Universität Duisburg-Essen

Virtueller Weiterbildungsstudiengang Wirtschaftsinformatik (VAWi)

Masterarbeit

Massiv-parallele Implementierung einer Heuristik zur Lösung des Electric Vehicle Scheduling Problems mit CUDA

Massively parallel implementation of a heuristic to the solution of the electric vehicle scheduling problem with CUDA

Vorgelegt der Fakultät für Wirtschaftswissenschaften der Universität Duisburg-Essen

Verfasser: Wolfgang Bongartz
5. Fachsemester
Von-Stauffenberg-Str. 8
53757 Sankt Augustin
Matrikelnummer 2222059

Gutachter: Prof. Dr. Natalia Kliewer
(Freie Universität Berlin)

Abgabe: 29.06.2017
Sommersemester 2017

Inhaltsverzeichnis

Abbildungsverzeichnis

Glossar

Ausrückfahrt

Eine Ausrückfahrt ist eine Leerfahrt, die von einem Depot an die Starthaltestelle einer Servicefahrt führt (Kliewer 2005).

Befehlsregister

(Instruction Register)
Das Befehlsregister ist ein Register des Prozessors, welches den nächsten auszuführenden Befehl enthält (Blume 1991).

Befehlszähler

(Program Counter, Instruction Counter, Programmzähler, Programmschrittzähler)
Der Befehlszähler ist ein Register des Prozessors, das die Speicheradresse des nächsten auszuführenden Befehls enthält. Ist der aktuelle Befehl abgearbeitet, wird der nächste Befehl aus der Speicheradresse ausgelesen, auf die der Befehlszähler verweist und in das Befehlsregister geschrieben. Der Befehlszähler wird inkrementiert, sodass er auf den nächst folgenden Befehl verweist. Der Befehl im Befehlsregister wird dekodiert und ausgeführt und der Zyklus beginnt von Neuem (Blume 1991).

Depot

(Betriebshof)
Ort für das Abstellen der Fahrzeuge eines Verkehrsunternehmens. I.d.R. verfügt ein Verkehrsunternehmen über mehrere Depots. Alle Fahrzeuge starten von einem Depot aus zur ersten Servicefahrt des Betriebstags und sie kehren am Ende des Betriebstags in das Depot zurück, von dem aus sie gestartet sind. Bis zum Beginn des nächsten Betriebstags bleiben sie im Depot geparkt. Ein Depot hat in der Realität eine begrenzte Kapazität an Stellplätzen für die Fahrzeuge (Kliewer 2005).

Einrückfahrt

Eine Einrückfahrt ist eine Leerfahrt, die von der Endhaltestelle einer Servicefahrt zurück ins Depot führt (Kliewer 2005).

Fahrzeugtyp

Die vom Verkehrsunternehmen eingesetzten Fahrzeuge haben bestimmte, für die Umlaufplanung relevante, Eigenschaften (bspw. die max. Passagieranzahl, die Antriebsart und die Bauform). Anhand dieser Eigenschaften sind die Fahrzeuge unterscheidbar. Ein Fahrzeugtyp bestimmt eine Menge dieser Eigenschaften so, dass sich jedes Fahrzeug genau einem Fahrzeugtypen zuordnen lässt. Für EVSP ist die Reichweite eine besonders relevante Eigenschaft (Kliewer 2005).

Fahrzeugtypgruppe

Eine Fahrzeugtypgruppe fasst die Fahrzeugtypen zusammen, die den Anforderungen bestimmter Servicefahrten gerecht werden (bspw. Niederflorbusse, behindertengerechte Busse) (Kliewer 2005). Zwei Fahrzeugtypgruppen sind kompatibel, wenn ihre Schnittmenge nicht leer ist.

Haltestelle

Haltestellen sind „...definierte und eindeutig benannte Orte für den Fahrgastwechsel...“ (Schnieder 2015).

Leerfahrt

(Deadrun)
Eine Leerfahrt wird ohne Fahrgäste durchgeführt und dient dazu, ein Fahrzeug oder dessen Fahrer an einen anderen Ort zu transportieren. Bspw. um einen Fahrerwechsel zu ermöglichen, um das Fahrzeug aufzuladen (bzw. es zu betanken) oder um eine Servicefahrt zu beginnen, die an einer entfernten Starthaltestelle beginnt. Leerfahrten werden unterschieden in Verbindungsfahrten, Einrückfahrten und Ausrückfahrten (Kliewer 2005).

Linie

Eine Linie ist eine regelmäßige Verkehrsverbindung zwischen einer Start- und einer End-Haltestelle, auf der Fahrgäste an bestimmten Haltestellen ein- und aussteigen können (Schnieder 2015). Eine Linie ist also eine Menge von Haltestellen, die von wenigen Abweichungen abgesehen während eines Betriebstages von einem Fahrzeug immer in der selben Reihenfolge abgefahren werden. Sie fasst damit alle Servicefahrten eines Betriebstages zusammen, die auf derselben Strecke verkehren.

Probleminstanz

Der Begriff *Probleminstanz* bezeichnet in der Komplexitätstheorie eine konkrete, vollständige Fragestellung. Im Unterschied zum Begriff *Problem*, der eine allgemeine Fragestellung bezeichnet, die (unendlich) viele Probleminstanzen umfassen kann. Das *Travelling Salesman Problem* ist bspw. ein Problem. Die Frage nach der kürzesten Rundreise für die Städte Köln, München, Hamburg und Berlin ist dagegen eine Probleminstanz des Travelling Salesman Problem.

Servicefahrt

(Service journey, Fahrplanfahrt, Fahrgastfahrt, Linienfahrt, Personenbeförderungsfahrt)
Eine Servicefahrt dient der Beförderung von Fahrgästen. Sie beginnt und endet zu definierten Zeitpunkten an bestimmten Haltestellen und hat einen vorgegebenen Verlauf. Die Start- und die Endhaltestelle können dabei durchaus auch identisch sein. Eine Servicefahrt kann einer Linie zugeordnet sein. Andernfalls handelt es sich meist um eine Sonderfahrt (bspw. eine Schulbusfahrt). Bei der Durchführung einer Servicefahrt wird eine bestimmte Fahrtstrecke zurückgelegt. Zwei Servicefahrten heißen *kompatibel* zueinander, wenn sie von ein und demselben Fahrzeug nacheinander absolviert werden können (ggf. mit zwischengeschalteter Verbindungsfahrt) (Kliewer 2005).

Travelling Salesman Problem

(Problem des Handelsreisenden, Rundreiseproblem)
"'Das TSP ist ein Optimierungsproblem, bei dem durch eine vorgegebene Menge von Orten von einem bestimmten Ausgangsort aus der kürzeste beziehungsweise kostengünstigste Rundreiseweg zu ermitteln ist; dabei gilt es zusätzlich, eine optimale Reihenfolge der Orte zu finden"' (*Brockhaus-Enzyklopädie online* 2014, Schlagwort "'Travelling Salesman Problem"').

Umlauf

(Fahrzeugumlaufplan, Umlaufplan)
Ein Umlauf ist die Fahrtroute, die ein Fahrzeug an einem Betriebstag absolviert. Es handelt sich also um eine Sequenz von Fahrzeugaktivitäten (Servicefahrten, Leerfahrten, Aufladevorgänge, Wartezeiten, etc.) die im Laufe eines Betriebstages von einem Fahrzeug ausgeführt werden. Ein Umlauf beginnt und endet immer in einem Depot (Kliewer 2005)..

Verbindungsfahrt

Eine Verbindungsfahrt ist eine Leerfahrt, die dazu dient, ein Fahrzeug und dessen Fahrer von der Endhaltestelle einer Servicefahrt zur Starthaltestelle einer kompatiblen Servicefahrt zu transportieren. Sie verbindet also gewissermaßen

zwei kompatible Servicefahrten (Kliewer 2005).

Abkürzungen

ACO

Ant Colony Optimization

ALU

Arithmetic Logic Unit

API

Programmierschnittstellen

AVX

Advanced Vector Extensions

CPU

Central Processing Unit

CSV

Comma-separated values

CUDA

Compute Unified Device Architecture

EVSP

Electric Vehicle Scheduling Problem

FPU

Floating Point Unit

GPGPU

General Purpose Computation on Graphics Processing Unit

GPU

Graphics Processing Unit

MDEVSP

Multi Depot Electric Vehicle Scheduling Problem

MDVSP

Multi Depot Vehicle Scheduling Problem

MIMD

Multiple Instruction Multiple Data

MISD

Multiple Instruction Single Data

MMU

Memory Management Unit

OpenCL

Open Computing Language

OpenGL

Open Graphics Library

OR

Operations Research

PC

Personal Computer

SA

Simulated Annealing

SDEVSP

Single Depot Electric Vehicle Scheduling Problem

SDVSP

Single Depot Vehicle Scheduling Problem

SFU

Special Funktion Unit

SIMD

Single Instruction Multiple Data

SIMT

Single Instruction Multiple Thread

SISD

Single Instruction Single Data

SM

Streaming Multiprocessor

SP

Streaming Processor

SSE

Streaming SIMD Extensions

VSP

Vehicle Scheduling Problem

ÖPNV

Öffentlicher Personennahverkehr

1. Einleitung

1.1. Problemumfeld und Motivation

Der Öffentlicher Personennahverkehr (ÖPNV) erlebt in Deutschland seit Jahren einen leichten, aber stetigen Aufwärtstrend. Die Fahrgastzahlen sind auch 2015 im Vergleich zum Vorjahr um 0,5% gestiegen. Dabei ist eine eindeutige Verschiebung zu beobachten: Im ländlichen Bereich nahmen die Fahrgastzahlen eher ab, während sie in Großstädten mit mehr als 500.000 Einwohnern deutlich zulegten (VDV 2016). Diese Verschiebungen scheinen sich insgesamt gut mit der demografischen Entwicklung in Deutschland erklären zu lassen, denn viele Menschen ziehen aus ländlichen Bereichen in die großen Städte (Kröhnert u. a. 2011). Dort wird der ÖPNV wegen des i.d.R. besseren Angebotes oft auch besser angenommen als auf dem Land[1] (Breitinger u. a. 2017).

Es ist zu erwarten, dass der Busverkehr vor allem in Großstädten weiter zunehmen wird. Nun werden die allermeisten der circa 36.000 in Deutschland verkehrenden Busse derzeit mit Diesel betrieben[2]. Gerade die Innenstadtbereiche deutscher Großstädte sind aber in besonderem Maße von verkehrsbedingten Luftschadstoffen wie

[1] Tatsächlich hat die Anzahl der Fahrgäste in Bussen 2015 um 1,3% abgenommen, während sie in Straßenbahnen um 2,4% gestiegen ist. Das ist gut damit erklärbar, dass große Städte meist über ein gut ausgebautes Straßen- bzw. U-Bahnnetz verfügen, was für eine Verlagerung eines Teils des ÖPNV vom Bus auf die Bahn führen dürfte. Ein Großteil des ÖPNV wird jedoch auch in Städten nach wie vor von Linienbussen bestritten (VDV 2016). Der Aufwärtstrend im ÖPNV scheint insgesamt vor allem durch das sich ändernde Mobilitätsverhalten vieler Menschen verursacht zu werden. So ist bspw. die Anzahl junger Menschen, die den Autoführerschein erwerben möchten in Deutschland seit Jahren rückläufig (DPA-Meldung 2014).

[2] Laut VDV (2016) fuhren 2015 in Deutschland nur 3,3% aller Linienbusse mit alternativen Antrieben. Derzeit fahren laut Bruge u. a. (2016) in Europa 88,9% aller Busse mit Diesel und nur 1,2% werden elektrisch betrieben (darin enthalten sind auch Busse, die an Oberleitungen betriebene werden).

Feinstäuben und Stickoxiden betroffen. Viele Städte und Kommunen streben daher den Umbau ihrer Busflotte auf Fahrzeuge mit emissionsarmen Antrieben an[3] (*Luftschadstoffe im Überblick* 2017). Vor allem elektrisch betriebene, mit Akkumulatoren ausgestattete Busse („Batteriebusse", „E-Busse")[4] stehen dabei im Fokus[5].

Dieser Flottenumbau wird wohl frühestens 2030 abgeschlossen sein, denn die typische Nutzungsdauer von Linienbussen liegt bei 10 bis 12 Jahren. Außerdem sind umfangreiche Investitionen in neue Fahrzeuge und die entsprechende Ladeinfrastruktur nötig, was eine besondere Herausforderung darstellt. Denn trotz der positiven Entwicklung der Passagierzahlen ist der ÖPNV meist defizitär[6]. Auch gilt die eingesetzte Technologie derzeit noch als wenig ausgereift[7] (Grah 2017; Becker 2017; Schwarzer 2016; Schwenn 2016).

[3] In naher Zukunft ist zumindest zeitweise mit der Sperrung vieler Innenstädte und Verkehrsschwerpunkte für Fahrzeuge mit Verbrennungsmotoren zu rechnen. Es ist zwar zu vermuten, dass auch Linienbusse zunächst von diesen Beschränkungen ausgenommen sein werden, dennoch dürfte der politische und juristische Druck auf Verkehrsunternehmen weiter zunehmen. Nicht zuletzt durch den sogenannten „Diesel-Skandal", durch das von der EU-Kommission angedrohte Vertragsverletzungsverfahren wegen der in vielen Städten nicht eingehaltenen Stickoxid-Grenzwerte und das 2015 in Paris auf der UN-Klimakonferenz ausgehandelte Klimaschutzabkommen. Schon jetzt streben Umweltverbände vielerorts Gerichtsurteile an, die Städte zu gravierenden Einschränkungen des innerstädtischen Verkehrs zwingen sollen. Auch Gesetzesänderungen sind zu erwarten (Kamann 2016; Becker 2017; AFP-Meldung 2017; Schwenn 2016).

[4] Mit „Batteriebus" bzw. „E-Bus" sind im Rahmen dieser Arbeit elektrisch betriebene Omnibusse gemeint, die ihre Antriebsenergie aus einem mitgeführten Traktionsakkumulator beziehen (siehe https://de.wikipedia.org/wiki/Batteriebus bzw. https://de.wikipedia.org/wiki/Traktionsbatterie).

[5] In vielen Städten wie bspw. in Hamburg und in Berlin laufen entsprechende Pilotprojekte (BMVI 2016). Auch andere Antriebskonzepte wie Brennstoffzellen werden z.T: schon seit Jahren untersucht (Bünnagel 2012).

[6] Im Vergleich zu 2013 ist die Summe der Fehlbeträge im ÖPNV in 2014 um 5,3% auf €3,124 Mrd. gestiegen (VDV 2016). Diese Fehlbeträge müssen i.d.R. von den Städten und Kommunen ausgeglichen werden. Mancherorts führt der Geldmangel zu einem Investitionsstau im ÖPNV(Breitinger u. a. 2017; Breitinger 2017).

[7] Insbesondere die beschränkte Reichweite, die eingeschränkte Zuverlässigkeit und die hohen Betriebskosten der E-Busse gelten als problematisch. Auch mangelt es an etablierten Standards bzgl. der Ladetechnik. Zudem ist die Auswahl an Serienfahrzeugen mit Batteriebetrieb noch gering. Bisher bieten vor allem die Unternehmen BYD (China) und Solaris (Polen) entsprechende Modelle an. Traditionell decken deutsche ÖPNV-Betriebe ihren Fahrzeugbedarf jedoch überwiegend bei deutschen Herstellern wie MAN und Mercedes-Benz. Letzterer hat ein E-Bus-Modell angekündigt, das frühestens 2019 auf den Markt kommen wird (Schwarzer 2016). Trotz aller Bedenken scheint der Betrieb einer reinen E-Bus-Flotte durchaus möglich zu sein: Laut Bruge u. a. (2016) gab es in 2015 weltweit schätzungsweise rund 173.000 E-Busse. Etwa 170.000 davon wurden in China betrieben. Laut Nikowitz (2016, S. 57) verfügt die chinesische 12-Millionen-Einwohner-Stadt Shenzhen über die weltweit größte E-Bus-Flotte. Bis Ende 2017 will die Stadt vollständig auf E-Busse umgestellt haben. Allerdings stammen die dort eingesetzten Busse allesamt vom chinesischen Anbieter BYD. Möglicherweise handelt es sich also um ein Prestige-Projekt. Wenn dem so ist dürfte das Interesse daran, über auftretende Probleme zu berichten, wohl eher gering sein.

Ein wichtiger Faktor für die Wirtschaftlichkeit teurer Investitionsgüter ist die möglichst optimale Planung des Ressourceneinsatzes. Bei Omnibussen machen die Anschaffungs- und Betriebskosten einen Großteil der Kosten aus[8]. Da diese Kosten bei E-Bussen noch höher sind als bei Bussen mit konventionellem Antrieb kommt der optimalen Fahrzeugeinsatzplanung vor dem Hintergrund der beschränkten finanziellen Mittel im ÖPNV eine besondere Bedeutung zu.

Die Fahrzeugeinsatzplanung (Fahrzeugumlaufplanung, Umlaufplanung) stellt einen Planungsschritt innerhalb eines komplexen mehrschrittigen Planungsprozesses des ÖPNV dar. Sie basiert auf einem fest vorgegebenen Fahrplan und weist allen während eines Betriebstages durchzuführenden Personenbeförderungsfahrten ein Fahrzeug zu. Der dabei entstehende Umlaufplan beinhaltet neben den Fahrplanfahrten auch alle nötigen Leerfahrten und alle Standzeiten. Ziel der Umlaufplanung ist es, die Betriebskosten sowie die Anzahl benötigter Fahrzeuge zu minimieren. Zur Erstellung der Umlaufplanung setzen ÖPNV-Betriebe Softwareprodukte ein, die auf etablierten Optimierungsverfahren aus dem Operations Research (OR) beruhen (Kliewer 2005, S. 3 und S. 7 ff.). Das zugrunde liegende Optimierungsproblem ist in der Literatur unter der Bezeichnung *Umlaufplanungsproblem* bzw. *Vehicle Scheduling Problem (VSP)* bekannt (siehe bspw. Suhl u. a. (2013, S. 220 ff.) und Kliewer, Mellouli u. a. (2006)).

Um die relativ geringen Reichweiten der E-Busse und die Verfügbarkeit der benötigten Ladeinfrastruktur angemessen berücksichtigen zu können ist es notwendig, die existierenden Optimierungsverfahren anzupassen bzw. neue Verfahren zu entwickeln. Das hierbei zu lösende Optimierungsproblem ist eine Erweiterung des VSP. Es kann für Instanzen realistischer Größe bislang nicht exakt gelöst werden. Die entsprechenden Forschungsarbeiten subsumieren unter dem Oberbegriff *Electric Vehicle Scheduling Problem (EVSP)* (siehe bspw. Adler (2014) und Kliewer, Reuer u. a. (2015)).

Für die Lösung des EVSP ist eine ausreichend leistungsfähige Computerhardware

[8] In 1995 machte der Personalaufwand noch über 50% der Aufwendungen der Unternehmen im Personenverkehr aus. In 2014 betrug er nur noch rund 36%. Im gleichen Zeitraum stieg der Materialaufwand (Anschaffungs- und Betriebskosten sowie Kosten für Roh-, Hilfs- und Betriebstoffe) von knapp 26% auf über 40% an (VDV 2016)

nötig, damit das Ergebnis innerhalb eines praxistauglichen Zeitraums ermittelt werden kann. Dabei sollte es sich nicht um teure Spezialhardware, sondern nach Möglichkeit um handelsübliche Personal Computer (PC) handeln. Für die Leistungsfähigkeit eines PCs ist insbesondere der Hauptprozessor, die Central Processing Unit (CPU), relevant. Wie im Abschnitt 3.2 erläutert werden wird, können moderne CPU mehrere Teilaufgaben parallel zueinander bearbeiten, um eine Gesamtaufgabe schneller lösen zu können. Es liegt also nahe, diese Fähigkeit auch für das Lösen des EVSP auszunutzen.

Nun hat die Entwicklung immer aufwändigerer 3D-Computerspiele seit 1991 dazu geführt, dass heute zahlreiche sehr leistungsfähige Grafikkarten für den Massenmarkt verfügbar sind. Diese Grafikkarten müssen als eigenständige Computersysteme mit eigenem Prozessor und eigenem Arbeitsspeicher angesehen werden. Sie sind auf das hochgradig parallele Ausführen bestimmter Operationen für die 3D-Grafikdarstellung spezialisiert. Ein solcher Prozessor, der als *Graphics Processing Unit (GPU)* bezeichnet wird, verfügt dazu über eine große Anzahl von Prozessorkernen und ist (bezogen auf seine Spezialisierung) wesentlich leistungsfähiger als eine CPU vergleichbaren Kaufpreises[9]. Seit 2008 existieren komfortable Programmierschnittstellen[10], mit denen sich diese Rechenleistung auch für andere Anwendungen nutzen lässt[11]. Da in einem PC durchaus auch mehrere Grafikkarten gleichzeitig betrieben werden können lässt sich mit relativ geringem finanziellem Aufwand ein Par-

[9] GPUs mit weit über Tausend Kernen sind heute zu Verkaufspreisen von deutlich unter €500,– verfügbar. Zwar ist die Taktfrequenz einer GPU geringer als die einer CPU und ihre Kerne sind auch nicht so vielseitig einsetzbar, aber die Anzahl der Kerne ist bei einer GPU oft um zwei Größenordnungen höher. Und mit der Anzahl der Kerne steigt die Anzahl der gleichzeitig ausführbaren Threads. Bspw. kann die CPU „Intel Core i7-6700K“ auf 4 Kernen bis zu 8 Threads gleichzeitig ausführen (mit einer Taktrate von etwa 4 GHz). Die Prozessoren einer Grafikkarte mit dem Chipsatz „Nvidia GeForce GTX 1060“ schafft bei etwa gleich hohem Kaufpreis die gleichzeitige Ausführung von bis zu 1.280 Threads (bei einer Taktrate von etwa 1,5 GHz). Die meisten der heute auf dem Markt verfügbaren Computerspiele sind ohne eine solch leistungsfähige Grafikkarte (also nur auf der CPU) nicht spielbar.

[10] Application Programming Interfaces (API)

[11] Auf diese Weise lassen sich bspw. bestimmte Matrixoperationen bei geschickter Programmierung leicht um den Faktor 500 beschleunigen. Solch gravierende Beschleunigungen sind in praxisrelevanten Anwendungen allerdings nicht erreichbar. Eine Beschleunigung um den Faktor 100 ist für manche Anwendung aber realistisch. So ist es möglich, komplexe physikalische Simulationen, für die früher ein Supercomputer nötig war nun ausreichend schnell auf einem PC ausführen zu lassen. In Form eines Programmierbeispiels für CUDA ist u.a. eine Partikelsimulation verfügbar, die unter https://developer.nvidia.com/accelerated-computing-toolkit abgerufen werden kann.

allelrechner von enormer Leistungsfähigkeit aufbauen[12]. Der interne Aufbau und die Arbeitsweise der Grafikhardware müssen bei der Programmierung allerdings berücksichtigt werden. Dies ist der Spezialisierung einer Grafikkarte auf 3D-Berechnungen geschuldet und führt zu speziellen Anforderungen, die nicht immer erfüllt werden können.

1.2. Zielsetzung und Vorgehen

Ziel der vorliegenden Masterarbeit ist es, ein heuristisches Lösungsverfahren für das Multi Depot Electric Vehicle Scheduling Problem (MDEVSP) so zu implementieren, dass es massiv-parallel auf der GPU einer modernen Grafikkarte ablaufen kann. Die Arbeit soll dazu beitragen die Frage zu klären, ob die speziellen Anforderungen an die Programmierung einer GPU für diesen Anwendungsfall erfüllt werden können[13], sodass eine optimale Lösung in deutlich geringerer Laufzeit ermittelt werden kann als bei ausschließlicher Benutzung der CPU.

Es sollen dazu zwei auf den Metaheuristiken *Ant Colony Optimization (ACO)*[14] und *Simulated Annealing (SA)*[15] basierende Lösungsverfahren konzipiert und implementiert werden. Das Ergebnis soll eine unter Windows lauffähige Konsolenanwendung sein, die in der Lage ist, eine in Form eines Textfiles vorliegende Problemdefinition *(Probleminstanz)* eines MDEVSP einzulesen und dann wahlweise auf der GPU oder der CPU zu lösen[16]. Dabei sollen alle auf der GPU bzw. auf der CPU

[12] Für den professionellen und den wissenschaftlichen Einsatz sind Grafikkarten lieferbar, deren Leistungsfähigkeit noch vor wenigen Jahren nur von Supercomputern in großen Rechenzentren erreicht wurde. Eines der Topmodelle des Grafikkartenherstellers Nvidia kostet derzeit einen mittleren vierstelligen Eurobetrag, kann 3.840 Threads gleichzeitig ausführen und schafft damit eine Rechenleistung von bis zu 12 TFlops(siehe http://www.nvidia.com/object/quadro-graphics-with-pascal.html). Auch in Supercomputern finden sich spezialisierte GPUs. Der derzeit mit rund 17,5 Petaflops leistungsfähigste derartige Supercomputer stammt von der Firma Cray und trägt den Namen „Titan“. Er steht aktuell auf Rang 3 der Top500-Liste der Supercomputer (siehe https://www.olcf.ornl.gov/titan/ und https://de.wikipedia.org/wiki/Titan_(Computer)).

[13] Siehe Unterabschnitt 3.3.8.

[14] Siehe Unterabschnitt 3.1.2.

[15] Siehe Unterabschnitt 3.1.1.

[16] Leider reichte die für diese Masterarbeit verfügbare Bearbeitungszeit nicht mehr aus, um den auf SA basierenden Lösungsansatz auch auf der GPU zu implementieren. Er steht daher nur für die CPU zur Verfügung. Die durch die GPU-Implementierung der ACO gewonnenen Erkenntnisse lassen sich aus Sicht des Autors allerdings auch auf SA übertragen.

verfügbaren Kerne ausgenutzt werden.

1.3. Aufbau der Arbeit

Diese Arbeit ist in sieben Kapitel unterteilt. Die Kapitel vier, fünf und sechs befassen sich mit der Konzeption und der Implementierung der Konsolenanwendung sowie mit der Auswertung der erzeugten Lösungen und machen damit den inhaltlichen Kern dieser Arbeit aus.

Das Kapitel 2 gibt einen Überblick über die Planungsaufgabe und stellt deren Zusammenhang zur operativen Planung im ÖPNV her. In Kapitel 3 wird auf die verwendeten Metaheuristiken und die für die Arbeit relevanten technischen Aspekte der Parallelverarbeitung und der GPU-Programmierung eingegangen. Kapitel 4 stellt die beiden in der Arbeit verfolgten Lösungsansätze vor und beschreibt, wie diese parallelisiert werden können. Die Konzeption und die Implementierung der Konsolenanwendung ist Gegenstand des Kapitels 5. Das Kapitel 6 befasst sich mit den von der Anwendung produzierten Daten und deren Auswertung. Das Kapitel 7 fasst die Ergebnisse der Arbeit schließlich zusammen und gibt einen kurzen Ausblick auf Fragestellungen, die aufbauend auf dieser Arbeit behandelt werden könnten.

2. Electric Vehicle Scheduling Problem - Problembeschreibung und Stand der Forschung

Sowohl die Aufbau- als auch die Ablauforganisation eines mit der Fahrgastbeförderung durch Busse im ÖPNV tätigen Verkehrsunternehmens wird üblicherweise in mehreren Schritten erstellt[1]. Die strategische Planung befasst sich mit der Aufbauorganisation, während die Ablauforganisation Gegenstand der operativen Planung ist.

Schritte der strategischen Planung:

1. **Bedarfsermittlung**

   Ermittlung des voraussichtlichen Beförderungsbefarfs.

2. **Angebotsplanung**

3. **Netzplanung**

   Festlegung der Haltestellen mit dem Ziel, möglichst kurze Reisezeiten zu erreichen. Definition des Streckennetzes.

Schritte der operativen Planung:

1. **Linienplanung**

[1] Die Ausführungen treffen natürlich auch auf die Fahrgastbeförderung mit anderen Verkehrsmitteln zu.

Festlegung des Linienverlaufs auf Basis der Bedarfsermittlung und dem Streckennetz, um eine optimale Integration aller Fahrten in das Gesamtverkehrssystem zu erreichen.

2. **Fahrplanerstellung**

   Festlegung der Taktfrequenzen und Abfahrtszeiten für alle Linien.

3. **Fahrzeugumlaufplanung**

   Planung des Fahrzeugeinsatzes.

4. **Personaleinsatzplanung**

   Planung der Tagesdienste (Schichtplan) und der Dienstreihenfolge (Dienstplan)

Die einzelnen Schritte der strategischen und auch die der operativen Planung sind komplex und ohne computergestützte Informations- und Entscheidungsunterstützungssysteme nicht in ausreichender Qualität durchführbar. Um einen kostenoptimalen Gesamtplan zu erstellen, der alle Nebenbedingungen berücksichtigt, müssten alle Planungsschritte gleichzeitig gelöst werden. Aufgrund der hohen Komplexität ist dies jedoch nicht möglich. Daher erfolgt eine Optimierung auf Ebene der einzelnen Planungsschritte. Der Planungsschritt, welcher den Schwerpunkt der vorliegenden Arbeit bildet ist die Fahrzeugumlaufplanung (Umlaufplanung), bei der festgelegt wird, mit welchen konkreten Fahrzeugen die im Rahmen der vorhergehenden Planungsschritte festgelegten Fahrten zur Personenbeförderung durchgeführt werden sollen (Kliewer 2005, Abschnitte 2.2.3 und 2.3). Das mit der Umlaufplanung verfolgte Hauptziel ist in der Regel, die Betriebs- und Investitionskosten des Fahrzeugeinsatzes zu minimieren[2]. Der Umlaufplan bildet wiederum die Grundlage der Personaleinsatzplanung.

[2]Dabei sind in der Praxis oft noch zahlreiche Nebenbedingungen zu erfüllen, die im Rahmen der vorliegenden Arbeit jedoch größtenteils nicht berücksichtigt werden. So steht die angestrebte Kostenminimierung im Widerspruch zur ebenfalls angestrebten Maximierung der Verfügbarkeit des Angebots. Denn jedes zusätzliche Fahrzeug erhöht natürlich sowohl die Verfügbarkeit als auch die Kosten. Bei der Planung ist also u.a. auch zu berücksichtigen, dass Fahrzeuge durch Wartungsarbeiten oder technische Defekte ausfallen werden. Auch das Vorsehen von Pufferzeiten bei Leerfahrten ist sinnvoll, damit der Plan möglichst robust gegen Verspätungen ist.

Die Erstellung eines optimalen Umlaufplans ist ein Optimierungsproblem, das in der Literatur als Vehicle Scheduling Problem (VSP) bezeichnet wird. Bedingt durch die Größe des Lösungsraums ist es mit den derzeit verfügbaren Computern nicht in akzeptabler Zeit exakt lösbar[3]. Daher kommen heuristische Lösungsverfahren zum Einsatz. Da neben der Anzahl der täglich durchzuführenden Servicefahrten vor allem die Anzahl der Depots maßgeblichen Einfluss auf die Problemkomplexität hat wird weiter zwischen *Single Depot Vehicle Scheduling Problem (SDVSP)* und *Multi Depot Vehicle Scheduling Problem (MDVSP)* unterschieden (Kliewer 2005; Kliewer, Mellouli u. a. 2006; