentation und passt sich auch dem DocString-Format an bzw. kann dieses parsen.

6.5 Wartbarkeit

Bei der Entwicklung unserer Software haben wir auch auf eine gewisse Wartbarkeit bzw. Erweiterbarkeit geachtet. Es soll anderen möglichst bequem und einfach möglich sein, unsere Software weiterzuentwickeln oder basierend auf unserer Software möglichst einfach ein Programm zu entwickeln, welches Fahrpläne nach einem abgeänderten Algorithmus erstellt. Also z.B. ohne das Gruppieren der Passagiere.

Um mit unserer Software nun eigene kleine Programme zu entwickeln, muss lediglich das Python-Package `abfahrt` importiert werden.

Folgender Beispielcode illustriert ein solches Programm:

```
from abfahrt import *
```

```
def run():
    input_ = Input()
    stations, lines, trains, passengers = input_.from_stdin()
    result = Result(input_)

    linelist = Linelist(lines)
    stationlist = Stationlist(stations, trains, result)
    travel_center = Travel_Center(trains, stationlist, linelist, result)

    groups = Groups(passengers)

    if not travel_center.check_plan():
        print("INVALID INPUT: Not all stations connected!")
        return

    while len(groups.route) != 0:
        group = groups.get_priority()
        start_station, end_station, group_size = travel_center.check_passengers(
            group)
        start_time_list, trainlist, available, __, __ = travel_center.check_train_in_stati
            start_station, group_size)
        if not available:
            groups.passengers_arrive(group)
            continue
        route_length = travel_center.time_count_length(
            start_station, end_station)
        train_fastest, train_fastest_start_time = travel_center.get_fastest_train_by_sta
            trainlist, start_time_list, route_length)
        travel_fastest = travel_center.time_count_train(
            start_station, end_station, train_fastest, train_fastest_start_time)
        travels = [travel_fastest]
        travel_center.determine_and_save_shortest_travel(
            travels, groups, group)
    print(result.to_output_text())
    return
run()
```

Die obige Code entspricht soweit dem von `main`, jedoch werden sämtliche Überprüfungen weggelassen, es geht bei diesem nur darum zu demonstrieren, dass auch ein eigener Algorithmus schnell und einfach mit unserer Software entworfen werden kann.

Auch die beiden Testmodule bzw. -programme können in einen anderen Programmcode einfach importiert und in diesem genutzt werden.

Folgender Beispielcode für den test_simulator:

```
from abfahrt import *  
  
ts = test_simulator("abfahrt/testfiles")  
ts.run()
```

Das Verzeichnis der Eingabedateien kann oben natürlich angepasst werden.

Folgender Beispielcode für den test_generator:

```
from abfahrt import *  
  
tg = test_generator()  
tg.run()
```

7 Auswertung

Die Qualität unserer Lösung hängt von den folgenden Kriterien ab:

- Ausführungsdauer
- Gesamtverspätung

Um die Qualität unserer Lösung hinsichtlich dieser auszuwerten, haben wir diese mit genügend Eingaben, gestaffelt nach jeweils unterschiedlichen Dimensionen, ausgeführt. Mit Dimension einer Eingabe ist die Anzahl der Bahnhöfe, Strecken, Züge und Passagiere, sowie die Größe der Kapazitäten dieser gemeint. Je größer eine Dimension ist, desto größer sind also diese Werte und desto stärker kann die Software hinsichtlich der Kriterien Effizienz oder Gesamtverspätung beansprucht bzw. beeinflusst werden. Um die beiden oben genannten Kriterien messen zu können, haben wir für jedes Kriterium zuerst solche Dimensionen aufgestellt und dann entsprechende Messungen durchgeführt.

7.1 Effizienz und Ausführungszeit

Für das Kriterium Effizienz bzw. Ausführungszeit/-dauer haben wir die folgenden Dimensionen genutzt:

Dimensionen bezüglich dem Streckennetz

Tabelle 1: Dimensionen bezüglich dem Streckennetz

Dimension:	Sehr Gering	Gering	Mittel	Groß	Sehr Groß	Sehr Sehr Groß
Parameter:	Bahnhöfe: 10 Strecken: 20	Bahnhöfe: 60 Strecken: 120	Bahnhöfe: 120 Strecken: 240	Bahnhöfe: 180 Strecken: 360	Bahnhöfe: 240 Strecken: 480	Bahnhöfe: 300 Strecken: 600

Dimensionen bezüglich dem Betrieb

Tabelle 2: Dimensionen bezüglich dem Betrieb

Dimension:	Sehr Gering	Gering	Mittel	Groß	Sehr Groß	Sehr Sehr Groß
Parameter:	Passagiere: 10	Passagiere: 200	Passagiere: 400	Passagiere: 600	Passagiere: 800	Passagiere: 1000

Da die Anzahl der Züge auf dem Streckennetz aus logischen Gründen fest auf die Summe der Kapazitäten aller Bahnhöfe nach oben begrenzt ist, macht es keinen Sinn, diese in die Dimensionen bezüglich dem Betrieb miteinzubeziehen. Deshalb werden wir diese als separate Größe betrachten, jedoch aber nicht die absolute Anzahl der Züge, sondern die relative Anzahl im Verhältnis zur Summe der Kapazitäten aller Bahnhöfe.

Dimensionen bezüglich der Anzahl der Züge

Tabelle 3: Dimensionen bezüglich der Anzahl der Züge

Dimension:	Gering	Mittel	Groß
Parameter:	Züge: 5%	Züge: 40%	Züge: 90%

Messung

Um eine Messung mit diesen Dimensionen durchzuführen, haben wir von jeder Kombination der Dimensionen (also bezüglich dem Streckennetz, dem Betrieb und der Anzahl der Züge) jeweils zehn Eingabedateien mit den entsprechenden Parametern generiert. Dabei haben wir den `test_generator` (siehe 6.3 Testen) verwendet. Der Inhalt der Eingabedateien wurden dann unserer Software als Eingabe übergeben und diese also jeweils mit dieser Eingabe ausgeführt. Die Ausführungszeit wurde anhand der Differenz der Start- und End-Zeit vor bzw. nach der Ausführung bestimmt. Die Ausführungszeit für eine bestimmte Kombination von Parametern ergab sich dann aus dem Durchschnitt aller Ausführungszeiten der zehn Eingabedateien mit dieser Kombination von Parametern. Bei dem Messen wurden, für dieses Kriterium, für die Kapazitäten überall standardmäßige (zufällige) Werte zwischen 1 und 40 festgelegt und diese nicht weiter bei den Messungen berücksichtigt, da diese sich nicht (explizit) auf die oben genannten Kriterien auswirken bzw. dabei keine bedeutende Rolle spielen.

