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

Inhaltsverzeichnis

Abkürzungsverzeichnis

SDA	Simple Dijkstra Algorithm
ADA	Advanced Dijkstra Algorithm
STP-Algorithmus	Simple Train Parallelization Algorithm

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 möglichst 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 dabei im Rahmen des InformatiCups 2022 entwickelt. Aufgabe des Wettbewerbs war es eine Software zu entwickeln, die basierend auf einer Textdatei, welche Züge, Strecken, Bahnhöfe und Passagiere beinhaltet, einen Fahrplan erstellt. Dieser sollte in der Hinsicht optimiert werden, dass die Gesamtverspätung aller Passagiere möglichst gering ist (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 wird überprüft, ob Syntax und die Berechnung der Boarding-, Abfahrts- und Ankunftszeiten korrekt durchgeführt wurde.

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 (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 ungerichtete Verbindungen zwischen zwei Bahnhöfen dar. Neben einer maximalen Zugkapazität werden sie zudem 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 ist, dass diese auch als „Wildcards“ vorkommen können. Das heißt, die Positionen dieser Züge sind zu Beginn noch nicht festgelegt und können beliebig gewählt werden. Zuletzt gibt es im Schienennetz noch Passagiere, welche neben einer Gruppengröße einen Zielbahnhof sowie eine Zielzeit besitzen. Kommt ein Passagier nach der Zielzeit am Ziel an, so ist er verspätet. Ziel der Aufgabe ist es, diese Verspätungen zu minimieren (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 Positionierung frei positionierbarer Züge sowie das Vermeiden von Kapazitätskonflikten sind nur einige interessante Herausforderungen, die diese zunächst so einfach wirkende Aufgabenstellung mit sich bringt. All das muss in einem Algorithmus 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 nicht möglich ist, einen „perfekten“ Algorithmus für alle Input-Probleme zu finden. Deshalb war es auch nie das Ziel, einen einzigen Algorithmus zu entwerfen. Stattdessen sollten mehrere unterschiedliche Algorithmen entwickelt 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 in gleicher Weise verwendet werden können. Diese sollen anschließend instanziiert und angewandt werden können, so dass es möglich ist mit weiteren Ansätzen 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.

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

Kapitel 3 der vorliegenden Arbeit beschäftigt sich mit dem Programmentwurf der Lösung. Dabei wird auf die Besonderheiten des Input- und Output-Parsings eingegangen, die den Rahmen der Lösung bilden. Weiter wird ein Interface für die Algorithmen sowie die Struktur der Algorithmen selbst kurz vorgestellt. Dieser Part ist zentral für die softwaretechnische Umsetzung der zuvor skizzierten Grundidee. Anschließend wird das Exception Handling in der Softwarelösung betrachtet. Schlussendlich stellt Kapitel 3 noch den Input Generator vor, der eigens dafür entworfen wurde, eigene Testdaten schnell und effizient zu generieren.

In Kapitel 4 werden schließlich die entworfenen Algorithmen selbst detailliert betrachtet. Das Kapitel umfasst dabei drei entwickelte Algorithmen sowie einen weiteren Abschnitt mit Ideen für weitere Lösungsansätze.

Im letzten Kapitel werden die Ergebnisse zusammengefasst und ein Fazit zur Lösung gezogen. Dabei soll speziell auf die Übertragbarkeit des Ansatzes auf die Realität eingegangen werden. Schlussendlich wird noch ein Ausblick mit weiterführenden Optimierungsideen für den entwickelten Lösungsansatz gegeben.

2. Theoretische Grundlagen

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

2.1 Annahmen bei der Berechnung

Commented [MS2]: MORITZ

In diesem Kapitel stellen wir zentrale Berechnungen → Besonderheiten vom Informaticup

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 Linie benötigt.

