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

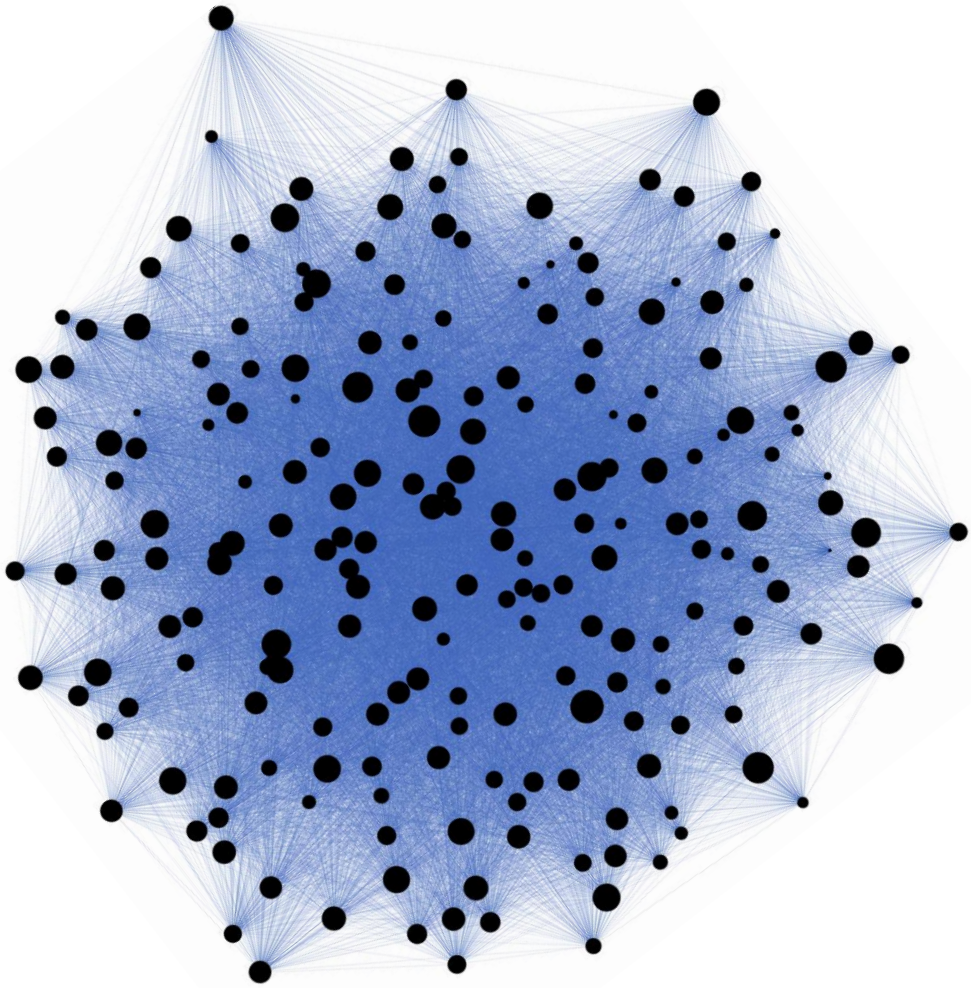

Abbildung 1: Visualisierung des Schienennetzes „large" (GitHub InformatiCup 2022)

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 der Softwarearchitektur 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 bewertet, wo die Stärken und Schwächen des dort verfolgten Ansatzes liegen. Außerdem werden Ideen für weitere Lösungsansätze genannt und erläutert. Abschließend werden alle Algorithmen in Bezug auf ihre Laufzeit sowie die Qualität ihrer Ergebnisse ausgewertet.

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.

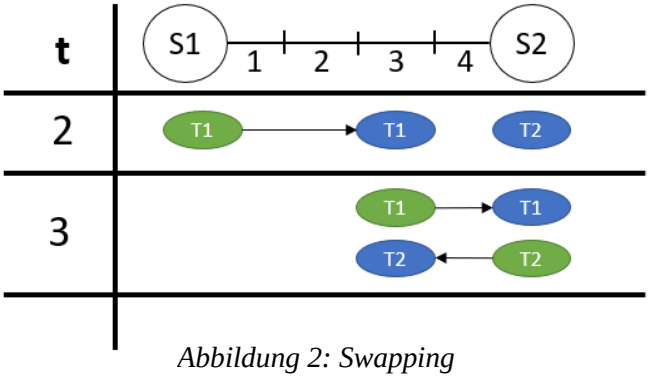

Abbildung 2: Swapping

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.

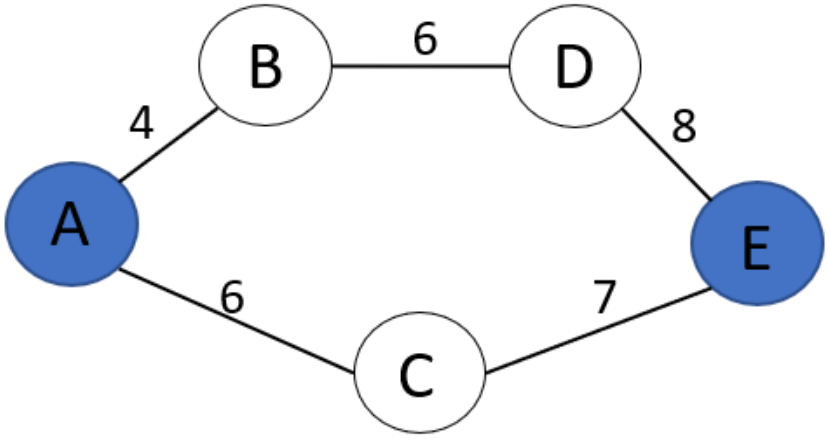

Abbildung 3: Einfaches Beispiel-Schema für den Dijkstra-Algorithmus
(in Anlehnung an Mönius et al. 2021, S.41ff.)

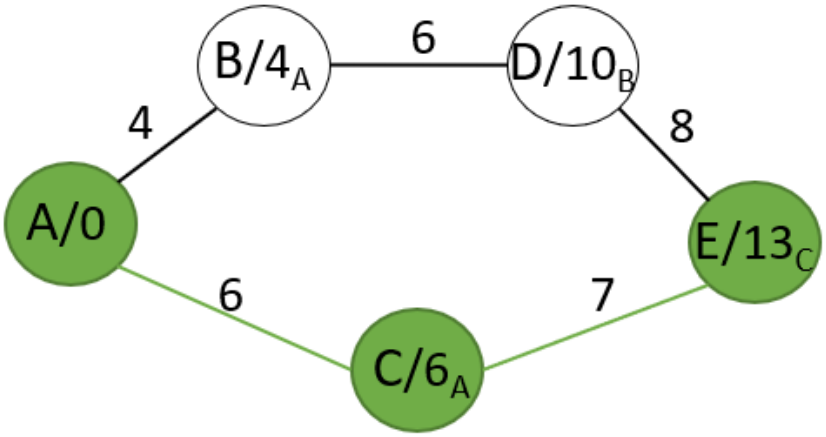

Abbildung 4: Kürzester Weg im Beispiel-Schema
(in Anlehnung an Mönius et al. 2021, S.41ff.)

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. 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 verschiedene Grundgedanken und Ansätze verfolgen, auf denselben Input angewendet werden. Dieses Kapitel stellt nacheinander einige solcher Algorithmen vor. Dabei werden jedoch nicht die Implementierungen im Detail dargelegt, sondern lediglich die grundsätzliche Ablauflogik vermittelt und stellenweise mit Codebeschreibungen veranschaulicht. Eine detailliertere Beschreibung würde aufgrund der Komplexität der Problemstellung den Rahmen dieser Arbeit sprengen.