Folgende Ausführungszeiten wurden gemessen:

Tabelle 4: Ausführungszeiten mit konstant 10 Zügen

Dimension:	10 Passagiere	200 Passagiere	400 Passagiere	600 Passagiere	800 Passagiere	1000 Passagiere
10, 20	0,0016	0,0027				
60, 120	0,0035	0,0376	0,0036	0,0061	0,0108	0,1
120, 240	0,0081	0,181	0,074	0,112	0,154	0,187
160, 360	0,0164	0,39	0,352	0,54	0,74	0,92
240, 480		0,669	0,776	1,141	1,55	1,969
360, 600	0,034	1,29	1,432	2,067	2,686	3,331
	0,068		2,55	3,821	5,075	6,413

Ausführungszeit

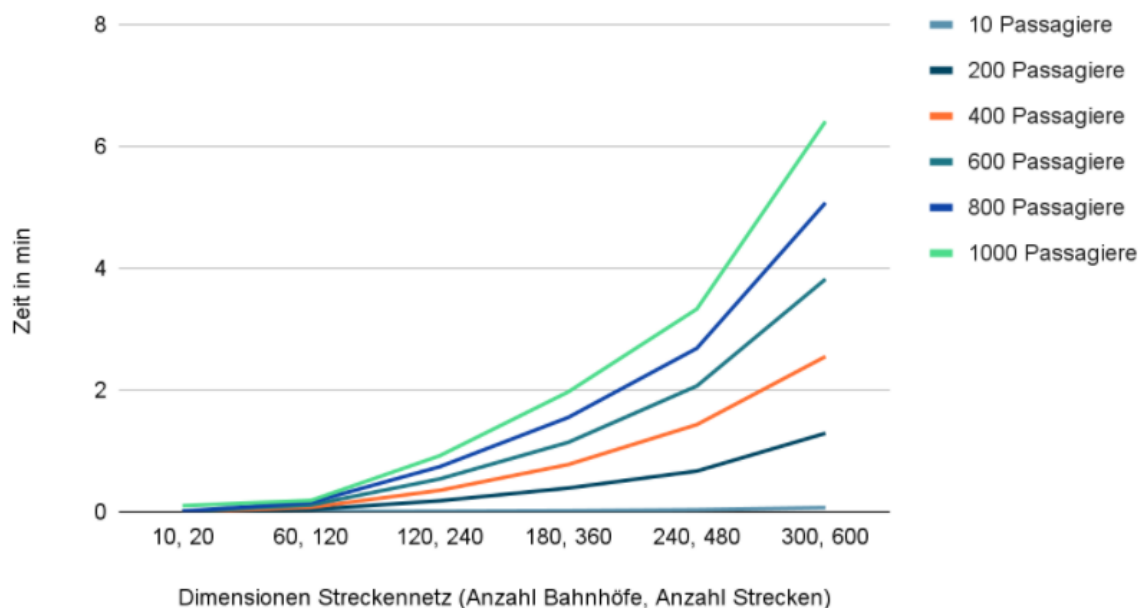


Abbildung 6: Ausführungszeit

Ausführungszeit mit 5 % Zügen von gesamter Bahnhofs-Kapazität

Tabelle 5: Ausführungszeit mit 5%-Zügen

Dimension:	10 Passagiere	200 Passagiere	400 Passagiere	600 Passagiere	800 Passagiere	1000 Passagiere
10, 20	0,0022	0,0043	0,0104	0,0164	0,0182	0,0219
60, 120	0,0067	0,0417	0,083	0,124	0,199	0,187
120, 240	0,0082	0,187	0,382	0,571	0,733	0,987
160, 360	0,016	0,396	0,803	1,178	1,618	2,03
240, 480	0,033	0,754	1,458	2,218	2,969	3,603
360, 600	0,0621	1,417	2,889	4,27	5,704	6,413

Ausführungszeit mit 5% Zügen

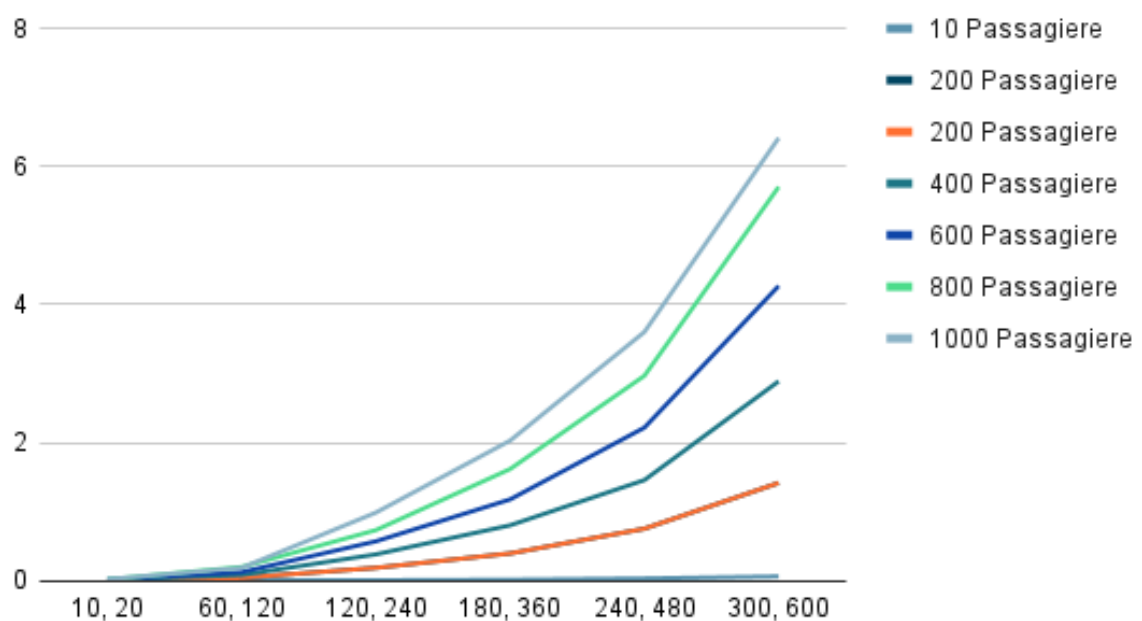


Abbildung 7: Ausführungszeit mit 5% Zügen

Ausführungszeit mit 40 % Zügen von gesamter Bahnhofs-Kapazität

Tabelle 6: Ausführungszeit mit 40%-Zügen

Dimension:	10 Passagiere	200 Passagiere	400 Passagiere	600 Passagiere	800 Passagiere	1000 Passagiere
10, 20	0,0023	0,0029	0,0039	0,0065	0,0141	0,0203
60, 120	0,0035	0,0376	0,074	0,102	0,154	0,187
120, 240	0,006	0,181	0,352	0,54	0,74	0,92
160, 360	0,0136	0,224	0,776	1,141	1,55	1,969
240, 480	0,0286	0,669	1,432	2,067	2,686	3,331
360, 600	0,056	1,134	2,206	3,236	5,075	6,187

