Entwicklung verschiedener Algorithmen zur Berechnung optimaler Zugfahrpläne im Rahmen des InformatiCup 2022

Timo Heiß

Nick Hillebrand

Moritz Schlager

Jonas Zagst

DHBW Ravensburg

15.01.2022

„The problem is not the problem; the problem is your attitude about the problem.”

Captain Jack Sparrow

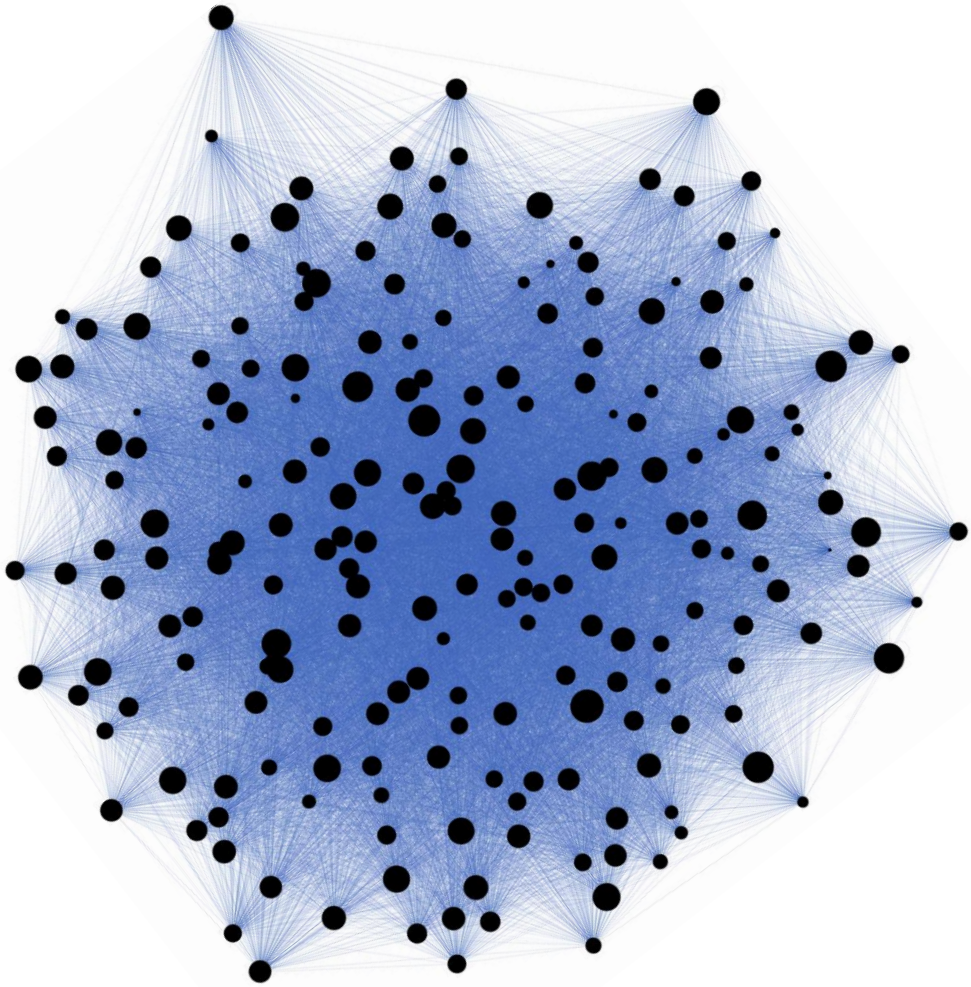

Abbildung 1: Visualisierung des Schienennetzes „large“ (GitHub InformatiCup 2022)

Inhaltsverzeichnis

Abkürzungsverzeichnis

SDI	Simple Dijkstra Algorithm
SPP	Simple Passenger Parallelization Algorithm
APP	Advanced Passenger Parallelization Algorithm
STP	Simple Train Parallelization Algorithm

Abbildungsverzeichnis

Listingverzeichnis

1. Einleitung

In Zeiten des Klimawandels spielen nachhaltige Alternativen auch beim Thema Mobilität zunehmend eine wichtige Rolle. Es ist gemeinhin bekannt, dass die Bahn – im Vergleich zu PKW oder Flugzeug – ein deutlich umweltfreundlicheres Verkehrsmittel darstellt. Dennoch präferieren viele Menschen immer noch das Auto, denn vielerorts ist die Bahn eine wenig attraktive Alternative. Die Gründe hierfür sind vielseitig und reichen von hohen Ticketpreisen bis hin zu schlecht ausgebauten Schienennetzen in den ländlicheren Regionen des Landes. Darüber hinaus sind es aber auch die häufigen Verspätungen oder sogar Ausfälle von Zügen, die der Attraktivität der Bahn in Deutschland schaden.

Insbesondere bei zuletzt genanntem Punkt kann nun von softwaretechnischer Seite unterstützt werden. Mit informationstechnischen Mitteln können Zugfahrpläne unter Berücksichtigung der Kapazitäten des Schienennetzes so gestaltet werden, dass die Züge zum gewünschten Zeitpunkt am richtigen Ort ankommen. Durch die Entwicklung möglichst optimaler Fahrpläne wird dann wiederum die Attraktivität der Bahn als Verkehrsmittel gesteigert.

Die vorliegende Arbeit zeigt einen Weg auf, wie möglichst optimale Zugfahrpläne für verschiedene Schienennetze berechnet werden können und stellt unterschiedliche, zu diesem Zweck entwickelte Algorithmen vor.

1.1 Problemstellung

Die in dieser Arbeit vorgestellte Methodik und die zugehörigen Algorithmen wurden im Rahmen des InformatiCups 2022 entwickelt. Aufgabe des Wettbewerbs war es eine Software zu entwickeln, die – basierend auf einer Textdatei mit der Problemstellung – einen Fahrplan erstellt. Dieser soll in der Hinsicht optimiert werden, dass die Gesamtverspätung aller Passagiere möglichst gering ist (vgl. InformatiCup 2022, S.1). Um die Gültigkeit des Fahrplans sicherzustellen, wurde im GitHub Repository des Wettbewerbs (GitHub InformatiCup 2022) ein auf der Programmiersprache GO basierendes Testing-Tool zur Verfügung gestellt. Auf Basis einer Input- und Output-Textdatei werden Syntax und Berechnung der Boarding-, Abfahrts- und Ankunftszeiten validiert.

Das Modell, auf das sich auch die nachfolgenden Ausführungen beziehen, sieht dabei wie folgt aus. Im Schienennetz gibt es vier Elemente: Bahnhöfe (nachfolgend auch Stationen genannt),

Strecken, Züge und Passagiere. Alle Aktionen in diesem Schienennetz werden in Runden durchgeführt, die damit als fiktive Zeiteinheiten zu betrachten sind (vgl. InformatiCup 2022, S.2).

An Stationen können Passagiere ein- und aussteigen. Neben Passagieren können sich an Stationen auch Züge befinden. Allerdings ist die maximale Zugkapazität eines Bahnhofs beschränkt. Ähnlich verhält es sich mit den Strecken. Diese stellen Verbindungen zwischen zwei Bahnhöfen dar und werden neben einer maximalen Zugkapazität durch eine Länge spezifiziert. Züge können sich auf den Strecken bewegen und somit Passagiere transportieren. Dabei haben die Strecken keine Richtung, können also von beiden Seiten befahren werden. Jeder Zug erhält darüber hinaus eine Passagierkapazität sowie eine Geschwindigkeit. Wie die Geschwindigkeit mit der Rundenzeit und der Länge einer Strecke zusammenspielt, wird im Abschnitt „Annahmen bei der Berechnung“ detailliert erläutert. Eine weitere Besonderheit bei den Zügen sind „Wildcards“. Das sind Züge, deren Positionen zu Beginn noch nicht festgelegt sind und beliebig gewählt werden können. Zuletzt gibt es im Schienennetz noch Passagiere, welche neben einer Gruppengröße einen Zielbahnhof sowie eine Zielzeit besitzen. Kommt ein Passagier später am Zielbahnhof an, so ist er verspätet. Aufgabe ist es nun, diese Verspätungen zu minimieren (vgl. InformatiCup 2022, S.2ff.).

Um dies bestmöglich zu erreichen, gibt es viele Aspekte zu berücksichtigen. Die Verteilung der Passagiere auf die verschiedenen Züge, die Wahl des Weges zum Ziel, die Platzierung frei positionierbarer Züge, sowie das Vermeiden von Kapazitätskonflikten sind nur einige interessante Herausforderungen, die diese zunächst so einfach wirkende Aufgabenstellung mit sich bringt. All diese Aspekte muss in der Entwicklung der hier vorgestellten Software-Lösung für die Erstellung von Zugfahrplänen berücksichtigt werden.

1.2 Lösungsansatz und Inhalt der Arbeit

An dieser Stelle soll nun kurz die Grundidee der entwickelten Lösung betrachtet werden. Diese basiert auf der Annahme, dass es aber einer gewissen Komplexität nicht möglich ist, einen „perfekten“ Algorithmus für alle Input-Probleme zu finden, da eine Simulation aller möglicher Szenarien nicht in annehmbarer Zeit lösbar wäre. Stattdessen kann sich nur durch unterschiedliche Ansätze Deshalb war es auch nie das Ziel, einen einzigen Algorithmus zu entwerfen. Stattdessen sollten mehrere unterschiedliche Algorithmen entwickelt werden, die dann gemeinsam in einer Algorithmensammlung angewendet werden.

Die Ansätze der Algorithmen können dabei so vielseitig sein wie die verschiedenen Input-Probleme selbst. Für unterschiedliche Input-Probleme sind es dann jeweils auch verschiedene Algorithmen, die die beste Lösung liefern. Trotzdem sollen alle entwickelten Algorithmen verwendet

werden können. Sie sollen jeweils instanziiert und angewandt werden können, so dass es möglich ist ein perfekt abgestimmtes Orchester an Algorithmen zur Lösung der Aufgabe einzusetzen. Am Ende wird als Output der Fahrplan mit der geringsten Gesamtverspätung ausgewählt. Dieser Lösungsansatz ist damit nicht nur für verschiedenste Input-Probleme geeignet, sondern ist gleichzeitig auch optimal erweiterbar und skalierbar.

Die vorliegende Arbeit stellt nun den zuvor skizzierten Lösungsansatz vor. In Kapitel 2 werden zunächst die theoretischen Grundlagen vermittelt, die für das Verständnis der praktischen Umsetzung der Lösung essenziell sind. Das umfasst einige Besonderheiten bei den Berechnungen sowie die Vorstellung des Dijkstra-Algorithmus, der mit dem „Kürzester-Weg-Problem“ eine zentrale Herausforderung der Aufgabenstellung löst und deshalb in allen entwickelten Algorithmen verwendet wird.

Kapitel 3 der vorliegenden Arbeit beschäftigt sich mit dem Programmentwurf der Lösung. Dabei wird auf die Besonderheiten des Input- und Output-Parsings eingegangen, die den Rahmen der Lösung bilden. Weiter wird ein Interface vorgestellt, dass von allen Algorithmen implementiert wird und diesen damit eine definierte Grundstruktur vorgibt. Anschließend wird das Exception Handling in der Softwarelösung betrachtet. Schlussendlich stellt das Kapitel noch den Input Generator vor, der eigens dafür entworfen wurde, eigene Testdaten schnell und effizient in großem Umfang zu generieren.

In Kapitel 4 werden schließlich die Sammlung aller entworfenen Algorithmen selbst detailliert betrachtet. Dabei wird jeweils die Idee und Umsetzung erklärt und jeweils bewertet wo die Stärken und Schwächen des verfolgten Ansatzes liegen. Außerdem werden in einem abschließenden Unterkapitel Ideen für weitere Lösungsansätze genannt und erläutert.

Im letzten Kapitel werden die Ergebnisse zusammengefasst und ein Fazit zur Lösung gezogen. Anschließend wird ein Ausblick mit weiterführenden Optimierungsideen für den entwickelten Lösungsansatz als Gesamteinheit gegeben. Zuletzt soll noch die Übertragbarkeit des Ansatzes auf die Realität geprüft und bewertet werden, um dann mit einer kurzen Reflexion des Entwicklungsprozesses zu schließen.

2. Theoretische Grundlagen

Die in Kapitel 1.1 bereits erläuterte Problemstellung wartet, im Detail betrachtet, mit vielen Herausforderungen in der Umsetzung auf. Im Folgenden sollen Berechnungsgrundsätze und theoretische Erklärungen zu Teilaspekten der Problemstellung erläutert werden, die in allen entwickelten Algorithmen Anwendung finden.

2.1 Annahmen bei der Berechnung

In diesem Kapitel werden die zentralen Berechnungen vorgestellt, die von allen Algorithmen der Softwarelösung genutzt werden, um gültige Fahrpläne zu berechnen. Diese Annahmen basieren ausschließlich auf den Vorgaben der Problemstellung (vgl. InformatiCup 2022) und haben deshalb nur eine konkrete Gültigkeit für den InformatiCup 2022.

2.1.1 Berechnung von Ankunftszeiten

Eine korrekte Berechnung der Abfahrts- und Ankunftszeiten einzelner Züge im Fahrplan ist unbedingt erforderlich, um einen gültigen Fahrplan bereitstellen zu können. Für die Kalkulation der Zeitdifferenz zwischen Abfahrt und Ankunft werden zunächst zwei Parameter benötigt. Die *Geschwindigkeit des Zugs (v)* und die *Länge der Strecke (s)*. So kann nach der einfachen Formel $t = \frac{s}{v}$ ermittelt werden, wie lange der Zug für das Befahren der Strecke benötigt.

Da gilt $s, v \in \mathbb{Q} \mid s, v > 0$ folgt damit, dass auch $t \in \mathbb{Q} \mid t > 0$ bei der Berechnung gilt. Es wird jedoch nach Vorgabe der Problemstellung nur in ganzen Zeiteinheiten gerechnet, weshalb hier immer aufgerundet wird, sodass gilt $t \in \mathbb{N}$. In Python kann dies mit der Funktion *math.ceil(x)* umgesetzt werden. Die so errechnete Zeitdifferenz kann nun genutzt werden, um basierend auf der Abfahrtszeit den Ankunftszeitpunkt zu berechnen. Dabei gilt *Abfahrtszeit + (t-1) = Ankunftszeit*. Hier lässt sich leicht erkennen, dass für *t=1* gilt *Abfahrtszeit = Ankunftszeit*. Dies bedeutet also, „dass ein Zug in derselben Runde an einem Bahnhof ankommt, in der er von einem anderen losgefahren ist“ (InformatiCup 2022, S.3). Fährt der Zug nun direkt weiter muss jedoch beachtet werden, dass *Abfahrtszeit > Ankunftszeit* gelten muss, dass der Fahrplan gültig ist, da pro Objekt nur eine Aktion zu einem Zeitpunkt definiert werden darf. Somit gilt: wenn $t_{Abfahrt} + (t_{Differenz} - 1) = t_{Abfahrt}$ dann $t_{Ankunft} = (t_{Abfahrt} + 1)$. Auch wenn Passagiere an der Station mit dem Zug interagieren sollen, muss beachtet werden, dass für das Ein- und Aussteigen jeweils eine Zeiteinheit benötigt wird, in der der Zug steht.

Handelt es sich jedoch um eine längere Strecke und der Zug fährt über mehrere Stationen hinweg ohne Passagiere ein- oder aussteigen zu lassen kann es vorkommen, dass die in einer Zeiteinheit zurückgelegte Strecke bis zu doppelt so lang ist, also sich demnach in bestimmten Fällen die Geschwindigkeit verdoppeln kann. Ersichtlich wird dies an folgendem Beispiel:

```
# Bahnhöfe: str(ID) int(Kapazität)
[Stations]
S1 10
S2 10
S3 10
S4 10

# Strecken: str(ID) str(Anfang) str(Ende) dec(Länge) int(Kapazität)
[Lines]
L1 S1 S2 4 1
L2 S2 S3 5 1
L3 S3 S4 1 1

# Züge: str(ID) str(Startbahnhof)/* dec(Geschwindigkeit) int(Kapazität)
[Trains]
T1 S1 2 50

# Passagiere: str(ID) str(Startbahnhof) str(Zielbahnhof) int(Gruppengröße)
int(Ankunftszeit)
[Passengers]
P1 S1 S4 50 10

Visualisiertes Schienennetz:
S1---(L1/s=4)---S2---(L2/s=5)---S3---(L3/s=1)---S4

Es ergibt sich damit der folgende Fahrplan:
1 - P1 Board T1
2 - T1 Depart L1 (zum Ende der Runde bei 2/4 auf L1)
3 - T1 Depart L2 (zum Ende der Runde bei 4/4 auf L1 und kann, da keine
    Passagiere aus- oder einsteigen müssen direkt abfahren und ist demnach
    schon bei 2/5 auf L2)-> T1 hat sich mit v*2 bewegt)
4 - (T1 ist zum Ende der Runde bei 4/5 auf L2)
5 - T1 Depart L3 (T1 kommt während der Runde bei S3 an, kann deshalb direkt
    abfahren und kommt noch zum Ende von Runde 5 bei S4 an)
6 - Detrain P1
```

Listing 1: Beispielhafte Demonstration der Berechnung von Ankunftszeiten

2.1.2 Swapping

Ein weiterer zentraler Aspekt der Problemstellung ist der Umgang mit begrenzten Kapazitäten an Stationen, Strecken, aber auch in Zügen, welche nur eine begrenzte Anzahl an Passagieren aufnehmen können. Beim *Swapping* handelt es sich um eine Lösung für beschränkte Kapazitäten in Bahnhöfen. Dabei wird die Tatsache ausgenutzt, dass „*während* einer Runde die Kapazitäten kurzzeitig überschritten werden dürfen, solange *am Ende* jeder Runde sämtliche Kapazitäten eingehalten werden“ (InformatiCup 2022, S.4).

Gegeben sei folgender Input:

```
# Bahnhöfe: str(ID) int(Kapazität)
[Stations]
S1 1
S2 1

# Strecken: str(ID) str(Anfang) str(Ende) dec(Länge) int(Kapazität)
[Lines]
L1 S1 S2 4 1

# Züge: str(ID) str(Startbahnhof)/* dec(Geschwindigkeit) int(Kapazität)
[Trains]
T1 S1 2 50
T2 S2 2 50

# Passagiere: str(ID) str(Startbahnhof) str(Zielbahnhof) int(Gruppengröße)
int(Ankunftszeit)
[Passengers]
P1 S1 S2 50 10

Visualisiertes Schienennetz:
S1---(L1/s=4)---S2
```

Listing 2: Beispielhafter Input, der Swapping erforderlich macht

Zunächst erscheint es als könnte P1 in dieser Ausgangsposition nicht durch T1 transportiert werden, da die Zielstation bereits voll belegt ist und sich auch der Strecke nur ein Zug gleichzeitig befinden kann. Auch durch eine Verschiebung aller Züge auf die nächste Station könnte hier keine Lösung gefunden werden, da dies in einer „Sackgasse“ enden würde. Es kann nun jedoch das *Swapping* zum Einsatz kommen. Grundgedanke ist dabei der Folgende: in der letzten Zeiteinheit, bevor ein Zug an der Station ankommt, fährt der Zug, welcher die Zielstation blockiert, los. Damit gibt es <u>während</u> der letzten Runde vor der Ankunft eine kurzzeitige Kapazitätsüberschreitung, da sich beide Züge auf der Strecke befinden. Am Ende der Runde sind die Positionen jedoch vollständig getauscht worden, womit dann alle Kapazitätsbeschränkungen wieder eingehalten werden und damit der Fahrplan gültig ist. Im konkreten oben gezeigten Beispiel würde der Fahrplan also wie folgt aussehen:

```
[Train:T1]
2 Depart L1

[Train:T2]
3 Depart L1

[Passenger:P1]
1 Board T1
4 Detrain
```

Listing 3: Fahrplan, der Swapping nutzt

Zur weiteren Veranschaulichung wurde das folgende Schaubild (Abbildung 2) erstellt. Dabei sind die Zeilen der Tabelle die Zeitpunkte und die Spalten stellen die Position der Züge T1 und T2 jeweils zu Beginn (grün) und Ende (blau) der Runde an.

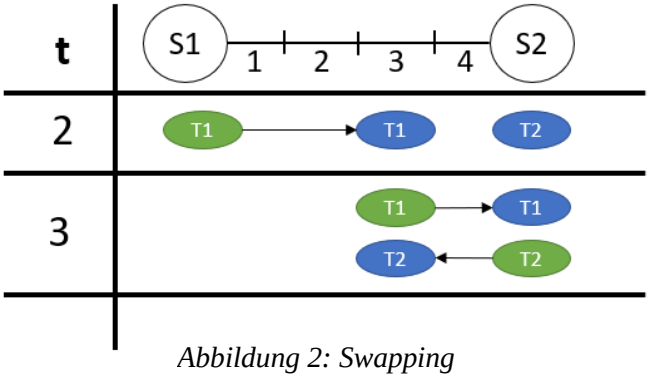

Abbildung 2: Swapping

Mit diesem Ansatz kann die Verschiebungsproblematik umgangen werden. Oft ist es theoretisch möglich auch ohne *Swapping* allein durch das Verschieben einer Menge von Zügen eine freie Station zu schaffen. Allerdings macht dies bei einem klassischen Ansatz komplexe rekursive Berechnungen unumgänglich in denen die hypothetische Kapazität potenzieller Zielstationen auswertet wird. Dabei müssen ebenfalls noch Zeitaspekte aufgrund unterschiedlichster Streckenlängen betrachtet werden, um in die beste Lösung zu finden. Damit ist weder in den Punkten Fehleranfälligkeit noch Laufzeit eine konkurrenzfähige Lösung zum *Swapping* gegeben.

2.2 Dijkstra-Algorithmus

Ein zentrales Problem der Aufgabenstellung ist das Finden des kürzesten Weges im Schienennetz zum gewünschten Ziel. Um einen solchen kürzesten Pfad zu finden, wurden bereits unterschiedlichste Algorithmen entwickelt. Zu den bekanntesten gehören neben dem *Bellman–Ford Algorithmus (*Stotz 2013) und *Floyd–Warshall Algorithmus (*Voroncovs 2015), auch der *Dijkstra-Algorithmus*. Dieser lässt sich vergleichsweise leicht auf andere Problemstellungen übertragen. Zudem löst er das Problem, den kürzesten Weg zwischen zwei Orten zu finden, sehr effizient und schnell (*Time Complexity:* $O(|\mathcal{N}|^2)$ (Voll 2014, S.33)) mit einer optimalen Lösung. Deshalb wird der Algorithmus auch für verschiedenste andere Problemstellungen verwendet, die nicht zwingend dem Grundproblem entsprechen müssen (vgl. Mönius et al. 2021, S.45-46).

Für das Problem der effizienten Fahrplanerstellung wird nur der kürzeste Pfad vom Startpunkt zum Ziel und nicht die kürzesten Pfade zu allen Punkten benötigt, um Züge effektiv einplanen zu können. Zudem ist in diesem realen Beispiel jede Streckenlänge immer eine positive Zahl. Der Dijkstra-Algorithmus hat eben diese Einschränkungen (vgl. Mönius et al. 2021, S.40-46) und wurde deshalb für die hier behandelte Lösung in allen Algorithmen implementiert.

Für den Dijkstra-Algorithmus werden die Stationen und Strecken in einem Graph-Objekt dargestellt, in dem Entfernungen zwischen zwei Knoten als Kantenlängen angegebenen werden. Im Folgenden werden diese Längen, wie auch in der Literatur, als Kantenkosten bezeichnet. Der Algorithmus versucht in seiner ursprünglichen Form immer den Weg mit den niedrigsten Kantenkosten zu den anderen Punkten des Graphen zu finden. (vgl. Velden 2014).

Jedem Knoten wird ein Attribut, das die Distanz zum Ausgangspunkt beschreibt, zugewiesen. Dieses wird beim Durchlaufen immer wieder überschrieben, wenn ein kürzerer Weg und damit eine kleinere Distanz zum Ausgangspunkt gefunden wurden. Außerdem wird der direkte Vorgängerknoten, relativ zum Ausgangpunkt gespeichert. Bereits besuchte Wege können somit nachvollzogen werden (ähnlich zum Aufbau einer *linked list*) (vgl. Mönius et al. 2021, S.40-45). Diese Eigenschaft ist von besonderer Bedeutung. Denn ansonsten ließe sich der Algorithmus nicht auf die in dieser Arbeit betrachtete Problemstellung übertragen, da die geringste Entfernung allein nicht zur Erstellung eines Fahrplans genügt.