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 *SDI*-Algorithmus angelehnt. Durch Anwendung unterschiedlicher Taktiken zur Parallelisierung von Passagieren in einzelnen Bereichen ist allerdings eine deutliche Verbesserung in der kumulierten Verspätung aller Passagiere bei weiterhin gültigen Fahrplänen zu verzeichnen. Außerdem wurde durch eine Überarbeitung der Struktur, wie beispielsweise der Auslagerung komplexer werdender Berechnungen in einzelne Methoden, die Wartbarkeit und Übersichtlichkeit deutlich verbessert. Im Folgenden werden dabei die beiden Algorithmen *Simple Passenger Parallelization Algorithm*

(SPP-Algorithmus) und *Advanced Passenger Parallelization Algorithm (APP-Algorithmus)* vorgestellt. Beide werden in der finalen Softwarelösung eingesetzt, da je nach Inputdatei der weniger komplexe *SPP*-Algorithmus zu besseren Ergebnissen im Vergleich zum *APP*-Algorithmus führt.

4.2.1 Idee und Umsetzung

Wie bereits beim *SDI*-Algorithmus (vgl. Kapitel 4.1) erläutert, werden mit einem vorab ausgewählten Zug alle Passagiere nacheinander transportiert. Dieser wird im Folgenden, wie auch im Quellcode, als *my_train* bezeichnet. Dabei wurden allerdings bei der Auswahl des Zuges Änderungen im Vergleich zum *SDI*-Algorithmus vorgenommen.

Bei der Ausführung der *Passenger Parallelization Algorithms* ist die Möglichkeit gegeben, mit der Instanziierung einen zusätzlichen Parameter *capacity_speed_ratio* anzugeben. Dieser hat einen Einfluss auf die Auswahl von *my_train*. Zunächst werden alle Züge mit einer Kapazität kleiner als die *biggest_passenger_group* aussortiert. Dann wird die verbleibende Liste sortiert, startend bei Zügen mit geringer Kapazität, welche sich mit hoher Geschwindigkeit fortbewegen, bis zu Zügen mit großer Kapazität aber langsamer Geschwindigkeit. 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“. Wildcard-Züge werden nur überprüft, wenn genügend Kapazität im Schienennetz frei ist und auch nur dann gesetzt, wenn sie durch die Evaluation des Parameters als *my_train* ausgewählt wurden. Inwiefern die Parametrisierung von zentraler Wichtigkeit für die Grundidee der Gesamtlösung ist, wird in Kapitel 4.3.3 am Beispiel des *STP*-Algorithmus näher erläutert.

In den *Passenger Parallelization Algorithms* wird die Auswahl von *my_train* dabei so implementiert, dass dieser in einer separaten Variable gespeichert wird und dadurch leicht ausgetauscht und durch einen andern ersetzt werden kann. So wäre die Erweiterung, um einen optionalen Auswahlalgorithmus oder gar die Implementierung einer Zug-Parallelisierung einfach umzusetzen (weitere Erweiterungsideen werden in Kapitel 4.4 behandelt).

Eine weitere Verbesserung der *Passenger Parallelization Algorithms* ist eine Speicherung 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 der *Dijkstra-Algorithmus* (vgl. Kapitel 2.2) den gewünschten Pfad berechnet, wird zunächst überprüft, ob dies bereits bei einem früheren Aufruf der Methode geschehen ist. Sofern dies der Fall ist, wird der kürzeste Pfad aus

dem *path_dict* ausgelesen und so unnötig redundante Berechnungen vermieden. Damit kann die Durchlaufzeit im Vergleich zum *SDI*-Algorithmus – vor allem bei mittelgroßen Schienennetzen mit vielen Passagieren (und damit vielen sich wiederholenden Berechnungen) – stark verkürzt werden (vgl. Kapitel 4.5). Die hier implementierte Version der *calculate_shortest_path()*-Methode wird auch im *STP*-Algorithmus (vgl. Kapitel 4.3) eingesetzt.

Die entscheidendste Veränderung zum *SDI*-Algorithmus, 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 werden. Ist nun ein Passagier P_1 ausgewählt, wird überprüft, ob auf dessen Weg weitere Passagiere P_n auf eine Beförderung warten. Dabei müssen deren Start- und Zielstationen auf dem zu kürzesten Weg von P_1*(initial_station)* nach P_1*(target_station)* liegen. Dies wird mit der Methode *find_passengers_along_the_way()* umgesetzt, die ein Dictionary zurückgibt, welches allen Stationen des Pfades Passagiere zuordnet, die den Voraussetzungen entsprechen. Während der schrittweisen Iteration durch den Pfad wird also überprüft, ob noch Kapazität in *my_train* vorhanden ist, um weiter Passagiere zu *boarden*. Diese werden dann unabhängig von ihrer *target_time* auf den entsprechenden Teilen des Weges mitgenommen und an ihrer individuellen Zielstation abgesetzt. Auch wenn der Zug vom Ziel des zuletzt transportierten Passagiers zum Start des nächsten Passagiers fährt wird geprüft, ob auf diesem Weg noch zusätzliche Passagiere mitgenommen werden können. Damit werden insgesamt weniger Zeiteinheiten benötigt, da einige Wege später nicht erneut abgefahren werden müssen.

Alle bisher beschriebenen Funktionen werden sowohl im *SPP*- als auch *APP*-Algorithmus eingesetzt. Um den Gedanken der Passagier-Parallelisierung weiterzuführen, wurden dem *APP*-Algorithmus noch weitere Features hinzugefügt:

Durch eine Modifikation der *find_passengers_along_the_way()*-Methode gibt diese auch Passagiere, deren Weg sich teilweise mit dem des Zuges überschneidet, zurück. Umgesetzt wird dies, indem zu Beginn der Ausführung für alle Passagiere der individuelle kürzeste Weg berechnet wird. Das so befüllte Dictionary wird beim Aufruf übergeben und untersucht. Da es auch vorkommen kann, dass der zusätzliche Passagier nur einen Teil der Strecke in Richtung seines Ziels mitgenommen wird und dennoch nicht am endgültigen Ziel angekommen ist, wurde die zusätzliche Instanzvariable *interim_target* als Attribut hinzugefügt. Ist dieses erreicht, hält der Zug beim Abfahren des Weges an und lässt den Passagier aussteigen. In einer späteren Runde wird er dann wieder abgeholt und zu seinem endgültigen Ziel gebracht.

Eine weitere Verbesserung des *APP* im Vergleich zum *SPP*-Algorithmus ist das *intelligente Swapping*. Hierbei wird, wenn *Swapping* (vgl. Kapitel 2.1.2) aufgrund von Kapazitätsengpässen erforderlich ist, ebenfalls die *find_passenger_along_the_way()*-Methode aufgerufen. So kann bei Bedarf dieser Zug ebenfalls von einem Passagier genutzt werden. Auch wenn der Fall nur sehr selten auftritt, kann so die sowieso erforderliche Zugbewegung genutzt werden, um die Gesamtverspätungszeit weiter zu senken.

4.2.2 Bewertung

Die Vorteile der *Passenger Parallelization Algotihms* liegen vor allem in der Parametrisierung, Neustrukturierung, Pfadspeicherung und Parallelisierung der Passagiere. Die positiven Aspekte der zugrundeliegenden Idee wurden bereits in der Bewertung des *SDI*-Algorithmus betrachtet (vgl. Kapitel 4.1.2).