Ausführungszeit mit 40% Zügen

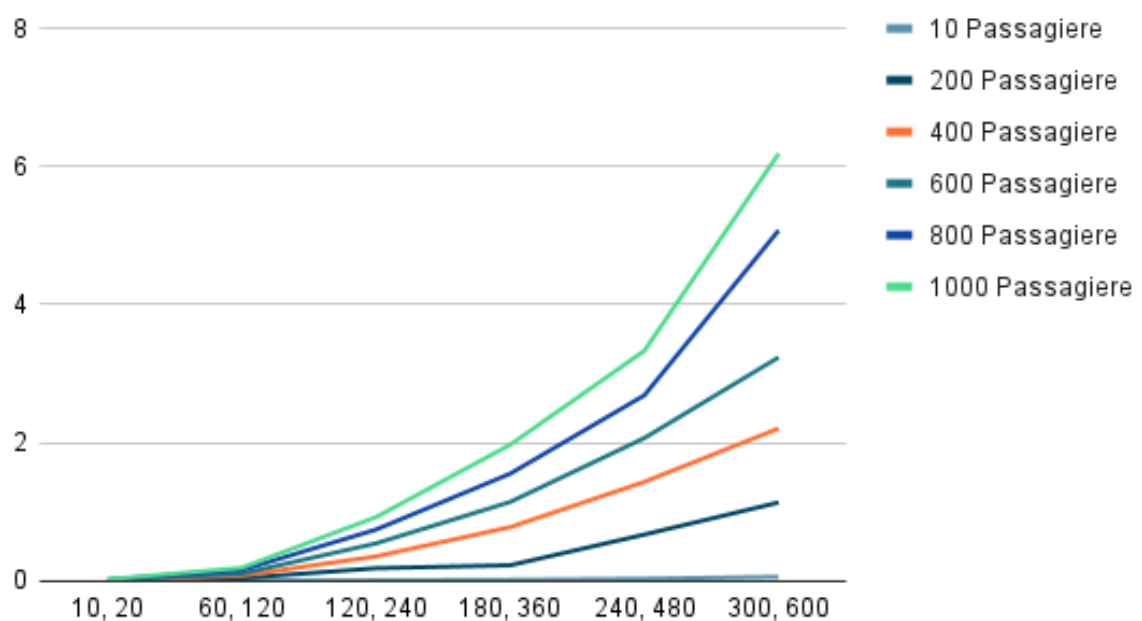


Abbildung 8: Ausführungszeit mit 40% Zügen

Ausführungszeit mit 90 % Zügen von gesamter Bahnhofs-Kapazität

Tabelle 7: Ausführungszeit mit 90%-Zügen

Dimension:	10 Passagiere	200 Passagiere	400 Passagiere	600 Passagiere	800 Passagiere	1000 Passagiere
10, 20	0,0018	0,0027	0,0031	0,005	0,0111	0,012
60, 120	0,0035	0,0323	0,084	0,122	0,162	0,183
120, 240	0,0081	0,171	0,352	0,521	0,72	1,01
160, 360	0,00164	0,39	0,781	1,122	1,64	1,978
240, 480	0,045	1,012	1,422	2,161	2,751	3,321
360, 600	0,068	1,124	2,54	3,821	5,176	6,552

Ausführungszeit mit 90% Zügen

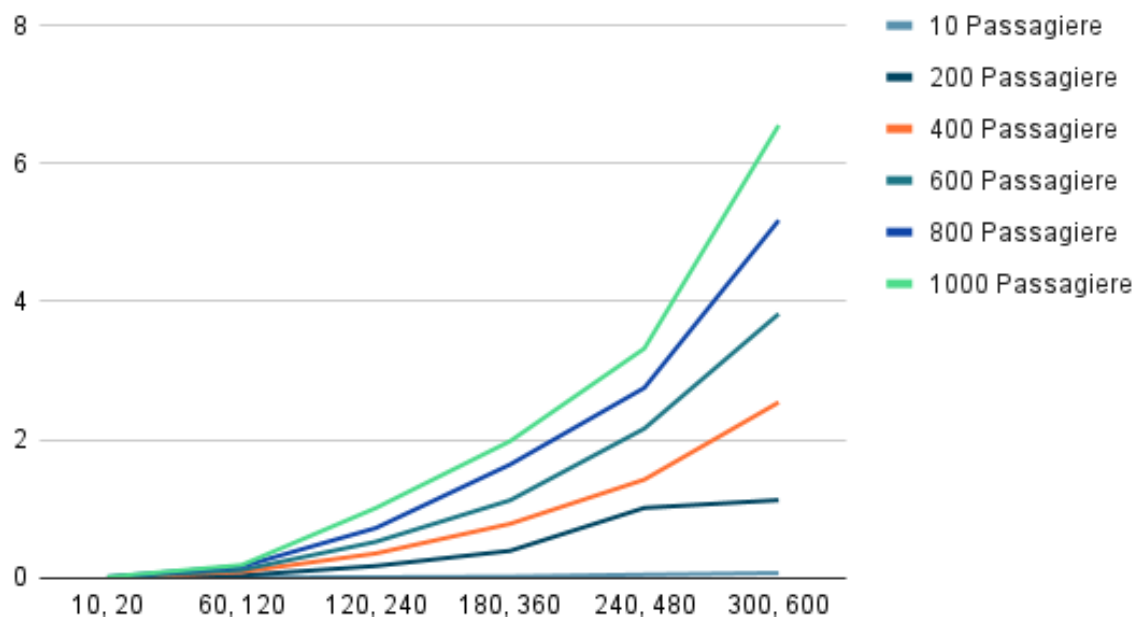


Abbildung 9: Ausführungszeit mit 90% Zügen

Die gemessenen Ausführungszeiten liegen alle in einem zeitlichen Bereich zwischen nahezu 0 bis 8 Minuten. Auch die längste Ausführung hat nicht länger als 10 Minuten andauert. Somit ist die Ausführungsdauer unseren Erwartungen entsprechend angenehm und unsere Software auch für die spontane Ermittlung von Fahrplänen geeignet, sofern die Dimensionen natürlich berücksichtigt werden. Eine wichtige Beobachtung bei der Messung der Ausführungszeit in Abhängigkeit der Anzahl der Züge ist, dass diese den Verlauf der Ausführungszeiten und die absoluten Werte dieser Ausführungszeiten kaum bzw. nur sehr gering beeinflusst hat. Zu erklären ist dies mit unseren Beschränkungen, welche wir eingeführt haben, wenn Züge in unserem Programmcode einen Multiplikator für die Zeitkomplexität darstellten. Als z.B. bei zehn Zügen die Zeitkomplexität verzehnfacht wurde, weil früher/zuvor bei dem Rufen von Zügen, für alle Züge eine Route berechnet wurde. Dies haben wir durch die Einführung von Beschränkungen und Optimierungen unterbunden. Indem z.B. die durchschnittliche Routenlänge (siehe 5. Theoretischer Ansatz) für jeden Zug als zu fahrende Route genutzt und dann berechnet wurde, wann dieser bei dem benötigten Bahnhof ankommen könnte, unter Hinzuziehung der frühesten Startzeit und seiner Geschwindigkeit natürlich. So war es nicht mehr nötig, mit dem Dijkstra-Algorithmus nach einer Route zu suchen. Dies hat sich wesentlich positiver auf die Ausführungszeit ausgewirkt.