Da gilt $s, v \in \mathbb{Q} \mid s, v > 0$ folgt damit, dass auch $t \in \mathbb{Q}$ 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 es vorkommen kann, dass ein Zug in derselben Runde an einer Station ankommt, in der er von einer anderen losgefahren ist (InformatiCup 2022, S.3). Fährt der Zug nun direkt weiter muss jedoch beachtet werden, dass dies nur für *Abfahrtszeit > Ankunftszeit* gültig ist und somit wenn $t_{Abfahrt} + (t_{Differenz} - 1) = t_{Abfahrt}$ gilt $t_{Ankunft} = (t_{Abfahrt} + 1)$. 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 somit 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
```

2.1.2 Swapping

Ein weiterer zentraler Aspekt der Problemstellung ist die Behandlung von 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 Kapazitäten kurzzeitig überschritten werden dürfen solange am Ende jeder Runde sämtliche Kapazitäten eingehalten werden. (InformatiCup 2022, S.4)

Gegeben sei eine Strecke 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
```

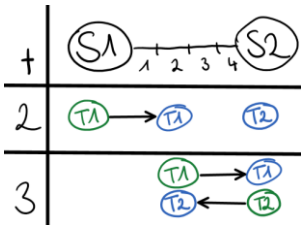
Zunächst erschein es als könnte P1 mit dieser Ausgangsposition nicht von T1 transportiert werden, da die Zielstation bereits voll belegt ist und auch auf der Strecke nur ein Zug gleichzeitig sein kann. Hier kann jedoch das *Swapping* zum Einsatz kommen. Grundgedanke ist: in der letzten Zeiteinheit, bevor ein Zug an der Station ankommt fährt der Zug, welcher die Zielstation blockiert, los. Damit gibt es während 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 Kapazitäten 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
```

Zur weiteren Veranschaulichung wurde das folgende Schaubild 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.



2.2 Dijkstra-Algorithmus

Für den Dijkstra-Algorithmus werden die Stationen und Strecken als Graph Objekt dargestellt, in dem Entfernungen als Kantenlängen angegebenen. 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. Dies lässt sich jedoch nur für positive Zahlen anwenden, was für die Fahrplanerstellung von Zügen jedoch keine Einschränkung darstellt (vgl. Velden 2014).

Jeder Station werden die Eigenschaften Distanz und ihre Vorgänger, relativ zum Ausgangpunkt zugewiesen. Beide werden beim Durchlaufen immer wieder überschrieben. Wichtig ist jedoch, dass der Vorgänger mit kürzestem Weg immer gemerkt wird. Somit lässt sich am Ende auch immer der gesamte Weg und nicht nur die Länge des Weges nachvollziehen (vgl. Mönius et al. 2021, S.40-45). Diese Eigenschaft war von besonderer Bedeutung da der Algorithmus sich sonst nicht auf das die Problemstellung übertragen ließe, da nur die geringste Entfernung nicht zur Erstellung eines Fahrplans genügt.

Eine weitere wichtige Eigenschaft des Dijkstra Algorithmus ist die Übertragbarkeit auf andere Problemstellungen. Zudem löst er das Problem den schnellsten 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)

Aus den oben genannten Gründen wurde der Dijkstra Algorithmus für die hier behandelte Lösung in allen Algorithmen implementiert. Jedoch mussten einige Anpassungen gemacht werden.

Als Basis wurde die Implementierung vom GitHub user mdsrosa gewählt (mdsrosa 2015), an der zusätzliche Anpassungen gemacht wurden, um den Anforderungen fachlich zu entsprechen.

Commented [MS3]: JONAS

Commented [GU4R3]: Frage Quellen? Einfach unten rein oder direkt im Text zitieren?

Commented [MS5R3]: Bitte auch im Text bei Nennung zitieren mit (Autor Jahr, Seite)

Commented [MS6R3]:

Commented [GU7R3]: ne sorry, meine auch wenn es kein direktes Zitat ist, soll ich nahc jedem Absatz ne Quelle nennen? Weil sehe ich sont auch nicht wirklich

Commented [MS8R3]: Ich würds nach dem entsprechenden Absatz immer nennen, sonst ist die Zuordnung sehr schwierig, wenn s nur unten drin steht... außerdem siehts besser aus ;-)

Commented [GU9R3]: und am Ende sollte noch schnell jemand drüberschauen ob es passt was ich zu unserer Anpassung geschreiben habe und das ergänzen. Da weiß ich ja nicht was genau gemacht wurde

Commented [MS10]: Diesen Absatz bitte noch deutlich mehr ausführen… am besten mit Beispiel erklären

Commented [MS11]: Hier bitte die korrekte Space und Time Complexity nennen

(hier Anpassungen erläutern)

Commented [MS12]: MORITZ

3. Programmentwurf

Commented [MS13]: JONAS

Nachdem die theoretischen Grundlagen für unseren Lösungsansatz vermittelt wurden, kann nun auf die konkrete Implementierung der Idee eingegangen werden. Hierzu wurde ein *Python3-Programm* entworfen, dessen Funktionalität und Aufbau im Folgenden beschrieben werden.

Um die Funktionalität unabhängig von dem 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. Das Output File wird bei erfolgreicher Durchführung des Programms in *stdout* ausgegeben.

3.1 Input Parsing

Commented [MS14]: NICK

Um mit der Inputdatei, die uns die Informationen über Züge, Strecken, Bahnhöfe und Passagiere gibt arbeiten zu können, ist es notwendig ein Programm zu schreiben, das die Datei liest, auf Korrektheit überprüft und die Information in einer Art und Weise aufarbeitet, das sie möglichst gut in einer objektorientierten Programmiersprache verwendet werden können. Ein derartiges Programm mit dem Namen *InputParser* haben wir daher als erstes entwickelt, um darauf aufbauend die Algorithmen entwickeln zu können. Zunächst haben wir dabei Klassen für Züge, Strecken, Bahnhöfe und Passagiere geschrieben, die diese Objekte entsprechend gut beschreiben. Diese Klassen werden beim Lesen der Inputdatei später verwendet, um die Informationen über z.B. die Züge in einer Liste von Zugobjekten aufzubereiten. Der gesamte Vorgang ist in 4 Methoden aufgeteilt, eine für Züge, eine für Strecken, eine für Bahnhöfe und eine für Passagiere, die dann in einer Hauptmethode aufgerufen werden. Diese vier Methoden funktionieren alle ähnlich, weshalb im Folgenden das Input Parsing 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. Die Klasse Train hat die folgenden Eigenschaften:

1. ID
2. Kapazität
3. Passagiere
4. Geschwindigkeit
5. Position
6. Startposition

Neben diese 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 die Eigenschaften des entsprechenden Zuges im gleichen Format zurückgibt, wie sie nachher auch in der Outputdatei stehen würden. Diese beiden Methoden gibt es auch in den Klassen für die Passagieren, die Bahnhöfe und die Strecken, lediglich die Eigenschaften sind andere. Nun wird näher auf die Erstellung der entsprechenden Zugobjekte (gilt natürlich analog für die Passagiere, die Bahnhöfe und die Strecken) aus der Inputdatei eingegangen. Um nun die Zugobjekte zu bekommen, wird die Methode parse_trains() aufgerufen (in der Hauptmethode). In der parse_trains() Methode wird zunächst ein Array deklariert, das nachher die Zugobjekte enthalten soll, die später am Ende der Methode zurückgegeben werden soll. Außerdem wird ein Boolean mit dem Namen train deklariert, der am Anfang den Wert *False* hat. Anschließend wird mit Hilfe einer *for-Schleife* über den Input iteriert, um jede einzelne Zeile der Datei zu behandeln. Dabei sind alle Zeilen bis zur Schlüsselzeichenkette "[Trains]" für das Einlesen der Züge irrelevant. Erst ab diesem Punkt beginnen die einzelnen Zeichenketten, die jeweils einen Zug beschreiben. Sobald die Schlüsselzeichenkette erkannt wurde, wird der Wert des Booleans auf *True* gesetzt und es wird versucht, alles was danach kommt als Zug einzulesen. Dabei 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, folgt später. Ist die aktuelle Zeile nun ein gültiger String, der einen Zug beschreibt, wird eine Instanz der Klasse Zug erstellt, indem dem Konstruktor alle 4 Teilstrings der aktuellen Zeile übergeben werden, diese werden dann den jeweiligen Eigenschaften zugeordnet. Ist es kein gültiger String, der einen Zug beschreibt, wird überprüft, ob es sich um einen anderen gültigen String handelt, also um einen String, der einen Passagier, einen Bahnhof, eine Strecke, aber auch um eine Schlüsselwortzeile, eine Leerzeile oder um einen Kommentar (dann würde der String mit "#" anfangen) handelt. Wenn es sich um eine andere gültige Zeile handelt, ist alles in Ordnung und die Zeile wird ignoriert, es wird einfach mit der nächsten Zeile fortgefahren, bis einmal über den ganzen Input iteriert wurde. Trifft allerdings nichts von alldem zu, dann handelt es sich um einen Verstoß gegen das vorgeschriebene Inputformat und es wird eine Exception geworfen, die dann abgefangen wird, um eine Fehlermeldung auszugeben. Nun also nur noch die Frage, wie wird die Gültigkeit beispielsweise eines Zugstrings geprüft? Wie bereits erwähnt, wird im Fall des Zuges die Methode check_train_string() aufgerufen. Dieser Methode wird dann der nach Leerzeichen aufgeteilte String übergeben. Die Zeile müsste wie folgt aussehen, um einen gültigen Zugstring darzustellen:

```
T1 S1 2 20
```

Das bedeutet für die Überprüfung der Gültigkeit nun, dass als aller erstes die Anzahl der aufgespaltenen Strings überprüft werden kann, wie man oben sehen kann, müssten es genau 4 Teilstrings sein, ist das nicht der Fall, kann direkt “False” zurückgegeben werden. Sind es genau 4 Teilstrings, muss allerdings weiter geprüft werden, der erste Teilstring muss in diesem Fall zwingend mit “T” beginnen, trifft dies nicht zu, wird direkt “False” zurückgegeben. Der zweite Teilstring kann mit “S” beginnen, da es eine Bahnhofs ID ist, und diese mit “S” für Station beginnen, er kann aber auch “*” sein (bedeutet, der Zug ist frei positionierbar), ist der zweite Teilstring nicht “*” oder beginnt mit “S”, kann direkt “False” zurückgegeben werden. Ist dies alles erfüllt, wird überprüft, ob nach “T” und nach “S” eine Zahl kommt, diese werden abgeschnitten und in einem Try – Except - Block versucht in einen Integer zu parsen, genauso wie die letzten beiden Teilstrings, schlägt dies fehl, sind das keine Ganzzahlen und im Except – Block wird die Exception abgefangen und die *CannotParseInputException* geworfen. Geht im Try Block alles gut, so handelt es sich um einen gültigen Zugstring und es wird “True” zurückgegeben. Für jede der 4 Teilmethoden, gibt es eine check Methode, die alle ähnlich funktionieren wie die hier vorgestellte check_train_string() Methode. In jeder der Nebenmethoden wird jede der check Methoden verwendet. Soll der InputParser verwendet werden, wird einfach, parse_input() aufgerufen, die alle 4 Methoden aufruft und eine Liste mit allen Instanzen zurückgibt, mit denen dann in den jeweiligen Algorithmen gearbeitet werden kann.

3.2 Output Parsing

Commented [MS15]: MORITZ

Der Output Parser übernimmt einige zentrale Aufgaben der entwickelten Software. Zentral besteht er dabei nur aus einer Methode *parse_output_files*, der eine Liste aller Algorithmen übergeben wird, welche zur Lösung des Input Files genutzt werden sollen. Diese wurden zuvor in der Main.py definiert. Aufgabe des Output Parsers ist es nun diese Algorithmen nacheinander auszuführen, was standardisiert mit der *solve()-Methode* geschieht, die jeder Algorithmus zwingend implementieren muss (siehe Kapitel 3.3). Konnte diese erfolgreich durchlaufen, 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_str, welcher am Ende zum Output File zusammengesetzt wird. In beiden Fällen wird hierbei auf Basis der Instanzvariable *journey_history* gearbeitet, welche alle Aktionen der ensprechenden Objekte enthält und während dem Durchlaufen des 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 gegebenen Eigenschaften 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 OutputParser übergreifend festzustellen. Es wird für jeden Algorithmus ein Output File 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 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 (siehe 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()
```

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.