Im Folgenden wird die Funktionsweise des Dijkstra-Algorithmus an einem einfachen Beispiel veranschaulicht. Wenn man beispielsweise in der untenstehenden Grafik auf dem kürzesten Weg von Konten A zum Knoten E gelangen will würde der Algorithmus im ersten Durchlauf zunächst B besuchen, da dieser die niedrigste Entfernung aufweist. Knoten A wird damit als besucht markiert, B wird der Wert 4 und der Vorgänger A zugewiesen. Im nächsten Zug wird der Knoten D besucht, da dieser der nächstgelegene noch nicht besuchte Knoten ist. Somit wird B als besucht markiert und D wird der Wert 10 (Summe aus dem Wert des Vorgängerknotens und der Distanz zu dieser (4+6)) zugewiesen. Nun ist der nächstgelegene, nicht besuchte Knoten jedoch C. Ihm wird der Wert 6 und A als Vorgänger zugewiesen. Im letzten Durchlauf wählt der Algorithmus den nächsten unbesuchten Knoten E welcher von C aus günstiger zu erreichen ist. Somit ist der schnellste Weg gefunden und es wird der Wert von C mit E aufsummiert, womit 13 die minimale Entfernung ist. Es wird also immer weiter iteriert, bis der tatsächlich kürzeste Weg gefunden ist. Ein bereits gemerktes Wegstück muss dabei nicht erneut durchlaufen werden.

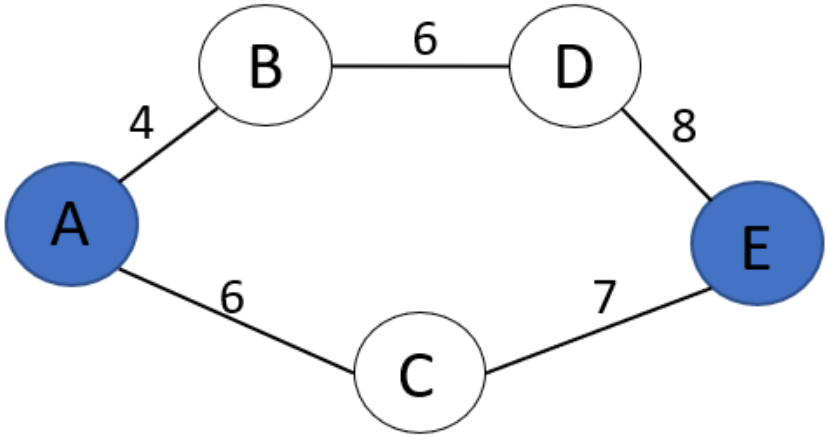

Abbildung 3: Einfaches Beispiel-Schema für den Dijkstra-Algorithmus
(in Anlehnung an Mönius et al. 2021, S.41ff.)

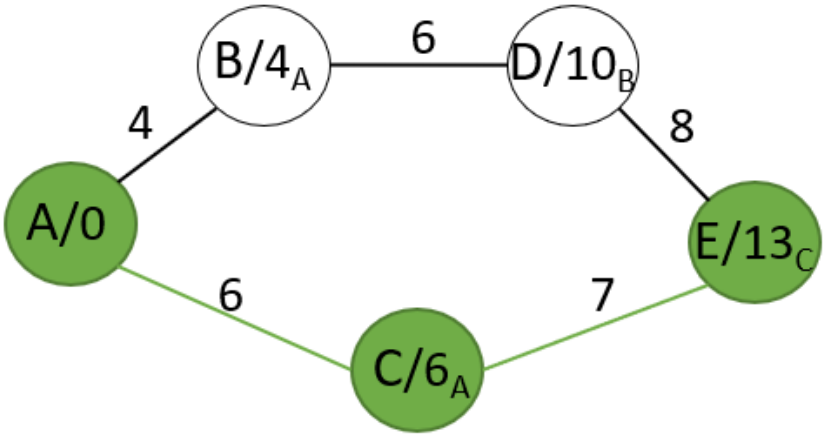

Abbildung 4: Kürzester Weg im Beispiel-Schema
(in Anlehnung an Mönius et al. 2021, S.41ff.)

Um allen Anforderungen der Problemstellung zu entsprechen, mussten jedoch einige Anpassungen gemacht werden. Als Basis wurde die Implementierung vom GitHub User *mdsrosa* gewählt (mdsrosa 2015). Zunächst wurde durch eine Modifikation des *Graph*, welcher das gesamte Schienennetz repräsentiert, und in der Initialisierung sichergestellt, dass alle Verbindungen zwischen Stationen bidirektional eingetragen sind. Außerdem wurde implementiert, dass neben einer Liste der besuchten Station und der Länge des Gesamtpfades auch eine Liste der verwendeten Strecken zurückgegeben wird. Dies macht es im Nachgang einfacher die Bewegung des Zuges durch eine Iteration durch ebendiese Liste zu simulieren.

3. Softwarearchitektur

Nachdem die theoretischen Grundlagen für den Lösungsansatz vermittelt wurden, kann nun auf die konkrete Implementierung der Idee eingegangen werden. Hierzu wurde ein *Python3-Programm* entworfen, dessen Funktionalität und Aufbau in den folgenden Subkapiteln beschrieben wird. Die Wahl auf Python als Programmiersprache fiel aufgrund der einfachen Handhabung durch dynamische Typisierung und die intuitive Erstellung mehrdimensionaler Listen. Bei der Entwicklung wurden die hierfür geltenden *coding conventions* eingehalten. Die Überprüfung erfolgte dabei durch die verwendete IDE (PyCharm). Auch wenn Python bezüglich der Performance nicht zu den besten Sprachen gehört, zeichnet es sich aus durch eine Vielzahl von Libraries, welche sich für die Anforderungen und mögliche Erweiterungen in der Zukunft besonders eignen. Vor allem für maschinelles Lernen, aber auch zur Visualisierung von komplexen Datenstrukturen, wie Zugfahrplänen, bietet Python etliche Bibliotheken, welche zukünftige Erweiterungen erleichtern. Eine weitere wichtige Python Library ist *pandas*, welche vom *STP-Algorithmus* (vgl. Kapitel 4.3) zur Erstellung und Manipulation von *DataFrames* verwendet wird. Darüber hinaus ist Python auch grundsätzlich eine mächtige Hochsprache mit gutem IDE Support, was die Entwicklung vereinfacht.

Um die Funktionalität unabhängig vom ausführenden Computer zu machen, wurde ein *Dockerfile* integriert mit dem ein unabhängig lauffähiges *Docker-Image* erstellt werden kann. Dieses kann auf jedem Computer immer gleich ausgeführt werden, dabei muss der Input beim Aufruf durch *stdin* an das Programm übergeben werden. Die Outputdatei wird bei erfolgreicher Durchführung des Programms in *stdout* ausgegeben. Dies alles geschieht in der Konsole, da bei einer bereits vorhandenen Input Datei kein Interesse besteht diese umständlich über ein GUI auszuwählen und dann die Berechnung zu starten und als Rückgabe ebenfalls schlicht eine Datei erwartet wird. Eine Benutzeroberfläche würde die Handhabung für diesen Anwendungsfall nur unnötig aufblähen und verkomplizieren. Erst wenn zusätzliche Anforderungen, wie eine dynamische Erstellung und Bearbeitung von Inputdateien durch den Inputgenerator (vgl. Kapitel 3.5) oder eine manuelle Parametrisierung der Algorithmen (vgl. Kapitel 4.3.3) zu den Anforderungen gehören kann ein Ansatz mit grafischer Benutzeroberfläche echten Mehrwert bringen.

3.1 Input Parsing

Um mit der Inputdatei, die Informationen über Züge, Strecken, Bahnhöfe und Passagiere enthält, arbeiten zu können, ist es notwendig ein Programm zu schreiben, das die Datei liest, auf Korrektheit überprüft und die Information so aufbereitet, dass sie möglichst gut in einer objektorientierten

Software verwendet werden können. Daher wurde zu Beginn ein *Input Parser* implementiert, um darauf aufbauend Algorithmen entwickeln zu können. Züge, Strecken, Bahnhöfe und Passagiere sind dabei als Klassen angelegt, die diese Objekte entsprechend gut beschreiben. Diese Klassen werden beim Lesen der Inputdatei später verwendet, um die Informationen über beispielsweise die Züge in einer Liste von Zugobjekten aufzubereiten. Der gesamte Vorgang ist in vier Methoden aufgeteilt, die jeweils eine der unterschiedlichen Klassen Stationen, Züge, Strecken und Passagiere parsen und in einer Hauptmethode aufgerufen werden. Diese vier Methoden funktionieren alle ähnlich, weshalb im Folgenden das Input Parsing ausschließlich am Beispiel der Züge näher betrachtet wird. Zunächst wird daher kurz die Klasse *Train* vorgestellt, die zur Aufbereitung der Informationen über die Züge verwendet wird. Sie hat dabei die folgenden Attribute:

```
def __init__(self, original_id, original_position, speed, capacity):
    self.id = ""
    self.original_id = original_id
    self.capacity = int(capacity)
    self.passengers = None
    self.speed = float(speed)
    self.position = ""
    self.original_position = original_position
    self.initial_position = self.position
    self.fixed_start = True
    self.journey_history = {}
```

Listing 4: Konstruktor der Train Klasse

Neben diesen Eigenschaften hat die Klasse Train auch eine *to_string()*-Methode, die die Eigenschaften des entsprechenden Zuges im gleichen Format zurückgibt, wie sie in der Inputdatei stehen. Außerdem gibt es noch eine *to_output()*-Methode, die zur Erstellung der Outputdatei genutzt wird. Diese beiden Methoden sind auch für die anderen Objekte des Schienennetzes vorhanden, lediglich die Eigenschaften unterscheiden sich.

Zunächst wird die Erstellung der Zugobjekte aus der Inputdatei im Detail betrachtet (gilt analog für die anderen Objekte). Um nun die Objekte einzulesen, wird die Methode *parse_trains()* aufgerufen. In ihr wird zunächst ein Array deklariert, das nachher die Zugobjekte enthalten soll, welche später den Rückgabewert der Methode verkörpern. Des Weiteren wird ein weiteres zusätzliches Array für Zugobjekte deklariert, um temporär Zugobjekte aufnehmen zu können, dazu später mehr.

Sobald die Schlüsselzeichenkette *[Trains]* erkannt wurde, wird versucht, alle folgenden Zeilen als Zug einzulesen. Tritt eine neue Zeile mit einer anderen Schlüsselzeichenkette auf, so werden die folgenden Zeilen ignoriert. Beim Parsing wird die aktuelle Zeile mit Hilfe der Methode *split()* anhand von Leerzeichen in ein Array von Strings zerlegt.

Dieses Array wird dann der *check_train_string()*-Methode übergeben, um zu überprüfen, ob es sich bei der aktuellen Zeile um einen gültigen String für einen Zug handelt. Wie genau die Validität des Strings festgestellt wird, wird später näher betrachtet. Ist die aktuelle Zeile ein Train-String, wird eine Instanz der Klasse Zug erstellt, indem dem Konstruktor alle vier Teilstrings der aktuellen Zeile übergeben werden. Wichtig ist hierbei, dass die Informationen über die ID und Position nicht direkt den Attributen ID beziehungsweise Position direkt zugeordnet werden, sondern den Attributen original_ID und original_position. Dies hat den Hintergrund, dass in den Algorithmen andere IDs verwendet werden (das Attribut Position kann ebenfalls eine ID, nämlich eine Bahnhofs-ID enthalten). Diese internen IDs beginnen immer mit dem Anfangsbuchstaben der Klasse (Objekte vom Typ *Train* starten also zum Beispiel immer mit einem "T") und sind durchgehend nummeriert. Dies ist unter anderem zur Nutzung des *STP-Algorithmus* notwendig. Die internen IDs werden aber erst am Ende der *parse_stations()*-Methode nach der Instanziierung der Zug-Objekte gesetzt.

Ist es weder ein Train-String noch eine andere Schlüsselzeichenkette, wird überprüft, ob es eine Leerzeile, oder ein Kommentar ist (dann würde die Zeile mit "#" beginnen). Ist dies der Fall, ist alles in Ordnung und die Zeile wird ignoriert. Trifft allerdings nichts von alldem zu, dann handelt es sich um einen Verstoß gegen das vorgeschriebene Inputformat und es wird eine *CannotParseInputException* geworfen, die dann abgefangen wird, um eine Fehlermeldung auszugeben.

Am Ende der *parse_trains()*-Methode müssen nun auch noch die intern verwendeten IDs gesetzt werden. Hierbei musste beachtet werden, dass die Wildcards nicht als ID, sondern als Wildcard behandelt (es wird "*" gesetzt) werden und daher direkt als interne ID übernommen werden.

Um die Gültigkeit des Train-Strings zu prüfen, wird im Fall des Zuges die Methode *check_train_string()* aufgerufen. Dieser Methode wird dann der nach Leerzeichen aufgeteilte String übergeben. Eine gültige Liste würde die vier Eigenschaften eines Zuges enthalten. Dies würde bedeuten, dass die Liste auch exakt vier Elemente haben müsste. Ist dies nicht der Fall wird direkt *False* zurückgegeben. Besteht der String allerdings aus vier Elementen, muss noch weiter geprüft werden, ob das Format der einzelnen Elemente korrekt ist. Dies wird überprüft, indem den Elementen in einem *try-except*-Block in die entsprechenden Typen gecastet werden, tritt dabei kein Fehler auf wird *True* zurückgegeben, tritt ein Fehler auf und es wird *False* zurückgegeben.

