Diese Arbeit wurde vorgelegt am LuFG Theorie hybrider Systeme

BACHELORARBEIT

PARALLELE BERECHNUNGEN IN DER ANALYSE VON VERSPÄTUNGEN IM SCHIENENVERKEHR

Lisa Wanko

Erstprüferin:
Prof. Dr. Erika Ábrahám

Zweitprüfer:
Prof. Dr.-Ing. Nils Nießen

Beraterin:
Dr. Rebecca Haehn

Aachen, 13.03.2023

Zusammenfassung

Diese Arbeit beschäftigt sich mit der parallelen Simulation von Verspätungen im Schienenverkehr auf der Basis eines symbolischen Simulationsalgorithmus für Eisenbahnfahrpläne. Dabei wurde untersucht, wie die Performance der Simulation durch die Nutzung von mehreren Threads beeinflusst wird. Die Ergebnisse zeigen, dass die Verwendung von Threads die Berechnungsgeschwindigkeit verbessern kann, jedoch abhängig von der Größe des Schienennetzes und der Anzahl der Threads auch einen Overhead verursachen kann, der zu einer ineffizienten Ausführung führt. Es wurden verschiedene Strategien zur Optimierung der Parallelisierung untersucht und diskutiert. Insgesamt bietet die Arbeit Einblicke in die Herausforderungen und Möglichkeiten der Parallelisierung eines Simulationsalgorithmus für Eisenbahnfahrpläne.

Inhalt

Kapitel 1

Einführung

Ein Schienennetz ist ein System von Eisenbahnschienen, auf denen Züge fahren können. Es besteht aus Gleisen, Bahnhöfen, Schienenweichen, Signalen und anderen Infrastruktureinrichtungen, die es ermöglichen, dass Züge sicher und effizient von einem Ort zum anderen befördert werden können. Ein Schienennetz kann sowohl innerhalb eines Landes als auch international verlaufen und es kann sowohl für den Personen- als auch den Güterverkehr genutzt werden. Die Entwicklung und der Ausbau von Schienennetzen spielt eine wichtige Rolle in der Verkehrsinfrastruktur eines Landes und trägt zur Verbesserung der Mobilität und zur Reduzierung von Emissionen bei. Folglich ist es sinnvoll, daran zu arbeiten, dass Schienennetze effizient genutzt werden können. Hierfür ist es von Bedeutung, Verspätungen so weit wie möglich zu minimieren, um sicherzustellen, dass Züge als verlässliches Transportmittel genutzt werden können und jeder Zug nur so lange wie nötig unterwegs ist. Um Verspätungen eliminieren zu können, ist es zunächst wichtig, zu ermitteln, wo sie auftreten können und durch welche Faktoren sie verursacht werden. Zu diesem Zweck haben Rebecca Haehn, Erika Ábrahám und Nils Nießen eine Simulation entwickelt, die genau diesen Aspekt betrachtet [HÁN21]. Ab sofort wird auf diese Simulation mit der Abkürzung HÁN-Simulation Bezug genommen.

1.1 Verwandte Arbeiten

In der Forschung und in kommerziellen Anwendungen wurden bereits zahlreiche Ansätze entwickelt, um den Betrieb von Schienennetzen zu verbessern. Ein gängiger Ansatz besteht darin, Data Science-Methoden zu nutzen. Hierbei werden Daten über ein existierendes Schienennetz gesammelt und analysiert, um beispielsweise Ursachen von Verspätungen zu identifizieren.

Dieser Ansatz ist besonders gut geeignet, um bereits vorhandene Fahrpläne zu untersuchen und festzustellen, wo häufig Verspätungen auftreten und was die Gründe dafür sind. Allerdings hat dieser Ansatz auch einige Nachteile. Zum einen ist er stark von der Qualität der gesammelten Daten abhängig. Unvollständige oder fehlerhafte Datensätze können zu falschen Ergebnissen führen und somit unbrauchbar sein. Zum anderen ist es notwendig, dass ausreichend Daten vorhanden sind, um eine aussagekräftige Analyse durchführen zu können. So können beispielsweise keine Fahrpläne untersucht werden, die noch nicht umgesetzt wurden und zu denen somit noch keine

Daten verfügbar sind. Wenn man einen Fahrplan analysieren möchte, bevor Daten verfügbar sind, z.B. um zu überprüfen, ob er umgesetzt werden sollte oder ob noch Veränderungen erforderlich sind, muss man auf eine andere Methode zurückgreifen.

Eine Möglichkeit ist die Simulation von Fahrplänen auf Schienennetzen. Hierbei sind lediglich ein Schienennetz als Graph und ein Fahrplan als Eingabe erforderlich. Obwohl in diesem Bereich bereits viele Ansätze verfolgt wurden, wurden viele davon zu kommerziellen Zwecken erstellt und es ist daher nicht möglich, bei den meisten zu überprüfen, wie genau die Simulation funktioniert und welche Algorithmen verwendet wurden.

Was allerdings häufig verwendet wurde sind verschiedene Monte-Carlo-Ansätze. Dabei werden sequenziell verschiedene Verspätungsszenarien hintereinander simuliert. Das Monte-Carlo-Verfahren wird zum Beispiel im Tool „OnTime“ eingesetzt. Die grundlegende Simulationstechnik des OnTime-Tools wird in der Arbeit „Stochastic modelling of delay propagation in large networks“ [BS12] beschrieben. Der Unterschied zwischen dieser Methode und dem Ansatz der HÁN-Simulation wird im folgenden Absatz erläutert.

Rebecca Haehn, Erika Ábrahám und Nils Nießen haben mit der HÁN-Simulation eine öffentlich zugängliche Simulation entwickelt, die auch statistische Methoden verwendet, um die Auswirkungen von anfänglichen Verzögerungen zu betrachten. Mit dieser Methode wird eine Wahrscheinlichkeitsverteilung verwendet, um den Aufenthaltsort der Züge anzuzeigen, anstatt sie auf diskrete Standorte zu beschränken. Dieser probabilistische Ansatz ermöglicht es, Wahrscheinlichkeiten anzugeben, zu denen sich ein Zug zu einem bestimmten Zeitpunkt an einem bestimmten Ort befindet. Dadurch kann das Verhalten des Netzwerks analysiert und mögliche Faktoren für Verzögerungen erkannt werden.
Im Gegensatz zu Monte-Carlo-Simulationen werden in diesem Ansatz alle möglichen Szenarien gleichzeitig simuliert. Dabei werden nicht alle Verspätungsszenarien einzeln dargestellt und verwendet, sondern es werden lokale, partielle Szenarien genutzt, um die Gesamtheit aller notwendigen Szenarien abzudecken. Die Details dieses Vorgehens werden in der Arbeit [HÁN21] ausführlich erläutert. Dieser Ansatz wurde bisher noch nicht implementiert und es ist daher wichtig zu testen, inwieweit die Simulation noch schneller gemacht werden kann, um ihre Nützlichkeit zu erhöhen, insbesondere bei der Simulation großer Netze über einen längeren Zeitraum. Die Geschwindigkeit, mit der die Simulation bestimmte Eingaben simulieren kann, ist also von großer Bedeutung für den Umfang, in dem die Simulation genutzt werden kann, und sollte daher maximiert werden. Daher versucht diese Arbeit, die Simulation durch Parallelisierung zu beschleunigen. Zu diesem Zweck wird Multithreading verwendet und die Karte des Schienennetzes (der Eingabegraph) in so viele Teile unterteilt, wie es Threads geben soll, die dann parallel rechnen. Für diese Simulation ist es von großer Bedeutung, dass Multi-Threading verwendet wird und nicht beispielsweise Multitasking. Der Grund dafür ist, dass beim Multitasking unabhängige Programmteile parallel ausgeführt werden, während beim Multi-Threading die einzelnen Threads miteinander kommunizieren können, da sie nicht vollständig unabhängig voneinander sind. Diese Kommunikation ist hier besonders wichtig, da Züge trotz der Aufteilung des Schienennetzes in den Zuständigkeitsbereich eines anderen Threads fahren können. An diesen Stellen müssen die Threads miteinander kommunizieren, um sicherzustellen, dass die Züge ihre vorgesehenen Routen problemlos abfahren können. Die Kommunikation zwischen den Threads bringt jedoch einige Herausforderungen mit sich,

auf die in der Arbeit eingegangen werden muss.

1.2 Struktur der Arbeit

Die Arbeit ist strukturiert wie folgt. In dem nächsten Kapitel werden zunächst die grundlegenden Kenntnisse erläutert, die für das Verständnis der vorliegenden Arbeit erforderlich sind. Die ersten Abschnitte „Schienennetz“ (2.1) und „Simulation von Schienenverkehr“ (2.2), sowie „Zeitplanung in der Simulation“ (2.2.2) und „Priorisierung und Scheduling von Zuginstanzen in der Simulation“ (2.2.3) beschreiben den Aufbau und die Funktionsweise der HÁN-Simulation [HÁN21].

Das Kapitel „Parallele Berechnung der Simulation“ (2.3) bezieht sich dann auf den Ansatz, den diese Arbeit verfolgen wird. Es wird erläutert, wie Cluster in diesem parallelisierten Ansatz genutzt werden. Und welche Anpassungen für die Parallelisierung und zur verbesserung der Laufzeit gemacht werden müssen. Dazu wird zunächst die Aufteilung in Cluster erklärt (2.3.1) und dann wird erläutert was bei der Zeitplanung für mehrere Threads beachtet werden muss (2.3.2). Im Unterabschnitt „Vermeidung von Zeitkonflikten zwischen benachbarten Clustern“ (2.3.3) wird beschrieben, wie Cluster am effizientesten parallel berechnet werden können. Es werden Ideen für Methoden vorgestellt, die testen, ob die einzelnen Cluster ihre Berechnungen problemlos ausführen können, ohne dass Fehler auftreten, und dass möglichst wenig Wartezeit für die Cluster entsteht.

Im nächsten Kapitel wird der Pseudocode für die eigentliche Implementierung beschrieben (3). Zunächst werden auch die verwendeten Notationen und Definitionen eingeführt. Dann werden die einzelnen Funktionen durchgegangen und erklärt, warum sie durchgeführt werden.

Im darauf folgenden Kapitel wird die zuvor durch Pseudocode beschriebene Simulation an einem vereinfachten Beispiel durchgeführt, um die Funktionsweise und den Ablauf der Simulation verständlicher zu machen (4). Dabei wird auch auf das Einlesen der Eingabedateien und die notwendigen Initialisierungen vor dem eigentlichen Start der Simulation eingegangen. Die Simulation wird Schritt für Schritt nach Clustern (bzw. Threads) aufgeteilt und ausgeführt.

Im nächsten Kapitel werden die experimentellen Ergebnisse vorgestellt und interpretiert (5). Abschließend wird eine Zusammenfassung gegeben und die erzielten Ergebnisse eingeordnet. Darauf folgt eine Aussicht auf Themen und mögliche Anpassungen der Simulation, die in Zukunft noch angegangen werden können (6).

Kapitel 2

Grundlagen

Das Kapitel ist in mehrere Abschnitte unterteilt: In Abschnitt 2.1 wird zunächst das Schienennetz erläutert, das in der HÁN-Simulation verwendet wird. Im Abschnitt 2.2 wird die Funktionsweise und Umsetzung bestimmter Aspekte erklärt, die in der HÁN-Simulation verwendet werden. In diesen beiden Abschnitten werden jedoch keine neuen Ideen oder Änderungen an der Simulation beschrieben, die in dieser Arbeit entwickelt wurden. Erst im Kapitel 2.3 werden die neuen Grundlagen für die parallele Umsetzung der Simulation erläutert. Dabei wird auch erörtert, wie die Funktionalitäten der HÁN-Simulation, die im vorherigen Abschnitt beschrieben wurden, dafür weiterverwendet oder angepasst werden müssen.

2.1 Schienennetz

Um eine Simulation eines Schienennetzes durchzuführen, ist es notwendig, zunächst ein Modell zu erstellen. Dabei gibt es makroskopische und mikroskopische Ansätze. Bei mikroskopischen Ansätzen werden viele Details berücksichtigt, was jedoch bedeutet, dass nur ein kleiner Ausschnitt der Realität simuliert werden kann. Auf der anderen Seite abstrahieren makroskopische Ansätze von vielen Details, um einen größeren Ausschnitt der Realität abbilden zu können. Die HÁN-Simulation ist ein Beispiel für einen makroskopischen Ansatz zur Simulation eines Schienennetzes.

Daher erfordert die Modellierung eines Schienennetzes die Abstraktion von verschiedenen Merkmalen, die in der realen Welt vorhanden sind. Dies ist notwendig, da es unmöglich ist, die Realität in allen Details abzubilden und ein Modell somit nicht mehr effektiv wäre. Darüber hinaus sind nicht alle Details für das vorgesehene Anwendungsgebiet des Modells von Bedeutung. In diesem Fall soll das Modell zur Simulation von Verspätungen verwendet werden. Daher können Elemente wie Schienenweichen und Signale, die die Simulation unnötig komplex machen würden, vernachlässigt werden. Die für die Simulation relevanten Merkmale werden im Folgenden näher erläutert.

2.2 Simulation von Schienenverkehr

Zunächst sind Bahnhöfe und die Schienen welche die Bahnhöfe verbinden relevant, da es notwendig ist zu bestimmen, wo sich Züge innerhalb des Schienennetzes be-

finden können. In diesem Zusammenhang sind auch die Züge selbst ein relevanter Aspekt. Des Weiteren wird ein Fahrplan benötigt, um die geplante Route der Züge sowie deren Positionen zu bestimmten Zeitpunkten festzulegen. Auf Basis dieser Informationen kann die Verspätung eines Zuges berechnet werden. Für den Rest dieser Arbeit werden daher folgende Begrifflichkeiten für die oben genannten benötigten Details benutzt.

Das Schienennetz wird als ein gerichteter Graph dargestellt, wobei die Bahnhöfe die Knoten des Graphen und die Schienenverbindungen zwischen den Bahnhöfen die Kanten sind. Knoten und Kanten sind dabei Elemente des Schienennetzes daher wird der Begriff Elemente als Begriff für Knoten und Kanten verwendet.
In der Simulation entsprechen die Züge denjenigen, die im Fahrplan festgelegt sind und zu bestimmten Zeiten eine bestimmte Strecke befahren sollen. Jeder Zug ist durch eine eindeutige ID gekennzeichnet und entspricht einem Zugtyp (bspw. ICE, RE, Güterzug).

2.2.1 Verspätungen in der Simulation

Es sollen Verspätungen berechnet werden, die durch Behinderungen von Zügen entstehen, die nicht mehr exakt ihrem Fahrplan folgen. Hierbei werden nur diejenigen Verspätungen betrachtet, die zu Beginn der Simulation schon vorhanden sind (Initialverspätungen) und die sich aus den daraus folgenden Behinderungen ergeben, wenn ein bereits verspäteter Zug einen anderen Zug behindert.
In der Simulation werden keine Verspätungen berücksichtigt, die während der Simulation aufgrund anderer Faktoren auftreten, die nicht von anderen Zügen verursacht werden. Beispiele in der realen Welt dafür sind Störungen von Signalen oder Komplikationen beim Ein- und Aussteigen von Passagieren an einem Bahnhof, die dazu führen, dass ein Zug länger als geplant halten muss. Um die Komplexität und die Rechenzeit der Simulation zu reduzieren, werden in dieser Arbeit nur Verspätungen berücksichtigt, die von Beginn an im Schienennetz existieren.

Dementsprechend wird für jeden Zug in der Simulation eine bestimmte diskrete Wahrscheinlichkeitsverteilung festgelegt, die bestimmt, welche Verspätung er hat. Zum Beispiel kann ein Zug mit einer Wahrscheinlichkeit von 60% pünktlich (0 Minuten Verspätung) sein und zu je 20% mit 10 oder 30 Minuten Verspätung fahren.
Die Simulation verwendet Zuginstanzen, um diese Verspätungen darzustellen. Jeder Zug mit einer eindeutigen ID und einer geplanten Abfahrtszeit kann in mehrere Instanzen aufgeteilt werden, von denen jede eine bestimmte Verspätung aufweist, aber immer noch die gleiche Strecke fährt. Diese Instanzen werden zu Anfang erzeugt und haben keine weiteren Auslöser für ihre Existenz. Während der Simulation können jedoch immer neue Zuginstanzen von allen Zügen entstehen, beispielsweise wenn eine Zuginstanz (A) von einer anderen Zuginstanz (B) eines anderen Zuges aufgehalten wird, die eine Verspätung aufweist. In diesem Fall teilt sich A in zwei Instanzen (A1 und A2), von denen jede eine eigene Wahrscheinlichkeit und Einschränkungen hat, unter denen sie existiert. Die Einschränkungen (Constraints) beschreiben dabei die Bedingungen, unter denen die Instanz existiert.
Es ist zu beachten, dass sämtliche Züge, die auf dem Schienennetz unterwegs sind und deren Fahrten konkret berechnet werden, in Wirklichkeit bereits Instanzen von

Zügen sind. Aus diesem Grund wird im Weiteren der Begriff „Zuginstanzen“ verwendet, um diesen Umstand zu verdeutlichen. Hierbei ist jedoch zu betonen, dass jede Zuginstanz einer bestimmten Zug-ID zugeordnet wird und nur in einem bestimmten Szenario auftritt.