7.2 Gesamtverspätung der Passagiere

Für dieses Kriterium haben wir einerseits mit teils riesigen Dimensionen gearbeitet/gemessen, um Abhängigkeiten zu prüfen, ob unsere Software auch z.B. wirklich optimiert, wenn mehr Kapazitäten bei den Strecken und Bahnhöfe vorliegen usw. und andererseits auch selbstständig (frei nach Hand) Eingaben erstellt und dabei darauf geachtet, dass diese von einem universalen Optimierer (also nicht unsere Software oder unser Algorithmus) realistisch lösbar sind und alle Passagiere zu der gewünschten Ankunftszeit ihr Ziel erreichen können.

Für die erste Variante wurden folgende Dimensionen für dieses Kriterium verwendet:

Dimensionen bezüglich der Kapazitäten

Tabelle 8: Dimensionen bezüglich dem Streckennetz

Dimension:	Gering	Mittel	Groß
Parameter:	Kapazität Bahnhof: 1 Kapazität Strecke: 1	Kapazität Bahnhof: 6 Kapazität Strecke: 6	Kapazität Bahnhof: 12 Kapazität Strecke: 12

Dimensionen bezüglich dem Betrieb

Tabelle 9: Dimensionen bezüglich dem Betrieb

Dimension:	Gering	Mittel	Groß
Parameter:	Passagiere: 2 Züge: 10	Passagiere: 200 Züge: 10	Passagiere: 400 Züge: 10

Messung: Variante mit generierten Eingaben

Um eine Messung mit diesen Dimensionen durchzuführen, haben wir von jeder Kombination der Dimensionen (also bezüglich dem Streckennetz und dem Betrieb) jeweils zehn Eingabedateien mit den entsprechenden Parametern generiert. Dabei haben wir den `test_generator` (siehe 6.3 Testen) verwendet. Bei dem Messen wurden, für dieses Kriterium, für die Anzahl der Bahnhöfe und Strecken überall standardmäßig Werte von 20 und 25 festgelegt und diese nicht weiter bei den Messungen berücksichtigt, da diese sich nicht wirklich auf die oben genannten Kriterien auswirken bzw. dabei keine bedeutende Rolle spielen, da die gewünschte Ankunftszeit hier in Abhängigkeit der kürzesten Routenlänge festgelegt wird, somit kommt es hier nur auf die Verzweigung von dem Streckennetz an, denn je weniger verzweigt dieses ist, desto unterschiedlicher fallen die gewünschten Ankunftszeiten der Passagiere aus und desto besser wird dann die Qualität der Lösung hinsichtlich der Minimierung der Gesamtverspätung getestet. Wenn also bei dieser Messung z.B. jeder Bahnhof mit jedem anderen Bahnhof verbunden wäre, dann würde man die Software nur mäßig bzw. gar nicht hinsichtlich der Priorisierung der Passagiere testen bzw. messen, da dann ja jeder Passagier im Schnitt die gleiche gewünschte Ankunftszeit besitzen würde.

Auch die Geschwindigkeit von allen Zügen wurde fest auf 1 festgelegt, die Länge der Strecken ebenso. Die Gruppengröße aller Passagiere sowie die Kapazität aller Züge wurde ebenfalls fest auf 1 festgelegt. Die gewünschte Ankunftszeit bzw. -runde wurde bei allen Passagieren der Eingaben als die vom Bahnhof ab der Startzeit bzw. Runde 0 optimalste bzw. frühestmögliche Ankunftszeit mit einem der Züge bei dem Ziel-Bahnhof festgelegt.

Danach wurde die Ausgabe bzw. der Fahrplan mit dem Bahn-Simulator (vom InformatiCup) validiert und der ausgegebene Score als (Vergleichs-)Wert für die Gesamtverspätung genutzt.

Folgende Gesamtverspätungen wurden gemessen:

Tabelle 10: Gesamtverspätungen nach Kapazitäten und Passagieren

Dimension:	2 Passagiere, 10 Züge	200 Passagiere, 10 Züge	400 Passagiere, 10 Züge
1, 1	2	42128	186329
6, 6	4	11047	48615
12, 12	6	12148	46146

Gesamtverspätung

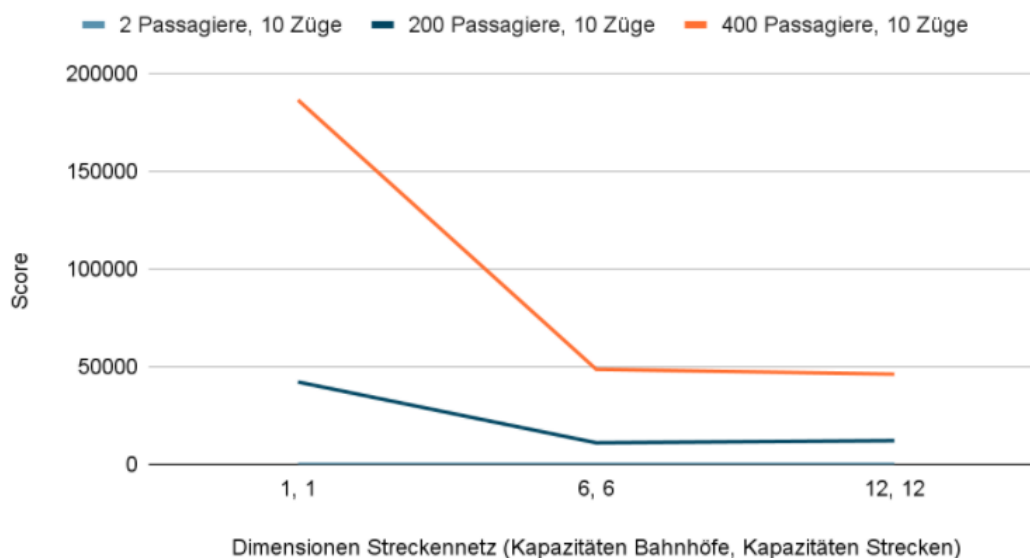


Abbildung 10: Gesamtverspätung generierter Eingaben

An den Gesamtverspätungen mit generierten Eingaben in Abhängigkeit der Kapazitäten ist die Gesamtverspätung bei 200, 400 Passagieren schon alleine aus diesem Grund so hoch, weil jeweils nur 10 Züge vorhanden waren. Somit war es natürlich nie möglich, die Passagiere auch nur annähernd entsprechend ihrer gewünschten Ankunftszeiten zu transportieren. Jedoch geht aus dem Verlauf hervor, dass der Algorithmus bei der Erhöhung der Anzahl an Bahnhöfen, die Passagiere früher an ihr Ziel bringen kann, somit hat sich dieser hinsichtlich seiner Fähigkeit zu Optimieren bewährt, da zusätzlich freie Kapazitäten aktiv wahrgenommen und genutzt werden bzw. diese in die Fahrplan-Erstellung miteinbezogen werden.