Für jede der vier Teilmethoden gibt es check-Methoden, die in ihrer Funktion alle der *check_train_string()*-Methode ähnlich sind. In jeder der Nebenmethoden wird die jeweilige check-Methode verwendet. Soll der *Input Parser* verwendet werden, wird *parse_input()* aufgerufen, die

alle vier Methoden aufruft und eine Liste mit allen Instanzen zurückgibt, mit der dann in den jeweiligen Algorithmen gearbeitet werden kann.

3.2 Das Interface ISolver und Algorithmus-Klassen

Die Grundidee des Lösungsansatzes auch in der softwaretechnischen Umsetzung deutlich. Damit die verschiedenen Algorithmen austauschbar und auf die gleiche Weise eingesetzt werden können, müssen diese auch gleich aufgebaut sein, also von außen betrachtet alle die grundlegend gleiche Struktur aufweisen. So können sie nicht nur gleich eingesetzt werden, der Anwender muss auch nicht die genaue Implementierung des Algorithmus kennen, sondern lediglich wie er diesen anzuwenden hat.

Um dies im Programmcode umzusetzen, bietet sich die Verwendung eines Interfaces an, welches alle Algorithmen implementieren. Damit wird vorgeschrieben, welche Methoden ein Algorithmus zwingend besitzen muss. Lediglich diese gemeinsamen Methoden werden dann vom Anwender, beziehungsweise im Output Parser, verwendet.

In diesem Fall wurde das Interface *ISolver* definiert. Genau genommen handelt es sich dabei nicht um ein Interface, da Python keine Interfaces zur Verfügung stellt, sondern um eine abstrakte Klasse, die lediglich abstrakte Methoden enthält. Trotzdem stellt sie damit ein Interface im weiteren Sinne dar, da *Pepper* Interfaces als „abstrakte Klassen, bei denen *alle* Methoden abstrakt sind“ (Pepper 2007, S. 237) bezeichnet.

In *ISolver* werden den implementierenden Klassen drei Methoden vorgeschrieben, die später allesamt im Output Parser relevant sind. Zunächst ist das die Methode *solve()*, in der die grundlegende Berechnung des jeweiligen Algorithmus durchgeführt werden soll. Diese Methode gibt einen Integer-Wert zurück, welcher die gesamte Verspätungszeit des durch den Algorithmus erstellten Fahrplans darstellt. Durch die Berechnungen in der Methode *solve()* soll die *journey_history* der unterschiedlichen Objekte vom Algorithmus befüllt werden. Die *journey_history* kommt als Instanzvariable von Passagieren und Zügen vor und speichert in Form eines *Dictionary* alle Aktionen (Board/Depart/Detrain) mit ihren jeweiligen Zeitpunkten. Die zweite Methode *get_trains_and_passengers()*, die *ISolver* vorschreibt, gibt dann eine Liste mit den Zügen und Passagieren zurück. Dadurch gelangt der Output Parser später an die Fahrplan-Daten, aus denen er den Output erstellt. Die dritte Methoden in *ISolver* ist die simple Methode *get_name()*, die lediglich den Namen des Algorithmus zurückgibt. Dieser spielt für die Benennung der Output-Dateien eine Rolle.

Damit nun alle Algorithmen diese Methoden besitzen, ist jeder Algorithmus der Lösung in einer eigenen Klasse realisiert, die das Interface *ISolver* implementiert (genau genommen: eine Klasse, die von der abstrakten Klasse *ISolver* erbt). Somit kann der Anwender einfach Instanzen der Algorithmus-Klassen erzeugen und diese so verwenden. Wie das nachfolgende Kapitel zeigen wird, wird genau das auch in der entwickelten Lösung getan.

3.3 Output Parsing

Um die Anforderungen bezüglich des Output Formats zu erfüllen, wird eine Klasse benötigt, die die Ergebnisse der Algorithmen in einen gültigen Fahrplan übersetzt. Dieser Output Parser besteht zentral aus nur einer Methode *parse_output_files()*, der eine Liste aller Algorithmen übergeben wird, welche zur Lösung der Inputdatei eingesetzt werden sollen. Diese wurden zuvor in der *Main.py* instanziiert. Dabei werden beim Aufruf der unterschiedlichen Algorithmen jeweils Kopien der vom InputParser erstellten Objekte und gegebenenfalls Parameter übergeben. Dies wird mit der Python Funktion *copy.deepcopy()*[1] umgesetzt.

Aufgabe des Output Parsers ist es nun, diese Algorithmen nacheinander auszuführen, was standardisiert mit der *solve()-Methode* geschieht, die jeder Algorithmus zwingend implementieren muss (vgl. Kapitel 3.3). Konnte diese erfolgreich durchlaufen werden, so werden die im Algorithmus genutzten Passenger- und Train-Objekte mithilfe eines Getters aufgerufen und erzeugen durch die in beiden Klassen implementierten Methode *to_output()* den Output-String. Dieser wird am Ende zu einer Outputdatei zusammengesetzt. In beiden Fällen wird dabei auf Basis der Instanzvariable *journey_history* gearbeitet, welche alle Aktionen der entsprechenden Objekte enthält und während dem Durchlaufen der *solve()*-Methode eines Algorithmus befüllt wird. Station- und Line-Objekte müssen hier nicht weiter berücksichtigt werden, da diese nur für die Berechnung notwendig sind, allerdings im Output nicht zusätzlich erwähnt werden, da in beiden Fällen alle Attribute nicht veränderbar sind.

Die *solve()*-Methode gibt zusätzlich die kumulierte Verspätungszeit aller Passagiere zurück und macht es so möglich den in dieser Hinsicht besten Algorithmus im Output Parser übergreifend festzustellen. Es wird für jeden Algorithmus eine Outputdatei nach der Namenskonvention *"output-" + solver.get_name() + ".txt"* erstellt, doch nur das an der Gesamtverspätung gemessen beste Ergebnis wird in *output.txt* geschrieben. So kann sichergestellt werden, dass aus der entwickelten

[1] c*opy.deepcopy()* konstruiert ein neues zusammengesetztes Objekt und fügt dann rekursiv Kopien der im Original gefundenen Objekte in dieses Objekt ein (vgl. Python 3 copy 2022). Damit kann sichergestellt werden, dass die Algorithmen nicht auf dieselben Objekte zugreifen. Dies ist unbedingt erforderlich, da die Algorithmen die Attribute der verschiedenen Objekte manipulieren.

Sammlung an Algorithmen immer der individuell Beste für das gegebene Problem ausgewählt wird.

Kann ein Algorithmus den gegebenen Input nicht lösen, wird eine *CannotSolveInput*-Exception geworfen, welche vom Output Parser abgefangen wird (vgl. Kapitel 3.4).

```
@contextmanager
def time_limit(seconds, msg=''):
    timer = threading.Timer(seconds, lambda: _thread.interrupt_main())
    timer.start()
    try:
        yield
    except KeyboardInterrupt:
        raise TimeoutException("Timed out for operation {}".format(msg))
    finally:
        # if the action ends in specified time, timer is canceled
        timer.cancel()
```

Listing 5: Methode zur Definition einer Begrenzung der Durchlaufzeit

Um zu verhindern, dass ein Algorithmus zu lange benötigt und damit eine zeitnahe Lösung blockiert, kann ein standardisiertes *time_limit* festgelegt werden. Dabei wird in einem separaten Thread ein Timer gestartet, der nach Ablauf des in Sekunden angegebenen Limits eine *TimeoutException* wirft, welche dann entsprechend abgefangen wird und im output.txt des jeweiligen Algorithmus mit --- *execution timed out* ---- vermerkt wird. Das Time-Limit wird in der Softwarelösung zunächst auf 600 Sekunden gesetzt.

3.4 Exception Handling

Bereits an verschiedenen Stellen der Arbeit wurden eigene Exceptions im Programmcode erwähnt. Wie im unmittelbar vorhergehenden Abschnitt beschrieben, kann der Output Parser eine *TimeoutException* werfen. Der Input Parser wirft eine *CannotParseInputException*, wenn ein ungültiger Input vorliegt, den er nicht parsen kann. Aber auch die Algorithmen selbst können Exceptions werfen. Teilweise handelt es sich dabei um interne Probleme des jeweiligen Algorithmus, die Exceptions werden dann auch direkt in der jeweiligen Algorithmus-Klasse abgefangen. Ein Beispiel hierfür zeigt Listing 6**Error! Reference source not found.**. Dort werden im *Simple Train Parallelization Algorithm* zwei Fälle abgefangen. Zunächst wird versucht, einen Passagier zu boarden und den Zug anschließend zu departen. Funktioniert eine dieser Aktionen nicht, so wird eine entsprechende Exception geworfen, welche wiederum sofort abgefangen wird. Mit diesem Vorgehen wird erreicht, dass die anschließenden Schritte (z.B. detrain) nie ausgeführt werden.

```
try:
    # depart train after boarding
    end_time = \
        self.depart_train(chosen_train, chosen_passenger.target_station,
self.time + 1)
    self.board_passenger(chosen_passenger, chosen_train)  # board passenger
on train
    # detrain passenger after arriving at target station
    self.detrain_passenger(chosen_passenger, chosen_train, end_time)
except CannotDepartTrain:
    pass
except CannotBoardPassenger:
    pass
```

Listing 6: Internes Abfangen von Exceptions im Simple Train Parallelization Algorithm

Allerdings können die Algorithmen nicht nur Exceptions werfen, die klassenintern abgefangen werden. Ein solcher Fall tritt ein, wenn ein Algorithmus ein Input-Problem schlichtweg nicht lösen kann. Dies kann insbesondere bei den stärker limitierten Algorithmen vorkommen, ist aber nicht weiter problematisch: Der Algorithmus wirft dann eine *CannotSolveInput*-Exception, welche im Output Parser abgefangen wird. Dieser wiederum setzt die Gesamtverspätung der Lösung des Algorithmus auf *sys.maxsize*[2], damit diese Lösung nicht ausgewählt wird, wenn andere Algorithmen ohne Fehler durchgelaufen sind. Hier kommt der große Vorteil dieses Vorgehens zum Tragen: Obwohl ein Algorithmus nicht lösen kann, ist das kein Problem, da die Ausnahme abgefangen wird und mit dem nächsten Algorithmus weitergemacht wird. Damit erhält man (im Idealfall) dennoch immer eine Lösung.

All die genannten, selbst geschriebenen Exceptions sind in einem gemeinsame Python-Skript *Errors.py* gesammelt und können für die Algorithmen, in denen sie benötigt werden, importiert werden. Die meisten dieser speziellen Exceptions habe jedoch keine eigene, tiefere Funktion. Sie sind also entweder schlichtweg leer oder besitzen lediglich wenige Instanzvariablen. Listing 7 veranschaulicht das anhand des Quellcodes der bereits zuvor betrachteten Exceptions *CannotDepartTrain* und *CannotBoardPassenger*.

[2] *sys.maxsize* ist eine Ganzzahl, die den Maximalwert darstellt, den eine Variable vom Typ *Py_ssize_t* annehmen kann. Dies schließt zum Beispiel die Länge von Listen, Dictionaries oder Strings ein. Im hier vorgestellten, konkreten Anwendungsfall geht es lediglich darum, eine möglichst hohe Ganzzahl zu verwenden. Da *sys.maxint* in Python 3 nicht mehr existiert, wurde hier die Konstante *sys.maxsize* gewählt, die hier ebenfalls den genannten Zweck erfüllt (vgl. Python 2 2022, Python 3 sys 2022).

```
class CannotDepartTrain(Exception):
    def __init__(self, time):
        self.time = time


class CannotBoardPassenger(Exception):
    pass
```

Listing 7: Implementierung der Exceptions CannotDepartTrain und CannotBoardPassenger

Neben eigenen Exceptions können aber auch nicht abgefangene *built-in* Exceptions auftreten. Obwohl bei der Entwicklung der Lösung intensiv getestet wurde, kann es doch noch zu Fehlern kommen, die beim Programmieren nicht vorausgesehen werden konnten. Damit nun das Programm nicht einfach abbricht, sobald ein Algorithmus eine Exception wirft, werden im Output Parser auch alle anderen Exceptions abgefangen. So kann sichergestellt werden, dass trotzdem alle Algorithmen zur Ausführung kommen und ein gültiger Fahrplan generiert wird.