2.2.2 Zeitplanung in der Simulation

In der Simulation wird eine bestimmte Zeitdauer simuliert, wobei für jeden Zeitschritt berechnet wird, welcher Zug ein Element wechselt, d.h. von einem Knoten auf eine Kante fährt oder umgekehrt. Eine Liste wird erstellt, um zu bestimmen, wann ein Zug ein Element wechseln kann und dementsprechend zu welchem Zeitpunkt etwas in der Simulation berechnet werden muss. Daraus ergeben sich dann die zu berechnenden Zeitschritte.

Wenn ein Zug ein neues Element betritt, wird der Fahrplan überprüft, um zu bestimmen, wann er das nächste Element betreten soll. Diese Zeit heißt „Earliest Possible Departure Time“ (epdt), und wird in die Liste eingetragen, da der Zug zu diesem Zeitpunkt das nächste Element erreichen möchte. Zudem gibt es eine konstante „Blocking Time", die gewartet werden muss, nachdem ein Zug ein Element verlassen hat, bevor der nächste Zug das Element betreten kann. Dies dient der Sicherheit, um sicherzustellen, dass nicht in derselben Sekunde ein Zug aus einem Bahnhof ausfährt und der nächste Zug bereits in den Bahnhof einfährt. Daher wird auch immer dann, wenn ein Zug ein Element gewechselt hat (z.B. von Knoten A auf Kante B), die Zeit, zu der dieser Wechsel stattgefunden hat, plus die Blocking Time in die Liste eingetragen. Denn zu dieser Zeit könnte ein anderer Zug den Knoten A betreten, der zuvor nicht einfahren konnte, weil der Knoten als „besetzt“ galt.

2.2.3 Priorisierung und Scheduling von Zuginstanzen in der Simulation

Bei der Simulation von Schienennetzen wird für jeden Zeitschritt t eine Berechnung durchgeführt, diese Berechnung wird Scheduling genannt. Zunächst werden alle Knoten geschedult, was bedeutet, dass für jeden Knoten alle eingehenden Kanten untersucht werden, um festzustellen, welche Zuginstanzen den Knoten zum Zeitpunkt t oder früher betreten möchten. Falls ausreichend Kapazität vorhanden ist, können alle Zuginstanzen eintreten. Wenn jedoch nicht genug freie Kapazität auf dem Knoten verfügbar ist, werden Priorisierungsregeln verwendet, um zu bestimmen, welche Zuginstanzen eintreten dürfen. In der HÁN-Simulation hängt die Priorisierung von verschiedenen Faktoren ab, wie z.B. dem Zugtyp, der geplanten Ankunftszeit und der Zug-ID. Zuginstanzen desselben Zuges werden als eine Einheit gezählt, da sie nur in verschiedenen Szenarien auftreten. Die Kapazität wird daher in Zügen und nicht in Zuginstanzen gemessen.

Anschließend werden alle Kanten für den Zeitpunkt t geschedult, indem der Ausgangsknoten jeder Kante untersucht wird, um zu bestimmen, welche Zuginstanzen zu diesem Zeitpunkt oder früher eintreten möchten. Wie bei den Knoten wird dann entschieden, welche Zuginstanzen eintreten dürfen. Es ist wichtig, dass zuerst alle Knoten und dann alle Kanten berechnet werden, da Züge durch Knoten durchfahren können,

während Kanten immer eine gewisse Zeit benötigen, um abgefahren zu werden. In der HÁN-Simulation werden alle Knoten und Kanten für jeden relevanten Zeitschritt durchlaufen und für jedes Element wird überprüft, ob ein Zug dieses betreten möchte.

2.3 Parallele Berechnung der Simulation

Das Ziel dieser Arbeit besteht darin, die HÁN-Simulation zu optimieren, um die Simulationszeit für größere Schienennetze über längere Zeiträume zu reduzieren. Hierbei wird angestrebt, die Geschwindigkeit durch die Implementierung von Multi-Threading, d.h. der parallelen Berechnung der Simulation, zu erhöhen. Im bisherigen Grundlagenkapitel wurde die Funktionsweise der HÁN-Simulation erläutert. In dem jetzt folgenden Abschnitt werden grundlegende Anpassungen vorgestellt, die zur Parallelisierung der Simulation erforderlich sind. Hierbei handelt es sich um Abänderungen der ursprünglichen Funktionsweise der HÁN-Simulation, für deren Konzeptionierung und Umsetzung die Ideen in dieser Arbeit entwickelt wurden.

2.3.1 Aufteilung in Cluster

Um die Simulation zu beschleunigen, soll die parallele Ausführung auf mehreren Threads genutzt werden, um die Leistung mehrerer CPU-Kerne auszuschöpfen. Dazu wird das Schienennetz in Cluster unterteilt, wobei jeder Knoten und jede Kante einem bestimmten Cluster zugeordnet wird. Die Anzahl der Cluster entspricht der Anzahl der Threads, die verwendet werden sollen. Es ist sinnvoll, den Graphen so aufzuteilen, dass möglichst wenige Kanten zwischen den Clustern verlaufen, da zwischen den Clustern synchronisiert werden muss, was zeitintensiver ist. Das liegt daran, dass wenn Knoten oder Kanten an der Grenze zwischen Clustern liegen, die Cluster auf die gleichen Variablen zugreifen müssen, was einen gesicherten Zugriff erfordert. Es ist daher wichtig, dass dieser Zugriff so selten wie möglich benötigt wird, um die Geschwindigkeit der Parallelisierung zu erhöhen. Jeder Knoten wird einem Cluster zugeordnet, und jede Kante gehört demselben Cluster wie ihr Ausgangsknoten an. Kanten, die zwischen Clustern verlaufen, gehören dem Cluster an, aus dem sie kommen und werden Außenkanten genannt. Kanten, die nur innerhalb eines Clusters verlaufen, also deren Start und Zielknoten im selben Cluster liegen, heißen Innenkanten.

Um die Simulation parallel auszuführen, wird für jedes Cluster ein Thread gestartet. Die einzelnen Threads berechnen dann nur die Knoten und Kanten, die dem ihnen zugewiesenen Cluster zugeordnet sind.

2.3.2 Zeitplanung für mehrere Threads

Wie im Abschitt 2.2.2 beschrieben wurde, wird eine Liste geführt in der festgehalten wird welche Zeitschritte als nächstes berechnet werden müssen. Bei der parallelen Ausführung ergibt sich dabei das folgende Problem. Wenn die Threads eine gemeinsame Liste haben und somit immer die gleiche Zeit berechnen, wodurch sie ständig aufeinander warten müssen, da nicht alle Cluster zu jedem Zeitpunkt etwas zu berechnen haben, wird die Simulation nicht schneller, sondern vermutlich langsamer, da wie oben beschrieben auch noch synchronisiert werden muss.

Es ist empfehlenswert, die Cluster in unterschiedlichen Geschwindigkeiten rechnen zu lassen, um eine höhere Effizienz der Simulation zu erreichen. Damit ist gemeint, dass nicht alle Cluster immer gleichzeitig dieselben Zeitpunkte berechnen, sondern es ist auch möglich, dass ein Cluster beispielsweise den Zeitpunkt $t = 5$ berechnet, während ein anderes Cluster erst den Zeitpunkt $t = 3$ berechnet. Zu diesem Zweck erhält jedes Cluster eine eigene Liste mit Zeitschritten, die als nächstes zu berechnen sind. Diese Liste beinhaltet nur Zeiten, die Knoten oder Kanten betreffen, welche sich innerhalb des jeweiligen Clusters befinden. Dadurch kann jedes Cluster seine eigene Liste von nächsten Schritten bearbeiten und somit schneller oder langsamer sein als seine Nachbarcluster. Somit können Cluster ihre Nachbarcluster überholen oder auch von ihnen überholt werden.

Im Abschnitt 2.2.3 wurde erläutert, dass in der HÁN-Simulation in jedem Zeitschritt alle Knoten und Kanten nacheinander überprüft werden, um festzustellen, ob eine Zuginstanz das betreffende Element betreten möchte oder nicht. Falls das der Fall ist wird das betroffene Element daraufhin berechnet. Zur Optimierung dieser Vorgehensweise wurde in dieser Arbeit folgender Ansatz verfolgt: Um zu vermeiden, dass alle Knoten und Kanten bei jedem Zeitschritt durchlaufen werden müssen, einschließlich derjenigen, bei denen nichts passiert, ist es sinnvoll, die Knoten und Kanten zu speichern, die zu einem bestimmten Zeitpunkt betrachtet werden müssen, weil ein Zug sie betreten möchte. Diese Information wird in der Liste der relevanten Zeitschritte gespeichert, die auch die jeweiligen Knoten/Kanten enthält, die zu diesem Zeitpunkt betrachtet werden sollen. Dadurch müssen nur noch diese betrachtet werden und nicht alle Elemente bei jedem Zeitschritt durchlaufen werden.

2.3.3 Vermeidung von Zeitkonflikten zwischen benachbarten Clustern

Da Züge auch zwischen den Clustern verkehren können, ist es nicht möglich, dass alle Cluster beliebig weit in der Zeit vorausrechnen können. Wenn ein Cluster (C1) zu einem bestimmten Zeitpunkt t_1 zuerst seine Knoten berechnen möchte, muss sichergestellt werden, dass kein Zug mehr von einem Nachbarcluster (C2) nach C1 geschickt werden kann, falls C2 noch bei t_2 ist und somit zeitlich zurückhängt, weil $t_2 < t_1$. Dazu werden in der Implementierung Testfunktionen verwendet („testeKnoten“ und „testeKanten“) die Ideen für diese Testfunktionen werden in den folgenden zwei Absätzen beschrieben.

Scheduling von Knoten

Bevor die Knoten berechnet werden können, muss sichergestellt werden, dass C2 keinen Zug mehr nach C1 schickt, der vor t_1 ankommt, da dieser dann auch von C1 zum Zeitpunkt t_1 berücksichtigt werden muss. Dazu müssen alle Kanten betrachtet werden, die von C2 nach C1 führen, und überprüft werden, ob ein Zug losgeschickt werden kann, der vor t_1 ankommen kann. Diese Kontrolle muss für alle Nachbarcluster gemacht werden.

Eine Möglichkeit zur Überprüfung, ob ein Zug in der Nähe ist, der die Kante vor dem Zeitpunkt t_1 erreichen wird, wäre die Betrachtung der Strecken von C2 nach c1. Hierbei könnten entweder nur die eingehenden Kanten oder auch die Startknoten sowie weitere Kanten berücksichtigt werden. Diese Berechnungen sind jedoch sehr

zeitaufwendig und würden daher vermutlich keinen Effizienzvorteil bringen.

In der Simulation wird daher eine feste Zeit festgelegt, die zur Überprüfung verwendet wird. Jedes Cluster erhält eine Liste seiner Nachbarcluster sowie die Zeit, die das betreffende Cluster c1 im Vergleich zu C2 bei der Berechnung seiner Knoten voraus sein darf. In dieser Liste wird die Zeit verwendet, die der schnellsten Verbindung entspricht, die ein Zug aus dem Nachbarcluster nehmen kann, um das jeweilige Cluster zu erreichen. Wie die Werte für die schnellste Verbindung berechnet werden, wird im Kapitel zum Pseudocode genauer erläutert (3). Die Überprüfung, die durchgeführt werden muss, um zu entscheiden, ob C1 bereits t_1 berechnen kann oder noch auf C2 warten muss, lautet wie folgt:

Gilt $t_2 \geq t_1$ - schnellste Verbindung?

Wenn C1 also auf der in Abbildung 2.1 dargestellten Zeitleiste den Zeitpunkt t_1 berechnen möchte und C2 zuletzt den Zeitpunkt t_2 berechnet hat, dann ist die obige Bedingung erfüllt. Das liegt daran, dass t_1 minus die schnellste Verbindung der Zeitpunkt t_3 auf der Zeitleiste ist. Das heißt, dass solange das Nachbarcluster von C1 schon mindestens den Zeitpunkt t_3 oder einen späteren Zeitpunkt berechnet hat, die obige Bedingung für C1 und den Zeitpunkt t_1 erfüllt ist und C1 somit seine Knoten berechnen darf. Denn selbst wenn ein Zug nach t_2 noch die schnellste Verbindung nutzt, kann er nicht vor t_1 in C1 ankommen.

Falls C2 jedoch erst bei t_4 mit seinen Berechnungen ist, muss C1 noch warten, bis C2 soweit fortgeschritten ist, dass die oben genannte Bedingung erfüllt ist.

Abbildung 2.1: Zeitleiste

Scheduling von Innenkanten

Nachdem ein Cluster seine Knoten berechnet hat, müssen die Kanten berechnet werden.Um eine Kante (k1) zu schedulen, wird ihr Ausgangsknoten betrachtet und geprüft, ob sich auf diesem eine oder mehrere Zuginstanzen befinden, die genau diese Kante k1 betreten wollen. Da bei allen Innenkanten sowohl der Ausgangs- als auch der Zielknoten bereits geschedult wurden, können die Innenkanten direkt nach den Knoten berechnet werden, ohne dass weitere Überprüfungen erforderlich sind.

Problematik beim Scheduling der Außenkanten

Die Außenkanten können nicht direkt im Anschluss geschedult werden. Das liegt daran, dass der Zielknoten bei Außenkanten in einem benachbarten Cluster liegt. Bei dem Scheduling einer Kante wird zunächst der Ausgangsknoten überprüft, um festzustellen, ob und wie viele Zuginstanzen von diesem Knoten die Kante betreten möchten. Anschließend wird überprüft, wie viel freie Kapazität diese Kante noch hat, um zu entscheiden, welche Zuginstanzen die Kante betreten dürfen. Dabei spielt der Zielknoten

eine Rolle, da er dafür verantwortlich ist, die Zuginstanzen wieder von dieser Außenkante herunterzunehmen, wodurch sich die freie Kapazität wieder vergrößert. Wenn das benachbarte Cluster zeitlich hinterherhängt, kann es sein, dass Zuginstanzen, die sich noch auf der Kante befinden, eigentlich bereits die Kante verlassen haben sollten. Wenn die Außenkanten einfach geschedult würden, ohne zu überprüfen wie weit das Nachbarcluster bereits gerechnet hat, würden inkorrekte Ergebnisse daraus resultieren. Daher ist es erforderlich, für die Außenkanten nochmals gesondert zu überprüfen, ob die benachbarten Cluster bereits weit genug gerechnet haben.

Ausnahmefall für das Scheduling von Außenkanten

Um die Außenkanten eines Clusters berechnen zu können, muss zunächst überprüft werden, ob dies für alle Außenkanten ohne Probleme möglich ist. Die Überprüfung wird für alle Außenkanten durchgeführt, und erst wenn der Test für alle Außenkanten erfolgreich ist, werden die Außenkanten des Clusters gemeinsam berechnet.
Die Überprüfung, ob die Außenkanten eines Clusters c1 zu einem bestimmten Zeitpunkt t berechnet werden können, erfolgt anders als bei den Knoten. Der Test muss nur für Außenkanten durchgeführt werden, deren Zielknoten in einem Cluster liegen, das zeitlich zurückhängt gegenüber dem Cluster, zu dem die Außenkante gehört. Wenn dieses Nachbarcluster also bereits mindestens bis zum Zeitpunkt t gerechnet hat, können alle Außenkanten, die in dieses Nachbarcluster führen, problemlos berechnet werden. Daher wird dies zuerst in der Testfunktion überprüft, und wenn das Zielcluster bereits mindestens t berechnet hat, wird direkt TRUE für diese Kante zurückgegeben. Der Rest der Überprüfung muss nur noch für Außenkanten durchgeführt werden, die in ein Cluster führen, das noch nicht bis t gerechnet hat.

Idee der Testfunktion für Außenkanten

Im Gegensatz zur Überprüfung bei den Knoten kann hier nicht vorhergesagt werden, ob eine bestimmte Zeit bezüglich eines Nachbarclusters C2 vorherberechnet werden kann. Es kann sein, dass C2 erst zum Zeitpunkt t oder später die Zuginstanzen von der Kante nimmt, und dies muss berücksichtigt werden. Zum Beispiel kann es sein, dass sich auf dem Zielknoten verspätete Zuginstanzen befinden oder die Zuginstanzen auf der Außenkante selber verspätet sind, sodass sich bei diesem Übergang die Zuginstanzen blockieren und auf der Kante warten müssen. Allerdings ist es nicht erkennbar ob der gerade beschriebene Fall eintritt oder zu welchem Zeitpunkt genau die Zuginstanzen von der Kante genommen werden. Damit trotzdem bestimmt werden kann, ob die Außenkanten berechnet werden können werden daher verschiedene Faktoren betrachtet.