Nun stellt sich noch die Frage, weshalb nicht nur der *APP*-Algorithmus zur Repräsentation der Passagier Parallelisierung genutzt wird, da dieser auf den ersten Blick betrachtet in jeder Hinsicht den *SPP*-Algorithmus schlägt. Allerdings muss beachtet werden, dass beim *APP*-Algorithmus Passagiere deutlich häufiger *boarden* und *detrainen*. Wie schon in Kapitel 2.1.1 bei den Annahmen in der Berechnung gezeigt werden dafür deutlich mehr Runden benötigt, da so nur selten Fälle auftreten, in denen Züge die gesamte Strecke durchfahren können und sich damit in speziellen Fällen mit doppelter Geschwindigkeit fortbewegen, was mit einer immensen Zeitersparnis einhergeht. Die Stärken des *APP*-Algorithmus werden also erst bei einer sehr großen Anzahl an Passagieren auf einem kleineren Schienennetzwerk ersichtlich (vgl. Kapitel 4.5). Hier kann er durch häufige Überschneidungen in den Wegen der Passagiere seine ganze Stärke ausspielen und wartet mit einem deutlich besseren Ergebnis im Vergleich zum *SPP*-Algorithmus auf. Bei kleineren Inputdateien ist jedoch in vielen Fällen der *SPP*-Algorithmus bei Betrachtung der kumulierten Verspätungszeit besser. Aus ebendiesen Gründen werden in der Ausführung der Softwarelösung beide Algorithmen angewendet, um immer die beste Lösung finden zu können.

In Bezug auf die Realität ist der *APP*-Algorithmus ein Abbild des anzunehmenden Passagierverhaltens. Steht ein Passagier an der Station bereit, an welcher ein Zug mit freien Kapazitäten in die gleiche Richtung abfährt, in der auch seine Zielstation liegt, wird er sicher einsteigen. Außerdem ist die kurze Laufzeit ein großer Vorteil für die produktive Anwendung. So ist beispielsweise ein Einsatz bei der Realtime-Berechnung denkbar. Auf großen Schienennetzen kann so ermittelt werden, welche Strecken bei dem Einsatz von nur einem Zug abgefahren werden müssten. Auf dieser Basis kann schnell eine Abschätzung vorgenommen werden, inwiefern die Parameter

Geschwindigkeit und Kapazität eine Rolle spielen und somit eine Kosten-Nutzen-Abwägung durchgeführt werden.

Größter Nachteil der in diesem Kapitel vorgestellten Algorithmen sind, wie auch schon bei der Bewertung des *SDI*-Algorithmus angemerkt, viele ungenutzte Kapazitäten, die durch eine Parallelisierung der Züge genutzt werden könnten.

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

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 Ergebnisqualität und Laufzeiten

Die vorhergehenden Abschnitte haben sich bereits einzeln mit einer Bewertung der Algorithmen auseinandergesetzt. Um nun die Algorithmen untereinander vergleichen zu können, wird in diesem Kapitel eine Analyse der Laufzeit und Ergebnisqualität vorgenommen. Dies ist für eine wissenschaftliche Auswertung unabdingbar. Zu diesem Zweck wurden auf einem Testsystem diese Daten erhoben. (Referenzsystem: Ryzen 5 5600X, 16 GB RAM, Linux Kernel 5.15)

Darüber hinaus wurden die Daten mit *matplotlib* in Python visualisiert. Die Grafiken sind in Abbildung 5 und Abbildung 6 dargestellt. Im Folgenden wird auf deren Kernaussagen eingegangen, um die in den vorhergehenden Kapiteln aufgestellten Thesen zu belegen.

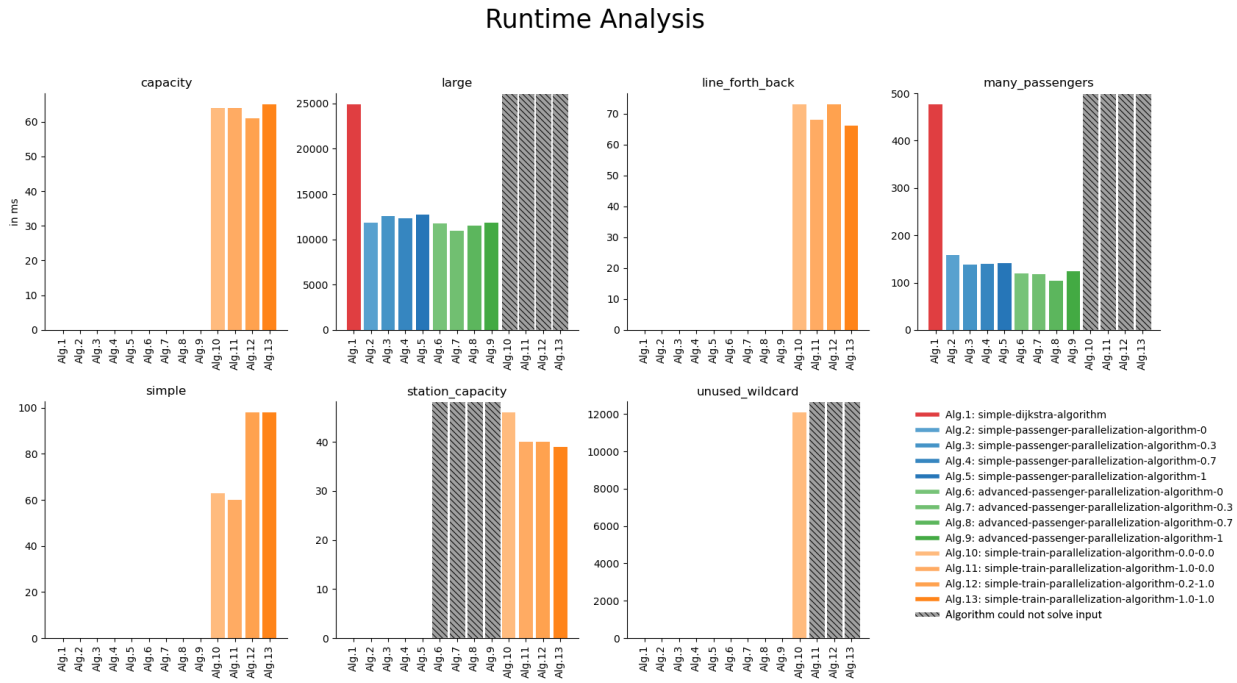

Abbildung 5: Runtime Analysis aller entwickelten Algorithmen

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

Bei der Laufzeitanalyse (Abbildung 5) lässt sich klar erkennen, dass die Instanzen des *STP*-Algorithmus die größte Laufzeit aufweisen und große Schienennetze nicht in annehmbarer Zeit lösen können. Aus diesem Grund wurden diese Zeiten nicht in die Grafik aufgenommen und gesondert markiert. Weiter muss zur Laufzeit gesagt werden, dass – wie oben bereits angemerkt – die *Passenger Parallelization Algorithms* eine erhebliche Verbesserung im Vergleich zum *SDI*-Algorithmus darstellen. Zuletzt sei noch hervorgehoben, dass bei kleinen Inputdateien die Algorithmen ohne Zug-Parallelisierung nahezu in Echtzeit einen gültigen Fahrplan bereitstellen.