3.5 Inputgenerator und manuelles Testing

Standardmäßig ist das Testen der entwickelten Algorithmen sehr aufwendig, da einzelne Inputdateien manuell geschrieben werden müssen, um diese dann für einen Test verwenden zu können. Das beschriebene Vorgehen hat allerdings offensichtliche Schwächen. So können damit nur kleine überschaubare Inputdateien erstellt werden, da viele Aspekte (z.B. doppelte Strecken, unverbundenen Bahnhöfe, etc.) im Blick behalten werden müssen. Damit ist die manuelle Erstellung von für Tests geeignete Inputdateien fehleranfällig. Außerdem können keine größeren Tests durchgeführt werden (für z.B. Laufzeitmessungen) und allgemein wäre das manuelle Erstellen mehrerer Inputdateien mühsam, da mehrere verschiedene Kombinationen getestet werden sollten (viele Bahnhöfe, provozierte Kapazitätsprobleme, usw.). Aus ebendiesen Gründen enthält die entwickelte Softwarelösung ein kleines Programm in Form einer optional ausführbaren Klasse, um diese Dateien zu generieren.

Bei seiner Ausführung liest dieser Inputgenerator zunächst einige Parameter ein, die zur Erstellung einer Inputdatei benötigt werden. Darunter fallen unter anderem die Anzahl der Bahnhöfe, die Anzahl der Passagiere, aber auch einige Spezifikationen der einzelnen Objekte, beispielsweise die maximale Kapazität eines Bahnhofs. Diese verschiedenen Attribute werden dann auf Basis der eingelesenen Spezifikationen zufallsbasiert generiert.

Wurden alle Objekte generiert, reicht es allerdings nicht aus, diese einfach in die Inputdatei zu schreiben. Zuvor muss die generierte Datei auf ihre Gültigkeit überprüft werden. Dabei wird zunächst überprüft, ob alle Bahnhöfe mindestens einmal über eine Strecke verbunden sind. Gibt es Bahnhöfe, die nicht verbunden sind, werden für diese Bahnhöfe Verbindungen generiert. Um die

Gültigkeit der Inputdatei zu wahren, müssen die Startpositionen aller Züge überprüft werden, damit die Kapazitäten der einzelnen Bahnhöfe nicht überschritten werden. Für den Fall, dass die Kapazität eines Bahnhofes überschritten wird, werden die Startpositionen einiger Züge durch "*" (Wildcard-Operator) ersetzt, so dass die Kapazität des betroffenen Bahnhofs genau eingehalten wird. Nun werden die Objekte in die Datei geschrieben und das Programm ist beendet.

Mit dem Inputgenerator steht nun also ein Tool zur Verfügung, dass es ermöglicht, einzelne Algorithmen schneller und besser auf bestimmte Inputszenarien zu testen. Ein weiteres Feature ermöglicht es, bestimmte Kombination von Parametern gleich mehrere Dateien zu generieren. Dazu wird zusätzlich die Anzahl der gewünschten Dateien eingelesen. Daraufhin werden aufgrund der einmal eingegebenen Parameter die gewünschte Anzahl an Dateien mit den gleichen Rahmenbedingungen generiert.

Außerdem besteht die Möglichkeit den Inputgenerator um eine automatisierte Generierung der dazugehörigen Outputdateien durch einen ausgewählten Algorithmus zu erweitern. Dazu muss lediglich eine kleine *for*-Schleife hinzugefügt werden, die über die generierten Dateien iteriert und diese nacheinander im Algorithmus ausführt. Von hier ist es auch nur noch ein kleiner Schritt, die generierten Ergebnisse mit Hilfe des in GO bereitgestellten Validation-Tools (GitHub Informati-Cup 2022) der Fahrpläne zu überprüfen. So könnten automatisiert Szenarien mit unterschiedlichen Parametern generiert und diese in großem Maßstab auf einen spezifischen Algorithmus validiert werden.

4. Algorithmen

Wie bereits in der Softwarearchitektur erläutert, sollen in der entwickelten Softwarelösung unterschiedliche Algorithmen, die jeweils unterschiedliche Grundgedanken und Ansätze verfolgen, auf denselben Input angewendet werden. Dieses Kapitel stellt nacheinander einige solcher Algorithmen vor.

4.1 Simple Dijkstra Algorithm

Im Folgenden wird der einfachste Algorithmus der Sammlung näher erläutert. Dieser stellt die Basis aller darauffolgenden Algorithmen da und nutzt nur einen Zug, um alle Passagiere sukzessive zu ihrem Ziel zu transportieren.

4.1.1 Idee und Umsetzung

Der *Simple Dijkstra Algorithm (SDI-Algorithmus)* ist eine sehr direkte Herangehensweise an die Problemstellung. Ziel bei der Entwicklung war es, einen soliden und einfachen Ansatz zu wählen, um jeden Input lösen zu können. Der Fokus liegt also vor allem auf der Rückgabe eines gültigen Fahrplans und nicht auf einer möglichst starken Reduzierung der Verspätungszeit.

Die Grundidee lautet dabei wie folgt: Mit einem einzigen, vorab ausgewählten Zug werden alle Passagiere nacheinander transportiert. Die Auswahl des Zugs geschieht auf Basis seiner Kapazität, da sichergestellt werden muss, dass die Zugkapazität groß genug ist, um die größte zu transportierende Passagiergruppe aufzunehmen. Handelt es sich bei dem ausgewählten Zug um einen *Wildcard-Train*, also mit noch nicht festgelegter Startposition, wird zunächst versucht, diesen an die Startstation des ersten Passagiers zu setzen. Um sichergehen zu können, dass dies möglich ist, wurde im Vorhinein die kumulierte freie Kapazität aller Stationen berechnet. Ist diese Null, wird gar nicht erst versucht, *Wildcard-Trains* zum Transport aller Passagiere einzusetzen. Sollte es nicht möglich sein einen Zug auszuwählen, wird die Variable *file_solvable* auf *False* gesetzt und damit in der weiteren Ausführung eine *CannotSolveInputException* geworfen, die dann im Output Parser aufgefangen und behandelt wird.

Wurde schließlich ein Zug ausgewählt nimmt dieser immer exakt einen Passagier auf, bringt diesen an seine Zielstation und fährt dann zum nächsten Passagier. Dieser Vorgang wird so oft wiederholt, bis jeder Passagier sein Ziel erreicht hat. Im Python Code wird dies mit einer *for*-Schleife umgesetzt, welche durch eine Liste aller Passagiere iteriert. Die Berechnung des kürzesten Weges, um den Passagier an sein Ziel zu bringen und dann zum nächsten Passagier zu navigieren, wird

dabei mit einer Implementierung des *Dijkstra-Algorithmus* umgesetzt (vgl. Kapitel 2.2). Die Methode *calculate_shortest_path()* berechnet hierbei den kürzesten Pfad zum Ziel und dessen Verlauf. Dieser Pfad wird dann mithilfe der *travel_selected_path()*-Methode Schritt für Schritt abgefahren. Dies ist notwendig, da ständig überprüft werden muss, ob gegen Kapazitätsvorschriften verstoßen wird. Gegebenenfalls wird das *Swapping* (vgl. Kapitel 2.1.2) angewandt, um solche Kapazitätskonflikte zu umgehen. Ist der Zug am Ziel angekommen, wird dies entsprechend in der *journey_history* des Objekts vermerkt. Diese dient später zur Generierung des Outputs. Falls der Zug nicht an der Zielstation ankommt und damit den Passagier nicht aufnehmen kann, wird die *CannotBoardPassenger*-Exception geworfen, die zu einem Abbruch des Algorithmus und einer entsprechenden Behandlung im Output Parser führt.

4.1.2 Bewertung

Der größte Vorteil eines so einfach gedachten Ansatzes ist die Stabilität auch bei komplexen Eingabedateien. Es kann sichergestellt werden, dass für jeden gültigen und denkbaren Input eine Lösung gefunden wird. Außerdem ist durch die geringe Komplexität in der Berechnung (vor allem dadurch begründet, dass intern keine Simulationen berechnet und verglichen werden) auch sichergestellt, dass die Lösung auch bei großen Schienennetzen (>10.000 Strecken) eine sehr kurze Berechnungszeit benötigt.

Nachteil dieses Ansatzes ist, dass es immer viele ungenutzte Kapazitäten, sowohl auf das gesamte Schienennetz, aber auch auf die individuelle Kapazität des genutzten Zuges betrachtet gibt. Dies führt in der Gesamtheit zu einer vergleichsweise hohen kumulierten Verspätungszeit.

4.2 Passenger Parallelization Algorithms

Die *Passenger Parallelization Algorithms* sind in ihrer Grundidee sehr eng an den *SDA* angelehnt. Durch vielfältige Verbesserungen in einzelnen Bereichen ist allerdings eine merkliche Verbesserung in der kumulierten Verspätung aller Passagiere bei weiterhin gültigen Fahrplänen zu bemerken. Außerdem wurde durch eine Überarbeitung der Struktur, wie beispielsweise der Auslagerung komplexer werdender Berechnungen in einzelne Methoden, die Wartbarkeit und Übersichtlichkeit deutlich verbessert. Da der *Simple Passenger Parallelization Algorithm (SPP-Algorithmus)*…

4.2.1 Idee und Umsetzung

Wie bereits beim *SDA* (vgl. Kapitel 4.1) erläutert, werden mit einem vorab ausgewählten Zug alle Passagiere nacheinander transportiert. Dieser wird im Folgenden, wie auch im Algorithmus, als

my_train bezeichnet. Dabei wurden allerdings bei der Auswahl des Zuges Änderungen zum *SDA* vorgenommen. Bei der Ausführung des *ADA* ist die Möglichkeit gegeben bei der Erstellung des Objekts einen zusätzlichen Parameter *capacity_speed_ratio* anzugeben. Dieser hat einen Einfluss auf die Auswahl von *my_train*, welcher für den Transport aller Passagiere genutzt wird. Zunächst werden aus der Liste aller Züge solche, die eine Kapazität < *biggest_passenger_group* besitzen, aussortiert. Dann wird die verbleibende Liste startend bei Zügen mit geringer Kapazität, welche sich mit hoher Geschwindigkeit fortbewegen bis zu Zügen mit großer Kapazität, aber langsamer Fortbewegung sortiert. Die Position des final ausgewählten Zuges in dieser Liste wird nun prozentual durch den übergebenen Parameter angegeben. Demnach erhält man für *capacity_speed_ratio =0* den „kleinsten, schnellsten Zug" und für *capacity_speed_ratio=1* den „größten, langsamsten Zug". Inwiefern die Parametrisierung von zentraler Wichtigkeit für die Grundidee der Gesamtlösung ist, wird im Kapitel 4.3.3 am Beispiel des *STP-Algorithmus* näher erläutert. Die Auswahl von *my_train* ist dabei so implementiert, dass dieser in einer separaten Variablen gespeichert wird und so leicht ausgetauscht und durch einen andern ersetzt werden kann. So ist die Implementierung eines optionalen Auswahlalgorithmus oder gar der Nutzung von mehr als nur einem Zug stark erleichtert (weitere Erweiterungsideen werden in Kapitel 4.4 behandelt).

Eine weitere Verbesserung des *ADA* ist eine Historisierung bereits berechneter kürzester Pfade zwischen zwei Stationen. Dazu wird zu Beginn ein *Python Dictionary path_dict* angelegt. Dieses wird der *calculate_shortest_path()*-Methode zur Berechnung des kürzesten Pfades zwischen zwei Stationen zusätzlich übergeben. Bevor diese durch den *Dijkstra-Algorithmus* (vgl. Kapitel 2.2) durchgeführt wird, wird zunächst überprüft, ob bereits eine Berechnung erfolgt ist, um unnötig redundante Berechnungen, welche sich negativ auf die Durchlaufzeit auswirken zu vermeiden.

Die größte Veränderung zum *SDA*, welche gleichzeitig auch die größte Verbesserung der kumulierten Verspätungszeit im direkten Vergleich bedeutet, ist die Parallelisierung des Passagiertransports. Dazu wird die Liste aller Passagiere zunächst nach dem Attribut *target_time* aufsteigend sortiert. So kann sichergestellt werden, dass Passagiere, die früher am Ziel ankommen wollen auch früher transportiert wollen. Ist nun ein Passagier P_1 ausgewählt, welcher transportiert werden soll, wird überprüft, ob auf dessen Weg weitere Passagiere P_n auf eine Beförderung warten deren Start- und Zielstationen auf dem zu kürzesten Weg von P_1*(initial_station)* nach P_1*(target_station)* liegen. Ist noch Kapazität in *my_train* vorhanden, werden Passagiere unabhängig von ihrer *target_time* noch auf den entsprechend benötigten Teilen des Weges mitgenommen. Damit kann jeder Passagier in insgesamt weniger Zeiteinheiten transportiert werden, da einige Wege später nicht erneut abgefahren werden muss.

4.2.2 Bewertung

Die Vorteile des *ADA* liegen vor allem in den Verbesserungen, die zum *SDA* gemacht wurden, also der Parametrisierung, Neustrukturierung, Pfadhistorisierung und Parallelisierung der Passagiere. Andere Vorteile der Herangehensweise wurden bereits in der Bewertung der simplen Version betrachtet (vgl. Kapitel 4.1.2)

Im Hinblick der Übertragbarkeit auf die Realität ist der *ADA* ein Abbild des anzunehmenden Passagierverhaltens. Steht dieser bereits an der Station bereit, an der ein Zug mit freien Kapazitäten die gleiche Station anfährt, die auch seine Zielstation ist, wird er sicher einsteigen. Außerdem ist die kurze Laufzeit (durch die Historisierung der kürzesten Pfade sogar noch performanter als der *SDA*) ein großer Vorteil für die Anwendung im realen Leben. Damit ist als Anwendung eine just-in-time Berechnung denkbar. Wenn auf großen Schienennetzen ermittelt werden soll welche Strecken bei dem Einsatz von nur einem Zug abgefahren werden müssten kann so schnell eine Abschätzung getroffenen werden inwiefern die Parameter Geschwindigkeit und Kapazität eine große Rolle spielen und so eine Kosten-Nutzen Abwägung durchgeführt werden.

Größter Nachteil des *ADA* sind, wie auch schon beim *SDA*, viele ungenutzte Kapazitäten, die durch eine Parallelisierung der Züge genutzt werden könnten. Ein Ansatz wie dies umgesetzt werden könnte wird im Folgenden anhand des *STP-Algorithmus* erläutert werden.

4.3 Simple Train Parallelization Algorithm

Alle bisher vorgestellten Algorithmen haben gemein, dass sie lediglich mit einem Zug zur selben Zeit arbeiten (Züge, die aufgrund des *Swapping* (vgl. Kapitel 2.1.2) abfahren, ausgenommen). Insbesondere bei großen Schienennetzen scheint es jedoch naheliegend, dass mehrere Züge zum gleichen Zeitpunkt abfahren und Passagiere damit gleichzeitig bzw. zeitlich parallel zu ihren Zielen bringen, um so die Gesamtverspätung erheblich zu minimieren. Ein solches paralleles Starten von möglichst vielen Zügen ist nun die Grundidee des *Simple Train Parallelization Algorithm (STP-Algorithmus)*. Dieser soll nachfolgend vorgestellt werden.

4.3.1 Idee und Umsetzung

Das parallele Starten von Zügen bringt diverse Probleme mit sich, die in dieser Form bei den bisher skizzierten Algorithmen nicht aufgetreten sind. Startet man einen Zug, so muss man hier bereits einen „Blick in die Zukunft“ werfen, um zu erkennen, ob es dort nicht zu Kapazitätsproblemen bei Strecken oder Stationen mit einem parallellaufenden Zug kommt. Für diesen „Blick in die

Zukunft“ kann man verschiedene Ansätze wählen. Im Rahmen des *Simple Train Parallelization Algorithm* wurde nun folgende Lösung implementiert:

Alle Attribute von Objekten im Schienennetz, die sich im Laufe der Zeit verändern, werden in einer Tabelle (hier: *Pandas DataFrame*) gespeichert. Zu den veränderlichen Attributen zählen unter anderem die aktuellen Kapazitäten von Stationen, Strecken und Zügen, die Positionen von Zügen und Passagieren im Schienennetz oder auch die Wahrheitswerte *is_in_train* im Falle der Passagiere bzw. *is_on_line* bei den Zügen. Jeder Zeitschritt des Zugfahrplans ist durch eine Zeile in dieser Tabelle dargestellt. Dabei entspricht der Index im DataFrame genau dem entsprechenden Zeitpunkt des Fahrplans.

Bei der Initialisierung einer neuen Instanz des *STP-Algorithmus* wird, neben der Tabelle selbst, auch gleich die erste Zeile (Index 0) erstellt. Diese enthält die Werte aus dem Input-Parser, wobei die Kapazitäten der Stationen gleich gemäß den aktuellen Positionen der Züge angepasst werden. Diese erste Zeile des DataFrames wird anschließend nur noch beim Setzen der Wildcard-Züge zu Beginn des Algorithmus verändert. Dies wird in Kapitel 4.3.3 unter dem Gesichtspunkt der Parametrisierung der entwickelten Algorithmen näher erläutert.

Die nachfolgende Ausführung zur Funktionsweise des *Simple Train Parallelization Algorithm* beginnt daher nach Setzen der Wildcard-Züge, also zum Zeitpunkt 1:

Grundsätzlich besteht die *solve()*-Methode des Algorithmus aus zwei verschachtelten *while*-Schleifen. Die äußere Schleife zählt dabei die Zeiteinheit („Runde“ (InformatiCup 2022, S. 2)) hoch. Sie bricht ab, sobald alle Passagiere an ihren Zielstationen angelangt sind. Die innere Schleife hingegen läuft standardmäßig so lange, bis es keinen freien, stehenden Zug mehr gibt, der einen Passagier mitnehmen kann. Dies kann unter Umständen jedoch sehr lange dauern und ist auch nicht immer von Vorteil, wie nachfolgende Ausführungen zeigen werden. Deshalb wird für diese Schleife ebenfalls gezählt, wie oft sie durchlaufen wird. Bei einer bestimmten Anzahl an Durchläufen wird unabhängig von ersterer Bedingung abgebrochen. Wie sich die Anzahl maximaler Durchläufe im Detail berechnet, wird in Abschnitt 4.3.3 erläutert.

Innerhalb der inneren Schleife passiert nun folgendes: Zunächst wird ein Passagier ausgewählt, der dann an sein Ziel gebracht werden soll. Dabei wird immer der Passagier mit der frühesten Ankunftszeit gewählt (ausgenommen sind bei der Auswahl Passagiere, die sich bereits auf einer Strecke befinden oder schon am Ziel angekommen sind). Anschließend wird ein Zug ausgewählt, der den Passagier an sein Ziel bringen soll. Aus allen möglichen Zügen (ausreichend Kapazität, nicht bereits auf einer Strecke, etc.) wird derjenige gewählt, der sich am nächsten am Passagier

befindet. Hierzu wird der kürzeste Weg mithilfe des laufzeitoptimierten Dijkstra-Algorithmus berechnet, der bei den Algorithmen in Kapitel 4.2 vorgestellt wurde.

Sofern ein Zug und ein Passagier ausgewählt werden konnten, wird weiter überprüft, ob deren Positionen bereits übereinstimmen. Ist dies der Fall, so kann der Passagier im Fahrplan einsteigen („board“). Beim Boarding werden zudem alle veränderlichen Attribute von Passagier und Zug im DataFrame entsprechend angepasst.

Im Anschluss an das Boarding fährt der Zug mitsamt Passagier in der darauffolgenden Zeiteinheit ab („depart“). Hierzu wird die Methode *depart_train()* aufgerufen, wobei der Methode ausgewählter Zug, Zielstation und Startzeit übergeben werden. Die Methode überprüft nun, ob der Zug überhaupt abfahren kann und nicht z.B. gerade andere Passagiere ablädt. Ist dies der Fall, so wird mithilfe des Dijkstra-Algorithmus der kürzeste Weg zur Zielstation berechnet. Die einzelnen Strecken auf diesem Weg werden dann in einer Schleife durchlaufen. Auch hierbei wird geprüft, ob der Zug überhaupt abfahren kann. Dazu müssen die Strecke (während der Fahrtzeit) und die Zielstation (ab Ankunft) frei sein. Hat die Zielstation nicht ausreichend Kapazität für den Zug frei, so wird geprüft, ob *Swapping* möglich ist. Ist dies nicht der Fall, so wird eine Exception geworfen, die Fahrt des Zuges wird an dieser Station abgebrochen. Kann der Zug aber Abfahren, so werden die entsprechenden Anpassungen im DataFrame vorgenommen. Ist dabei *Swapping* notwendig und möglich, so wird hierfür wieder die *depart_train()*-Methode verwendet. Sobald der Zug sein Ziel erreicht, gibt die Methode die Ankunftszeit zurück.

Diese wird anschließend verwendet, um den Passagier aussteigen zu lassen. Die entsprechende Methode *detrain_passenger()* wird unter Angabe der Ankunftszeit aufgerufen. Der Passagier wird abgeladen („detrain”) und die Attribute im DataFrame werden wieder entsprechend angepasst.

Der beschriebene Ablauf gilt aber nur für den Fall, dass sich Passagier und Zug an derselben Station befinden. Ist dem nicht so, muss zunächst der Zug auf kürzestem Weg zum Passagier gebracht werden. Auch dazu wird die *depart_train()*-Methode verwendet.

Der dargelegte Prozess aus Passagier- und Zugauswahl und Versuch des Boarding, Abfahren sowie Aussteigen des Passagiers wird in der inneren Schleife ausgeführt und somit für eine Zeiteinheit so oft wiederholt, bis die Schleife abbricht. Dadurch werden, wenn möglich, verschiedene Züge mit Passagieren (zeitlich) parallel gestartet. Diesem Grundprinzip verdankt der *Simple Train Parallelization Algorithm* seinen Namen.

Wie bereits dargelegt, ist der Algorithmus fertig, sobald alle Passagiere am Ziel sind. Abschließend wird die gesamte Verspätungszeit des erstellten Fahrplans berechnet. So kann die Lösung schlussendlich bewertet und mit den anderen Lösungen verglichen werden.

4.3.2 Bewertung

An dieser Stelle soll eine Bewertung des *STP-Algorithmus* hinsichtlich dessen Vor- und Nachteile vorgenommen werden. Generell ist dabei anzumerken, dass es sich beim *Simple Train Parallelization Algorithm* – wie der Name schon impliziert – um einen Algorithmus handelt, der die Zugparallelisierung so einfach wie möglich umsetzt. Darüber hinaus kann er jedoch an diversen Stellen optimiert werden.

Ungeachtet dessen muss festgehalten werden, dass die Idee der Parallelisierung von Zügen insbesondere bei kleineren Schienennetzen mit mehreren Passagieren gut angewandt werden kann. So liefert der Algorithmus zum Beispiel für das kleine Input-Problem "Kapazität" (GitHub InformatiCup 2022) sehr gute Ergebnisse. Während die anderen Algorithmen hier nicht ohne Verspätungszeit lösen können, erstellt der *STP-Algorithmus* einen optimalen Fahrplan ohne Verspätungen, da er als einziger mehrere Züge nutzt.

Jedoch kommen bei größeren Schienennetzen Laufzeitprobleme zum Tragen. Insbesondere der "Large"-Input (siehe *GitHub InformatiCup 2022*) kann nicht innerhalb einer akzeptablen Zeit gelöst werden. Dies ist in erster Linie darauf zurückzuführen, dass der Algorithmus sehr viele Berechnungen (z.B. mit dem Dijkstra-Algorithmus) durchführt. Die meisten davon resultieren nicht in einer Aktion (Board/Depart/Detrain) für den Fahrplan, sondern testen lediglich, ob eine Aktion möglich ist oder wie die optimale Lösung aussieht. Darüber hinaus kann es vorkommen, dass sich der Algorithmus bei großen Schienennetzen in einer Endlosschleife verfängt und dieselben Aktionen dabei periodisch wiederholt. Dies darf jedoch nicht als Grenze der Grundidee verstanden werden, sondern ist lediglich auf kleinere Fehler in der Implementierung zurückzuführen.

Ferner muss am Algorithmus kritisiert werden, dass immer nur ein Passagier pro Zug gleichzeitig transportiert wird, auch wenn dessen Kapazität einen weiteren Passagier zulassen würde. So könnte die kumulierte Verspätungszeit um ein Vielfaches minimiert werden, so dass der Algorithmus in der Gesamtbetrachtung deutlich besser abschneidet.

4.3.3 Parametrisierung der Algorithmen am Beispiel des STP-Algorithmus

Die Grundidee der Gesamtlösung und deren Umsetzung wurde bereits in den vorangegangenen Kapiteln thematisiert. Es werden Instanzen verschiedener Algorithmen erzeugt, die dasselbe Input-Problem lösen. Ausgewählt wird schließlich das Ergebnis mit der geringsten Gesamtverspätung. Dies kann man nun noch weitertreiben und nicht nur verschiedene Algorithmen instanzieren, sondern auch denselben Algorithmus mehrmals mit verschiedenen Parametern instanziieren. In Abhängigkeit der Parameter löst derselbe Algorithmus dann das Input-Problem auf eine etwas andere Weise. Dies soll nun am Beispiel der Parametrisierung des *Simple Train Parallelization Algorithm* näher erläutert werden.

```
def __init__(self, input_from_file, parallelization_factor=1.0,
set_wildcards=1.0):
```

Listing 8: Konstruktor der Klasse des Simple Train Parallelization Algorithm

Listing 8 zeigt den Konstruktor des *STP-Algorithmus*. Neben dem Parameter *input_from_file*, den alle Algorithmen besitzen und über den der geparste Input übergeben wird, besitzt dieser Konstruktor noch zwei weitere optionale Argumente: die Parameter *parallelization_factor* und *set_wildcards*. Beide sollen anschließend kurz erläutert werden, bevor dargestellt wird, wie der Anwender diese Parameter nutzen kann.

Der Parameter *parallelization_factor* bezieht sich auf die maximale Anzahl an Schleifendurchläufen der inneren Schleife des Simple Train Parallelization Algorithm. Standardmäßig ist diese maximale Anzahl definiert als das Minimum aus Stationen, Strecken und Passagieren des Schienennetzes. Damit können - vorausgesetzt jeder Schleifendurchlauf resultiert in einem abfahrenden Zug – maximal diese Anzahl an Zügen zeitlich parallel abfahren. Der Parameter *parallelization_factor* gibt nun dem Anwender etwas Kontrolle über die Anzahl an Schleifendurchläufen und damit an parallelen Zügen. Dies kann insbesondere bei Laufzeitproblemen ein interessantes Werkzeug sein. Denn es ist davon auszugehen, dass der Großteil der Schleifendurchläufe nicht in einer erfolgreichen Zugabfahrt mündet. Der *parallelization_factor* ist eine Fließkommazahl zwischen 0 und 1. Die Anzahl maximaler Schleifendurchläufe, zuvor festgelegt durch die Minima (s.o.), wird dann mit diesem Faktor multipliziert. Insbesondere bei größeren Schienennetzen, bei denen der Algorithmus standardmäßig nicht in akzeptabler Zeit löst, ist eine Reduktion dieses Faktors also ein durchaus denkbarer Ansatz.

Einen weiteren Parameter stellt das Argument *set_wildcards* dar. Dieses bezieht sich auf das Setzen der Wildcard-Züge. Dabei können verschiedene Strategien angewandt werden. Zum Beispiel

können immer alle Wildcards gesetzt werden, damit möglichst viele Züge zum Transport von Passgieren zur Verfügung stehen. Andererseits können aber auch so wenig wie möglich Wildcard-Züge gesetzt werden (d.h. nur einer; wenn noch andere Züge vorhanden sind, sogar keiner). Gedanke hinter dieser Strategie ist es dann, Kapazitätskonflikte möglichst zu vermeiden. Zwischen diesen beiden Extremen muss nun bei der Implementierung einer Strategie zum Setzen der Wildcards abgewogen werden. Der Parameter *set_wildcards* aber erlaubt es, dass diese Entscheidung nicht bereits bei der Implementierung des Algorithmus getroffen werden muss: Dieser Parameter stellt nämlich den relativen Anteil der Wildcard-Züge dar, die in der jeweiligen Instanz des Algorithmus gesetzt werden sollen. Ist der Parameter also 1, so werden alle Wildcards gesetzt (=Standardwert), wohingegen bei 0 schließlich kein Zug gesetzt wird, außer es ist sonst kein Zug zur Verfügung. In diesem Fall wird der Wildcard-Zug mit der höchsten Kapazität gesetzt. Damit kann auch bei diesem Parameter der Anwender verschiedene Variationen des Algorithmus für sein Input-Problem testen und die für dieses spezifische Problem beste Wildcard-Strategie anwenden.

Zusammengefasst bieten die Parameter also eine Möglichkeit für den Anwender nicht nur den für sein Problem besten Algorithmus auszuwählen, sondern diesen auch noch durch Finetuning der Parameter möglichst optimal an seinen Anwendungsfall anzupassen.

In der Lösung, die für den InformatiCup 2022 abgegeben wird, soll der Anwender selbst aber nicht aktiv in den Lösungsprozess eingreifen. Dadurch wird es schwierig, mit diesem Ansatz den besten Algorithmus für jeden Sonderfall zu instanziieren. Trotzdem wurden im Rahmen der Abgabe möglichst verschiedene Variationen der parametrisierbaren Algorithmen (*SPP*, *APP* und *STP*) instanziiert, deren Parameter sich stark unterscheiden. So sollte für die meisten Fälle zumindest ein guter Algorithmus vorhanden sein, wenn auch nicht der optimal angepasste Algorithmus. Ein anderer Ansatz, um diese Problematik zu lösen, wird später im Ausblick dieser Arbeit noch diskutiert.

4.4 Weiterführende Algorithmen und Features

Nun wurden insgesamt vier verschiedene Algorithmen mit unterschiedlichen Grundideen vorgestellt. Trotzdem gibt es noch eine Reihe weiterer Ansätze, auf deren Basis Algorithmen implementiert werden können. Obwohl neben den vier gezeigten keine weiteren Algorithmen umgesetzt wurden, seien in diesem Kapitel noch einige Ideen für weitere Algorithmen skizziert.

Ein erster naheliegender Ansatz ist dabei die Kombination von bereits vorhandenen Algorithmen zu einer neuen, besseren Lösung. Dabei kommen insbesondere der *STP*- und der *APP-Algorithmus* infrage. Während der *STP-Algorithmus* die Züge parallelisiert und dabei deren Kapazitäten nicht vollständig ausnutzt, werden im *APP-Algorithmus*, bei jedoch nur einem verwendeten Zug, die

Zugkapazitäten ausgereizt. Nimmt man nun die Vorteile beider Algorithmen und kombiniert diese, erhält man einen deutlich besseren Algorithmus.

Ein gänzlich neuer Algorithmus hingegen kann auf Basis der folgenden Idee entwickelt werden: Bislang können Passagiere auch „zu früh“ am Ziel ankommen. Diese Zeit geht aber nicht positiv in die Verspätungszeit ein und wird damit „verschenkt“. Denn sie könnte an anderen Stellen noch ausgenutzt werden, an denen durch bisherige Ansätze noch Verspätungen entstehen. Ein Algorithmus, der Passagiere nicht zu früh an ihr Ziel kommen lässt, könnte die kumulierte Verspätungszeit hier noch erheblich reduzieren.

Ebenfalls ein neuer Ansatz ist derjenige, dass Pfade nicht aufgrund der kürzesten Distanz ausgewählt werden, sondern auch die Passagiere entlang des Weges betrachtet werden. Für einen Zug wird dann nicht der kürzeste Pfad ausgewählt, sondern eine Route berechnet, auf der möglichst viele Passagiere in sinnvoller Reihenfolge eingesammelt und stückweise ans Ziel gebracht werden. Denkt man diese Idee weiter, könnte bei großen Schienennetzwerken mithilfe eines *k-means Clustering-Algorithmus* festgestellt werden, wo vielbefahrene Routen in diesem Schienennetzwerk liegen. Diese könnten dann zu Standardrouten für Züge zusammengefasst werden[3].

Auch wenn keine dieser Ideen bislang implementiert wurde, stellt eine Erweiterung der Softwarelösung um diese Algorithmen kein großes Problem dar. Denn der Lösungsansatz, verschiedene Algorithmen austauschbar zu verwenden, macht die Softwarelösung optimal skalierbar, insbesondere hinsichtlich der Implementierung weiterer Algorithmen.

4.5 Auswertung von Performance und Laufzeiten

[3] Die skizzierte Idee stammt aus dem Bereich der algorithmischen Auswertung von Standortdaten (vgl. Grossenbacher 2011) und wurde auf das hier behandelte Themengebiet übertragen.

5. Schlussbetrachtung

5.1 Zusammenfassung und Fazit

An dieser Stelle werden die Inhalte der Arbeit nun zusammengefasst und ein Fazit gezogen. Die Zusammenfassung orientiert sich dabei am Aufbau der Softwarelösung, um so nicht nur einen Überblick über die Inhalte zu vermitteln, sondern auch den zeitlich-logischen Ablauf des Programms darzulegen.

Die Programmausführung beginnt mit dem Einlesen der Inputdatei via *stdin*. Diese wird dem Input Parser übergeben, der die Inhalte der Datei in Objekte überführt. Anschließend werden die vier verschiedenen Algorithmen je mehrmals mit unterschiedlicher Parametrisierung instanziiert. Es werden jeweils Kopien der vom Input Parser erstellten Objekte weitergegeben. Die Instanzen der Algorithmen werden in einer Liste zusammengefasst und dem Output Parser übergeben. Dieser führt sukzessive die *solve()*-Methoden der einzelnen Algorithmus-Instanzen aus. Nach Beendigung aller Algorithmen wird das beste Ergebnis anhand der kumulierten Verspätungszeit ausgewählt. Diese Version des Fahrplans wird in den *stdout* geschrieben und so an den Benutzer übergeben.

Zusammenfassend lässt sich festhalten, dass dieser grundlegende Ansatz und dessen Umsetzung in der Softwarearchitektur die Unterschiedlichkeit der Algorithmen ausnutzt, um für verschiedenste Input Dateien bestmögliche Fahrpläne zu erstellen. Das Konzept ist optimal durch komplexere Algorithmen erweiterbar, liefert aber bereits mit den aktuellen Implementierungen gute und insbesondere gültige Ergebnisse in sehr kurzer Laufzeit. Dennoch muss angemerkt werden, dass die Algorithmen in der einzelnen Betrachtung noch erhebliches Potential aufweisen, da sie auf weitestgehend simplen Grundideen beruhen. Dies stellt jedoch keine allzu große Limitierung dar. Denn der modularen Architektur der Softwarelösung können sehr einfach weiterentwickelte Algorithmen hinzugefügt werden.

5.2 Ausblick

In diesem Abschnitt sei nun noch eine Idee skizziert, um den vorgestellten Lösungsansatz zu optimieren. Diese setzt an einer Schwachstelle der Lösung an, die leicht auf den ersten Blick zu erkennen ist: Im erläuterten Lösungsansatz werden alle instanziierten Algorithmen ausgeführt und erst im Nachhinein wird das beste Ergebnis ausgewählt. Dies ist nicht nur suboptimal aus Sicht

des Programmentwurfs, sondern kann insbesondere bei großen Inputdateien problematisch in Bezug auf die Laufzeit der Lösung werden.

Hier wäre es sinnvoll, bereits vor Ausführung der einzelnen Algorithmen zu wissen, welcher das beste Ergebnis liefern wird. Wählt der Anwender selbst für jeden Input einen Algorithmus aus, so kann er bereits vor der Ausführung anhand der in dieser Arbeit vorgestellten Vor- und Nachteile einschätzen, welcher Algorithmus das beste Ergebnis erzeugt. Diese Aufgabe könnte nun eine *Künstliche Intelligenz* bzw. ein *Machine Learning Modell* übernehmen.

Hierbei kann überwachtes Lernen (*supervised learning*) eingesetzt werden, wobei das Optimierungsproblem als Klassifizierungsproblem betrachtet wird. Eine Inputdatei wird dabei stark in Tabellenform komprimiert, so dass die wesentlichen Informationen (z.B. Anzahl an Zügen, Strecken, …) einheitlich in einer Zeile dargestellt werden können. Für jede Zeile wird in einer weiteren Spalte der für diese Daten performanteste Algorithmus hinzugefügt. Diese Spalte stellt das Zielattribut („Label“) dar. Hat man genügend Daten gesammelt, kann man verschiedene Modelle mit diesen Daten trainieren (z.B. einfache *Decision Trees*). Wurde ein gutes Modell trainiert, kann dieses anschließend verwendet werden, um für ein Input-Problem den besten Algorithmus auszuwählen. Bei der Wahl des Algorithmus sollte der komplexitätsreduzierte Datensatz und ein einfacheres Modell ausreichen, da sich die Algorithmen und ihre Vorteile hinsichtlich verschiedener Probleme relativ deutlich unterscheiden. Für eine intelligente Parameterwahl werden jedoch komplexere Modelle und Daten nötig sein.

Zusammenfassend lässt sich sagen, dass diese Optimierung den Lösungsansatz erheblich verbessern könnte, insbesondere mit Blick auf Laufzeit und Programmentwurf. Allerdings konnte diese Idee aus zeitlichen Gründen nicht umgesetzt werden, wurde hier aufgrund ihres enormen Potenzials aber trotzdem dargelegt.

5.3 Übertragbarkeit auf die Realität

Abschließend soll noch die Übertragbarkeit des vorgestellten Lösungsansatzes auf reale Probleme bewertet werden. Zunächst wird dabei auf die Grenzen der Aufgabenstellung eingegangen.