Fall 1 der Testfunktion für Außenkanten

Für die Kontrolle des ersten Faktors, muss für jede Außenkante überprüft werden, ob es genügend freie Kapazitäten für alle Zuginstanzen gibt, die auf die Kante fahren wollen. Wenn dies der Fall ist, kann die Kante problemlos geschedult werden, da es unerheblich ist, ob noch Zuginstanzen entfernt werden oder nicht. Wenn jedoch

nicht genügend freie Kapazitäten vorhanden sind, muss der zweite Faktor betrachtet werden.

Fall 2 der Testfunktion für Außenkanten

Dafür müssen alle Zuginstanzen berücksichtigt werden, die sich auf der Außenkante befinden. Es werden insbesondere die epdt-Zeiten der Zuginstanzen untersucht. Wenn noch keine dieser Zuginstanzen ihre epdt vor dem Zeitpunkt t erreicht hat, bedeutet dies, dass sich auch die freien Kapazitäten dieser Kante bis zum Zeitpunkt t nicht mehr ändern werden, da Zuginstanzen auf keinen Fall vor ihrem epdt das Element wechseln können. Dann kann die Kante ebenfalls geschedult werden.

Fall 3 der Testfunktion für Außenkanten

Falls nicht ausreichend freie Kapazität zur Verfügung steht und mindestens eine Zuginstanz ihre epdt erreicht hat, ist es nicht möglich, diese Kante zu schedulen, bevor das benachbarte Cluster ausreichend weit gerechnet hat. Es ist möglich, dass beim nächsten Schedulen des Zielknotens der ausgehenden Kante die oben beschriebene Überprüfung positiv ausfällt. Dies ist jedoch nicht sicher, da es genauso gut möglich ist, dass das benachbarte Cluster bis zum Zeitpunkt t keinen Zug mehr abnimmt, und das Ergebnis der Überprüfung somit negativ bleibt, was im Voraus nicht bekannt ist. Da für jede Außenkante eines Clusters die oben beschriebene Überprüfung durchgeführt werden muss, ist es wahrscheinlich effizienter, alle Cluster zu sammeln, auf die gewartet werden muss. Anschließend kann kann gewartet werden, bis alle diese Cluster mindestens bis zum Zeitpunkt t berechnet haben. Hierfür speichert man das Cluster des Zielknotens einer ausgehenden Kante, falls diese die Überprüfung nicht bestanden hat. So kann man darauf warten, dass alle diese Cluster bis zum Zeitpunkt t berechnet sind.
Anstatt für jede dieser Kanten immer wieder zu überprüfen, ob das benachbarte Cluster den Zielknoten geschedult hat und ob die Überprüfung nun positiv ausfällt, wird in der Simulation auf diese Zielcluster gewartet, bis sie ebenfalls bis zum Zeitpunkt t berechnet haben. Obwohl durch eine regelmäßige erneute Überprüfung eventuell auch schon früher weitergerechnet werden könnte, als wenn das benachbarte Cluster bei t angekommen ist, ist diese immer wieder notwendige Überprüfung wahrscheinlich ein so großer Overhead, dass sie keinen Effizienzvorteil bringen würde.

Kapitel 3

Pseudocode

In diesem Kapitel wird die parallele Ausführung der Simulation anhand von Pseudocode erläutert. Dabei werden die bereits vorgestellten Testfunktionen sowie die verwendeten Datenstrukturen und Objekte genau erläutert.

3.1 Benutzte Notationen

Im Pseudocode der Simulation werden die benötigten Objekte als n-Tupel dargestellt, wobei jede Stelle des Tupels ein Attribut des Objekts für die Simulation repräsentiert. Die Attribute können aus den folgenden Mengen stammen:

$\mathbb{N}$: die natürlichen Zahlen dabei gilt in dieser Arbeit $0 \in \mathbb{N}$
$\mathbb{R}$: die reellen Zahlen
$\mathbb{B}$: ein Wahrheitswert (wahr oder falsch)

Die Notation 2^X bedeutet, dass das entsprechende Attribut eine Menge von Elementen aus X enthält anstatt nur eines einzelnen Wertes, wobei X eine der oben genannten Mengen darstellt. Attribute, die nur einen Wert enthalten, werden in Kleinbuchstaben geschrieben, während Attribute, die eine Menge von Werten enthalten, in Großbuchstaben geschrieben werden. Um auf spezifische Objekte im Pseudocode zu verweisen, werden Tupel verwendet, wobei irrelevante Stellen des Tupels mit einem Stern (*) gekennzeichnet werden. Das bedeutet, dass an dieser Stelle des Tupels beliebige Werte stehen können, da das entsprechende Attribut für das Objekt in diesem Fall irrelevant ist.

3.2 Definitionen

Die Menge ZugInstanzen besteht aus einer Teilmenge aller 4-Tupel aus $\mathbb{N}\times\mathbb{N}\times\mathbb{N}\times\mathbb{R}$. Für ein Tupel $(id, zid, neid, epdt) \in$ ZugInstanzen ist id die eindeutige ID der jeweiligen Zuginstanz aus den natürlichen Zahlen. zid (Züge ID) identifiziert den Zug, zu dem die Instanz gehört. Es kann mehrere Instanzen desselben Zugs geben. $neid$ (Nächstes Element ID) ist die ID des nächsten Infrastrukturelements, das die Zuginstanz betreten möchte, also ein Knoten oder eine Kante. Es ist immer klar, ob ein

Knoten oder eine Kante gemeint ist, da diese abwechselnd aufeinander folgen. Wenn sich die Zuginstanz auf einer Kante befindet, ist *neid* eine Knoten-ID und umgekehrt. *epdt* ist die „earliest possible departure time“, also die Zeit, zu der die Zuginstanz das Element, auf dem sie sich derzeit befindet, frühestens verlassen möchte.

In dieser Arbeit wird der Begriff KNOTEN als eine Teilmenge aller 5-Tupel aus $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times 2^{\mathbb{N}} \times \mathbb{N}$ definiert. Jedes Tupel $(id, cid, kap, ZID, zanz) \in$ KNOTEN repräsentiert einen Knoten. Dabei steht *cid* für die ID des Clusters, in dem sich der Knoten befindet, *kap* für die Kapazität des Knotens, ZID für die Menge der IDs der Zuginstanzen, die sich im Knoten befinden, und *zanz* für die Anzahl der Züge, die sich auf dem Knoten befinden. In diesem Zusammenhang bezieht sich der Begriff „Züge“ tatsächlich auf Züge und nicht auf Instanzen von Zügen. Dies liegt daran, dass Instanzen eines Zuges sich gegenseitig ausschließen, da sie nie gleichzeitig auftreten können, da es sich nur um einen real existierenden Zug handelt. Die Anzahl der Züge, die sich auf einem Element befinden, wird verwendet, um zu prüfen, ob noch weitere Instanzen eines Zuges darauf fahren können. Instanzen desselben Zuges können jedoch immer beliebig viele auf demselben Element sein, ohne dass sich die Anzahl freier Plätze auf dem Element verringert, da sie wie bereits erwähnt niemals gleichzeitig auftreten können, da es sich nur um einen Zug handelt.

Der Begriff INNENKANTEN beschreibt eine Teilmenge aller 7-Tupel aus $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times 2^{\mathbb{N}} \times \mathbb{N}$. Ein Tupel $(id, cid, skid, zkid, kap, ZID, zanz) \in$ INNENKANTEN besteht aus einer eindeutigen ID für die Kante (id) und auch dem Cluster (cid), in dem sich die Kante befindet. Zusätzlich wird angegeben, welche beiden Knoten ($skid$ für den Startknoten und $zkid$ für den Zielknoten) die Kante verbindet. kap, ZID und $zanz$ werden für Innenkanten genauso definiert wie für Knoten. Innenkanten sind somit alle Kanten, deren Start- und Endknoten im selben Cluster liegen.

Der Begriff AUSSENKANTEN bezieht sich auf eine Teilmenge von 8-Tupeln aus $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times 2^{\mathbb{N}} \times \mathbb{N}$, wobei $(id, acid, zcid, skid, zkid, kap, ZID, zanz)$ ein Tupel ist, das die Ausgangs- ($acid$) und Ziel-Cluster ($zcid$) der Kante enthält. $skid, zkid, kap, ZID$ und $zanz$ werden genauso wie für INNENKANTEN definiert.

Der Begriff CLUSTER beschreibt eine Teilmenge von 8-Tupeln aus $\mathbb{N} \times 2^{\mathbb{N}} \times 2^{\mathbb{N}} \times 2^{\mathbb{N}} \times \mathbb{R} \times \mathbb{B} \times 2^{\mathbb{N}} \times 2^{\mathbb{N}}$. In einem Tupel $(id, KID, IKID, AKID, t, k, NCID, WCID)$ bezeichnet id dabei die eindeutige ID des Clusters und $KID/IKID/AKID$ sind die IDs aller Knoten/Außenkanten/Innenkanten in dem Cluster (Knoten ID, Innenkanten ID, Außenkanten ID). Die aktuelle Zeit des Clusters wird durch t angegeben. Der Wahrheitswert k ist TRUE, wenn die Knoten bis zur aktuellen Zeit t berechnet wurden. Er ist FALSE, wenn alle Knoten und Kanten bis zum aktuellen Zeitpunkt berechnet wurden und der Zeitschritt abgeschlossen ist. Die IDs der benachbarten Cluster werden durch $NCID$ (Nachbarcluster ID) und die IDs der Cluster, auf die aktuell gewartet werden muss, durch $WCID$ (Warten auf Cluster ID) bezeichnet.

Die Menge NAECHSTEZEITSCHRITTE besteht aus einer Teilmenge aller 5-Tupeln von Elementen aus $\mathbb{N} \times \mathbb{R} \times 2^{\mathbb{N}} \times 2^{\mathbb{N}} \times 2^{\mathbb{N}}$, die Teil der nächsten Zeitschritte in einem Cluster sind. Ein Element $(id, t, KID, IKID, AKID)$ aus NAECHSTEZEITSCHRITTE besteht aus der Zeit (t) und den Mengen an Knoten (KID), InnenKanten ($IKID$) und AußenKanten ($AKID$), die zu diesem Zeitpunkt im Cluster (id) betrachtet wer-

den sollen, da Zuginstanzen diese betreten möchten.

Die Menge KuerzesterPfadNach besteht aus einer Teilmenge aller 3-Tupel, die aus $\mathbb{N} \times \mathbb{N} \times \mathbb{R}$ sind. Dabei gibt ein Tupel $(cid, ncid, t)$ die Zeit (t) an, die für die schnellste Verbindung vom Nachbarcluster mit der ID *ncid* in das Cluster mit der ID *cid* benötigt wird.

3.3 Funktionen

Im Anhang 6.3.2 ist eine Zusammenstellung der zuvor vorgestellten Mengen und Funktionen zu finden, die beim Verfolgen des Pseudocodes hilfreich sein kann, um den Überblick zu behalten.

3.3.1 Ablauf der Simulation

Nachdem die Eingabedateien eingelesen wurden (dies wird in Kapitel 4.1 anhand eines Beispiels genauer erläutert), werden die Threads gestartet. Dabei werden 2 Threads gestartet, die jeweils die Main-Funktion (1) und die zugewiesene Cluster-ID als Eingabe erhalten, sodass jedes Cluster von einem Thread simuliert wird. Dies geschieht in einer übergeordneten Hauptfunktion, die auch für das Einlesen und Starten der Threads verantwortlich ist. In dieser Funktion wird auch festgelegt, dass die Threads nach Beendigung ihrer Main-Funktion wieder zusammengeführt werden. Die Zeitmessung für die Simulationsdauer wird von vor dem Starten der Threads bis nach der Zusammenführung durchgeführt.

In der Simulation wird für jeden Zeitschritt, der in der Liste NaechsteZeitschritte enthalten ist, geplant, welche Knoten und Kanten zu diesem Zeitpunkt betrachtet werden sollen. Jeder Eintrag in NaechsteZeitschritte enthält eine *cid*, die einem Cluster zugeordnet ist. Da jeder Thread für die Simulation eines eigenen Clusters zuständig ist, betrachtet jeder Thread nur die Einträge, die die *cid* des zugehörigen Clusters enthalten.

Für jedes Cluster wird ein Thread gestartet und in jedem Thread wird die Main-Funktion einmal mit der Eingabe einer Cluster-ID aufgerufen. Jeder Thread erhält dabei eine eigene Cluster-ID.
Jeder Thread durchläuft eine While-Schleife und berechnet in jeder Iteration einen weiteren Zeitschritt. Die Schleife wird so lange durchlaufen, bis NaechsteZeitschritte keine Einträge mehr enthält, was bedeutet, dass alle Threads fertig gerechnet haben und alle Zuginstanzen in ihrem Zielknoten angekommen sind. Das funktioniert, da wenn eine Zuginstanz ihr Element wechselt, ihre neue epdt in die NaechsteZeitschritte Liste geschrieben wird. Wenn sie jedoch am Ende in ihrem Zielknoten ist, gibt es keine weitere epdt und somit wird auch kein Eintrag mehr zur Liste hinzugefügt.

Algorithm 1 Main($cid \in \mathbb{N}$)

while NaechsteZeitschritte $\neq \emptyset$ **do**
  $time = \min t : (cid, t, *, *, *) \in$ NaechsteZeitschritte
  BerechnungProClusterUndTimestep($cid, time, sd$)

Der Algorithmus 1 zeigt, wie die While-Schleife für jedes Cluster durchlaufen wird.

Dabei meint *sd* eine Konstante, die angibt, wie lange ein Cluster wartet (in BerechnungProClusterUndTimestep), bevor es erneut überprüft, ob es weiter rechnen kann, wenn es gerade auf Nachbarcluster wartet. Die Main-Funktion ruft BerechnungProClusterUndTimestep jeweils für das kleinste t aus NaechsteZeitschritte auf, welches auch zu dem Cluster gehört.

Die While-Schleife sorgt dafür, dass NaechsteZeitschritte des aktuellen Clusters solange überprüft wird, bis kein Cluster mehr einen Eintrag in NaechsteZeitschritte hat. Es ist nicht ausreichend, nur das aktuelle Cluster zu überprüfen, da es vorkommen kann, dass in Zukunft noch ein Zug von einem anderen Cluster in das aktuelle gefahren kommt, obwohl die eigenen NaechsteZeitschritte leer sind.

3.3.2 Ablauf einzelner Simulationsschritte

Algorithm 2 BerechnungProClusterUndTimestep($cid \in \mathbb{N}, time \in \mathbb{R}, sd \in \mathbb{R}$)

1: let $(cid, *, *, *, t, k, *, WCID) \in$ Cluster
2: let $(cid, time, KID, IKID, AKID) \in$ NaechsteZeitschritte
3: **if** testeKnoten(cid,$time$) = True **then**
4: **for each** $(id, cid, *, *, *) \in$ Knoten mit $id \in KID$ **do**
5: berechneKnoten($cid, id, time$)
6: **for each** $(id, cid, *, *, *, *, *) \in$ InnenKanten mit $id \in IKID$ **do**
7: berechneKante($cid, id, time$)
8: $t' = time$
9: $k' =$ True
10: $Cluster \leftarrow Cluster \setminus \{(cid, *, *, *, t, k, *, WCID)\}$
11: $Cluster \leftarrow Cluster \cup \{(cid, *, *, *, t', k', *, WCID)\}$
12: **if** testeKanten($cid, time$) = True **then**
13: **for each** $(id, *, *, *, *, *, *, *) \in$ AussenKanten mit $id \in AKID$ **do**
14: berechneKante($cid, id, time$)
15: NaechsteZeitschritte $\leftarrow$ NaechsteZeitschritte $\setminus$ $\{(cid, time, KID, IKID, AKID)\}$
16: $k'' =$ False
17: Cluster $\leftarrow$ Cluster $\setminus\{(cid, *, *, *, t', k', *, WCID)\}$
18: Cluster $\leftarrow$ Cluster $\cup\{(cid, *, *, *, t', k'', *, WCID)\}$
19: **else**
20: **while** $\neg\forall(cid', *, *, *, t', k', *, *) \in$ Cluster $\wedge$ $cid' \in WCID : (time < t') \vee (time = t' \wedge k' =$ True) **do**
21: **wait**
22: **for each** $(id, *, *, *, *, *, *, *) \in$ AussenKanten mit $id \in AKID$ **do**
23: berechneKante($cid, id, time$)
24: **else**
25: schlafen für SchlafDauer (sd)

Die Funktion „BerechnungProClusterUndTimestep“ (2) ist dafür verantwortlich, zu prüfen, welche Berechnungen bereits durchgeführt werden können, und diese dann entsprechend auszuführen. Dazu werden die Funktionen „testeKnoten/-Kanten“ und „berechneKnoten/-Kante“ verwendet.