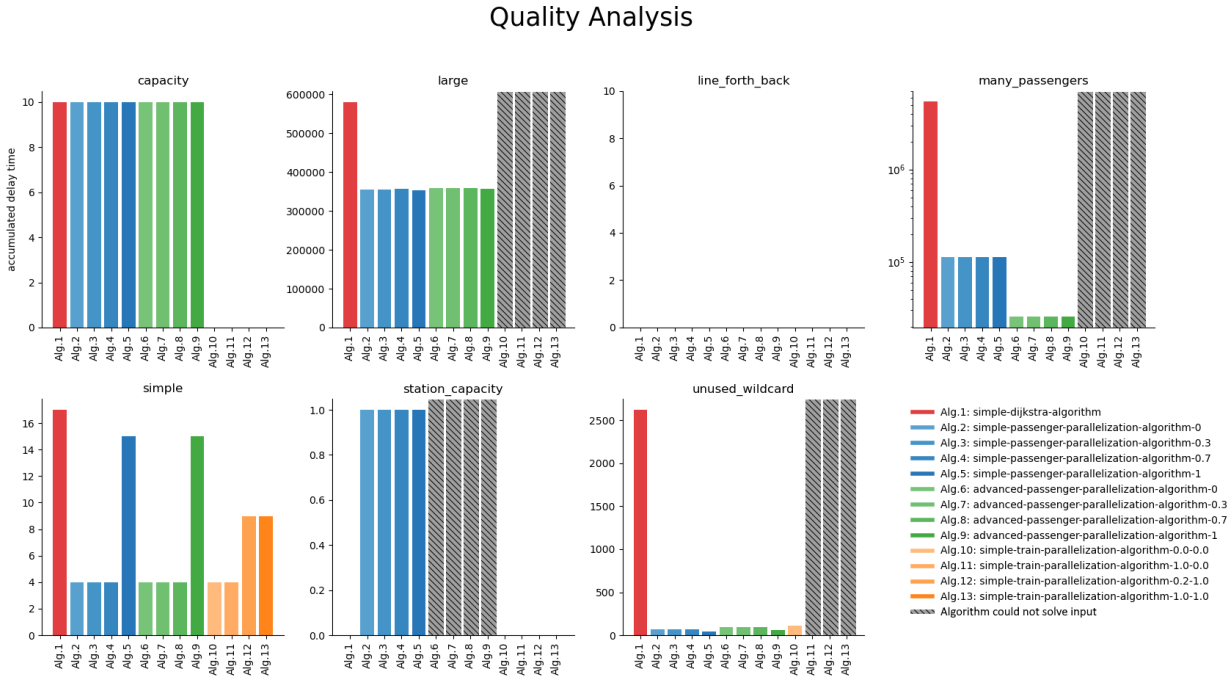

Abbildung 6: Quality Analysis aller entwickelten Algorithmen

Neben der Laufzeit eines Algorithmus ist aber vor allem die Qualität seiner Ergebnisse relevant. Diese wird, wie in der Problemstellung beschrieben, an der kumulierten Verspätungszeit aller Passagiere gemessen. Es wird deutlich, dass die verschiedenen Algorithmen für diverse Inputdateien unterschiedlich gute Ergebnisse liefern. Damit bewahrheitet sich die zu Beginn geäußerte Vermutung, dass kein Algorithmus für alle Probleme perfekt angewandt werden kann. Vielmehr ist es das Orchester an Algorithmen, dass im Zusammenklang die beste Lösung bereitstellt. Die Ergebnisse der Qualitätsanalyse bestätigen somit auch den gewählten Lösungsansatz.

Weiter zeigt die Auswertung mit einer konstruierten Inputdatei (*many_passengers*, Anhang 2), dass der weiterentwickelte *APP*-Algorithmus in speziellen Fällen durchaus eine erhebliche Verbesserung gegenüber dem *SPP*-Algorithmus darstellt. Auffällig ist dabei die sehr hohe Verspätungszeit aller Algorithmen. Diese wurde bewusst durch eine niedrige *target_time* der

Passagiere provoziert, da das Ziel bei der Generierung nicht etwa eine niedrige kumulierte Verspätungszeit war, sondern der Vergleichbarkeit der beiden genannten Algorithmen dienen sollte.

Ebenso zeigen die Grafiken (Abbildung 6), dass sich unterschiedliche Parametrisierungen eines Algorithmus (hier jeweils im selben Farbton dargestellt) auf das Ergebnis auswirken. Auch wenn der Effekt teilweise marginal ausfällt, rechtfertigt dies die Verwendung von Parametern, damit so durch Finetuning der Algorithmen die Ergebnisse für unterschiedliche Schienennetze verbessert werden können.