3.3 Das Interface ISolver und Algorithmus-Klassen

Wie die Erläuterung des Output Parsings gezeigt hat, wird die Grundidee des Lösungsansatzes auch in der softwaretechnischen Umsetzung deutlich. Damit nun aber, wie im Output Parser gewollt, die verschiedenen Algorithmen austauschbar und auf die gleiche Weise eingesetzt werden können, müssen diese auch gleich aufgebaut sein. Zumindest von außen müssen die Algorithmen alle die 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, das alle Algorithmen implementieren. Damit wird vorgeschrieben, welche Methoden ein Algorithmus zwingend besitzen muss. Lediglich diese gemeinsamen Methoden werden dann vom Anwender bzw. 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, denn in *Pepper 2007* werden Interfaces bezeichnet als „abstrakte Klassen, bei denen *alle* Methoden abstrakt sind."

In *ISolver* werden den implementierenden Klassen drei Methoden vorgeschrieben, die 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()* sollen die *journey_history* vom Algorithmus befüllt werden. Diese kommen als Instanzvariablen von Passagieren und Zügen vor und speichern 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 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 im Abschnitt zum Output Parser erwähnt, wird genau das auch in unserer Lösung gemacht. Weitere Möglichkeiten, die der Anwender beim Instanziieren der Algorithmen hat, werden in späteren Kapiteln noch vorgestellt.

3.4 Exception Handling

Commented [MS16]: TIMO

Bereits an verschiedenen Stellen der Arbeit wurden eigene Exceptions im Programmcode erwähnt. Der Input Parser wirft eine *CannotParseInputException*, wenn ein ungültiger Input vorliegt, den er nicht parsen kann. 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 1. Dort werden im *Simple Train Parallelization Algorithm* zwei Fälle abgefangen: Es 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. Damit 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 1: Internes Abfangen von Exceptions im Simple Train Parallelization Algorithm

Allerdings können 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, da dieser Fall abgedeckt wird: 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*[1], 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) trotzdem 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 eigenen Exceptions habe jedoch keine eigene, tiefere Funktion. Sie sind also entweder schlichtweg leer oder besitzen lediglich wenige Instanzvariablen. Listing 2

[1] *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 2022).

veranschaulicht das anhand des Quellcodes der bereits zuvor betrachteten Exceptions *CannotDepartTrain* und *CannotBoardPassenger*.

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


class CannotBoardPassenger(Exception):
    pass
```

Listing 2: 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.

3.5 Inputgenerator

Am Anfang war das Testen der entwickelten Algorithmen sehr aufwendig, da zuvor einzelne Inputdateien selbst geschrieben werden mussten, um diese dann für einen Test verwenden zu können. Das hat allerdings offensichtliche Schwächen, so können damit nur kleine überschaubare Inputdateien erstellt werden, da viele Punkte (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 verschieden Kombinationen getestet werden sollten (viele Bahnhöfe, provozierte Kapazitätsprobleme, etc.). Daher haben wir uns dazu entschieden, ein Programm zu schreiben, um uns diese Dateien zu generieren. Dieses liest zunächst einige Parameter ein, die zur Erstellung einer Inputdatei benötigt werden, unter anderem die Anzahl der Bahnhöfe, die Anzahl der Passagiere, und vieles mehr, aber auch einige Spezifikationen der einzelnen Objekte müssen eingelesen werden, beispielsweise die maximale Kapazität eines Bahnhofs. Die verschiedenen Attribute werden dann auf Basis der eingelesenen Spezifikationen Zufallsbasiert generiert (beispielsweise die Kapazität eines einzelnen Bahnhofs). Wurden alle Objekte generiert, reicht es allerdings nun 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. Anschließend wird überprüft, ob es doppelte Strecken gibt (das kann bei einer zufallsbasierten Generierung der Strecken passieren) und gelöscht. Das schließt ebenfalls “umgedrehte” Strecken ein (hierbei sind die gleichen Bahnhöfe verbunden, aber in einer anderen Reihenfolge im String vorhanden). Um die Gültigkeit der Inputdatei zu wahren, mussten die Startpositionen aller Züge überprüft werden, um die Kapazitäten der einzelnen Bahnhöfe nicht zu überschreiten. Für den Fall, dass die Kapazität eines Bahnhofes überschritten wird, werden die Startpositionen einiger Züge durch “*” 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 stand uns nun also ein Tool zur Verfügung, dass uns ermöglichte, einzelne Algorithmen schneller und besser auf bestimmte Inputszenarien zu testen. Mit der Zeit wurde allerdings bemerkt, dass es auch sinnvoll sein kann für eine bestimmte Kombination von Parametern gleich mehrere Dateien zu generieren (bisher musste man in diesem Fall das Programm mehrmals aufrufen und immer wieder die gleichen Parameter eingeben). Daher wird als letztes vor der Generierung nun auch die Anzahl der gewünschten Dateien eingelesen, daraufhin werden aufgrund der einmal eingegebenen Parameter die gewünschte Anzahl an Dateien generiert.

4. Algorithmen

4.1 Simple Dijkstra Algorithm

Commented [MS17]: MORITZ

4.1.1 Idee und Umsetzung

Der Simple Dijkstra Algorithm (SDA) ist eine sehr einfache Herangehensweise an die Problemstellung. Ziel bei der Entwicklung war es als Basis für die Algorithmensammlung einen sehr soliden und einfachen Ansatz zu wählen, um jeden Input lösen zu können.

4.1.2 Vor- und Nachteile

4.2 Advanced Dijkstra Algorithm

Commented [MS18]: MORITZ

- Historisierung der bereits berechneten Strecken

- Mitnahme von Passagieren auf dem Weg

- mytrain statt train[0].. flexiblerer erweiterbarer Ansatz

-

Parametrisierung wird in 4.3.3 näher erläutert

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 *Swappings* (Kap. 2.1.1) 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. Eine 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

Commented [MS19]: TIMO

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 in diesem Fall bereits einen „Blick in die Zukunft“ werfen, um zu erkennen, ob es dort nicht zu

Kapazitätsprobleme bei Strecken oder Stationen mit einem parallellaufenden Zug kommt. Für diesen „Blick in die Zukunft“ kann man nun verschiedene Ansätze wählen. Im Rahmen des *Simple Train Parallelization Algorithm* wurde nun folgende Lösung implementiert:

Alle Attribute von Strecken, Stationen, Zügen oder Passagieren (allgemein: von Objekten im Schienennetz), die sich im Laufe der Zeit verändern können, 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 oder auch Wahrheitswerte wie *is_in_train* im Falle der Passagiere bzw. *is_on_line* bei den Zügen.

Jeder Zeitschritt des Zugfahrplans ist dargestellt durch eine Reihe in dieser Tabelle. Dabei entspricht der Index des DataFrames genau dem entsprechenden Zeitpunkt des Fahrplans (Pandas DataFrames beginnen standardmäßig bei Index 0).

Bei der Initialisierung einer neuen Instanz des *STP-Algorithmus'* wird, neben der Tabelle selbst, auch gleich die erste Reihe (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 Reihe 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 („Runden“) hoch. Sie bricht ab, sobald für alle Passagiere deren aktuelle Position mit ihrem Ziel übereinstimmt. Dann sind alle Passagiere am Ziel, der Algorithmus ist fertig.