Zunächst wird überprüft, ob die Knoten für den Zeitpunkt *time* bereits berechnet werden dürfen. Wenn nicht, kann keine Berechnung durchgeführt werden und der Thread wartet kurz, bevor er erneut überprüft. Die Dauer des Wartens (in der Variable *sd*) ist variabel, danach sucht die Hauptfunktion neue Einträge in der Liste NaechsteZeitschritte. Es ist wichtig, erneut den kleinsten Eintrag in NaechsteZeitschritte zu suchen, da ein False-Ergebnis der Funktion „testeKnoten“ bedeutet, dass es für Nachbarcluster noch möglich ist, Zuginstanzen auf ihre ausgehenden Kanten zu senden, die vor dem aktuell getesteten Zeitpunkt *time* in diesem Cluster ankommen. Wenn dies der Fall ist, wird eine Zeit in NaechsteZeitschritte hinzugefügt, die kleiner ist als *time*, und dieser Eintrag muss dann zuerst noch geschedult werden. Dieser Eintrag darf nicht übersehen werden, was allerdings passieren würde, wenn nicht abgewartet wird, bis *time* berechnet werden darf. Wenn „testeKnoten“ True zurückgibt, werden die Knoten sowie die Innenkanten berechnet. An dieser Stelle wird auch der Wert von t für das Cluster angepasst und die Variable k auf True gesetzt, um anzuzeigen, dass bis jetzt nur die Knoten für den aktuellen Zeitpunkt t berechnet wurden.

Im nächsten Schritt wird getestet, ob auch die Außenkanten berechnet werden können. Falls dies der Fall ist, werden diese berechnet und die Variable k wird wieder auf False gesetzt, um anzuzeigen, dass alle Elemente für den Zeitschritt t berechnet wurden. Aus diesem Grund wird auch das Tupel für den Zeitschritt t aus der Menge NaechsteZeitschritte entfernt. Wenn die Außenkanten nicht berechnet werden dürfen, wartet das Cluster so lange, bis alle Nachbarcluster, auf die es warten muss, mindestens auch bis zum Zeitschritt t gerechnet haben und scheduled die Außenkanten dann erst. Hierbei genügt es, wenn die Knoten des Nachbarclusters für den Zeitschritt t geschedult wurden und somit die Außenkanten des Nachbarclusters noch nicht geschedult wurden. In diesem Fall gilt $t_{nachbar} = t$ und $k_{nachbar} =$ True. Der Grund dafür liegt darin, dass es für die Berechnung der Außenkanten relevant ist, ob der Zielknoten der Außenkante bereits Zuginstanzen abgenommen hat. Wenn die Knoten des Nachbarclusters bereits für t geschedult wurden, ist dies bereits geschehen.

3.3.3 Testfunktionen

In der Funktion „BerechnungProClusterUndTimestep“ werden Testfunktionen verwendet, um zu überprüfen, ob Knoten und Kanten bereits berechnet werden dürfen. Diese werden im Folgenden genauer erklärt.

testeKnoten

Die Testfunktion für Knoten, „testeKnoten“ (3), überprüft jedes Nachbarcluster, um festzustellen, ob es mindestens soweit gerechnet hat, dass selbst der schnellste Zug auf der kürzesten Strecke nicht mehr zur aktuellen Zeit des Clusters (*time*) dort ankommen kann. Wenn dies nicht der Fall ist, wird das Nachbarcluster zur WartenAuf-Liste hinzugefügt. Nur wenn auf kein Cluster gewartet werden muss, wirdTrue zurückgegeben.

Die Liste KuerzesterPfadNach wird beim Einlesen des Fahrplans befüllt. Hierbei wird bei jeder Nutzung einer Kante zwischen zwei Clustern überprüft, ob es bereits einen Eintrag für eine Verbindung zwischen diesen beiden Clustern gibt. Wenn ja, wird überprüft, ob die Zeit, die der Zug laut Fahrplan benötigt, um diese Kante abzufahren, kürzer ist als die Zeit, die bereits in der Liste steht. In diesem Fall wird die Zeit

Algorithm 3 testeKnoten($cid \in \mathbb{N}, time \in \mathbb{R}$)

```
1: let (cid, *, *, *, *, *, NCID, WCID) ∈ Cluster
2: WCID = ∅
3: for each (id', *, *, *, t', *, *, *) ∈ Cluster mit id' ∈ NCID do
4:     (cid, id', timeTo) ∈ KuerzesterPfadNach
5:     if time − timeTo > t' then
6:         WCID' = WCID ∪ id'
7:         Cluster ← Cluster \{(cid, *, *, *, *, *, NCID, WCID)}
8:         Cluster ← Cluster ∪{(cid, *, *, *, *, *, NCID, WCID')}
9: if WCID = ∅ then
10:    return True
11: else
12:    return False
```

in der Liste aktualisiert. Wenn es noch keinen Eintrag für eine Verbindung zwischen diesen beiden Clustern gibt, wird ein neuer Eintrag erstellt und die Zeit, die der Zug für die Kante benötigt, wird in die Liste KuerzesterPfadNach eingetragen.

Dies führt dazu, dass nachdem der gesamte Fahrplan eingelesen wurde, in der Liste KuerzesterPfadNach für alle existierenden Verbindungen zwischen zwei Clustern die kürzeste Zeit eingetragen ist, in der ein Zug die Strecke von einem Cluster ins andere überqueren kann. Wenn es zwischen zwei Clustern keine direkte Verbindung gibt, werden diese Cluster auch nicht als Nachbarcluster angesehen. Die Menge $NCID$ (Nachbarcluster ID) aus dem Tupel $(id, KID, IKID, AKID, t, k, NCID, WCID)$, welches ein Cluster beschreibt, wird auch an dieser Stelle befüllt. Ein Cluster sieht andere Cluster also nur als seine Nachbarcluster an, wenn es von diesen Nachbarclustern eine Verbindung in das eigene Cluster gibt.

testeKanten

Im Algorithmus „testeKanten“ (4) wird für jede Außenkante des Clusters geprüft, ob sie bereits berechnet werden kann. Dazu wird in Zeile 6 über alle Außenkanten iteriert, die sich in der Liste NaechsteZeitschritte befinden und zum angegebenen Zeitpunkt *time* geschedult werden sollen sowie zum angegebenen Cluster *cid* gehören. Es wird geprüft, welcher der folgenden drei Fälle vorliegt:

- Es gibt genug freie Kapazität für alle Zuginstanzen
- Es gibt nicht genug freie Kapazität, aber die freie Kapazität kann sich bis *time* auch nicht mehr ändern
- Es gibt nicht genug freie Kapaziät und diese kann sich bis *time* noch verändern

Im Folgenden werden die genauen Bedingungen für die drei genannten Fälle und das Verhalten der Simulation in diesen Fällen näher erläutert.

1. Es gibt ausreichend freie Kapazität auf der Kante, sodass alle Zuginstanzen, die darauf fahren wollen, dies auch können. Dafür iteriert die for-Schleife in Zeile 9 über die Zuginstanzen, um festzudstellen wie viele Zuginstanzen die jeweilige Außenkante betreten möchten. Dies wird durch die if-Abfrage in Zeile 10

Algorithm 4 testeKanten($cid \in \mathbb{N}, time \in \mathbb{R}$)

```
1: let (cid, *, *, *, *, *, *, WCID) ∈ Cluster
2: WCID' ← ∅
3: Cluster ← Cluster \{(cid, *, *, *, *, *, *, WCID)}
4: Cluster ← Cluster ∪{(cid, *, *, *, *, *, *, WCID')}
5: (cid, time, *, *, AKID') ∈ NaechsteZeitschritte
6: for each (id, *, zcid, skid, *, kap, ZID, zanz) ∈ AussenKanten mit id ∈ AKID'
   do
7:     (skid, cid, *, ZID', *) ∈ Knoten
8:     Anzahl ← 0
9:     for each (id'', *, neid, epdt) ∈ ZugInstanzen mit id'' ∈ ZID' ∧ epdt ≤ time
   do
10:        if neid = id then
11:            Anzahl ← Anzahl + 1
12:    if kap − zanz ≥ Anzahl then
13:        continue
14:    else
15:        for each (id', *, epdt) ∈ ZugInstanzen mit id' ∈ ZID do
16:            if time > epdt then
17:                WCID ← WCID ∪ zcid
18:                break
19: if WCID = ∅ then
20:    return True
21: else
22:    return False
```

überprüft. Wenn die Zuginstanz genau diese Kante betreten möchte, wird der Zähler in Zeile 11 erhöht. Wenn genug freie Kapazität vorhanden ist, wird die nächste Kante überprüft. Daher wird in Zeile 12 überprüft, ob die Anzahl der freien Plätze größer ist als die Anzahl der Zuginstanzen, die die Kante betreten möchten. Wenn dies der Fall ist, wird mit „continue“ direkt zur nächsten Außenkante in der for-Schleife aus Zeile 6 gesprungen, da die überprüfte Kante berechnet werden darf.

2. Wenn die Kante nicht genügend freie Kapazität hat, damit alle Züge darauf fahren könnten, wird geprüft, ob die*epdt* aller Zuginstanzen, die sich aktuell auf der Kante befinden, kleiner als *time* ist (siehe if-Abfrage in Zeile 16). Wenn dies der Fall ist, kann die Kante berechnet werden, da sich bis zum Zeitpunkt *time* nichts an der freien Kapazität ändern kann.

3. Ist jedoch nicht genügend Kapazität frei, damit alle Zuginstanzen auf die Kante fahren können (also nicht Fall 1), und es gibt mindestens eine Zuginstanz, deren *epdt* bereits erreicht ist (also nicht Fall 2), tritt Fall 3 ein. In diesem Fall wird das Zielcluster dieser Kante zur Menge $WCID$ hinzugefügt (Cluster, auf die gewartet werden muss), da die Kante noch nicht berechnet werden kann. Dies geschieht in Zeile 17, und in Zeile 18 wird die for-Schleife aus Zeile 15 durch „break“ direkt verlassen, da es nicht mehr notwendig ist, die restlichen Zuginstanzen dieser Außenkante zu überprüfen. Dies liegt daran, dass eine Zuginstanz, deren *epdt* vor *time* erreicht wird, ausreichend ist, um zu zeigen, dass

diese Kante nicht direkt geschedult werden kann, da diese Zuginstanz die Kante noch verlassen könnte und somit die freie Kapazität wieder erhöht würde. Daher kann direkt mit der nächsten Außenkante in der for-Schleife aus Zeile 6 fortgefahren werden.

Wenn auf kein Cluster gewartet werden muss (dh alle Züge auf die Außenkanten fahren können oder keiner seine *epdt* erreicht hat und daher auch normal geschedult werden kann), wird TRUE zurückgegeben. Andernfalls wird FALSE zurückgegeben, was in der if-Abfrage in Zeile 19 passiert.

3.3.4 Scheduling von Knoten und Kanten

In der BerechnungProClusterUndTimestep (2) wird nach dem jeweiligen Test, ob Knoten oder Kanten berechnet werden dürfen, die Funktion berechneKnoten (5) bzw. berechneKante (6) aufgerufen. Diese Funktionen wurden bereits in der nicht parallelisierten Version der Simulation verwendet.

In der Funktion berechneKnoten (5) wird für jeden Knoten, der von einer Zuginstanz betreten werden soll, eine Anfrage erstellt. Anschließend prüft die Update-Funktion alle Anfragen für diesen Knoten und entscheidet aufgrund der Anfragen und der verfügbaren Kapazität, welche Zuginstanzen den Knoten betreten dürfen. Die Anfragen enthalten auch Informationen zum Zugtyp, zu seiner aktuellen epdt und zu seiner ID, sodass sie verwendet werden können, um zu entscheiden, welche Zuginstanzen den Knoten betreten dürfen. Die Funktion aktualisiert auch die Position der entsprechenden Zuginstanzen.

Die Funktion berechneKanten (6) arbeitet ähnlich wie die Funktion berechneKnoten und verwendet ebenfalls die Update-Funktion, um zu entscheiden, welche Zuginstanzen die Kante betreten dürfen. Für eine detailliertere Beschreibung dieser Funktionen wird auf das Paper von Rebecca Haehn, Erika Ábrahám und Nils Nießen verwiesen ([HÁN21]). Eine Änderung, die gegenüber der ursprünglichen Version der Simulation vorgenommen wurde, besteht in der Hinzufügung der Funktionen KnotenAktualisieren und KantenAktualisieren. Diese Funktionen dienen dazu, die Liste NAECHSTEZEITSCHRITTE zu aktualisieren, wenn eine Zuginstanz das Element gewechselt hat, auf dem sie sich befand. Sie aktualisieren auch *zanz*, also die Anzahl an Zuginstanzen, die sich derzeit auf den einzelnen Knoten und Kanten befinden. Diese Funktionen erhalten als Eingabe die ID des Elements, das gerade geschedult wurde, sowie die Id des Cluster, zu dem dieses Element gehört.

Algorithm 5 berechneKnoten($cid \in \mathbb{N}, kid \in \mathbb{N}$,t$\in \mathbb{R}$)

1: Scheduling des Eingabeknoten *kid* wie in der originalen Version der Simulation.
2: $IKID \in Innenkanten$ ist die Menge aller Innenkanten, von denen eine Zuginstanz auf *kid* gefahren ist
3: $AKID \in Aussenkanten$ ist die Menge aller Außenkanten, von denen eine Zuginstanz auf *kid* gefahren ist
4: **for each** $ikid \in IKID$ **do**
5: InnenKantenAktualisieren($cid, ikid, t$)
6: **for each** $akid \in AKID$ **do**
7: AußenKantenAktualisieren($acid, akid, t$)
8: KnotenAktualisieren(cid, kid, t)

Algorithm 6 berechneKante($cid \in \mathbb{N}, kid \in \mathbb{N}$,t$\in \mathbb{R}$)

1: Scheduling der Eingabekante *kid* wie in der originalen Version der Simulation.
2: $skid \in Knoten$ ist der Startknoten der berechneten Kante, von der Zuginstanzen auf die Kante gefahren sind
3: KnotenAktualisieren(cid,$skid, t$)
4: KantenAktualisieren(cid, kid, t)

Algorithm 7 KnotenAktualisieren($cid \in \mathbb{N}, kid \in \mathbb{N}$)

1: let $(kid, cid, *, ZID, zanz) \in Knoten$
2: $Z \leftarrow \emptyset$
3: $zanz \leftarrow 0$
4: **for each** $(id, zid, neid, epdt) \in Zuginstanz$ mit $id \in ZID$ **do**
5: $(cid, epdt, *, IKID, AKID) \in NaechsteZeitschritte$
6: **if** $\exists(neid, cid, kid, *, *, *, *) \in InnenKanten$ **then**
7: $IKID \leftarrow IKID \cup \{neid\}$
8: **else**$\exists(neid, cid, kid, *, *, *, *) \in AussenKanten$
9: $AKID \leftarrow AKID \cup \{neid\}$
10: **if** $zid \notin Z$ **then**
11: $Z \leftarrow Z \cup zid$
12: $zanz \leftarrow zanz + 1$

Die Funktion „KnotenAktualisieren“ (7) aktualisiert sowohl NaechsteZeitschritte als auch *zanz* des gerade berechneten Knotens. Sie wird in berechneKnoten für jeden Knoten aufgerufen, den neue Zuginstanzen „betreten“ haben, die eine neue *epdt* erhalten haben (diese wird in „berechneKnoten“ aktualisiert) und zu dieser auf eine Kante (innen oder außen) wechseln möchten. Daher werden die Kanten, auf die gewechselt werden soll, als IKID oder AKID zur entsprechenden *epdt* hinzugefügt in NaechsteZeitschritte. Dazu wird über alle Zuginstanzen iteriert, die sich derzeit auf dem Knoten befinden, und das Element gesucht, das als nächstes betreten werden soll (*neid* $\rightarrow$ nächstes Element-ID). Je nachdem, ob es sich um eine Innen- oder Außenkante handelt, wird es der entsprechenden Menge in NaechsteZeitschritte mit der neuen *epdt* dieser Zuginstanz hinzugefügt. Die If-Abfrage in Zeile 6 überprüft, ob es sich um eine Innen- oder Außenkante handelt, und funktioniert so, weil die IDs unter allen Kanten eindeutig sind und es somit keine Innen- und Außenkanten mit derselben ID gibt. Obwohl es für die Aktualisierung von NaechsteZeitschritte nicht unbedingt notwendig wäre, wird hier tatsächlich über alle Zuginstanzen auf dem Knoten iteriert, da dies für die Aktualisierung von *zanz* erforderlich ist. Auch das Iterieren über alle Instanzen für die Aktualisierung von NaechsteZeitschritte bringt keine Nachteile, da in NaechsteZeitschritte Mengen von Knoten, Innen- und Außenkanten für die einzelnen *epdt* gespeichert sind. Bei erneutem Hinzufügen passiert daher nichts, und das jeweilige Element ist trotzdem nur einmal in der Menge vorhanden.