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

```
# Bahnhoefe
[Stations]
S1 1
S2 1
S3 1
S4 1
S5 1
S6 1
S7 1
S8 1
S9 1
S10 1
S11 1
S12 1
S13 1
S14 1
S15 1
S16 1
S17 1
S18 1
S19 1
S20 1

# Strecken
[Lines]
L1 S10 S3 5 1
L2 S6 S13 20 1
L3 S3 S20 15 1
L4 S2 S15 2 1
L5 S13 S4 18 1
L6 S3 S14 7 1
L7 S13 S10 13 1
L8 S15 S7 19 1
L9 S9 S10 12 1
L10 S9 S10 16 1
L11 S9 S8 20 1
L12 S1 S13 14 1
L13 S5 S2 10 1
L14 S8 S9 19 1
L15 S10 S1 20 1
L16 S10 S19 18 1
L17 S1 S2 15 1
L18 S4 S14 19 1
L19 S9 S11 17 1
L20 S13 S12 18 1
L21 S7 S2 16 1
L22 S18 S6 15 1
L23 S3 S2 17 1
L24 S2 S13 11 1
L25 S9 S4 10 1
L26 S20 S13 2 1
L27 S14 S13 3 1
L28 S15 S14 18 1
L29 S17 S20 18 1
L30 S5 S4 9 1
L31 S4 S19 13 1
L32 S13 S15 18 1
L33 S16 S5 12 1
L34 S19 S4 10 1
L35 S8 S10 12 1
L36 S17 S3 1 1
L37 S7 S18 1 1
L38 S14 S19 1 1
L39 S17 S6 15 1
L40 S4 S8 1 1
L41 S11 S12 13 1
L42 S9 S8 12 1
L43 S9 S7 3 1
L44 S8 S10 13 1
L45 S3 S6 17 1
L46 S12 S16 11 1
L47 S14 S6 9 1
L48 S20 S19 5 1
L49 S4 S10 10 1
L50 S5 S18 20 1
L51 S7 S20 18 1
L52 S8 S10 3 1
L53 S9 S1 16 1
L54 S10 S2 4 1
L55 S12 S11 17 1
L56 S14 S3 9 1
L57 S15 S17 20 1
L58 S17 S5 6 1
L59 S18 S4 13 1
L60 S19 S19 10 1
L61 S20 S7 3 1

# Zuege
 [Trains]
T1 S10 4 142

# Passagiere
[Passengers]
P1 S15 S1 4 1900
P2 S15 S12 1 155
P3 S6 S14 5 1167
P4 S14 S17 5 745
P5 S10 S18 5 1108
P6 S7 S20 2 1246
P7 S3 S18 2 951
P8 S1 S9 1 646
P9 S9 S19 3 1813
P10 S4 S6 1 3
P11 S20 S13 1 261
P12 S3 S13 3 811
P13 S2 S20 4 1831
P14 S15 S12 1 233
P15 S9 S10 3 772
P16 S5 S12 3 906
P17 S10 S17 4 1240
P18 S1 S2 3 1330
P19 S20 S4 4 602
P20 S15 S2 4 898
P21 S17 S3 2 1734
P22 S10 S6 4 1208
P23 S3 S8 5 8
P24 S10 S18 1 1797
P25 S19 S1 1 746
P26 S6 S19 5 220
P27 S2 S1 1 42
P28 S17 S19 1 1952
P29 S13 S2 3 296
P30 S17 S15 3 1329
P31 S13 S4 3 1805
P32 S1 S11 2 309
P33 S1 S4 3 194
P34 S8 S15 5 174
P35 S14 S1 3 1500
P36 S16 S12 3 1540
P37 S14 S13 1 772
P38 S16 S10 4 925
P39 S13 S2 1 264
P40 S20 S15 1 585
P41 S17 S13 3 1831
P42 S17 S20 2 45
P43 S9 S19 5 1669
P44 S2 S10 5 694
P45 S16 S19 1 1519
P46 S8 S7 3 1134
P47 S1 S17 4 1497
P48 S12 S2 5 1590
P49 S17 S14 5 393
P50 S17 S6 5 1473
P51 S13 S20 4 696
P52 S20 S19 2 1380
P53 S4 S13 4 584
P54 S7 S6 1 774
P55 S9 S3 2 705
P56 S12 S8 5 537
P57 S6 S5 2 1575
P58 S1 S18 1 1326
P59 S16 S11 5 754
P60 S17 S13 3 1551
P61 S7 S11 4 1410
P62 S19 S18 4 1308
P63 S3 S13 1 806
P64 S6 S4 2 702
P65 S17 S16 4 1142
P66 S20 S12 1 651
P67 S2 S16 5 1521
P68 S7 S13 3 1740
P69 S1 S5 5 1688
P70 S6 S20 4 1748
P71 S5 S16 3 612
P72 S1 S5 5 113
P73 S7 S14 1 509
P74 S13 S7 4 265
P75 S5 S15 5 243
P76 S14 S15 1 1500
P77 S10 S18 1 1642
P78 S19 S1 4 616
P79 S5 S12 1 385
P80 S13 S6 4 1391
P81 S16 S2 3 1709
P82 S13 S2 3 112
P83 S6 S11 2 240
P84 S18 S10 5 803
P85 S19 S12 5 1564
P86 S15 S14 1 1254
P87 S6 S3 3 503
P88 S16 S13 3 415
P89 S7 S9 4 485
P90 S15 S9 1 1268
P91 S17 S5 1 1164
P92 S4 S20 1 150
P93 S5 S7 1 1187
P94 S18 S11 4 1983
P95 S16 S19 5 620
P96 S3 S19 5 30
P97 S4 S8 1 359
P98 S12 S9 3 1926
P99 S5 S6 3 944
P100 S16 S15 5 1062
P101 S9 S13 5 1818
P102 S3 S2 4 1182
P103 S19 S5 3 1159
P104 S15 S5 5 799
P105 S8 S17 3 857
P106 S7 S17 3 1210
P107 S15 S20 5 131
P108 S6 S12 2 1239
P109 S16 S14 3 901
P110 S20 S15 4 604
P111 S3 S5 3 444
P112 S20 S4 2 390
P113 S11 S10 4 668
P114 S5 S20 4 500
P115 S14 S3 4 1761
P116 S16 S14 3 1559
P117 S1 S2 2 421
P118 S18 S4 2 285
P119 S12 S8 4 1169
P120 S4 S8 3 1559
P121 S3 S19 5 940
P122 S12 S15 4 1178
P123 S2 S11 5 1358
P124 S2 S8 3 1031
P125 S10 S12 4 917
P126 S5 S4 3 1989
P127 S8 S10 1 1115
P128 S12 S18 2 1524
P129 S6 S12 5 1040
P130 S2 S12 4 1137
P131 S14 S11 1 302
P132 S8 S12 1 410
P133 S18 S1 5 295
P134 S15 S19 3 1555
P135 S3 S17 1 520
P136 S8 S7 1 1235
P137 S13 S3 5 1797
P138 S9 S8 3 1677
P139 S2 S12 1 628
P140 S18 S8 4 862
P141 S14 S13 1 48
P142 S14 S16 3 1088
P143 S4 S9 1 571
P144 S18 S6 2 1095
P145 S5 S13 1 307
P146 S8 S12 3 1727
P147 S10 S9 4 1033
P148 S16 S3 1 1511
P149 S18 S7 4 573
P150 S13 S18 3 990
P151 S17 S12 2 497
P152 S18 S19 2 183
P153 S1 S19 4 40
P154 S9 S2 5 1737
P155 S18 S20 2 1196
P156 S15 S3 3 294
P157 S1 S4 2 1695
P158 S5 S12 4 592
P159 S18 S6 5 1007
P160 S5 S4 3 1688
P161 S19 S7 5 1269
P162 S11 S20 4 1429
P163 S6 S1 4 1212
P164 S20 S17 3 919
P165 S9 S17 2 122
P166 S15 S14 4 1516
P167 S17 S19 3 1073
P168 S14 S5 1 195
P169 S14 S9 1 1116
P170 S3 S14 2 910
P171 S2 S15 4 88
P172 S6 S9 5 1441
P173 S14 S7 4 966
P174 S13 S8 4 1171
P175 S4 S1 5 136
P176 S9 S16 2 1634
P177 S10 S19 5 401
P178 S13 S5 4 283
P179 S16 S4 3 1720
P180 S10 S11 3 1563
P181 S13 S4 5 1774
P182 S5 S4 2 1805
P183 S7 S13 2 1609
P184 S11 S9 2 1395
P185 S4 S20 2 217
P186 S19 S17 5 704
P187 S13 S4 3 1616
P188 S4 S17 4 1928
P189 S1 S2 3 1463
P190 S10 S14 4 498
P191 S18 S15 1 147
P192 S5 S6 4 1576
P193 S14 S17 3 1653
P194 S6 S18 4 1706
P195 S5 S8 5 986
P196 S16 S15 1 922
P197 S20 S19 2 45
P198 S16 S9 4 1086
P199 S7 S6 5 1238
P200 S19 S17 2 1708
P201 S9 S17 5 1300
P202 S12 S11 1 1716
P203 S16 S3 5 853
P204 S13 S16 3 396
P205 S17 S5 3 1583
P206 S11 S7 3 1932
P207 S4 S14 5 1993
P208 S2 S19 2 133
P209 S3 S12 1 712
P210 S8 S13 5 1467
P211 S6 S17 4 1660
P212 S1 S2 4 1732
P213 S1 S3 1 1951
P214 S13 S7 2 823
P215 S3 S11 5 518
P216 S9 S5 5 1119
P217 S9 S4 1 638
P218 S18 S12 2 1797
P219 S8 S7 3 122
P220 S2 S14 2 1527
P221 S7 S6 3 1502
P222 S7 S6 2 1360
P223 S3 S20 2 1442
P224 S13 S4 1 1436
P225 S13 S12 1 529
P226 S18 S8 4 511
P227 S15 S12 1 1914
P228 S5 S4 4 886
P229 S11 S13 1 1617
P230 S10 S9 2 91
P231 S14 S16 4 149
P232 S15 S16 2 782
P233 S5 S15 1 416
P234 S15 S14 2 980
P235 S3 S17 5 1730
P236 S9 S5 3 1649
P237 S5 S19 3 693
P238 S9 S18 2 747
P239 S19 S2 1 534
P240 S11 S18 3 15
P241 S3 S4 3 1978
P242 S19 S18 5 1759
P243 S13 S11 1 1341
P244 S13 S20 5 94
P245 S5 S9 1 1928
P246 S6 S19 2 1149
P247 S10 S7 5 1290
P248 S2 S12 2 1211
P249 S13 S15 3 1314
P250 S16 S18 3 1136
P251 S14 S8 4 1826
P252 S15 S7 2 1623
P253 S12 S10 3 560
P254 S9 S3 3 278
P255 S6 S13 3 1001
P256 S19 S1 1 17
P257 S3 S13 5 604
P258 S20 S7 3 601
P259 S17 S8 5 196
P260 S11 S16 5 1004
P261 S1 S18 4 893
P262 S7 S11 3 1414
P263 S2 S9 3 1623
P264 S10 S18 1 108
P265 S7 S17 1 77
P266 S12 S6 1 447
P267 S16 S11 1 353
P268 S18 S3 2 280
P269 S18 S20 2 1427
P270 S18 S17 3 474
P271 S15 S18 3 1808
P272 S15 S6 4 234
P273 S5 S15 3 1717
P274 S13 S11 3 146
P275 S1 S18 5 336
P276 S5 S4 4 763
P277 S4 S18 1 1477
P278 S15 S1 1 1109
P279 S17 S15 5 1289
P280 S11 S7 5 409
P281 S7 S4 2 1352
P282 S14 S6 5 402
P283 S2 S7 3 993
P284 S9 S19 5 531
P285 S14 S7 1 1788
P286 S9 S2 5 1859
P287 S11 S2 4 460
P288 S1 S4 5 1544
P289 S18 S6 1 836
P290 S15 S13 3 515
P291 S8 S13 1 782
P292 S20 S15 4 1632
P293 S18 S5 5 1874
P294 S17 S8 5 1766
P295 S20 S4 4 748
P296 S8 S7 4 1225
P297 S12 S2 4 110
P298 S17 S2 2 1161
P299 S13 S3 5 1817
P300 S6 S14 4 1137
P301 S5 S11 5 1626
P302 S6 S12 2 596
P303 S3 S9 3 1055
P304 S4 S3 5 1129
P305 S7 S3 5 738
P306 S7 S17 2 577
P307 S5 S3 2 454
P308 S7 S20 3 1023
P309 S8 S6 1 1666
P310 S1 S13 5 1699
P311 S3 S9 4 1778
P312 S6 S7 2 1438
P313 S4 S9 4 781
P314 S20 S19 1 1069
P315 S9 S2 4 1838
P316 S15 S14 5 728
P317 S11 S7 3 260
P318 S17 S11 5 939
P319 S11 S14 1 462
P320 S20 S8 3 1160
P321 S11 S10 4 549
P322 S15 S14 1 207
P323 S14 S16 1 522
P324 S10 S5 1 1353
P325 S13 S3 1 987
P326 S20 S19 4 1689
P327 S19 S6 2 1972
P328 S3 S11 4 425
P329 S18 S2 4 113
P330 S2 S20 1 1798
P331 S10 S17 5 865
P332 S3 S14 3 1386
P333 S11 S2 5 224
P334 S14 S20 5 232
P335 S17 S4 3 1369
P336 S5 S2 5 1918
P337 S9 S19 3 1492
P338 S4 S6 1 581
P339 S2 S7 2 984
P340 S12 S17 3 68
P341 S3 S14 2 177
P342 S12 S16 2 18
P343 S16 S3 3 1423
P344 S18 S2 3 1231
P345 S11 S13 1 1246
P346 S5 S17 2 31
P347 S15 S8 1 1242
P348 S13 S6 5 283
P349 S17 S3 3 1858
P350 S16 S4 2 1427
P351 S14 S11 3 554
P352 S1 S20 1 1152
P353 S5 S3 1 357
P354 S4 S20 5 1083
P355 S10 S9 4 228
P356 S4 S8 4 1167
P357 S15 S9 4 1547
P358 S3 S2 5 1340
P359 S18 S3 4 514
P360 S7 S6 5 920
P361 S10 S2 3 729
P362 S9 S13 2 1654
P363 S4 S5 4 444
P364 S15 S9 5 205
P365 S8 S16 2 404
P366 S4 S5 5 1637
P367 S2 S1 1 1561
P368 S10 S2 2 746
P369 S4 S12 5 1858
P370 S9 S11 5 262
P371 S2 S8 4 287
P372 S16 S17 3 1181
P373 S1 S7 4 666
P374 S18 S20 5 1773
P375 S2 S4 3 1312
P376 S6 S10 2 295
P377 S7 S16 4 1489
P378 S9 S10 3 1733
P379 S14 S2 5 1061
P380 S19 S11 4 1638
P381 S9 S18 2 1110
P382 S14 S16 3 573
P383 S4 S1 1 89
P384 S19 S18 5 278
P385 S16 S20 1 530
P386 S13 S7 3 1057
P387 S16 S8 3 1850
P388 S19 S16 3 7
P389 S7 S18 3 1807
P390 S16 S18 4 1214
P391 S11 S6 3 1638
P392 S13 S14 3 1851
P393 S13 S2 4 1129
P394 S11 S2 1 259
P395 S18 S13 3 1883
P396 S14 S15 2 1474
P397 S18 S12 1 128
P398 S7 S9 1 534
P399 S5 S16 2 1876
P400 S17 S18 3 1738
P401 S4 S18 3 675
P402 S7 S5 4 1522
P403 S13 S8 2 769
P404 S20 S8 5 1355
P405 S11 S7 4 1059
P406 S14 S18 4 1099
P407 S2 S3 4 824
P408 S12 S19 1 1226
P409 S14 S1 5 1199
P410 S4 S15 5 417
P411 S2 S1 4 1704
P412 S12 S1 5 341
P413 S5 S20 2 889
P414 S8 S18 3 1363
P415 S11 S9 5 295
P416 S7 S16 1 887
P417 S7 S12 3 64
P418 S8 S14 2 1548
P419 S9 S1 5 1442
P420 S18 S13 3 132
P421 S5 S13 1 1674
P422 S1 S14 2 242
P423 S5 S10 5 1825
P424 S14 S8 3 188
P425 S1 S6 5 1451
P426 S7 S8 1 1129
P427 S11 S16 3 1292
P428 S12 S20 2 866
P429 S3 S5 1 753
P430 S12 S2 3 1145
P431 S3 S13 1 281
P432 S7 S16 1 721
P433 S17 S18 3 87
P434 S3 S6 2 1531
P435 S7 S6 5 1338
P436 S19 S11 4 339
P437 S12 S9 5 842
P438 S5 S15 5 621
P439 S18 S6 2 290
P440 S14 S4 2 1281
P441 S9 S14 5 460
P442 S16 S6 3 1409
P443 S6 S9 2 1687
P444 S11 S18 5 104
P445 S11 S16 1 1765
P446 S3 S2 3 1919
P447 S16 S10 4 1275
P448 S11 S10 3 360
P449 S3 S2 4 722
P450 S15 S18 4 644
P451 S17 S5 4 1159
P452 S1 S6 4 1167
P453 S14 S17 3 336
P454 S7 S6 4 1597
P455 S9 S18 3 1521
P456 S3 S7 3 633
P457 S18 S19 3 1043
P458 S14 S13 3 1195
P459 S12 S11 3 1203
P460 S15 S9 5 397
P461 S13 S6 5 263
P462 S4 S19 5 1159
P463 S16 S7 2 216
P464 S4 S1 1 584
P465 S20 S17 5 839
P466 S5 S11 5 498
P467 S15 S3 2 770
P468 S5 S10 3 1278
P469 S19 S10 3 1639
P470 S14 S8 3 1882
P471 S10 S12 5 1446