Diesbezüglich ist anzumerken, dass die Aufgabenstellung aus betriebswirtschaftlicher Perspektive stark begrenzt ist. So wird angenommen, dass Bahnunternehmen stets das Ziel haben, die Verspätung der Passagiere zu minimieren und damit die Kundenzufriedenheit zu maximieren. Dies ist in der Realität jedoch nicht der Fall: Das Ziel jedes Unternehmen ist es, den Gewinn zu maximieren, was durchaus in einem Zielkonflikt mit der Aufgabenstellung stehen kann. Wird ein Programm zur Erstellung eines in Bezug auf die Verspätung optimalen Fahrplans entwickelt, kann es

vorkommen, dass dort Züge mit sehr wenigen Passagieren fahren. Dies wäre insbesondere dann betriebswirtschaftlich nicht erstrebenswert, wenn der Umsatzverlust durch eine reduzierte Kundenzufriedenheit geringer ist, als der Verlust, der entsteht, wenn man einen Zug für eine geringe Anzahl von Passagieren weite Strecken fahren lässt. Zusammengefasst ist es also nicht unbedingt realistisch, dass der Fahrplan gänzlich nach den Wünschen der Passagiere ausgerichtet ist. Vielmehr versucht man die am häufigsten genutzten Strecken anzubieten, die Passagiere müssen sich nach diesem Fahrplan richten.

Allerdings stellt nicht nur der betriebswirtschaftliche Kontext der Aufgabenstellung eine Limitierung bei der Übertragbarkeit dar. Auch konkrete fachliche Aspekte der Aufgabe lassen sich nur schwer auf reale Bedingungen übertragen. Dies hat insbesondere das Kapitel „Annahmen bei der Berechnung“ gezeigt. Durch die Vorgabe der Problemstellung, dass ein Zug in der gleichen Zeiteinheit, in der er ankommt, auch wieder von einer Station abfahren kann ist es in bestimmten Fällen möglich eine Verdoppelung der Geschwindigkeit zu erreichen, was so in der Realität verständlicherweise nicht möglich ist. Des Weiteren stellt auch das *Swapping* einen Vorgang dar, der physikalisch ohne eine Ausweichschiene nicht umgesetzt werden kann. Stattdessen würde die im zugehörigen Kapitel (2.1.1) beschriebenen Verschiebungsproblematik wieder auftreten.

Unmittelbar aus den Limitierungen der Aufgabenstellung resultieren auch die Grenzen der hier vorgestellten Algorithmen in ihrer Praxisrelevanz. Sie beruhen allesamt auf den genannten Angaben bei der Berechnung und sind darüber hinaus in Bezug auf eine minimale Gesamtverspätung optimiert. Dass dies nicht unbedingt sinnvoll ist, haben obige Erläuterungen gezeigt. Unabhängig von diesen Aspekten muss angemerkt werden, dass die bereits implementierten Algorithmen in ihren Grundideen jeweils sehr simpel sind. Damit stellen auch die Grundgedanken der Algorithmen nicht zwingend einen Mehrwert für reale Problemstellungen dar.

Anders verhält es sich jedoch beim gesamten Lösungsansatz. Die Idee, verschiedene Algorithmen mit Vor- und Nachteilen bei unterschiedlichen Problemen zu verwenden und so im Zusammenspiel ein optimales Ergebnis zu erzeugen, lässt sich durchaus auf die Realität übertragen. Zudem bildet dieser Ansatz einen guten Rahmen für beliebige, auf konkrete reale Probleme angepasste Algorithmen, die anschließend wie in dieser Arbeit beschrieben verwendet werden können. Geht man darüber hinaus noch weiter, trainiert und implementiert ein Machine Learning Modell für die Auswahl des besten Algorithmus, so stellt diese Lösung in ihrem Kerngedanken durchaus einen Ansatz dar, der auch für reale Probleme effektiv eingesetzt werden kann.

Dabei stellt die Erstellung von Zugfahrplänen nicht den einzigen Anwendungsfall dar. Da im vorgestellten Lösungsansatz beliebige Algorithmen verwendet werden können, kann mit diesem

Grundgedanken Software entwickelt werden, die nicht nur im Rahmen von Logistikproblemen eingesetzt werden kann, sondern im Allgemeinen auf Optimierungsprobleme verschiedenster Art anwendbar ist.

5.4 Reflexion

Nachdem in dieser Arbeit nun ausführlich auf die entwickelte Software eingegangen wurde, soll dieser Abschnitt noch genutzt werden, um kurz die Errungenschaften und das Gelernte bei der Umsetzung der Lösung zu reflektieren.

Die Problemstellung des InformatiCup 2022, welche auf den ersten Blick eng begrenzt erscheint, entpuppt sich bei der detaillierten Betrachtung als Herausforderung auf unterschiedlichen Ebenen. Diese liegt neben der Entwicklung der Software vor allem im präzisen Verständnis der Berechnungsvorschriften. Auch die Nutzung von Docker zur zuverlässigen, systemunabhängigen Ausführung der Software brachte einige Hürden mit sich. Darüber hinaus konnten die Kenntnisse im Bereich der objektorientierten Programmierung mit Python vertieft und weiterentwickelt werden. Ebenso konnte der analytische Umgang mit Problemen und insbesondere das Finden von Lösungen für diese Probleme anhand der Aufgabenstellung sehr gut erlernt werden. Auch das kann als wichtige Errungenschaft betrachtet werden. Denn für Unternehmen gehört in der heutigen Zeit die Fähigkeit eines Mitarbeiters Probleme zu lösen, zu den wichtigsten Soft Skills.

Dass der InformatiCup 2022 komplexe Probleme bereitstellte, zeigt vor allem ein Blick auf das Schienennetz „large“ (GitHub InformatiCup 2022). Betrachtet man die Visualisierung dieses Schienennetzes (Abbildung 1), das gleich nach dem Deckblatt dieser Arbeit abgebildet ist, so bekommt man schnell den Eindruck, man stünde vor einem unlösbaren Problem.

Doch die Tatsache, dass die im Zuge dieser Arbeit entwickelte Software eine funktionierende Lösung auch für dieses Problem bietet, zeigt: Am Ende ist eben nicht das Problem selbst das Problem, sondern nur die Haltung, mit der man an das Problem herangeht.

Quellenverzeichnis

[GitHub InformatiCup 2022] GitHub Repository zum InformatiCup 2022. URL: https://github.com/informatiCup/informatiCup2022 *(letzter Zugriff: 08.01.2022)*

[Grossenbacher 2011] Grossenbacher, Timo. *Möglichkeiten der algorithmischen Auswertung von Standortdaten*. Universität Zürich, 2011. URL: https://files.ifi.uzh.ch/hilty/t/examples/fach-arbeiten/Algorithmische_Auswertung_Standortdaten_Grossenbacher.pdf *(letzter Zugriff: 15.01.2022)*

[InformatiCup 2022] Aufgabenstellung Informaticup2022. URL: https://github.com/informati-Cup/informatiCup2022/blob/main/informatiCup%202022%20-%20Abfahrt!.pdf *(letzter Zugriff: 08.01.2022)*

[mdsrosa 2015] Modified Python implementation of Dijkstra's Algorithm, 2015. URL: https://gist.github.com/mdsrosa/c71339cb23bc51e711d8 *(letzter Zugriff: 11.01.2022)*

[Mönius et al. 2021] Mönius, Katja; Steuding, Jörn; Stumpf, Pascal. *Algorithmen in der Graphentheorie - Ein konstruktiver Einstieg in die Diskrete Mathematik*. 1. Auflage. Wiesbaden: Springer Wiesbaden GmbH, 2021, S. 40-46

[Pepper 2007] Pepper, Peter. *Programmieren Lernen: Eine Grundlegende Einführung Mit Java.* 3. Auflage. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, S.237

[Stotz 2013] Stotz, Richard. *Der Bellman-Ford-Algorithmus*. TU München, 2013. URL: https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-bellman-ford/index_de.html *(letzter Zugriff: 15.01.2022)*

[Python 2 2022] Python 2 Dokumentation der Bibliothek sys – System-specific parameters and functions. URL: https://docs.python.org/2/library/sys.html *(letzter Zugriff: 10.01.2022)*

[Python 3 sys 2022] Python 3 Dokumentation der Bibliothek sys – System-specific parameters and functions. URL: https://docs.python.org/3/library/sys.html *(letzter Zugriff: 10.01.2022)*

[Python 3 copy 2022] Python 3 Dokumentation der Bibliothek copy. URL: https://docs.python.org/3/library/copy.html *(letzter Zugriff: 14.01.2022)*

[Velden 2014] Velden, Lisa. *Der Dijkstra-Algorithmus*. TU München, 2014. URL: https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-dijkstra/index_de.html *(letzter Zugriff: 10.01.2022)*

[Voll 2014] Voll, Robert. *Methoden der mathematischen Optimierung zur Planung taktischer Wagenrouten im Einzelwagenverkehr*. TU Dortmund, 2014. URL: https://d-nb.info/1095598791/34 *(letzter Zugriff: 11.01.2022)*

[Voroncovs 2015] Voroncovs, Aleksejs. *Der Floyd-Warshall Algorithmus*. TU München, 2015. URL: https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-floyd-warshall/index_de.html *(letzter Zugriff: 15.01.2022)*

Anhang

InformatiCupPy

1. Installation

Dieses Kapitel beschreibt die Installation der Anwendung auf einem beliebigen Betriebssystem.

1.1 Voraussetzungen

Die minimalen Voraussetzungen für die Ausführung der Software sind ein von Docker offiziell unterstütztes Betriebssystem sowie eine offiziell unterstützte Plattform. Wir empfehlen unterstützte **Linux Distributionen**, sowie **x86_64** als Plattform. Hier muss die **Docker Engine** lauffähig installiert sein, um die Ausführung der Anwendung zu ermöglichen (Installationsanleitung: Linux, Windows, macOS). Zusätzlich wird empfohlen git installiert zu haben (Git Installationsanleitung).

1.2 Installation der Anwendung über git

Wir empfehlen die Installation der Anwendung über **git**. Hierfür sollte man im Terminal, in einer beliebigen shell (empfohlene Referenzshells: bash oder zsh), in den gewünschten Verzeichnis/Ordner für die installation navigieren, was in den meisten shells durch `cd {Pfad zu gewünschtem Installationsordner}` erreicht werden kann. Anschließend wird die Anwendung via `git clone {Pfad zu Repository auf Gitprovider}` installiert.

2. Ausführung

Dieses Kapitel beschreibt die Ausführung der Software mithilfe von docker auf verschiedensten Betriebssystemen.

2.1 Bau eines lauffähigen Docker Image

Wenn die Anwendung korrekt installiert wurde, findet sich in ihr das **Dockerfile**. Anhand von diesem kann man mit **docker**, auf allen unterstützten Betriebssystemen, einfach ein lauffähiges **Docker Image** der Anwendung erstellen. Hierfür sollte im Terminal der Befehl `docker build {Pfad zum Rootverzeichnis der Anwendung (in der das Dockerfile liegt)} -t [Name des Image]` ausgeführt werden (in einigen Systemen kann es nötig sein dies Administrator Berechtigungen auszuführen (in Linux `sudo` vor den Befehl)).

2.2 Ausführung des Docker Image

Wenn das Docker Image ohne Fehler erstellt wurde, kann es mit dem Befehl (Referenz Linux mit zsh und bash shell) `docker run -i [Name des Image] < {Pfad zum Inputfile} > {Pfad zum gewünschten Outputfile}` ausgeführt werden. Hier wird das Inputfile durch `< {Pfad zum Inputfile}` in stdin des Containers gelesen, wodurch dieser damit arbeiten kann. Der stdout Output des Containers wird durch `> {Pfad zum gewünschten Outputfile}` in ein gewähltes Outputfile geparst, dort findet sich dann der berechnete Fahrplan (je nach Betriebssystem sollte hierfür die Dateiendung *.txt* gewählt werden).

3. Anmerkungen zu diesem Handbuch

Dieses Handbuch soll die Installation und Nutzung der Software möglichst jedem Benutzer unabhängig von dessen Wahl des Betriebssystems ermöglichen. Jedoch können wir nicht für die Allgemeingültigkeit dieser Anweisungen garantieren. Wir konnten die Software vor allem auf Linux und Windows Testen, da wir kein MacOS verwenden. Zudem sind die hier gezeigten Befehle alle auf die Linux Shells, welche wir getestet haben (bash und zsh) bezogen, es könnte auf anderen Shells und Betriebssystemen eine andere Syntax erforderlich sein. Falls es zu Problemen kommt empfehlen wir diese Ressourcen: Linux, Windows und macOS.

Anhang 1: Handbuch zur Softwarelösung