Anschließend wird in derselben Schleife auch *zanz* aktualisiert, was die Anzahl der Züge auf dem Knoten angibt. Hierbei handelt es sich tatsächlich um Züge und nicht um Zuginstanzen, da dieses Attribut nur in der Funktion testeKanten verwendet wird, welche mit Zügen arbeitet. Der *zanz*-Zähler (der zu Beginn der Funktion in Zeile 3 auf 0 zurückgesetzt wurde) wird nur dann um 1 erhöht, wenn die Zug-ID noch nicht

gezählt wurde. Eine Menge (Z, welche zu Beginn in Zeile 2 leer initialisiert wurde) wird verwendet, um zu speichern, welche Zug-IDs bereits gezählt wurden. Die if-Abfrage in Zeile 10 überprüft für jede Zuginstanz, ob der Zug, zu dem sie gehört, bereits gezählt wurde. Somit sind am Ende der Funktion NaechsteZeitschritte und *zanz* für den entsprechenden Knoten aktualisiert. Die Funktion KnotenAktualisieren wird für jeden Knoten aufgerufen, der von einer neuen Zuginstanz betreten wurde, um sicherzustellen, dass alle Knoten nach diesem Zeitschritt wieder die richtigen Werte für *zanz* und haben.

Nachdem ein Knoten von der Funktion „berechneKnoten“ geschedult wurde, ändert sich nicht nur die freie Kapazität des Knotens, sondern auch die der Kanten, von denen die Zuginstanzen kamen. Da der Knoten nun weniger freie Plätze hat, haben die Kanten wiederum mehr freie Plätze. Deshalb muss auch der Wert von *zanz* für die vorherigen Kanten aktualisiert werden. Die korrekten Werte in *zanz* sind wichtig, da diese in der Funktion „testeKanten“ verwendet werden, um zu überprüfen, ob die Kanten für den nächsten Schritt berechnet werden können.

Daher ruft „berechneKnoten“ auch die Funktion „KantenAktualisieren“ auf, jedoch für die Kanten, von denen Zuginstanzen entfernt wurden. Wie „KantenAktualisieren“ (9 und 8) genau funktioniert, wird im Folgenden erklärt. Bei der Betrachtung des vorliegenden Pseudocodes könnte die Annahme entstehen, die Aktualisierung von NaechsteZeitschritte sei in diesem Fall unnötig, da nur Zuginstanzen entfernt wurden und keine neuen hinzugekommen sind, weshalb es auch keine neuen Einträge für NaechsteZeitschritte geben sollte. Im Code für die Simulation werden jedoch auch an dieser Stelle Einträge hinzugefügt, die sicherstellen, dass das entsprechende Element nach Ablauf der Blocking Time (die in 2.2.2 kurz vorgestellt wurde) erneut geschedult wird und die Blocking Time eingehalten wird. Im Pseudocode wurde jedoch das gesamte Konzept der Blocking Time aus Gründen der Einfachheit weggelassen, insbesondere weil es die Änderungen die für die Parallelisierung notwendig sind nicht beeinflusst.

Algorithm 8 InnenKantenAktualisieren($cid \in \mathbb{N}, kid \in \mathbb{N}$)

```
1: let (kid, cid, ∗, zkid, ∗, ZID, zanz) ∈ InnenKanten
2: for each (id, zid, zkid, epdt) ∈ ZugInstanzen mit id ∈ ZID do
3:     let (cid, epdt, KID, ∗, ∗) ∈ NaechsteZeitschritte
4:     KID = KID ∪ zkid
5:     Z = ∅
6:     zanz = 0
7:     if zid ∉ Z then
8:         Z = Z ∪ zid
9:         zanz = zanz + 1
```

Die Funktion KantenAktualisieren ist in zwei Unterfunktionen unterteilt, nämlich InnenKantenAktualisieren und AußenKantenAktualisieren. Der Grund dafür ist, dass Innenkanten einen Eintrag mit derselben Cluster-ID wie ihre eigene *cid* in NaechsteZeitschritte hinzufügen. Außenkanten fügen jedoch einen Eintrag für das benachbarte Cluster hinzu. Daher müssen sie die Zielcluster-ID (*zcid*) als *cid* in NaechsteZeitschritte schreiben, wie in Zeile 1 und 3 von Algorithmus 9 zu sehen ist. Algorithmus 8 verwendet an dieser Stelle einfach seine eigene *cid*. Ansonsten folgen diese

Algorithm 9 AußenKantenAktualisieren($cid \in \mathbb{N}, kid \in \mathbb{N}$)

1: let $(kid, cid, zcid, *, zkid, *, ZID, zanz) \in AussenKanten$
2: **for each** $(id, zid, zkid, epdt) \in ZugInstanzen$ mit $id \in ZID$ **do**
3: let $(zcid, epdt, KID, *, *) \in NaechsteZeitschritte$
4: $KID = KID \cup zkid$
5: $Z = \emptyset$
6: $zanz = 0$
7: **if** $zid \notin Z$ **then**
8: $Z = Z \cup zid$
9: $zanz = zanz + 1$

beiden Funktionen im Wesentlichen dem gleichen Vorgehen wie KnotenAktualisieren. Knoten werden also der *kid* in NaechsteZeitschritte zum Zeitpunkt *epdt* hinzugefügt und anschließend wie oben beschrieben *zanz* aktualisiert.

Kapitel 4

Beispielausführung der Simulation

Im folgenden soll die Simulation anhand eines kleinen Beispiels durchgespielt werden, um ein besseres Verständnis für den Prozess zu erlangen. Hierfür wird das Schienennetz in Abbildung 4.1 verwendet, welches durch den Fahrplan mit den beiden Zügen aus Abbildung 4.2 und 4.3 befahren wird.

4.1 Einlesen von Graph und Fahrplan

Zunächst muss das Schienennetz in zwei Cluster unterteilt werden. Aufgrund der geringen Anzahl von nur zwei Knoten in diesem Beispiel ist die Zuordnung eindeutig, wobei der Knoten 0 in Cluster 0 und der Knoten 1 in Cluster 1 liegt. Eine Veranschaulichung der Cluster-Aufteilung ist in Abbildung 4.4 dargestellt.

Als erste Schritte werden der Graph und der Fahrplan eingelesen. Die Graph-Datei enthält Attribute für alle Knoten und Kanten, einschließlich des Clusters, dem sie zugeordnet sind. Dadurch ist es möglich, direkt zu erkennen, welche Elemente zu welchem Cluster gehören. Außerdem wird auch die Kapazität für jedes Element mit eingelesen. In diesem Beispiel haben alle Elemente des Graphen (4.1) die Kapazität 1. Bei dem Einlesen des Fahrplans wird auch die KuerzesterPfadNach Liste, wie bereits zuvor erwähnt, direkt initialisiert. Da der Fahrplan nach dem Graphen eingelesen wird, ist bereits bekannt, welche Elemente im Schienennetz vorhanden sind und welchem Cluster sie zugeordnet sind. Demzufolge kann bei der Einlesung der Fahrzeiten für eine Kante überprüft werden, ob es sich um eine Außenkante handelt, d.h. ob das Startcluster sich von dem Zielcluster unterscheidet. Im verwendeten Beispiel-

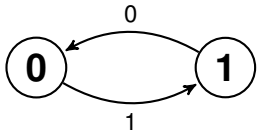

Abbildung 4.1: zwei Knoten Beispiel Netz

Knoten	Ankunftszeit	Abfahrtszeit
0	0.5	1.0
1	5.0	6.0

Abbildung 4.2: Beispiel Fahrplan Zug 0 für das Schienennetz aus 4.1

Knoten	Ankunftszeit	Abfahrtszeit
1	0.0	1.5
0	4.0	4.5

Abbildung 4.3: Beispiel Fahrplan Zug 1 für das Schienennetz aus 4.1

Graphen führen beide Kanten von einem Cluster zu einem anderen, daher handelt es sich bei beiden um Außenkanten. Die Fahrtzeit für Kanten kann durch Subtraktion der Ankunftszeit an einem Knoten von der Abfahrtszeit am vorherigen Knoten berechnet werden.

Beim Einlesen des Fahrplans für Zug 0 (4.2) wird erkannt, dass die Verbindung zwischen Knoten 0 und 1 eine Dauer von 4 Minuten hat. Nach der Feststellung, dass diese Verbindung eine ausgehende Kante ist, wird überprüft, ob es bereits einen Eintrag in der Liste KuerzesterPfadNach gibt. Diese Liste enthält Einträge der Form $(cid, ncid, t) \in \mathbb{N} \times \mathbb{N} \times \mathbb{R}$ und wird verwendet, um zu überprüfen, ob Knoten bereits berechnet wurden. Wenn es einen Eintrag mit $cid = 1$ und ncid = 0 gibt, wird überprüft, ob die neu entdeckte Verbindung von 4 Minuten kürzer ist als der bereits vorhandene Eintrag t. Wenn dies der Fall ist, wird t aktualisiert, da die schnellste Verbindung benötigt wird, um sicherzustellen, dass kein Zug schneller das Cluster wecheln kann als es der Wert in der Liste angibt. Da es sich um die erste eingelesene Verbindung handelt, gibt es noch keinen Eintrag, daher wird ein neuer Eintrag erstellt: $(1, 0, 4) \in$ KuerzesterPfadNach.

Anschließend wird auch ein Eintrag für die Rückrichtung gefunden, nämlich (0, 1, 2.5) $\in$ KuerzesterPfadNach. In diesem Beispiel gibt es nur eine Verbindung in beide Richtungen, daher muss für keine der Verbindungen ein Minimum ermittelt werden. Die beiden Einträge enthalten automatisch die minimal benötigte Zeit zwischen den Clustern.

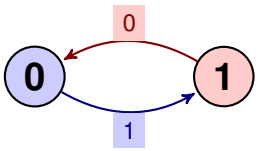

Abbildung 4.4: Beispiel aus 2 Knoten Cluster Aufteilung

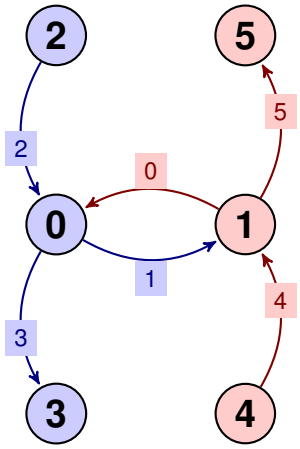

Abbildung 4.5: Graph aus 4.4 mit hinzugefügten Start und Ziel Knoten

4.2 Initialisierungsfunktion

Des Weiteren gibt es eine Initialisierungsfunktion, die beispielsweise dafür sorgt, dass NaechsteZeitschritte mit den im Fahrplan festgelegten Werten initialisiert wird. Dabei wird nur der erste Halt jedes Zuges in die Liste aufgenommen und die Liste wird im Laufe der Simulation ergänzt, da Verzögerungen auftreten können.

Die Funktion „initialize“ führt eine weitere Aufgabe aus, die bisher nicht erwähnt wurde, da sie keinen Einfluss auf die Simulation selbst hat. Jedes Cluster erhält einen „imaginären“ Start- und Zielknoten, die eine unbegrenzte Kapazität aufweisen und als Zugdepot oder Zugparkplätze fungieren. Das bedeutet, dass alle Züge von Anfang an existieren, auch wenn sie zu Beginn noch nicht abfahren. Um zu vermeiden, dass alle Züge von Anfang an auf ihrem Startknoten stehen und dabei das Element blockieren, auf dem sie sich befinden, warten sie im Startknoten ihres Clusters auf ihren Einsatz. Daher gibt es von diesen Zugdepotknoten eine Kante zu jedem Knoten im Cluster, und die Reisezeit auf diesen Kanten beträgt immer 0, damit die Züge direkt zu ihrem eigentlichen Startknoten gelangen können. Ebenso funktionieren die Endknoten, sowohl die Knoten als auch alle Kanten, die zu ihnen führen, haben eine unbegrenzte Kapazität und die Reisezeit auf den Kanten beträgt 0. Dies ist erforderlich, damit Züge am Ende ihrer Route nicht einfach auf dem letzten Knoten ihrer Route verbleiben.Das Beispielnetzwerk mit diesen Start- und Zielknoten ist in Abbildung 4.5 dargestellt. Knoten 2 ist der Startknoten von Cluster 0 und Knoten 3 der Zielknoten des Clusters. Knoten 4 ist der Startknoten von Cluster 1 und Knoten 5 der Zielknoten des Clusters.

Eine weitere Aufgabe der Funktion initialize besteht darin, die Wahrscheinlichkeitsverteilung bereitzustellen und die ersten Zuginstanzen zu erstellen. In dem Beispiel für dieses Kapitel wird davon ausgegangen, dass alle Züge zu 100% pünktlich losfahren. Das wird so gemacht, da das Vorgehen der Simulation trotzdem zu verstehen ist und die Komplexität dann nicht durch verschiedene Zuginstanzen erhöht wird.

Wie bereits erwähnt, wird die Variable NaechsteZeitschritte initialisiert. Deshalb hat NaechsteZeitschritte beim Start den Inhalt, der in Abbildung 4.6 dar-

cid	t	K	IK	AK	ZugID
1	0.0	0			1
0	0.5	1			0

Abbildung 4.6: NaechsteZeitschritte nach Initialisierung

gestellt ist.

Um den Inhalt von NaechsteZeitschritte leichter zu verstehen, wird er in diesem Kapitel als Tabelle dargestellt (4.8). In der Tupel Notation, sähe das erste Beispiel wie folgt aus: $(1, 0.0, 0, ,)$. Dies bedeutet, dass für die Cluster-ID 1 zum Zeitpunkt 0 nur der Knoten 0 betrachtet werden soll und die Menge der zu betrachtenden Innen- und Außenkanten (IK, AK) leer ist. Um das Verständnis weiter zu erleichtern, wurde in Abbildung 4.6 eine zusätzliche Spalte hinzugefügt, die anzeigt, welche Züge sich zu diesem Zeitpunkt bewegen möchten. In unserem Beispiel möchte sich zu jedem Zeitpunkt nur ein Zug bewegen, weshalb diese Angabe in der Tabelle gut funktioniert.

4.2.1 Durchführung der Simulation

Gemäß Kapitel 2.2.2 werden sogenannte „blockingtimes“ verwendet, um einen Sicherheitsabstand zwischen zwei Zügen zu gewährleisten. Dies bedeutet, dass nachdem ein Zug ein Element verlassen hat, eine gewisse Zeit gewartet werden muss, bevor ein neuer Zug dieses Element betreten darf. Für die Implementierung wurde eine Blocking time von 2 Minuten gewählt. Dies bedeutet, dass nach jedem berechneten Zeitschritt ein neuer Eintrag in der Liste NaechsteZeitschritte hinzugefügt wird, der den Zeitpunkt $t + 2$ und die Elemente enthält, die gerade verlassen wurden. Zum Beispiel, wenn zum Zeitpunkt t=5 der Knoten 1 geplant wird und ein Zug von Kante 1 auf Knoten 1 fährt, wird ein Eintrag für t=7 erstellt, bei dem die Kante 1 betrachtet werden soll.

In der Simulation sind vor allem dann blocking Zeiten relevant, wenn es dazu kommt, dass mehrere Züge hintereinander oder sogar gleichzeitig ein Element betreten wollen. In solchen Situationen ist es erforderlich, dass eine Zuginstanz eine bestimmte Zeit wartet, bevor sie das Element betreten kann, damit die Sicherheit gewährleistet ist. Dies ist oft der Fall, wenn Zuginstanzen Verspätungen haben und sich nicht mehr genau an ihren Fahrplan halten können. Allerdings sollte der Fahrplan so gestaltet sein, dass, wenn alle Züge pünktlich sind, keine gegenseitige Behinderung auftritt. Die Simulation ermöglicht es auch, dies zu ermitteln.

Der Fahrplan, der in unserem Beispiel verwendet wird, wurde so erstellt, dass es ohne Verspätungen keine Probleme gibt und keine Zuginstanzen aufeinander warten müssen. Darüber hinaus wurde entschieden, für die beispielhafte Durchführung keine Initialverspätungen zu verwenden und somit davon auszugehen, dass alle Züge pünktlich losfahren. In diesem Beispiel möchte keine Zuginstanz bei einem Eintrag, der durch eine Blockingzeit begründet ist, eine Bewegung durchführen. Aus diesem Grund werden diese Einträge in der beispielhaften Durchführung der Simulation weggelassen, um Schritte zu vermeiden, bei denen keine Zuginstanz das Element wechselt, da sich ansonsten die Anzahl aller Schritte insgesamt verdoppeln würde.

Die Tabelle mit allen Zeiten, einschließlich der Blocking Zeiten, wird jedoch der Vollständigkeit halber in Abbildung 4.7 aufgeführt, während für die beispielhafte Aus-

führung der Simulation getrennt nach Threads die Tabelle ohne Blocking Zeiten verwendet wird (4.8). In der letzten Spalte von 4.7 wurde bei den Einträgen, die durch die Blocking Time verursacht wurden, anstatt einer ZugID „blocked“ geschrieben, da zu diesem Zeitpunkt kein Zug das Element betreten möchte. Dies ist jedoch nur aufgrund des einfachen Beispiels ohne Verspätungen der Fall. Die Pfeile, die am Rand der Tabelle 4.7 zu sehen sind, zeigen an, nach welchen berechneten Einträgen die entsprechenden "blocked“ Einträge hinzugefügt wurden. Um die Übersichtlichkeit zu erhöhen, wurden die durch das Betreten und Verlassen eines Elements des Zuges 1 verursachten "blocked“ Einträge auf der rechten Seite und die durch Zug 0 verursachten Einträge auf der linken Seite eingetragen. Die Information, welcher Zug einen „blocked“ Eintrag hervorgerufen hat, ist für die Berechnungen nicht relevant. Die Aufteilung der Pfeile erleichtert jedoch die Verfolgung der Pfeile in der Tabelle. Es wird später beim Durchgehen der Simulation für die beiden Threads erklärt, warum Einträge zu t=4.5 und t=6 in Klammern gesetzt sind.