P472 S3 S20 4 1897
P473 S19 S13 2 883
P474 S18 S5 4 842
P475 S17 S2 4 1453
P476 S15 S8 3 57
P477 S1 S6 3 1044
```

```
P478 S1 S9 1 1912
P479 S6 S18 5 350
P480 S17 S2 3 308
P481 S15 S7 4 1418
P482 S4 S20 5 39
P483 S1 S3 4 1624
P484 S15 S16 2 644
P485 S10 S8 3 121
P486 S15 S19 2 1300
P487 S1 S17 4 1342
P488 S9 S18 4 780
P489 S6 S4 5 659
P490 S15 S9 5 726
P491 S17 S3 5 382
P492 S2 S18 4 1996
P493 S11 S4 1 1873
P494 S7 S16 2 992
P495 S20 S18 4 919
P496 S20 S16 3 1249
P497 S8 S10 1 658
P498 S8 S7 4 631
P499 S5 S4 1 1555
P500 S1 S11 2 1692
P501 S4 S5 4 525
P502 S6 S7 5 40
P503 S20 S12 2 868
P504 S14 S11 1 498
P505 S7 S20 2 734
P506 S10 S13 1 1570
P507 S11 S6 3 658
P508 S12 S9 4 402
P509 S10 S4 5 925
P510 S17 S6 3 1140
P511 S4 S9 1 1286
P512 S7 S8 5 1270
P513 S4 S20 2 1770
P514 S19 S7 3 156
P515 S13 S6 5 1494
P516 S13 S12 5 204
P517 S13 S11 5 725
P518 S14 S5 2 1844
P519 S3 S7 1 9
P520 S8 S15 3 1404
P521 S11 S20 2 1937
P522 S14 S15 2 1869
P523 S18 S11 5 1752
P524 S9 S8 5 1787
P525 S18 S13 1 822
P526 S13 S17 1 1804
P527 S18 S19 1 1278
P528 S11 S19 3 1218
P529 S14 S10 2 259
P530 S17 S9 4 1794
P531 S18 S8 4 1461
P532 S14 S10 1 268
P533 S13 S15 3 1681
P534 S12 S8 3 176
P535 S17 S9 2 1971
P536 S5 S8 2 793
P537 S9 S4 1 615
P538 S13 S16 3 231
P539 S19 S13 3 1896
P540 S21 S20 5 6
P541 S1 S8 4 149
P542 S9 S1 1 1365
P543 S1 S7 3 1085
P544 S3 S14 5 433
P545 S7 S13 5 463
P546 S12 S10 1 1813
P547 S1 S11 1 1911
P548 S16 S6 1 1504
P549 S10 S14 4 1565
P550 S15 S3 1 858
P551 S19 S4 5 1401
P552 S15 S9 2 1631
P553 S14 S2 5 1772
P554 S16 S7 4 788
P555 S11 S7 1 1343
P556 S18 S10 4 9
P557 S12 S10 4 318
P558 S12 S16 3 857
P559 S2 S5 3 717
P560 S7 S10 1 423
P561 S19 S5 4 838
P562 S18 S20 1 1066
P563 S19 S13 1 560
P564 S10 S3 1 247
P565 S4 S17 1 157
P566 S3 S20 4 1601
P567 S19 S18 5 62
P568 S7 S13 1 148
P569 S4 S8 4 1692
P570 S2 S6 4 107
P571 S3 S13 1 1037
P572 S6 S1 3 481
P573 S6 S7 2 1386
P574 S15 S19 4 1926
P575 S8 S10 4 528
P576 S10 S12 1 580
P577 S15 S1 1 226
P578 S1 S7 5 1383
P579 S19 S12 3 181
P580 S10 S11 1 449
P581 S12 S20 5 728
P582 S1 S9 4 595
P583 S15 S5 3 619
P584 S20 S8 4 851
P585 S4 S6 1 1669
P586 S17 S2 5 724
P587 S2 S11 2 782
P588 S12 S11 3 519
P589 S19 S18 3 406
P590 S19 S1 3 569
P591 S7 S10 3 1874
P592 S5 S20 1 1548
P593 S12 S10 5 736
P594 S3 S11 5 1875
P595 S6 S3 5 1752
P596 S4 S17 5 664
P597 S19 S13 5 1841
P598 S6 S15 4 876
P599 S11 S5 4 146
P600 S4 S14 1 1222
P601 S15 S11 1 74
P602 S11 S2 4 281
P603 S16 S19 2 800
P604 S5 S11 5 1805
P605 S20 S11 3 1466
P606 S11 S12 5 1206
P607 S11 S17 3 1112
P608 S1 S8 5 409
P609 S5 S1 5 63
P610 S1 S3 2 1173
P611 S13 S12 4 424
P612 S18 S19 4 1689
P613 S5 S4 2 513
P614 S3 S5 3 245
P615 S20 S19 3 1854
P616 S12 S14 5 651
P617 S5 S4 2 53
P618 S5 S9 5 1290
P619 S6 S1 2 1252
P620 S8 S18 3 754
P621 S7 S6 2 1498
P622 S4 S10 1 1217
P623 S7 S19 2 232
P624 S1 S12 3 1081
P625 S18 S1 4 963
P626 S10 S3 2 1094
P627 S3 S16 3 343
P628 S19 S9 1 920
P629 S2 S1 3 1877
P630 S9 S8 4 989
P631 S12 S19 2 672
P632 S16 S12 1 21
P633 S14 S1 3 1615
P634 S13 S15 5 530
P635 S13 S20 2 750
P636 S10 S20 1 1724
P637 S16 S19 3 244
P638 S9 S6 2 691
P639 S6 S4 1 1172
P640 S14 S4 5 210
P641 S9 S4 1 598
P642 S3 S6 4 1392
P643 S19 S5 1 790
P644 S13 S3 4 955
P645 S19 S16 3 262
P646 S18 S12 4 215
P647 S7 S9 1 371
P648 S17 S14 1 1332
P649 S3 S20 2 145
P650 S6 S1 2 1096
P651 S8 S2 5 649
P652 S14 S19 5 682
P653 S9 S8 1 110
P654 S1 S15 1 431
P655 S16 S6 5 881
P656 S5 S2 1 127
P657 S3 S10 5 604
P658 S3 S1 2 791
P659 S18 S17 3 164
P660 S8 S20 4 984
P661 S17 S16 5 1281
P662 S20 S19 1 344
P663 S4 S11 1 1239
P664 S5 S6 1 1517
P665 S4 S1 3 1772
P666 S10 S4 4 351
P667 S2 S4 5 1049
P668 S17 S13 1 753
P669 S2 S1 3 1603
P670 S5 S1 4 1043
P671 S5 S6 4 1369
P672 S15 S16 1 1669
P673 S1 S12 2 1202
P674 S20 S11 5 1405
P675 S16 S1 2 162
P676 S8 S19 1 1963
P677 S9 S14 5 284
P678 S11 S15 3 1373
P679 S5 S8 1 1667
P680 S10 S12 3 639
P681 S17 S4 5 845
P682 S12 S16 1 682
P683 S3 S18 4 114
P684 S3 S15 4 1718
P685 S18 S17 3 238
P686 S4 S16 3 1580
P687 S19 S3 1 1041
P688 S2 S1 2 506
P689 S3 S1 3 1516
P690 S8 S20 4 1416
P691 S19 S2 3 1318
P692 S5 S17 3 846
P693 S16 S2 5 632
P694 S11 S16 1 1264
P695 S10 S9 5 1074
P696 S18 S20 5 1214
P697 S10 S7 5 1696
P698 S19 S9 1 890
P699 S10 S13 2 881
P700 S17 S18 5 1765
P701 S5 S7 4 1056
P702 S14 S17 1 1969
P703 S19 S6 2 885
P704 S15 S1 4 227
P705 S7 S6 1 676
P706 S13 S7 2 467
P707 S13 S4 3 1595
P708 S2 S17 4 1365
P709 S4 S5 4 1934
P710 S17 S19 2 265
P711 S16 S7 3 777
P712 S7 S11 2 1709
P713 S20 S13 1 746
P714 S6 S20 4 1561
P715 S18 S15 5 1809
P716 S6 S7 1 1486
P717 S4 S10 1 366
P718 S20 S14 1 794
P719 S3 S6 1 632
P720 S19 S2 5 1262
P721 S13 S10 1 1262
P722 S19 S3 5 1624
P723 S20 S16 4 384
P724 S15 S19 5 570
P725 S5 S6 4 1807
P726 S17 S16 5 786
P727 S13 S12 5 596
P728 S13 S11 3 691
P729 S6 S13 3 650
P730 S16 S15 1 354
P731 S20 S3 4 323
P732 S6 S12 3 1645
P733 S14 S15 3 1168
P734 S8 S7 3 1945
P735 S13 S2 1 1272
P736 S10 S9 5 682