Messung: Variante mit manuellen Eingaben

Um diese Messung durchzuführen, wurden zuerst manuell 20 Eingaben bzw. Eingabedateien erstellt. Die Anzahl der Bahnhöfe variierte zwischen fünf und zehn und die Anzahl der Strecken entsprechend der Anzahl der Bahnhöfe zwischen acht und 24, die Kapazitäten der Bahnhöfe und Strecken variierten zwischen eins und drei und die Längen der Strecken betrugen alle genau eins. Die Anzahl der Züge variierte zwischen 1 und 10, die Gruppengröße der Passagiere war strikt auf 1 festgelegt und die Kapazität der Züge variierte wieder zwischen 1 und 8. Die Gruppengröße der Passagiere variierte deshalb nicht, weil der Bahn-Simulator (vom InformatiCup) den ausgegebenen Score aus der Summe der Produkte der Verspätungen der Passagiere und der jeweiligen Gruppengröße berechnet. Ist diese Gruppengröße hingegen bei allen Passagieren nur 1, ist der Score somit nur alleine für die Gesamtverspätung repräsentativ. Nachdem in diesen Größenordnungen Eingaben erstellt wurden, wurden die gewünschten Ankunftszeiten der Passagiere festgelegt. Der Wert von diesem Parameter ist entscheidend für eine richtige Messung der Gesamtverspätungen, denn beträgt diese z.B. (stark untertrieben) bei jedem Passagier eins oder null, dann ist logischerweise keine realistische Lösung möglich, die gewünschten Ankunftszeiten einzuhalten, ohne dass sich ein Passagier verspätet. Deshalb wurden diese anhand selbständig ausfindig gemachter Routen und ausgedachter Möglichkeiten bei z.B. einem vollen Bahnhof diesen zu umfahren usw., die gewünschte Ankunftszeit bei jedem Passagier entsprechend dieser Optimierungsmöglichkeiten festgelegt. Für einen (fiktiven) universalen Optimierer von solchen Fahrplänen, sollte es also mögliche sein, die gewünschten Ankunftszeiten einzuhalten und eine entsprechende Lösung zu finden. Danach wurde diese Eingaben bzw. Eingabedateien mit unserer Software getestet und dann die Ausgaben bzw. Lösungen mit dem Bahn-Simulator (vom InformatiCup) validiert. Die ermittelten Score-Werte stellten dann jeweils einen Wert für die Gesamtverspätung dar.

Folgende Gesamtverspätungen wurden gemessen:

Tabelle 11: Gesamtverspätungen manueller Eingaben

Gesamtverspätungen (Score):	Anzahl Eingaben (absolut)	Anzahl Eingaben (relativ)
0	9	45%
0-9	5	25%
10-19	2	10%
20-39	2	10%
>= 40	2	10%

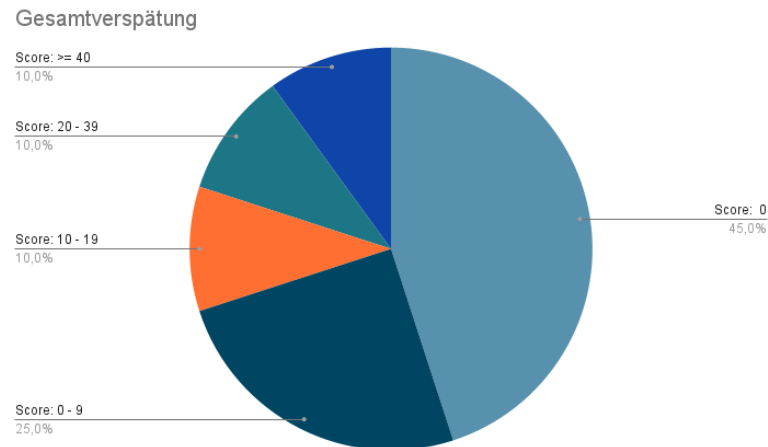


Abbildung 11: Gesamtverspätung manueller Eingaben

Die obigen Ergebnisse der Messung der Gesamtverspätungen mit manuell erstellten Eingaben zeigen, dass der Algorithmus hinsichtlich der per Hand als optimal festgelegten bzw. bestimmten gewünschten Ankunftszeiten, die Passagiere im Durchschnitt recht zufriedenstellend transportieren lässt. Für 45 Prozent der Eingaben bzw. Eingabedateien gab es sogar keine Verspätungen. Den relativ zweitgrößten Anteil an Verspätungen hatte bzw. lag in dem Intervall von 0 bis 9 Score-Punkten, also ein verhältnismäßig tolerierbarer Verspätungszeitraum. Die weniger tolerierbar bzw. ansatzweise als kritischer zu betrachtenden Intervalle der Gesamtverspätungen, nahmen einen Anteil von insgesamt 30 Prozent der gesamten Gesamtverspätungen ein.

8 Diskussion

8.1 Berechnung der kürzesten Routen/Pfade

Zur Berechnung der (kürzesten) Routen kamen für uns mehrere Such-Algorithmen infrage. Man entscheidet grundsätzlich zwischen den informierten (heuristischen) und den uninformierten (blinden) Algorithmen. Die uninformierten Algorithmen, wie z.B. der Dijkstra-Algorithmus, die Breiten- oder Tiefen-Suche und der Bellman-Ford Algorithmus kamen bei uns zuerst nicht infrage, da diese zu rechenintensiv sein könnten, für unsere Graphen bzw. Streckennetze, welche ja auch in größeren Dimensionen vorliegen könnten. Die informierten Algorithmen wie z.B. der A*-Algorithmus erschienen uns da besser geeignet, da diese sich in ihrer Rechenzeit begrenzen bzw. optimierter verhalten. Wir haben uns schließlich dennoch für den Dijkstra-Algorithmus entschieden, obwohl dieser zu den uninformierten Such-Algorithmen gehört, da dieser unter anderem leichter zu implementieren war.

Dijkstra-Algorithmus

Der Dijkstra-Algorithmus löst das Problem der kürzesten Pfad-Berechnung von einem Knoten zu allen anderen Knoten in einem Graphen [10]. Die Zeitkomplexität beträgt im Worst-Case $O(E + V^2)$, im Average-Case $O(E + V^2)$ und im Best-Case $O(E + V^2)$ [23] [10]. Die Platzkomplexität beträgt $O(V)$ [10]. Dieser erschien uns einerseits als am leichtesten zu implementieren und andererseits im Verhältnis dafür der Algorithmus mit der im Schnitt besten Effizienz bzw. Zeitkomplexität. Der Bellman-Ford-Algorithmus z.B. hat eine Zeitkomplexität von $O(E * V)$ im Worst-Case wie auch Average-Case und ist damit weniger effizient [6]. Die Breiten- oder Tiefen-Suche hingegen hat nur eine Zeitkomplexität von $O(E + V)$ und eine Platzkomplexität von $O(V)$ und ist somit effizienter als der Dijkstra-Algorithmus, jedoch kann dieser bzw. können diese Algorithmen nur den kürzesten Pfad ohne gewichtete Kanten berechnen und schieden damit natürlich sofort aus. Die uniformierte Kostensuche (engl. Uniformed-Cost-Search), welche eine Weiterentwicklung von der Breiten-Suche ist und auch auf gewichtete Kanten ausgelegt ist, kam ebenfalls in Frage. Diese hat eine bessere Zeitkomplexität als der Dijkstra-Algorithmus, erschien uns jedoch schwieriger/aufwendiger zu implementieren. Dies war auch der Grund für das Ausscheiden der heuristischen bzw. informierten Algorithmen. Zudem ermittelt der Dijkstra-Algorithmus nicht nur den kürzesten Pfad zu einem bestimmten Knoten, sondern zu allen anderen Knoten im Graphen, dies hätte man nutzen können um eventuell Pfade zu diesen zwischenspeichern (engl. cachem). Der Dijkstra-Algorithmus funktioniert zudem auch in nicht vollständig zusammenhängenden Graphen, was uns bei der weiteren Entwicklung unserer Software als sehr nützlich erwies [22]. Aus diesen Gründen haben wir uns für den Dijkstra-Algorithmus entschieden.

A*-Algorithmus

Anders als die uninformierten Algorithmen, wie den Dijkstra-Algorithmus, nutzt der A*-Algorithmus eine Heuristik um eine optimale Lösung zu berechnen [5]. Der A*-Algorithmus verbraucht zwar mehr Speicherplatz als der Dijkstra-Algorithmus und muss dazu mehr Operationen durchführen, je Knoten, jedoch ist der A* Algorithmus insgesamt schneller, da dieser immer nur die optimalen Knoten mit optimalen Kosten prüft,

mittels seiner Heuristik [5]. Da stellt sich die Frage, warum nicht der A*-Algorithmus genutzt wurde?

Da wir nicht nur mit festen Beispielen/Graphen arbeiten, also dadurch dass jedes Zug-Netz einzigartig ist, kann keine perfekte Heuristik erstellt werden. Im echten Leben, mit einem festen, statischen Zug-Netz kann der A* Algorithmus auch nicht eingesetzt werden, da Strecken und Bahnhöfe immer neu gebaut bzw. verlegt oder auch abgerissen werden. Dann müsste die Heuristik stets erneuert werden. Mit dem Dijkstra-Algorithmus muss keine Heuristik gepflegt/verwendet werden, aber dafür sind die Rechenzeiten auch größer [19] [5].

Kruskal-Algorithmus

Der Kruskal-Algorithmus gehört, wie der Dijkstra-Algorithmus, zu den Greedy-Algorithmen. Der Kruskal-Algorithmus kann einen minimalen Spannbaum in einem Graphen berechnen und ist somit eher ungeeignet den minimalen/kürzesten Pfad von einem Knoten zu einem anderen Knoten zu berechnen [14]. Dieser hat außerdem eine Zeitkomplexität von $O(V * \log(V) + E * \log(E))$ [15]. Dieser ist damit weniger/gleich effizient wie der Dijkstra-Algorithmus mit einer Zeitkomplexität von $O(E + V^2)$ [23] [10].

Prim-Algorithmus

Der Prim-Algorithmus gehört, ebenfalls wie der Dijkstra-Algorithmus, zu den Greedy-Algorithmen. Der Prim-Algorithmus berechnet hingegen, wie der Kruskal-Algorithmus, nur einen minimalen Spannbaum von einem Graphen mit der Einbeziehung von allen Knoten in dem Graphen [16]. Dieser ist somit, wie der Kruskal-Algorithmus, dazu ungeeignet, den kürzesten Pfad von einem Knoten zu einem anderen zu berechnen. Außerdem hat dieser eine Zeitkomplexität von $O(V * \log(V) + E * \log(E))$ [17]. Diese übertrifft somit nicht die des Dijkstra-Algorithmus mit $O(E + V^2)$ [23] [10].

Uniform-Cost-Search

Uniform-Cost Search oder kurz UCS ist eine Weiterentwicklung der Breiten-Suche. Der UCS-Algorithmus ist effizienter bzw. hat eine bessere/gleiche Zeitkomplexität und, durch seine Vorrangwarteschlange, ist der UCS-Algorithmus auch effizienter in dem Speicherverbrauch bzw. der Platzkomplexität wie der klassische Dijkstra-Algorithmus [2]. Jedoch berechnet der UCS-Algorithmus normalerweise nur den kürzesten Pfad von einem Knoten zu einem bestimmten anderen Knoten im Graphen [21]. Wir hatten jedoch die Überlegung, dass es optimal wäre, für jeden Knoten den kürzesten Pfad zu alle anderen Knoten (einmal) zu berechnen und dann zwischenspeichern (engl. cachen). Dafür ist der UCS-Algorithmus jedoch nicht ausgelegt, deshalb haben wir den Dijkstra-Algorithmus diesem vorgezogen. Dieser ist dafür nämlich ausgelegt.

Außerdem war uns der Dijkstra-Algorithmus vertrauter/bekannter, da wir diesen bereits in anderen Vorlesungen/Übungen verwendet hatten.

Wenn man jedoch nur den kürzesten Pfad von einem Start- zu einem Ziel-Knoten berechnen möchte, dann sollte man, im Gegensatz zu dem Dijkstra-Algorithmus, den UCS-Algorithmus nutzen. [2] [20].

8.2 Priorisierung der Passagiere und Ankunftszeit

Bei unserem Algorithmus haben wir uns darauf beschränkt, die Passagiere mit der kleinsten/frühesten Ankunftszeit, auch frühestmöglich oder besser gesagt so schnell und früh wie möglich ihrem Ziel zu transportieren. Dies unabhängig davon, ob die transportierten Passagiere zu früh ankommen könnten oder pünktlich. Jedoch wird die Gesamtverspätung durch einen zu früh ankommenden Passagier keineswegs verbessert bzw. verringert. Zu früh ankommende Passagiere haben also eigentlich keinen direkten positiven Einfluss auf die Gesamtverspätung aller Fahrten. Jedoch einen klaren indirekten Einfluss, denn je früher ein Passagier mit einem Zug bei seinem Ziel ankommt, desto früher ist auch der entsprechende Zug wieder für andere Fahrten verfügbar. Auf genau diesen Aspekt bzw. diese Annahme haben wir, unter anderem, bei der Entwicklung unserer Software gesetzt. Auf praktischen Seite hat das depriorisieren von Passagieren mit später bzw. hoher gewünschter Ankunftszeit auch einen gewissen Vorteil bzw. Sinnmäßigkeit, denn ein Passagier, welcher eine lange Strecke eingeplant hat, hat normalerweise auch eher Verspätungen oder Verzögerungen bei der Fahrt mit-eingeplant, als jemand, der jeden Morgen knapp eine Stunde vor Arbeitsbeginn in den noch spätmöglichsten Zug einsteigt, um morgens auf keinen Fall zu früh aufzustehen. Dieser verlässt sich also normalerweise eher auf die geplanten Ankunftszeiten und das Management der Züge und Fahrpläne, als ein Tourist, welcher von Berlin nach Paris oder an die Nordsee in den Urlaub fährt und somit eigentlich keine Eile hat.

8.3 Freigabe nur in den Nachbar-Bahnhof

Die Freigabe von einem Bahnhof könnte theoretisch auch in einen ganz anderen Bahnhof als einen der Nachbar-Bahnhöfe erfolgen. Damit wäre der Spielraum für das Freigeben größer, denn es würden dann grundsätzlich mehr Bahnhöfe zur Verfügung stehen. Jedoch haben wir uns bewusst gegen diese Idee/Strategie entschieden, da das Freigeben von einem Bahnhof immer auch mit dem Verlust von Zeit verbunden ist, in welcher der Zug (welche zum Freigeben verwendet wird) in den anderen Bahnhof gefahren wird, denn dieser kann währenddessen nicht genutzt werden. Aus diesem Grund beschränkt sich unsere Strategie nur auf die Nachbar-Bahnhöfe. Dies hat vor allem aber auch den Grund, dass das direkte Tauschen/Wechseln von Zügen unabhängig von den Kapazitäten der Strecke oder dem Bahnhof möglich ist (siehe 5.1.3 Freigeben einer Station).

8.4 Dauer des Algorithmus

Die Dauer der Berechnung und Erstellung eines Fahrplans hängt bei diesem Problem von vielen Faktoren ab und wurde bei uns auch bei den ersten Überlegungen bezüglich den ersten Ideen bzw. der Ideensammlung nachgiebig berücksichtigt. Eine Software, welche z.B. die kürzeste Route mittels Backtracking bzw. Brute-Forcing ermittelt, schlossen wir natürlich bereits im Vorhinein aus, da diese Methode weder empfohlen wird und vor allem bei einer kurzen Berechnung der Schleifendurchläufe bezüglich der Komplexität von $O(N^N)$ im Worst-Case, bei z.B. 200 Bahnhöfen, würde $200^{200} \rightarrow \infty$ gegen unendlich gehen. Damit ist diese Methode z.B. schon ab 200 Bahnhöfen für diese Software nicht mehr realistisch einsetzbar, da wir quasi gegen unendlich lange für die Berechnung der kürzesten Route im Worst-Case benötigen würden, für einen Zug bzw. Passagier. Schon bei $800 * 200^2$ Schleifendurchläufen (getestet in Python auf einem Computer mit durchschnittlicher Leistung) beansprucht die Ausführung einige Sekunden. Die Optimierung bzw. Festlegung der Priorität bezüglich der Dauer der Berechnung war also ein wichtiger Aspekt bei unserer Entwicklung. Eine Software mit der Aufgabe einen Fahrplan zu berechnen und zu optimieren, hat normalerweise bei z.B. der Deutschen Bahn oder ähnlichen Organisationen natürlich Tage dazu Zeit bzw. wird darauf festgelegt, einen Fahrplan möglichst optimiert z.B. erst in der nächsten Woche fertigzustellen. Jedoch gibt bzw. sollte es auch Software geben, welche in kritischen Situationen wie z.B. bei plötzlich eintretenden Ereignissen (wie z.B. bei einer blockierten Strecke durch einen Unfall) in möglichst kurzer Zeit, Tage sind da z.B. definitiv zu viel, einen neuen Fahrplan erstellt, welcher dann an die Ereignisse angepasst ist bzw. die neuen Umstände bei dem neuen Fahrplan miteinbezogen hat. Wir haben uns zum Ziel gemacht, mit unserer Software auch eine solche Anforderung zu erfüllen bzw. eine solche Software bieten zu können.

9 Ausblick

Die von uns entwickelte Software kann auf Basis dem von uns entworfenen bzw. ausgedachten Algorithmus und den Eingabeparametern entsprechend der Aufgabenstellung, selbstständig einen gesamten Fahrplan berechnen und optimieren. Dabei spielt die Anzahl der Passagiere, der Bahnhöfe oder der Strecken sowie Züge keine Rolle, abgesehen von der nötigen Berechnungsdauer natürlich.

Jedoch ist diese Software natürlich (noch) nicht perfekt und kann sowohl bei dem Kriterium der Berechnungsdauer bzw. Komplexität sowie der Gesamtverspätung der Passagiere an der ein oder anderen Stelle natürlich noch verbessert/optimiert werden. Auch möglich ist, die gesamte Software mit ihren Bestandteilen so weiterhin zu nutzen, jedoch den Grundalgorithmus vollständig umzukrempeln. Dafür haben wir z.B. stetig auf eine ausgewogene Unabhängigkeit zwischen den Modulen/Klassen geachtet. Eine mögliche Idee für eine geeignete Weiterentwicklung wäre z.B. eine Erweiterung in der Klasse `Travel_Center` zu implementieren, welche das Einsteigen von Passagieren in einen durch einen Bahnhof fahrenden Zug prüft bzw. tätigt. Eine andere dazu passende Erweiterung wäre das Umleiten von Zügen, welche bereits auf einer Strecke fahren, um während dem Transportieren von Passagieren noch weitere Passagiere (wenn möglich) zusätzlich aufzunehmen (von einem Zwischenbahnhof), wenn damit die gewünschte Ankunftszeit der bisherigen Passagiere nicht überschritten wird und die neuen Passagiere dasselbe Ziel haben. Dies sind alles Ideen, welche auf Basis unserer Software leicht umsetzbar sind, denn abgesehen von der Unabhängigkeit der Software-Komponenten, haben wir auch auf eine sehr gründliche und anständige (Entwickler-)Dokumentation des Programmcodes geachtet (siehe 6.4 Programmierstil). Es ist möglich, sich eine solche Dokumentation für unsere Software generieren zu lassen und diese im Internet-Browser bequem zu begutachten (siehe 3. Installation).

10 Fazit

Die Verspätung der Züge ist ein Problem, das den Schienenverkehr seit vielen Jahren begleitet. Die Gründe für Zugverspätungen liegen sowohl in geplanten als auch in ungeplanten Verzögerungen. Die unplanmäßige Verzögerungen, z. B. durch schlechtes Wetter, sind unvorhersehbar, während die planmäßige Verspätungen durch optimierte Zeitplanung und Algorithmen minimiert werden können. Im Rahmen des Wettbewerbs InformatiCup wurden die geplanten Verspätungen beobachtet/behandelt.

Der Wettbewerb vereinfacht den realen Eisenbahnverkehr auf ein Modell, das aus Bahnhöfen, Strecken, Zügen und Passagieren besteht. Das Ziel des Wettbewerbs ist es, die Gesamtdifferenz, d.h. die Gesamtverspätung, zwischen der gewünschten und der tatsächlichen Ankunftszeit der Passagiere zu minimieren. Vor diesem Hintergrund haben wir einen Algorithmus entwickelt, der die Planung von Zügen mit den kleinen Verspätungen ermöglicht. Auf der Softwareebene haben wir eine Software entwickelt, die Modellinformationen gemäß dem Eingabeformat parsen, den Algorithmen zur Bahnplanung aufrufen und den Fahrplan in einem bestimmten Format ausgeben kann. Darüber hinaus haben wir das Ergebnis des Algorithmus und seine Performance mit verschiedenen Hintergründen analysiert und diskutiert.