Als nächstes werden die beiden Threads gestartet und jedem wird ein Cluster als Eingabe zugewiesen. Beide Threads rufen dann die Hauptfunktion (1) auf. Zunächst wird der Eintrag mit dem kleinsten t-Wert aus NaechsteZeitschritte gesucht, der der Cluster-ID entspricht, die der jeweilige Thread zugewiesen bekommen hat.

Da es für jedes Cluster in diesem Beispiel nur einen Eintrag gibt, ist dies automatisch der Eintrag mit dem kleinsten t-Wert. Daraufhin wird die Berechnung gestartet. Da die Simulation parallel ausgeführt wird, kann sie nicht immer in derselben Reihenfolge ablaufen. Die Threads können unterschiedlich schnell rechnen, solange sie sich an die „Regeln“ halten, d.h. die Testfunktionen müssen eine positive Rückmeldung geben, bevor weitergerechnet wird. Um zu verdeutlichen, welche Zeitpunkte in diesem Beispiel überhaupt berechnet werden müssen und welche von anderen abhängen, wird eine Tabelle erstellt. In der Tabelle und im Diagramm werden die Einträge, die sich auf Cluster 0 beziehen, blau markiert und die Einträge, die sich auf Cluster 1 beziehen, rot markiert. Welches Cluster gemeint ist, wird jedoch auch in der ersten Spalte (cid) der Tabelle 4.8 angegeben. In der Tabelle steht t für den zu berechnenden Zeitpunkt, K für die Menge an Knoten, IK für die Menge an Innenkanten und AK für die Menge an Außenkanten, die vom Cluster *cid* zum Zeitpunkt t berechnet werden sollen. Die Zug-ID gibt die ID des Zuges an, der das entsprechende Element betreten möchte. Die # Spalte ist nur zur Nummerierung, sodass Zeilenangaben bezüglich der Tabelle besser erkennbar sind. Die Pfeile links und rechts von der Tabelle 4.8 zeigen die Abhängigkeiten zwischen den Einträgen an. Die Abhängigkeiten sind dadurch begründet, dass die Einträge in der Liste immer dann hinzugefügt werden, wenn ein Zug ein neues Element betritt, welches er zu einem bestimmten Zeitpunkt wieder verlassen möchte (dieser wird in die Liste geschrieben). Die Einträge für einen Zug werden also nach und nach hinzugefügt, wenn sich der Zug über die Elemente bewegt. Aus diesem Grund sind die Einträge, die denselben Zug betreffen, voneinander abhängig und können nur in der Reihenfolge hinzugefügt und abgearbeitet werden, in der der Zug die betroffenen Elemente abfährt. Die Pfeile auf der rechten Seite markieren die Abhängigkeiten der Einträge von Zug 1, während die Pfeile auf der linken Seite die Abhängigkeiten der Einträge von Zug 0 markieren.

Es ist wichtig zu beachten, dass die vorliegende Tabelle nun chronologisch sortiert ist, jedoch muss dies nicht zwangsläufig der Fall sein, wenn die Simulation durchgeführt wird. Insbesondere weil beide Threads gleichzeitig rechnen. Es gibt jedoch nicht unendlich viele mögliche Reihenfolgen für das Scheduling von Knoten und Kanten zu bestimmten Zeitpunkten, da bestimmte Einschränkungen vorhanden sind. Zur Er-

#	cid	t	K	IK	AK	ZugID
1	1	0.0	1			1
2	0	0.5	0			0
3	0	1			1	0
4	1	1.5			0	1
5	0	2		4		blocked
6	0	2.5		2		blocked
7	0	3	0			blocked
8	1	3.5	1			blocked
9	0	4	0			1
10	0	4.5		3		1
11	0	(4.5)	3			1
12	1	5	1			0
13	1	6			0	blocked
14	1	6		5		0
15	1	(6)	5			0
16	0	6.5	0			blocked
17	1	7			1	blocked
18	1	8	1			blocked

Abbildung 4.7: Zeitlicher Ablauf der Simulation mit den blockingtime Einträgen

#	cid	t	K	IK	AK	ZugID
1	1	0.0	1			1
2	0	0.5	0			0
3	0	1			1	0
4	1	1.5			0	1
5	0	4	0			1
6	0	4.5		3		1
7	0	(4.5)	3			1
8	1	5	1			0
9	1	6		5		0
10	1	(6)	5			0

Abbildung 4.8: Zeitlicher Ablauf der Simulation

klärung dieser Einschränkungen und zur Darstellung der verfügbaren Möglichkeiten werden die beiden Cluster separat betrachtet. Es wird aufgezeigt, welche Abhängigkeiten bestehen und wie die Funktionen „testeKnoten“ und „testeKanten“ sicherstellen, dass auf ein benachbartes Cluster gewartet werden muss, bevor fortgefahren wird.

4.2.2 Ausführung Thread 0

In diesem Abschnitt wird die Simulation schrittweise ausgeführt, so wie es Thread 0 in der Simulation tut. Die Nummerierung bezieht sich darauf, welcher Eintrag des Clusters 0 in Tabelle 4.8 gemeint ist.

1. Um den ersten Eintrag des Clusters C0 mit t=0.5 zu berechnen, muss sichergestellt werden, dass C0 nicht weiter als 2,5 Minuten im Vergleich zu C1 vorausrechnet. Das liegt daran, dass ein Knoten berechnet werden soll und dies die Bedingung der Testfunktion „testeKnoten“ ist, die KuerzesterPfadNach benutzt(und der Eintrag aus KuerzesterPfadNach, der hier gebraucht wird, gibt eine Zeit von 2,5 Minuten vor. Siehe Erläuterung von KuerzesterPfadNach in Abschnitt 4.2). Da die Simulation hier jedoch noch ganz am Anfang ist, kann C1 nicht weiter als 0,5 Minuten zurückliegen (also noch bei dem Startzeitpunkt t=0 sein), was bedeutet, dass die Differenz zwischen den beiden Clustern nicht größer als 2,5 Minuten ist. Somit darf C0 seinen ersten Eintrag berechnen. Nachdem der Knoten berechnet wurde, wird ein weiterer Eintrag in NaechsteZeitschritte eingefügt. Dieser Eintrag bedeutet, dass Zug 0 den Knoten 0 zum Zeitpunkt t=1 wieder verlassen möchte, um Kante 1 zu betreten. Daher wird der entsprechende Eintrag, wie in der dritten Zeile von Abbildung 4.8 zu

sehen ist, zu NaechsteZeitschritte hinzugefügt. In diesem Fall ist dies auch der nächste Eintrag, den C0 bearbeitet und ausführt, da es zu diesem Zeitpunkt keine weiteren Einträge für C0 gibt und es daher automatisch der Eintrag mit dem kleinsten t für C0 ist.

2. Beim zweiten Eintrag möchte C0 eine Außenkante zum Zeitpunkt t=1 berechnen. Es ist jedoch notwendig, dass zuerst die Funktion testeKnoten eine positive Rückmeldung gibt. Wie auch im ersten Schritt ist das Argument hier, dass C1 nicht weiter zurückliegen kann als noch beim Zeitpunkt t=0 zu sein, was bedeutet, dass die Differenz zwischen beiden Clustern nicht größer als 2,5 Minuten sein kann, wenn C0 t=1 berechnen möchte. Daher gibt die Funktion testeKnoten bereits einTrue zurück. Für den zweiten Eintrag soll Cluster C0 eine Außenkante zum Zeitpunkt t=1 berechnen. Zunächst muss jedoch die Funktion testeKnoten positiv ausfallen, was bedeutet, dass C1 nicht weiter zurückliegen darf als zum Zeitpunkt t=0, was eine Differenz von höchstens 2,5 Minuten bedeutet, da C0 erst zu t=1 berechnen will. Folglich gibt die Funktion testeKnoten bereitsTrue zurück. Damit C0 jedoch tatsächlich eine Außenkante berechnen kann, muss auch die Funktion testeKantenTrue zurückgeben. Der erste Fall der Funktion überprüft, ob genügend freie Kapazität auf der Kante vorhanden ist, damit alle Zuginstanzen, die die Kante betreten möchten, dies auch tun können. Das passiert natürlich für alle Außenkanten eines Clusters, aber in diesem Beispiel hat C0 nur eine Außenkante, daher ist es auch die einzige, die überprüft wird. In der aktuellen Situation ist auf der Kante noch keine Zuginstanz vorhanden, da C0 sie selbst auf die Kante geschickt haben müsste, was jedoch nicht der Fall ist. Die Kante hat eine Kapazität von 1 und genau eine Zuginstanz möchte die Kante betreten, daher fällt der Test direkt nach Überprüfung des ersten Falls positiv aus. Somit kann auch die Kante in jedem Fall direkt berechnet werden. Nachdem dieser Knoten berechnet wurde, wird ein neuer Eintrag in die Liste NaechsteZeitschritte hinzugefügt. Dieser Eintrag, der in Zeile 8 der Abbildung 4.8 zu sehen ist, besagt, dass Zug 0 (der gerade die Kante 1 betreten hat) die Kante zum Zeitpunkt t=5 wieder verlassen will, um Knoten 1 zu erreichen. C1 wird sich darum kümmern müssen.

3. Der nächste Eintrag, der Cluster 0 zugeordnet ist, tritt zum Zeitpunkt t=4 auf. Dieser Eintrag hängt von der Berechnung von C1 bis zum Zeitpunkt t=1.5 ab, wie anhand der Pfeile ersichtlich ist. Ohne diese Berechnung würde der Eintrag noch nicht existieren. In diesem Beispiel hat C0 dann keine weiteren Einträge in der Liste. Der nächste sichtbare Eintrag, t=4.5, hängt ebenfalls vom vorherigen Eintrag ab, t=4. Daher kann an dieser Stelle angenommen werden, dass C1 bereits bis t=1.5 berechnet hat. C0 würde vorher einfach warten, weil es für C0 nichts zu tun gibt. Allerdings hat C1 noch Einträge in der Liste NaechsteZeitschritte, sodass diese nicht leer ist und die while-Schleife in der Main-Funktion für C0 weiterhin nicht beendet wird. C0 möchte nun einen Knoten berechnen, darf aber wie bereits erwähnt nicht mehr als 2,5 Minuten in die Zukunft voraus berechnen. Da C1 jedoch bereits bis t=1.5 berechnet hat und es eine kleinergleich-Abfrage ist, ist es erlaubt, weiter zu rechnen.

4. Der nächste Zeitpunkt, zu dem C0 eine Berechnung durchführen soll, ist wie bereits erwähnt t=4.5. Dieser Eintrag wurde auch gerade bei der letzten Berechnung von C0 zu NaechsteZeitschritte hinzugefügt. Es handelt sich um

eine Innenkante, die zum Endknoten des Clusters führt, also dem Knoten im Cluster, an dem Züge ankommen, die ihre Route zu Ende gefahren sind. Bevor die Berechnung durchgeführt werden kann, muss zunächst die Funktion TesteKnoten aufgerufen werden, da Knoten immer zuerst geplant werden müssen. Tatsächlich gibt diese Funktion FALSE zurück, wenn C1 noch nicht bis zum Zeitpunkt t=1.5 gerechnet hat, da der Abstand zwischen den beiden Clustern dann größer als 2.5 Minuten wäre. Das bedeutet, dass C1 noch die Chance hätte, einen Zug auf die Kante 0 zu schicken, die zu C0 führt, und dass dieser Zug vor dem Zeitpunkt 4.5 ankommt. Daher darf die Kante noch nicht berechnet werden. Wenn C1 jedoch bereits bis mindestens t=2 gerechnet hätte (der Zeitpunkt, zu dem die Berechnung stattfinden soll, minus KUERZESTERPFADNACH aus diesem Nachbarcluster: 4.5 - 2.5 = 2), würde der Test positiv ausfallen und die Berechnung fortgesetzt werden können. Bei der Betrachtung der ganzen Tabelle kann festgestellt werden, dass C1 bereits bis t=5 gerechnet haben muss, da dies der nächsthöhere Eintrag von C1 ist. Wenn das also schon geschehen ist und C1 mindestens bei t=5 ist, gibt die Funktion TesteKnoten TRUE zurück. Da hier nur eine Innenkante und keine Außenkante berechnen werden soll, ist das ausreichend und die Kante kann dann einfach berechnet werden. Wie bereits in Abschnitt 4.2 beschrieben, haben die Kanten zu den Zugspeicherknoten für alle Züge eine Reisezeit von 0 Minuten. Dies führt dazu, dass für den Knoten 3 und den Zeitpunkt t=6 ein neuer Eintrag in NAECHSTEZEITSCHRITTE erstellt wird. Da bei jedem Zeitschritt zuerst alle Knoten und dann alle Kanten berechnet werden, ist die Berechnung für den aktuellen Zeitschritt (t=6) für C0 bereits abgeschlossen. Dies bedeutet, dass der Knoten 3 einfach bei der nächsten Gelegenheit berechnet wird. In der Tabelle 4.8 sieht es jetzt so aus, als gäbe es für C0 keinen Eintrag mehr, was bedeutet, dass kein Zug mehr erwartet wird, der sich noch in C0 bewegen möchte. Jedoch werden an dieser Stelle die blocking times benötigt, damit die Züge von der letzten Kante noch in den Endknoten gelangen können. Daher wird dieser Knoten nicht zum Zeitpunkt t=4.5 berechnet, sondern zum Zeitpunkt t=6.5, da dies der nächste Zeitpunkt ist, zu dem C0 durch die blocking times noch einen Eintrag in seiner Liste hat (siehe Tabelle 4.7, Zeile 16).

Für diese letzten Knoten ist es kein Problem, wenn sie nicht zu einem bestimmten Zeitpunkt berechnet werden, da sowohl die Kanten als auch die Knoten eine unbegrenzte Kapazität haben und hier kein Fahrplan mehr eingehalten werden muss. Das Ziel ist lediglich, dass die Züge nicht einfach auf dem letzten Knoten, der in ihrem Fahrplan steht, verbleiben.

5. Wie bereits erläutert, wird als nächstes der Knoten 3 zum Zeitpunkt 6.5 berechnet, da die blocking times diese Verzögerung verursachen. Da schon für die Berechnung von Kante 3, C1 seinen Zeitschritt t=5 berechnet haben muss und keine Differenz von mehr als 2.5 zwischen den Zeitpunkten 5 und 6.5 besteht, kann dies direkt geschehen. Daraufhin ist C0 endgültig mit all seinen Berechnungen fertig.

Nachdem C0 die oben stehenden Berechnungen abgeschlossen hat, hat es keine Einträge mehr in der Liste NAECHSTEZEITSCHRITTE. Trotzdem wird die while-Schleife in der main-Funktion erst beendet, wenn auch C1 keine Einträge mehr hat. Dies liegt daran, dass nicht ausgeschlossen werden kann, dass C1 noch einen Zug zum Cluster C0 schickt. Die Tabelle, die den vollständigen Zeitverlauf der Simulation

zeigt, zeigt zwar, dass auch keine weiteren Berechnungen mehr für C0 dazu kommen, ist aber erst nach einem vollständigen Durchlauf der Simulation verfügbar.

4.2.3 Ausführung Thread 1

In diesem Abschnitt wird die Simulation schrittweise für Cluster 1 ausgeführt, analog zum vorherigen Abschnitt. Die Nummerierung bezieht sich dabei darauf, welcher Eintrag des Clusters 1 in Tabelle 4.8 gemeint ist.

1. Auch für C1 wurde bei der Initialisierung ein Eintrag in NaechsteZeitschritte hinzugefügt, der von Anfang an vorhanden ist (siehe 4.6). Es handelt sich um t=0.0 und Knoten 1, der von Zug 1 als erstes betreten werden soll. Wie bei den Berechnungen für C0 muss auch hier zuerst die Funktion „testKnoten“ bestanden werden. Diese überprüft, ob die Berechnung des Zeitschrittes gegenüber eines Nachbarclusters zeitlich zu weit voraus geht. Wie in 4.1 bei der Beschreibung von KuerzesterPfadNach erwähnt, darf die Zeit, die vorab berechnet wird, in diese Richtung maximal 4 Minuten betragen. Da in diesem Fall der Zeitpunkt t=0 berechnet werden soll und kein Cluster eine negative Zeit aufweisen kann, wird die Bedingung der Funktion „testKnoten“ automatisch erfüllt (es wird keinem Cluster gegenüber zu weit in die Zukunft berechnet und erst recht nicht mehr als 4 Minuten). Da es sich um einen Knoten handelt, ist „testKnoten“ die einzige Funktion, die erfüllt sein muss, und dieser Zeitschritt kann problemlos berechnet werden. Nach dieser Berechnung wird ein neuer Eintrag in NaechsteZeitschritte hinzugefügt. Dies ist der Eintrag in Zeile 4 von 4.8, da dies der Zeitpunkt t=1.5 ist, zu dem Zug 1 den Knoten 1, den er gerade betreten hat, verlassen möchte, um zur Kante 0 zu wechseln.