P737 S19 S15 4 602
P738 S13 S19 1 1566
P739 S9 S19 5 309
P740 S6 S18 2 133
P741 S16 S4 2 1820
P742 S20 S12 3 1556
P743 S11 S20 5 70
P744 S19 S16 4 1579
P745 S10 S3 3 405
P746 S13 S16 3 1467
P747 S9 S6 5 430
P748 S14 S7 3 937
P749 S1 S3 5 1674
P750 S8 S15 4 1430
P751 S16 S12 5 627
P752 S3 S8 4 223
P753 S3 S2 5 1227
P754 S7 S4 3 1443
P755 S4 S13 2 1026
P756 S6 S7 4 488
P757 S2 S5 1 645
P758 S19 S5 4 986
P759 S12 S11 4 283
P760 S19 S14 3 1065
P761 S7 S8 1 417
P762 S8 S12 2 427
P763 S10 S4 3 19
P764 S17 S7 1 91
P765 S14 S6 4 1851
P766 S8 S13 4 475
P767 S16 S12 3 1316
P768 S3 S1 5 1666
P769 S11 S18 4 114
P770 S14 S17 2 1087
P771 S17 S10 3 329
P772 S3 S13 1 1810
P773 S11 S5 3 742
P774 S19 S12 1 1941
P775 S5 S16 5 1922
P776 S14 S13 2 1616
P777 S18 S13 1 301
P778 S15 S17 3 1700
P779 S20 S10 3 1633
P780 S5 S11 1 325
P781 S12 S1 5 1762
P782 S12 S2 3 57
P783 S4 S9 4 1454
P784 S14 S18 5 202
P785 S8 S14 5 1216
P786 S19 S18 1 843
P787 S7 S12 1 434
P788 S6 S15 1 1672
P789 S19 S18 4 1212
P790 S4 S7 4 1983
P791 S6 S12 1 1144
P792 S1 S12 1 311
P793 S8 S17 3 1184
P794 S20 S9 3 1508
P795 S3 S6 1 552
P796 S5 S17 2 1904
P797 S15 S1 3 659
P798 S14 S13 5 1207
P799 S13 S14 1 51
P800 S4 S5 2 1870
P801 S1 S19 1 192
P802 S8 S9 1 1703
P803 S18 S19 5 1446
P804 S8 S18 2 1718
P805 S11 S7 4 1875
P806 S11 S14 2 1583
P807 S1 S4 1 285
P808 S9 S8 5 337
P809 S19 S2 5 675
P810 S10 S4 3 1860
P811 S4 S8 2 1701
P812 S16 S5 4 864
P813 S8 S9 2 75
P814 S13 S6 5 1152
P815 S14 S8 3 82
P816 S15 S16 3 1836
P817 S7 S6 5 1176
P818 S8 S7 1 1918
P819 S5 S1 5 513
P820 S11 S9 4 1838
P821 S11 S6 5 415
P822 S10 S17 3 1767
P823 S19 S18 3 317
P824 S14 S6 4 278
P825 S7 S20 1 1076
P826 S10 S20 2 165
P827 S13 S10 5 1026
P828 S5 S4 2 1458
P829 S11 S18 1 1693
P830 S7 S12 5 1773
P831 S19 S14 5 318
P832 S11 S20 2 1339
P833 S11 S2 1 194
P834 S11 S10 1 946
P835 S3 S2 3 476
P836 S9 S20 3 1214
P837 S6 S7 2 1648
P838 S18 S15 4 922
P839 S13 S8 2 1121
P840 S9 S1 3 1431
P841 S9 S14 2 577
P842 S14 S13 1 625
P843 S3 S19 4 1078
P844 S11 S16 5 334
P845 S8 S15 4 4
P846 S9 S18 2 432
P847 S2 S1 5 205
P848 S16 S2 4 1628
P849 S9 S7 3 740
P850 S19 S20 3 308
P851 S18 S20 1 744
P852 S17 S4 3 1753
P853 S4 S10 4 1434
P854 S11 S16 5 626
P855 S5 S10 2 880
P856 S17 S10 3 1204
P857 S19 S18 2 1323
P858 S1 S5 5 457
P859 S16 S2 2 1120
P860 S6 S1 1 152
P861 S18 S15 1 646
P862 S5 S15 4 500
P863 S2 S13 5 500
P864 S5 S2 5 1080
P865 S6 S11 2 117
P866 S15 S6 3 1614
P867 S16 S20 2 1331
P868 S8 S18 4 353
P869 S8 S6 4 1172
P870 S18 S1 4 948
P871 S2 S6 2 553
P872 S8 S18 1 809
P873 S5 S14 2 1017
P874 S15 S13 3 1863
P875 S21 S20 3 781
P876 S10 S3 3 797
P877 S20 S18 5 774
P878 S20 S5 4 1642
P879 S6 S1 2 1099
P880 S4 S2 3 1292
P881 S15 S2 2 1837
P882 S17 S18 2 710
P883 S20 S5 1 566
P884 S8 S5 2 421
P885 S10 S12 3 1905
P886 S21 S20 3 69
P887 S7 S3 5 633
P888 S11 S2 3 229
P889 S9 S11 3 1164
P890 S8 S5 3 1766
P891 S16 S15 4 1907
P892 S3 S17 5 1027
P893 S20 S6 3 1420
P894 S2 S18 5 1099
P895 S8 S14 3 556
P896 S10 S2 2 1892
P897 S5 S16 3 946
P898 S11 S10 5 421
P899 S8 S16 1 1553
P900 S15 S12 5 1701
P901 S14 S18 3 1721
P902 S15 S5 2 536
P903 S12 S3 2 580
P904 S4 S14 2 1255
P905 S19 S6 4 1019
P906 S3 S2 4 124
P907 S1 S7 1 540
P908 S4 S16 1 1040
P909 S12 S19 2 1977
P910 S12 S13 5 1374
P911 S20 S7 1 305
P912 S4 S17 5 891
P913 S1 S16 5 1847
P914 S12 S19 5 573
P915 S14 S2 5 1679
P916 S20 S3 2 649
P917 S14 S1 4 829
P918 S1 S14 4 1528
P919 S10 S19 2 1430
P920 S8 S9 2 187
P921 S9 S5 2 1270
P922 S11 S16 2 1612
P923 S8 S11 5 1633
P924 S8 S7 2 280
P925 S7 S2 5 1503
P926 S11 S2 4 1807
P927 S18 S17 2 1066
P928 S11 S12 4 1616
P929 S1 S12 4 353
P930 S13 S20 2 716
P931 S7 S19 2 62
P932 S14 S7 4 1387
P933 S10 S9 4 690
P934 S9 S12 4 1829
P935 S20 S7 4 1965
P936 S12 S17 4 773
P937 S6 S8 1 1986
P938 S13 S20 4 1149
P939 S17 S13 3 390
P940 S7 S6 3 1332
P941 S9 S12 4 593
P942 S4 S18 4 824
P943 S12 S17 1 41
P944 S4 S3 5 976
P945 S16 S9 2 1514
P946 S10 S18 5 1190
P947 S2 S13 4 628
P948 S5 S7 1 250
P949 S15 S11 3 1614
P950 S3 S15 3 1917
P951 S14 S18 5 288
P952 S8 S11 2 1765
P953 S1 S9 1 1665
P954 S2 S7 5 1822
P955 S17 S16 1 454
P956 S9 S8 1 1427
P957 S19 S5 1 126
P958 S10 S16 2 1833
P959 S19 S4 3 274
P960 S17 S4 2 932
P961 S11 S8 3 1212
P962 S2 S3 4 1853
P963 S5 S15 1 1480
P964 S6 S9 1 751
P965 S9 S11 1 1440
P966 S12 S5 5 1284
P967 S19 S18 2 1262
P968 S16 S12 1 336
P969 S19 S7 5 107
P970 S9 S4 4 1715
P971 S20 S19 4 1008
P972 S6 S13 4 1601
P973 S15 S6 2 1111
P974 S12 S9 5 1275
P975 S15 S20 3 1803
P976 S18 S6 1 924
P977 S15 S17 5 66
P978 S2 S20 4 1119
P979 S11 S14 2 793
P980 S10 S16 3 1711
P981 S8 S7 5 716
P982 S5 S12 5 1503
P983 S16 S5 2 695
P984 S3 S5 5 303
P985 S7 S15 4 1120
P986 S8 S1 1 953
P987 S3 S5 2 1060
P988 S13 S17 1 1406
P989 S10 S2 2 136
P990 S18 S19 1 1260
P991 S1 S2 5 1955
P992 S11 S5 1 167
P993 S11 S4 2 1756
P994 S5 S15 1 1949
P995 S6 S7 3 280
P996 S7 S17 2 1849
P997 S13 S3 4 1148
P998 S6 S19 3 1415
P999 S5 S3 1 1081
```

Anhang 2: Inputfile mit vielen Passagiere