Der Algorithmus und vorallem die Ergebnisse dieser Auswertungen waren für uns relativ zufriedenstellend. Wir verfügten am Ende über eine Software, welche die zuvor an sie gestellten Anforderungen nun erfüllen konnte. Eine dieser war z.B. die Ausführungszeit, wir hatten uns nämlich vorgenommen, dass bei der Berechnung von auch einigen Hundert Bahnhöfen und Strecken sowie Passagieren, die Berechnung von einem Fahrplan einige Minuten bzw. definitiv nicht eine Stunde oder länger andauert. Diese Anforderung konnte unsere Software, so haben wir es in der Auswertung festgestellt, erfüllen. Auch die Gesamtverspätung hielt sich in Grenzen, jedoch gab es im Gegensatz zur Ausführungszeit, wesentlich mehr Ausschweifungen, also neben vorwiegend geringen Gesamtverspätungen, auch verhältnismäßig große Gesamtverspätungen, welche durchaus auch schwieriger hinzunehmen waren. Diese hatten ihre Ursache hauptsächlich in der anfokusierten Entwicklung eines vor allem validen Fahrplan-Berechners, wir haben, wie in der Diskussion beschrieben, ja doch einige Aspekte bezüglich der Optimierung ausgelassen bzw. weniger beachtet. Trotzdem waren auch mit den im Durchschnitt ausgewerteten Gesamtverspätungen relativ zufrieden. Außerdem haben wir auch den Aspekt der Weiterentwicklung, wie im Ausblick beschrieben, sehr stark beachtet und bei unserer Implementierung der Software auch miteinbezogen. Unser Ziel war nämlich auch die Software in einer Weise zu implementieren, sodass diese von Dritten leicht weiterentwickelt und verbessert werden kann.

11 Eigenständigkeitserklärung

Hiermit erklären wir, dass wir die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

Alle sinngemäß und wörtlich übernommenen Textstellen aus fremden Quellen wurden kenntlich gemacht.

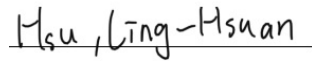
Berkehan Ünal,



Han Yang,



Ling-Hsuan Hsu,



Lukas Dreyer,



Hannover, den 13. Januar 2022

Literatur

- [1] Deutsche Bahn AG. Erläuterung Pünktlichkeitswerte November 2021. https://www.deutschebahn.com/de/konzern/konzernprofil/zahlen_fakten/puenktlichkeitswerte-1187696? Zugegriffen: 2022-01-03.
- [2] Baeldung CS. Comparison Between Uniform-Cost Search and Dijkstra's Algorithm. <https://www.baeldung.com/cs/uniform-cost-search-vs-dijkstras>. Zugegriffen: 2022-01-04.
- [3] Sustainable Finance Berlin Deutsche Bahn AG, Investor Relations. Deutsche Bahn Integrierter Bericht 2020. 2021.
- [4] Mark Dingler, Amanda Koenig, Sam Sogin, and Christopher PL Barkan. Determining the causes of train delay. In *AREMA Annual Conference Proceedings*, 2010.
- [5] GeeksForGeeks. A* search algorithm. <https://www.geeksforgeeks.org/a-search-algorithm/>. Zugegriffen: 2022-01-07.
- [6] GeeksForGeeks. Bellman ford algorithm (simple implementation). <https://www.geeksforgeeks.org/bellman-ford-algorithm-simple-implementation/>. Zugegriffen: 2021-11-26.
- [7] GeeksForGeeks. Timsort. <https://www.geeksforgeeks.org/timsort/>. Zugegriffen: 2021-11-30.
- [8] Zülfükar Genc. Ein neuer Ansatz zur Fahrplanoptimierung im ÖPNV: Maximierung von zeitlichen Sicherheitabständen. <https://kups.ub.uni-koeln.de/950/>. Zugegriffen: 2021-11-20.
- [9] Hackernoon. Timsort — the fastest sorting algorithm you've never heard of. <https://hackernoon.com/timsort-the-fastest-sorting-algorithm-youve-never-heard-of-36b28417f399>. Zugegriffen: 2021-11-26.
- [10] iq.opengenus.org. Dijkstra's algorithm: Finding shortest path between all nodes. <https://iq.opengenus.org/dijkstras-algorithm-finding-shortest-path-between-all-nodes/>. Zugegriffen: 2022-01-07.
- [11] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series. p. 85 - 103. Plenum Press, New York, 1972.
- [12] Ratthaphong Meesit and John Andrews. Ranking the critical sections of railway networks. *Proceedings of the Institution of Mechanical Engineers, Part F: Journal of Rail and Rapid Transit*, 235(3):361 – 376, 2021.
- [13] Nishkarsh. Complexity cheat sheet for python operations. <https://www.geeksforgeeks.org/complexity-cheat-sheet-for-python-operations/>. Zugegriffen: 2022-01-08.

- [14] Uni Paderborn. 16. Minimale Spannbäume. https://cs.uni-paderborn.de/fileadmin/informatik/fg/ti/Lehre/SS_2017/DuA/16.pdf. p. 21.
Zugegriffen: 2022-01-06.
- [15] Uni Paderborn. 16. Minimale Spannbäume. https://cs.uni-paderborn.de/fileadmin/informatik/fg/ti/Lehre/SS_2017/DuA/16.pdf. p. 39.
Zugegriffen: 2022-01-06.
- [16] Uni Paderborn. 16. Minimale Spannbäume. https://cs.uni-paderborn.de/fileadmin/informatik/fg/ti/Lehre/SS_2017/DuA/16.pdf. p. 11 - 20.
Zugegriffen: 2022-01-06.
- [17] Uni Paderborn. 16. Minimale Spannbäume. https://cs.uni-paderborn.de/fileadmin/informatik/fg/ti/Lehre/SS_2017/DuA/16.pdf. p. 20.
Zugegriffen: 2022-01-06.
- [18] Anushka Chandrababu A. Abhilasha G. N. Srinivasa Prasanna R. Sanat, Tarun Dutt. Optimizing schedule of trains in context of a large railway network. <https://ieeexplore.ieee.org/document/8569410>. Zugegriffen: 2021-11-24.
- [19] Stuart Russell and Peter Norvig. Artificial intelligence: A modern approach (4th edition) (pearson series in artificial intelligence). p. 191 - 196. Pearson, 2020.
- [20] Stuart Russell and Peter Norvig. Artificial intelligence: A modern approach (4th edition) (pearson series in artificial intelligence). p. 72 - 74. Pearson, 2020.
- [21] Towards Data Science. Search Algorithm: Dijkstra's Algorithm and Uniform-Cost Search, with Python.
<https://towardsdatascience.com/search-algorithm-dijkstras-algorithm-uniform-cost-search-with-python-ccbee250ba9>. Zugegriffen: 2022-01-07.
- [22] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. Introduction to algorithms. p. 85 - 103. The MIT Press, 2009.
- [23] Uni Tübingen. 7. Graphenalgorithmen. http://www.ra.cs.uni-tuebingen.de/lehre/uebungen/ss05/Algorithmen/Algorithmen_2005_Kap_07_Graphen.pdf.
Zugegriffen: 2022-01-07.