2. Der soeben hinzugefügte Eintrag in der Tabelle entspricht dem neuen Zeitschritt, der von Thread C1 berechnet werden soll. Falls Thread C0 bereits die ersten beiden Einträge in Zeile 2 und 3 in Abbildung 4.8 berechnet hat, existiert auch bereits der Eintrag in Zeile 8 für den Zeitpunkt t=5. Da jedoch immer der Eintrag mit dem kleinsten t-Wert als nächstes betrachtet wird und 1,5 kleiner als 5 ist, wird der Eintrag in Zeile 4 (also t=1,5) zuerst berechnet.

   Wie üblich muss auch hier zunächst die Testfunktion „testKnoten“ ein positives Ergebnis liefern. Da es hier jedoch um den Zeitpunkt t=1,5 geht und wie bereits erwähnt vier Minuten im Vergleich zu Nachbarclustern voraus gerechnet werden darf, gibt es hier kein Problem. Daher wird „testKnoten“ in jedem Fall True zurückgeben.

   Da Kante 0 eine Außenkante ist, muss auch die Testfunktion „testeKanten“ True zurückgeben bevor der Zeitschritt berechnet werden darf. Dazu wird zunächst überprüft, ob ausreichend freie Kapazität für alle Zuginstanzen vorhanden ist, die die Kante betreten möchten. Wie in der Ausführung von Thread 0, ist dies auch hier der Fall, da unter keinen Umständen bereits ein Zug auf dieser Kante sein kann. Thread C1 hat keinen Zug auf diese Kante geschickt, da es nur zwei Züge gibt und der andere Zug zum Zeitpunkt t=1,5 definitiv auf Kante 1 ist.

   Daher kann Kante 0 problemlos berechnet werden. Dadurch wird auch ein neuer Eintrag in NaechsteZeitschritte geschrieben, nämlich der Eintrag in Zeile 5 in Abbildung 4.8. Dieser Eintrag entspricht dem Zeitschritt 4, da zu diesem

Zeitpunkt Zug 1 die Kante 0, auf der er sich gerade befindet, verlassen wird, um Knoten 0 zu erreichen.

3. Der nächste zu berechnende Zeitschritt für C1 ist t=5. Jedoch muss der Eintrag für diesen Zeitschritt (4.8, Zeile 8) von C0 in die Liste NaechsteZeitschritte geschrieben werden, da der Zug 0 von C0 nach C1 fahren soll. Dieser Eintrag wird von C0 in die Liste geschrieben, nachdem C0 die Außenkante 1 zum Zeitpunkt t=1 berechnet hat. Wenn C0 also noch nicht bis t=1 gerechnet hat, wird es für C1 vorerst keinen Eintrag in NaechsteZeitschritte geben und C1 muss warten. Wenn C0 bis t=1 gerechnet hat, kann C1 mit der Berechnung von t=5 fortfahren.

   Dazu muss die Funktion testeKnotenTrue zurückgeben. C1 möchte t=5 berechnen und C0 hat mindestens bis t=1 gerechnet, so dass die maximale Differenz 4 Minuten beträgt. Das ist genau der Wert des Eintrags aus KuerzesterPfadNach und da es ein kleiner gleich ist, gibt testeKnoten hierTrue zurück. Da nur ein Knoten berechnet werden soll, muss testeKanten nicht angewendet werden.

   Wie zuvor wird nach der Berechnung ein neuer Eintrag in NaechsteZeitschritte geschrieben. Der Zeitpunkt t=6 wird hinzugefügt, zu dem der Zug 0 den Knoten 0 verlassen möchte, den er gerade betreten hat, um auf die Kante 5 zu wechseln, die zum Zugspeicherknoten führt, welcher der Zielknoten des Clusters ist.

4. Als nächstes soll dann also genau dieser Eintrag berechnet werden, da es auch keinen anderen Eintrag für C1 gibt. Da die Kante 5 eine Innenkante ist, reicht es aus, wenn die Funktion testeKnotenTrue zurückgibt. In dem letzten Schritt wurde festgestellt, dass C0 mindestens bis t=1 berechnet haben muss, damit es den zuletzt berechneten und den aktuellen Eintrag für C1 geben kann. Wenn C0 immer noch bei t=1 ist, kann C1 den Zeitschritt t=6 nicht berechnen, da die Differenz größer als 4 Minuten ist, und die Funktion testeKnoten False zurückgibt. Daher muss C1 warten, bis C0 mindestens bis t=2 gerechnet hat, was genau eine Differenz von 4 Minuten bedeutet. Da der nächste Zeitschritt, der größer als t=2 ist, den C0 berechnet, t=4 ist, ist die Bedingung für die Funktion testeKanten erst erfüllt, wenn C0 seinen Zeitschritt t=4 berechnet hat. Der Grund dafür ist, dass C0 vorher noch die Möglichkeit hat, einen Zug auf die Außenkante nach C1 zu schicken, der vor t=6 dort ankommen kann und somit zu diesem Zeitpunkt berücksichtigt werden müsste. Wenn C0 also seinen Zeitschritt t=4 berechnet hat, kann auch C1 seinen Zeitschritt t=6 berechnen, da die Funktion testeKnoten jetztTrue zurückgibt und die Funktion testeKanten nur für Außenkanten genutzt werden muss, was in diesem Fall also nicht erforderlich ist. In diesem Schritt wird erneut ein neuer Eintrag in NaechsteZeitschritte erstellt, und zwar für den Endknoten 5 von Cluster C1 zum Zeitpunkt 6. Da die Berechnung des Clusters C1 für den Zeitpunkt 6 bereits abgeschlossen ist, wird der Knoten 5 bei der nächsten Gelegenheit mit berechnet. Dies geschieht zum Zeitpunkt 7, wie in Zeile 17 von Abbildung 4.7 dargestellt.

5. Zum Zeitpunkt 7 soll also der Knoten 5 berechnet werden. Da bereits sichergestellt wurde, dass das Cluster C1 bis zum Zeitpunkt 4 berechnet wurde, kann die Berechnung direkt durchgeführt werden, da die Differenz zwischen 7 und 4 nicht größer als 4 Minuten ist.

Da nun auch Cluster C1 alle seine Berechnungen abgeschlossen hat, ist die Liste NaechsteZeitschritte tatsächlich vollständig leer, da keiner der beiden Cluster noch Einträge hat. Folglich verlassen beide Cluster in diesem Moment die While-Schleife in der Hauptfunktion und die beiden Threads werden wieder mit dem Hauptthread zusammengeführt. Nach der Simulation werden Statistiken und Zeitmessungen ausgegeben sowie eine Datei, die das Ergebnis der Simulation darstellt und angibt, welche Züge mit welcher Wahrscheinlichkeit und Verspätung enden. Anschließend wird die Simulation beendet.

Kapitel 5

Experimentelle Ergebnisse

In diesem Kapitel geht es um die Umsetzung der zuvor präsentierten Idee und den daraus resultierenden Ergebnissen.

Die ursprüngliche Implementierung der Simulation wurde in C++ geschrieben. Um die Parallelisierung der Simulation umzusetzen, wurde Multithreading verwendet. Hierbei wurde die Thread-Bibliothek ab C++11 genutzt, welche das Erstellen und Nutzen von Threads unterstützt. Außerdem bietet sie Synchronisationsmechanismen wie Mutexe, Bedingungsvariablen und atomare Variablen an, um Raceconditions und Deadlocks zu vermeiden. Durch die Verwendung dieser Mechanismen kann der Zugriff auf gemeinsam genutzte Ressourcen sicher und effektiv erfolgen, weshalb sie auch in der parallelisierten Version der Simulation genutzt wurden.

Im Gegensatz zur originalen Version, die die Simulation sequenziell ausführt, nutzt die parallelisierte Version C++ Multithreading, um die Simulation auf zwei Threads auszuführen.

Die parallelisierte Implementation funktioniert aktuell nur für kleine Beispiele, wie zum Beispiel den Graphen aus Abbildung 5.1 und 3 Züge, die über dieses Schienennetz fahren. Es lag zeitlich leider nicht mehr im Rahmen dieser Arbeit auch größere Beispiele zu testen. Daher kann nicht anhand von Experimentellen Ergenissen beurteilt werden, ob die parallele Ausführung der Simulation für große Schienennetze einen zeitlichen Vorteil bietet.

5.1 Ermittelte Ergebnisse

Die Originale Version der Simulation führt die Berechnungen sequenziell aus und benötigt für das Beispiel aus Abbildung 5.1 und den Fahrplänen aus 5.2, 5.3 und 5.4 im Release-Modus nach Compiler-Optimierung nur 1 ms. Die parallelisierte Version benötigt für dasselbe Beispiel und mit einer Aufteilung in Clustern, wie es anhand der eingefärbten Knoten in Abbildung 5.1 zu sehen ist, jedoch 7 ms.

5.2 Interpretation der Ergebnisse

Das Ergebnis, dass die parallele Ausführung durch Einteilung in Cluster nur unter bestimmten Voraussetzungen effizient funktionieren kann, ist nachvollziehbar. Wie

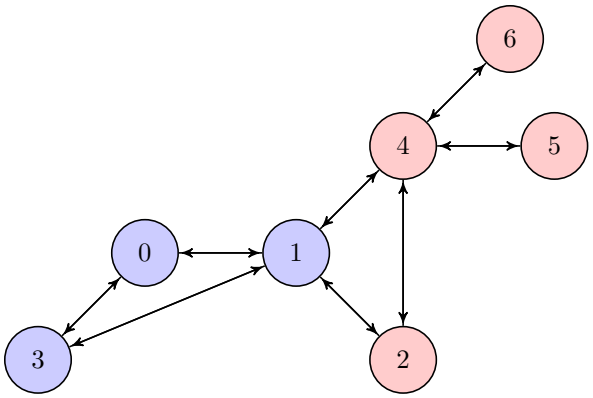

Abbildung 5.1: kleines Beispiel Netz

Knoten	Ankunftszeit	Abfahrtszeit
0	0.0	0.5
1	3.5	4.0
2	7.0	7.5

Abbildung 5.2: Beispiel Fahrplan Zug 0 für das Schienennetz aus 5.1

Knoten	Ankunftszeit	Abfahrtszeit
3	0.0	0.5
1	4.1	4.6
4	6.0	6.5
5	8.0	8.5

Abbildung 5.3: Beispiel Fahrplan Zug 1 für das Schienennetz aus 5.1

Knoten	Ankunftszeit	Abfahrtszeit
6	0.0	0.5
4	5.4	5.9
2	7.6	8.1

Abbildung 5.4: Beispiel Fahrplan Zug 2 für das Schienennetz aus 5.1

im Kapitel 2.3 beschrieben, ist die notwendige Synchronisation, wenn Kanten zwischen Clustern verlaufen, zunächst mit einem Overhead verbunden. Daher sollten die Cluster so eingeteilt werden, dass möglichst wenige Kanten zwischen ihnen verlaufen.

Wenn ein Schienennetz mit mehreren hundert bis tausend Knoten und Kanten vorliegt und nur 2 Cluster daraus gebildet werden, dann bedeutet dies, dass vergleichsweise viele Kanten vorhanden sind, an denen nicht synchronisiert werden muss. Denn nur an Außenkanten, die zwischen Clustern verlaufen, muss synchroniesiert werden. Wenn man dann noch darauf achtet, die Cluster sinnvoll einzuteilen, indem sie annähernd gleich groß sind und so wenige Kanten wie möglich die Grenze überschreiten, kann die parallelisierte Version noch effizienter arbeiten.

Daher ist es bei kleinen Beispielen fast immer effizienter, sequenziell vorzugehen, da der Overhead durch die Synchronisation viel zu groß ist. Mit dem Schienennetz aus 5.1 und dem Fahrplan der drei Züge aus 5.2, 5.3 und 5.4 fährt Zug 0 nur 2 Kanten ab, von denen eine eine Außenkante ist, an der synchronisiert werden muss. Bei Zug 1 ist es eine von insgesamt drei Kanten, die abgefahren werden, die eine Außenkante ist, an der synchronisiert werden muss, während Zug 2 zwei Kanten ohne erforderliche Synchronisation abfährt. Das bedeutet, dass an 2 von 7 Kanten synchronisiert werden muss, das ist eine sehr hohe Quote, die bei größeren Schienennetzen und auch Fahrplänenen automatisch kleiner wird.

Des Weiteren wird die Parallelisierung auch dadurch beschleunigt, dass die einzelnen Threads gegenseitig zeitlich vorausrechnen können und nicht zu oft aufeinander warten müssen. Wenn es nur 2 Cluster und somit Threads gibt und einer von ihnen warten muss, wird die Ausführung in diesem Moment bereits fast wieder sequenziell, jedoch mit einem erheblichen Overhead durch die Synchronisation. Das oben beschriebene Beispiel ist nicht gut geeignet um eine zeitliche Verbesserung hervorzurufen. Das liegt daran, dass ein längeres Fahrplan-Beispiel als 9 Minuten besser geeignet wäre, um eine größere gegenseitige Vorlaufzeit für die Berechnungen der einzelnen Cluster zu ermöglichen, da die Simulationszeit insgesamt zu kurz ist, um wesentlich vorauszuberechnen.

Ein weiterer wichtiger Faktor für die parallele Ausführung ist, dass einige Berechnungen nur einmal am Anfang ausgeführt werden müssen. Dazu gehört beispielsweise die Einteilung in Cluster, die möglichst optimal die oben beschriebenen Bedingungen erfüllen sollen. Außerdem werden auch die Kürzeste-Pfad-Nach-Werte zwischen allen Clustern am Anfang einmalig berechnet. Obwohl solche Berechnungen nicht wiederholt werden müssen und die Einteilung in Cluster sogar nur einmal für jedes Schienennetz erfolgt, lohnen sich solche Berechnungen erst bei ausreichend großen Schienennetzen mit großem Fahrplan, sodass die Simulation im Verhältnis mehr Zeit spart als sie für diese zusätzlichen Berechnungen aufwendet.

5.3 Probleme

Die aktuelle Implementierung der Simulation funktioniert nur für kleine Beispiele mit paralleler Ausführung, da unerwartete Probleme aufgetreten sind. In den meisten Fällen waren diese Probleme auf Synchronisationsprobleme zwischen den beiden Threads zurückzuführen. Die Threads müssen an vielen Stellen Informationen miteinander teilen, damit die Simulation korrekt ablaufen kann. Zum Beispiel muss jeder Thread einen Eintrag in der NaechsteZeitschritte-Liste für das andere Cluster schreiben, wenn ein Zug das Cluster wechselt, damit das andere Cluster darüber infor-

miert ist. Wenn ein Thread einen Eintrag für ein anderes Cluster schreibt, muss eine geschützte Zugriffsmethode, also Synchronisation verwendet werden. Andere Informationen, die zwischen den Clustern geteilt werden müssen, sind die zuletzt berechnete Zeit, die boolsche Variable k, die angibt, ob erst Knoten oder auch schon Kanten berechnet wurden, und ob die anderen Cluster noch Einträge in der NaechsteZeitschritte-Liste haben. Diese Zugriffe können einfach durch Mutexe geschützt werden, da es sich um Variablen und Datenstrukturen handelt, die nur für die parallele Version hinzugefügt wurden und in der sequenziellen Implementierung nicht vorhanden waren. Allerdings müssen auch die Zuginstanzen selbst zwischen den Clustern hin und her fahren und somit von einem Element eines Clusters zu einem Element eines anderen Clusters wechseln. Wenn ein Zug ein Element gewechselt hat, muss das zuvor besetzte Element in dem anderen Cluster auch noch blockiert werden durch die Blocking Time. Diese Funktionen werden wie in Kapitel 3 beschrieben durch die Funktionen *berechneKnoten* und *berechneKanten* umgesetzt. Da diese Funktionen bereits in der ursprünglichen Code-Version für die sequenzielle Ausführung verwendet wurden und übernommen sind, war es schwierig, diese Vorgänge für die Verwendung in einer Multithreading-Umgebung anzupassen, ohne die Funktionen vollständig neu zu schreiben.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Das Ziel dieser Arbeit bestand darin, die Geschwindigkeit einer Simulation zu steigern, indem die parallele Ausführung durch Multithreading implementiert wird. Dabei wurde die Simulation aus einer früheren Arbeit [HÁN21] verwendet und das zu simulierende Schienennetz in zwei Cluster aufgeteilt, die von zwei Threads parallel simuliert wurden. Jeder Thread durfte die Nachbarcluster zeitlich überholen, wenn er schneller als diese mit der Berechnung der einzelnen Zeitschritte in seinem Cluster vorankam. Allerdings war es nicht möglich, dass die Cluster uneingeschränkt zeitlich vorausrechnen konnten. Es musste getestet werden, ob es für jedes Cluster überhaupt problemlos möglich war, den nächsten Zeitschritt zu berechnen. Zur Steigerung der Effizienz wurden diese Testfunktionen in Knoten- und Kanten-Tests aufgeteilt, wobei die Kanten-Tests nur für Außenkanten benötigt wurden. Durch diese Aufteilung konnten Knoten und Innenkanten häufig vor den Außenkanten berechnet werden, wodurch die Wartezeit auf die Berechnung aller Elemente des Clusters reduziert wurde.

Es gab verschiedene Möglichkeiten, die Testfunktionen zu implementieren, um eine weitergehende Vorausberechnung zu ermöglichen. Diese führten jedoch zu unterschiedlichem hohem Overhead durch die benötigten Berechnungen. In dieser Arbeit wurden einfache Konstanten für die Knoten-Tests verwendet, um eine schnelle Vorausberechnung ohne Berechnungen während der Laufzeit zu ermöglichen. Bei den Außenkanten-Tests wurden einfache Überprüfungen durchgeführt, ob ausreichend freie Kapazität auf der Kante vorhanden war oder ob sich die Kapazität nicht mehr ändern konnte, da kein Zug mehr vor dem zu berechnenden Zeitpunkt das Ende der Kante erreichen würde. Diese einfachen Tests ermöglichten trotzdem eine gewisse Vorausberechnung. Die weitere Erforschung von Testfunktionen, die eine weitergehende Vorausberechnung ermöglichen, aber mehr Zeit für die Berechnung der Testergebnisse benötigen, wurde als Ausblick in Abschnitt 6.3 dargestellt.

Zur Veranschaulichung des Vorgehens der Simulation wurde Pseudocode verwendet und anhand eines schrittweise durchgegangenen Beispiels erläutert. Experimentelle Ergebnisse wurden ebenfalls präsentiert, wobei nur kleine Beispiele getestet wurden, da unerwartete Probleme bei der Synchronisation aufgetreten sind. Die Ergebnisse

zeigten, dass bei kleinen Beispielen eine sequentielle Ausführung sinnvoller ist, da eine parallele Ausführung einen zu großen Overhead erzeugt. Bei großen Beispielen könnte jedoch eine Zeitersparnis durch parallele Ausführung erzielt werden. Die genaue Auswirkung hängt jedoch von Faktoren wie den gewählten Testfunktionen, der Größe der Eingabegraphen und der Länge der zu simulierenden Zeit ab und muss weiter untersucht werden.

6.2 Diskussion

In den Experimenten wurde festgestellt, dass bei kleinen Schienennetzwerken eine sequentielle Ausführung aufgrund des Overheads bei der parallelen Ausführung sinnvoller ist. Bei großen Schienennetzwerken kann eine parallele Ausführung jedoch zu einem Zeitgewinn führen. Die genaue Höhe dieses Gewinns hängt von Faktoren wie den gewählten Testfunktionen und der Größe der Schienennetzwerke ab. Es gibt auch andere Faktoren, die die Parallelisierung beeinflussen, wie z.B. die Länge der zu simulierenden Zeit und die Zahl der Threads.

Insgesamt zeigt diese Arbeit, dass die parallele Ausführung der Simulation durch Multithreading bei großen Schienennetzwerken eine effektive Möglichkeit darstellen kann, um die Laufzeit zu verbessern. Die Wahl der Testfunktionen spielt dabei eine entscheidende Rolle, und weitere Untersuchungen sind erforderlich, um optimale Testfunktionen zu finden. Zudem müssen in zukünftigen Arbeiten auch andere Faktoren, die die Parallelisierung beeinflussen, berücksichtigt werden, um die Effizienz weiter zu verbessern.

6.3 Ausblick

Um die vermutete Beschleunigung der Simulationszeit bei Verwendung großer Beispiele zu belegen, ist es zunächst sinnvoll, die parallelisierte Version der Simulation für diese Fälle nutzbar zu machen. Danach könnte auf die im folgenden ausgeführten Ideen eingegangen werden.

Ein wichtiger Faktor für die Effizienz der parallelisierten Simulation ist die Anzahl der verwendeten Threads. In dieser Arbeit wurde stets die Verwendung von zwei Threads beibehalten, aber es gibt noch viel Spielraum für weitere Experimente in diesem Bereich. Eine Erhöhung der Anzahl der Threads würde zunächst zu einer schnelleren Berechnung führen. Jedoch hängt dies auch von der Größe des zu simulierenden Schienennetzes ab, da ab einer bestimmten Anzahl an Threads der Overhead zu groß wird und sich eine weitere Erhöhung der Threads nicht mehr lohnt. Dies liegt daran, dass bei allen Kanten zwischen den Threads synchronisiert werden muss und je mehr Threads verwendet werden, in desto mehr Teile muss der Graph zerlegt werden. Dadurch liegen immer mehr Kanten auf den Grenze zwischen den Clustern. Daher kann in weiteren Arbeiten versucht werden herauszufinden, bei welcher Graphengröße welche Anzahl an Threads optimale Resultate bringt.

Wie in 2.3.3 schon teilweise erwähnt, gibt es bei den Testfunktionen auch noch einige Optionen, deren Effekt auf die Geschwindigkeit der Ausführung der Simulation ausgetestet werden kann.

6.3.1 Testfunktionen für Knoten

In dieser Arbeit wurden feste Zeitwerte verwendet, um die Vorrechnungsdauer von einem Cluster zum anderen in Bezug auf seine Knoten zu bestimmen. Der Vorteil dieser Methode war eine schnelle Überprüfung, da diese konstanten Werte bereits beim Einlesen des Fahrplans berechnet wurden und daher während der Simulation nicht mehr berechnet werden mussten, sondern nur noch ausgelesen und verwendet werden konnten. Es sei jedoch darauf hingewiesen, dass diese Zeitwerte die minimale Vorrechnungsdauer darstellen, die als problemlos zu berechnen betrachtet werden kann.

Jedoch unterscheiden sich alle anderen beschriebenen Verfahren grundlegend, da in jedem Fall während der Simulation Zuginstanzen gesucht, Zeiten ausgelesen und berechnet werden müssen. Diese Verfahren haben den Vorteil, dass für längere Zeiträume mit Sicherheit gesagt werden kann, ob eine zeitliche Vorrechnung möglich ist oder nicht. Wenn dies der Fall ist, kann weiter in die Zukunft vorberechnet werden, als mit der in dieser Arbeit präsentierten Methode.
Der Nachteil besteht jedoch darin, dass ein erheblich größerer Rechenaufwand erforderlich ist, der längere Zeit benötigt. Es müsste erst ausprobiert werden, ob der zeitliche Vorteil, seinen Nachbarclustern gegenüber weiter voraus rechnen zu können zu können, den größeren Rechenaufwand überwiegt.

Eine erste Idee zur Verlängerung der vorrechenbaren Zeit wäre, jedem Knoten eine eigene, konstante Vorrechnungszeit zuzuweisen. Eine Voraussetzung dafür ist, dass der Knoten mindestens eine eingehende Kante aus einem benachbarten Cluster hat. Diese Zeiten könnten ebenfalls beim Einlesen des Fahrplans berechnet werden. Es würde dann für jeden Knoten die schnellste Verbindung aus jedem Nachbarcluster gesucht werden, aus dem dieser Knoten direkt erreichbar ist. Wenn ein Knoten also aus einem Nachbarcluster über mehr als eine Kante direkt zu erreichen ist, würde auch bei diesem Vorgehn wieder das Minimum gewählt werden.

Allerdings wäre es dann nicht mehr pauschal möglich, zu überprüfen, wie weit Cluster C1 im Vergleich zu seinem Nachbarcluster C2 vorrechnen darf. Stattdessen müsste immer überprüft werden, welche Knoten bei dem nächsten Zeitschritt berechnet werden sollen. Wenn dann in der Menge der Knoten KID in NaechsteZeitschritte überprüft wurde, welche Knoten überhaupt zu dem aktuell kleinsten t in der Liste berechnet werden sollten, kann jeder dieser Knoten überprüfen, ob er von einem anderen Cluster aus erreichbar ist und wenn ja, in welcher Zeit. Dann kann das Minimum all dieser Zeiten verwendet werden, um zu bestimmen, wie weit voraus gerechnet werden darf.

Dieser Ansatz bietet insbesondere dann einen Vorteil, wenn die schnellste Verbindung (wie in dieser Arbeit verwendet) wesentlich schneller ist als die meisten anderen Verbindungen und auch selten von Zügen genutzt wird. In diesem Fall würden die Knoten oft noch nicht berechnet, obwohl dies bereits möglich wäre. Der vorgestellte Ansatz würde dem entgegenwirken.

Weiterführende Untersuchungen könnten darauf abzielen, festzustellen, ob auf den eingehenden Kanten zum Zeitpunkt t tatsächlich Zuginstanzen vorhanden sind und ob diese ihre *epdt* vor t erreichen. Wenn dies nicht der Fall ist, muss der betreffende Knoten nicht warten und kann bei der Suche nach dem Minimum übersprungen werden, sofern die *epdt* der Zuginstanzen noch nicht erreicht ist. Dieser Ansatz setzt

den vorherigen voraus, da jeder Knoten nach wie vor seine eigene konstante Zeit hat. Im Gegensatz zum vorherigen Ansatz müssen jedoch für alle Kanten, die von einem Cluster zu einem Knoten führen, Zeitwerte gespeichert werden, anstatt nur die minimale Zeit für die Verbindung zu einem Knoten. Dies liegt daran, dass alle Kanten auf das Vorhandensein von Zuginstanzen überprüft werden müssen, die den Knoten erreichen können, wie bereits beschrieben. Nur weil auf der schnellsten Verbindung keine Zuginstanz vorhanden ist, die den Knoten bis zum Zeitpunkt t erreichen kann, bedeutet dies nicht automatisch, dass dies auch für langsamere Verbindungen gilt. Auch bei diesem Ansatz muss erneut untersucht werden, ob der zusätzliche Aufwand durch die erhöhte Anzahl von Berechnungen dem zeitlichen Gewinn durch das weitere Vorrechnen überlegen ist.

6.3.2 Testfunktionen für Kanten

Im Abschnitt, in dem die Testfunktion für Kanten beschrieben wird, wurde bereits erwähnt, dass es nicht notwendig ist, nach der Überprüfung der ersten beiden Fälle (3.3.3) zu warten, bis die zu erwartenden Cluster gebildet werden und die Zeit t erreicht ist.

Eine mögliche Herangehensweise besteht darin, den zweiten Fall zu nutzen, bei dem überprüft wird, ob mindestens eine Zuginstanz auf der Außenkante ihre epdt vor t erreicht hat. Es ist grundsätzlich nicht notwendig, dass das Nachbarcluster bis zum Zeitpunkt t berechnet wird, sondern es ist möglich, dass das Nachbarcluster den Zug schon vor seiner epdt von der Kante nimmt.

Wenn es also nicht genügend freie Kapazität gibt und mehrere Zuginstanzen auf der Kante eine epdt kleiner als t haben, kann es sein, dass nachdem nur eine Zuginstanz vom Zielknoten aufgenommen und somit von der Kante wurde, genügend freie Kapazität für alle Zuginstanzen vorhanden ist, die auf die Kante fahren möchten. In diesem Fall kann die Kante bereits geschedult werden, bevor das Nachbarcluster bis zum Zeitpunkt t berechnet hat.

Im worst-case müsste gewartet werden, bis die Nachbarcluster auch bis zum Zeitpunkt t berechnet haben, bevor weiter fortgeschritten werden kann. In diesem Fall würde die Testfunktion erneut ausgeführt werden, während das in dieser Arbeit vorgestellte Verfahren in der Lage ist, sofort fortzufahren. In allen anderen Fällen, wenn also bereits vor dem Erreichen von t weiter fortgeschritten werden kann, spart dieser Ansatz vermutlich Zeit.

Ein alternativer Ansatz setzt bei Fall 1 der in dieser Arbeit verwendeten Testfunktion an. Hier wird überprüft, ob genügend freie Kapazität auf der Kante vorhanden ist, um allen Zügen, die auf ihr fahren möchten, dies auch zu ermöglichen. In diesem Fall werden die Züge gezählt und nicht die Zuginstanzen. Die Zuginstanzen können immer einem Zug zugeordnet werden und daher gibt es nur einen realen Zug auf dieser Kante, auch wenn sich mehrere Instanzen desselben Zuges in der Simulation auf dieser Kante befinden.

Es ist jedoch auch möglich, dass Zuginstanzen verschiedener Züge sich gegenseitig ausschließen. Dies liegt daran, dass die Zuginstanzen alle an ein Szenario gebunden sind, in dem sie auftreten. Ein Beispiel dafür wäre folgende Situation:

- Zug A fährt zu 50% pünktlich los: Instanz A1

- Zug A fährt zu 50% mit 5 Minuten Verspätung los: Instanz A2
- Zug B fährt zu 100% pünktlich los: Instanz B

Instanz B wurde von A2 unterwegs aufgehalten, wodurch B sich in B1 und B2 aufteilt. B1 wurde von A2 aufgehalten und konnte erst 5 Minuten später weiterfahren, während B2 pünktlich weiterfahren konnte. Da A2 nicht in allen Szenarien existiert und somit nicht in jedem Fall die Instanz B aufhalten kann, iexistiert auch B1 nicht in allen Szenarien.

Wenn die Instanzen A1 und B1 später gleichzeitig die gleiche Außenkante betreten möchten, schließen sie sich gegenseitig aus und müssen nicht als separate Zuginstanzen gezählt werden, wenn es um die Kapazität auf der Kante geht, obwohl sie zu unterschiedlichen Zügen gehören. Da B1 nur existiert, wenn der Zug A mit einer 5-minütigen Verspätung losfährt und B aufhält, existieren die beiden Zuginstanzen A1 und B1 nicht im gleichen Szenario und dürfen beide die Kante betreten, auch wenn nur ein freier Platz vorhanden ist.

Bei der Überprüfung der freien Kapazität für Fall 1 der Testfunktion unter Beachtung der Szenarien, ist es wichtig zu berücksichtigen, ob dieses Vorgehen einen zeitlichen Vorteil bietet oder ob die damit verbundenen Berechnungen einen zu großen Overhead verursachen, der letztendlich zu keiner Beschleunigung führt. Es sollte sorgfältig geprüft werden, ob dieses Vorgehen in der praktischen Anwendung effektiv ist.

Literaturverzeichnis

[BS12] Thorsten Büker and Bernhard Seybold. Stochastic modelling of delay propagation in large networks. *Journal of Rail Transport Planning & Management*, 2(1):34–50, 2012.

[HÁN21] Rebecca Haehn, Erika Ábrahám, and Nils Nießen. Symbolic simulation of railway timetables under consideration of stochastic dependencies. In Alessandro Abate and Andrea Marin, editors, *Quantitative Evaluation of Systems*, pages 257–275, Cham, 2021. Springer International Publishing.

Im Pseudocode verwendete Mengen:

ZugInstanzen: $(id, zid, neid, epdt) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{R}$
Knoten: $(id, cid, kap, ZID, zanz) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times 2^{\mathbb{N}} \times \mathbb{N}$
InnenKanten: $(id, cid, skid, zkid, kap, ZID, zanz) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times 2^{\mathbb{N}} \times \mathbb{N}$
AußenKanten: $(id, acid, zcid, skid, zkid, kap, ZID, zanz) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times 2^{\mathbb{N}} \times \mathbb{N}$
Cluster: $(id, KID, IKID, AKID, t, k, NCID, WCID) \in \mathbb{N} \times 2^{\mathbb{N}} \times 2^{\mathbb{N}} \times 2^{\mathbb{N}} \times \mathbb{R} \times \mathbb{B} \times 2^{\mathbb{N}} \times 2^{\mathbb{N}}$
NächsteZeitschritte: $(id, time, KID, IKID, AKID) \in \mathbb{N} \times \mathbb{R} \times 2^{\mathbb{N}} \times 2^{\mathbb{N}} \times 2^{\mathbb{N}}$
KürzesterPfadNach: $(cid, ncid, t) \in \mathbb{N} \times \mathbb{N} \times \mathbb{R}$

Funktionen und ihre Eingabeparameter sowie der Typ der Ausgabe:

BerechnungProClusterUndTimestep($cid \in \mathbb{N}, time \in \mathbb{R}, sd \in \mathbb{R}$) $\rightarrow$ void
testeKnoten($cid \in \mathbb{N}, time \in \mathbb{R}$) $\rightarrow \mathbb{B}$
testeKanten($cid \in \mathbb{N}, time \in \mathbb{R}$) $\rightarrow \mathbb{B}$
berechneKnoten($cid \in \mathbb{N}, kid \in \mathbb{N}, time \in \mathbb{R}$) $\rightarrow$ void
berechneKante($cid \in \mathbb{N}, kid \in \mathbb{N}, time \in \mathbb{R}$) $\rightarrow$ void
KnotenAktualisieren($cid \in \mathbb{N}, kid \in \mathbb{N}$) $\rightarrow$ void
InnenKantenAktualisieren($cid \in \mathbb{N}, kid \in \mathbb{N}$) $\rightarrow$ void
AußenKantenAktualisieren($cid \in \mathbb{N}, kid \in \mathbb{N}$) $\rightarrow$ void