Die innere Schleife hingegen läuft prinzipiell so lange, bis es keinen freien, stehenden Zug mehr gibt, der einen Passagier mitnehmen kann. Dies kann jedoch unter Umständen sehr lange dauern und nicht immer von Vorteil sein, wie nachfolgende Ausführungen zeigen werden. Deshalb wird auch für diese Schleife gezählt, wie oft sie durchlaufen wird, und bei einer bestimmten Anzahl an Durchläufen wird unabhängig von ersterer Bedingung abgebrochen. Wie genau sich die Anzahl maximaler Durchläufe 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 in Zügen

befinden oder schon am Ziel angekommen sind). Anschließend wird ein Zug ausgewählt, der den gewählten Passagier an sein Ziel bringen soll. Aus allen möglichen Zügen (ausreichend Kapazität, nicht bereits auf einer Strecke, etc.) wird derjenige ausgewählt, der sich am nächsten am Passagier befindet. Hierzu wird der kürzeste Weg mithilfe des Dijkstra-Algorithmus' berechnet, der auch im Algorithmus aus Kapitel 4.2 verwendet wird.

Sofern ein Zug und ein Passagier ausgewählt werden konnten (wenn nicht, bricht die innere Schleife ab), wird weiter überprüft, ob deren Position bereits übereinstimmt. Ist dies der Fall, so kann der Passagier geboardet werden. Beim Boarding werden 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“).

Grob erklären, was die voraussetzungen sind, insb. Swapping, wann abgebrochen wird.

Hat der Zug das Ziel erreicht oder muss er an einer früheren Station abbrechen (z.B. aufgrund von Kapazitätskonflikten), so wird der Passagier abgeladen („detrain”). Dabei werden die Attribute im DataFrame wieder entsprechend angepasst.

Der beschriebene Prozess aus Passagier- und Zugauswahl und Versuch des Boarding, Departing sowie Detraining wird in der inneren Schleife ausgeführt und somit für eine Zeiteinheit („Runde”) 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.

Am Ende des Algorithmus steht schließlich nicht nur der Output, sondern auch eine Tabelle als Pandas DataFrame, die nochmals über detailliertere Informationen zum Fahrplan verfügt. Diese Tabelle wird am Ende des *STP-Algorithmus* genutzt, um die gesamte Verspätungszeit im erstellten Fahrplan zu berechnen. So kann die Lösung schlussendlich bewertet und mit den anderen verglichen werden.

4.3.2 Vor- und Nachteile

An dieser Stelle soll nun eine Bewertung des *STP-Algorithmus* hinsichtlich dessen Vor- und Nachteile vorgenommen werden.

Generell ist anzumerken, dass es sich beim *Simple Train Parallelization Algorithm* – wie der Name schon impliziert – um einen sehr einfachen Algorithmus handelt, der noch an diversen Stellen optimiert werden kann.

Ungeachtet dessen muss festgehalten werden, dass die Idee der Parallelisierung von Zügen insbesondere bei kleineren Schienennetzen mit mehreren Passagieren gut angewendet werden kann. So liefert der Algorithmus zum Beispiel für das Input-Problem "Kapazität" (siehe *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.

Jedoch kommen bei größeren Schienennetzen als Input 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.

Weiter muss am Algorithmus kritisiert werden, dass viele intelligente Ansätze aus den ersten beiden Algorithmen nicht berücksichtigt werden. So wird zum Beispiel immer nur ein Passagier pro Zug gleichzeitig transportiert, auch wenn dessen Kapazität einen weiteren Passagier zulassen würde und dieser dasselbe Ziel hätte. Ebenso werden die Passagiere auch schon früher als sie eigentlich müssen an ihr Ziel gebracht. Diese Zeit könnte jedoch noch für andere Passagiere genutzt werden. Alle hiergenannten Kritikpunkte könnten am Algorithmus noch verbessert werden, um die Verspätungszeiten weiter zu minimieren.

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 instanziieren, 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 3: Konstruktor der Klasse des Simple Train Parallelization Algorithm

Listing 3 zeigt den Konstruktor des Simple Train Parallelization Algorithm. 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 (wobei auch über 1 möglich wäre). 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 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. Denn beim Setzen dieser Züge 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 andere Züge noch vorhanden 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* erlaubt es nun, dass diese Entscheidung nicht bereits bei der Implementierung des Algorithmus' getroffen werden muss: Dieser Parameter stellt den relativen Anteil der Wildcard-Züge dar, die in

der jeweiligen Instanz des Algorithmus' gesetzt werden. Ist der Parameter also 1, so werden alle Wildcards gesetzt (=Standardwert). Beträgt der Wert nur 0.5, wird hingegen nur die Hälfte der Wildcard-Züge gesetzt. Bei 0 wird schließlich kein Zug gesetzt, außer es ist ansonsten kein Zug zur Verfügung. In diesem Fall wird der Wildcard-Zug mit der höchsten Kapazität gesetzt. Somit kann auch hier 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 seinen Anwendungsfall besten Algorithmus auszuwählen, sondern diesen auch noch durch Finetuning der Parameter möglichst optimal an seinen Anwendungsfall anzupassen.

Im Rahmen 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 des Simple Train Parallelization Algorithm instanziiert, deren Parameter sich stark unterscheiden. So sollte für die meisten Fälle zumindest ein guter Algorithmus vorhanden sein, wenn auch nicht der optimal angepasste Algorithmus. Ein anderer Ansatz, um diese Problematik zu lösen, wird später im Ausblick dieser Arbeit noch diskutiert.

4.4 Weiterführende Algorithmen und Features

Commented [MS20]: ALLE: IDEEN

Kombination aus STP und ADA

Zusätzliche Idee als zusätzliches Feature der Software: Knotenpunkte mit niedriger Kapazität feststellen und dann ausgeben, wo noch nachgebessert werden könnte um das Ergebnis drastisch zu verbessern

Commented [MS21]: Sollen wir sowas einbauen und wenn ja wo?

5. Schlussbetrachtung

5.1 Zusammenfassung und Fazit

5.2 Ausblick

Commented [MS22]: TIMO

In diesem Abschnitt seien nun noch einige Ideen skizziert, um den vorgestellten Lösungsansatz weiter zu verbessern und zu optimieren.

Dabei soll zunächst an einer Schwachstelle der Lösung angesetzt werden, 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 eher unschön aus programmiertechnischer sich, sondern kann insbesondere bei großen Input-Dateien 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 selbst einen Algorithmus aus, so kann er bereits vor 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. Nachfolgend seien zwei Ansätze kurz skizziert:

Beim ersten Ansatz handelt es sich um überwachtes Lernen (*supervised learning*), wobei das Optimierungsproblem als Klassifizierungsproblem betrachtet wird. Ein Input-Problem wird dabei stark komprimiert, so dass die wesentlichen Informationen (z.B. Anzahl an Zügen, Strecken, …) einheitlich in einer Zeile (=Datensatz) einer Tabelle dargestellt werden können. Für jeden Datensatz wird in einer weiteren Spalte der für diese Daten am besten performende 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.

Ansatz mit Reinforcement Learning → mit Tom reden?

Laufzeitminimierung durch intelligente Auswahl der Algorithmen (Stichworte: Evolutionäre Algorithmen??, Machine Learning (Reinforcement Learning?))

5.3 Übertragbarkeit auf die Realität

Commented [MS23]: NICK & TIMO

Grenzen der Aufgabenstellung

Kostentechnische Erwägungen (nicht jeder Passagier muss ankommen) – höhere Verspätung in Kauf nehmen, wenn die Kundenzufriedenheit nicht zu stark sinkt

Nicht die Algorithmen, sondern Idee ist eher auf die Realität anwendbar.

Die Aufgabenstellung ist auch aus betriebswirtschaftlicher Perspektive leider bei ihrer Übertragung auf die Realität stark begrenzt. So wird angenommen, dass Bahnunternehmen stets das Ziel haben die Verspätung der Passagiere zu minimieren, bzw. Die Kundenzufriedenheit zu maximieren. Das ist aber nicht der Fall, das Ziel der Unternehmen ist es, ihren Gewinn zu maximieren, das kann durchaus auch einen Zielkonflikt mit der Aufgabenstellung ergeben. Wenn man ein Programm zur Erstellung eines optimalen Fahrplans bezüglich der Verspätung entwickelt, kann es durchaus sein, dass dort Züge mit sehr wenigen Passagieren fahren, dies wäre betriebswirtschaftlich nicht erstrebenswert, wenn der Verlust, der durch eine geringere Kundenzufriedenheit geringer wäre, als der Verlust der entsteht, wenn man einen Zug für eine geringe Anzahl von Passagieren weite Strecken fahren lässt. Es ist nicht realistisch, dass der Fahrplan komplett nach den Wünschen der Passagiere ausgerichtet ist, vielmehr versucht man die am häufigsten genutzten Strecken anzubieten und die Passagiere müssen sich nach dem Fahrplan richten.

Quellenverzeichnis

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

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

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

[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 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)*

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

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

[Mönius et al. 2021] Algorithmen in der Graphentheorie - Ein konstruktiver Einstieg in die Diskrete Mathematik. Wiesbaden: Springer Wiesbaden GmbH, 2021, S. 40-46

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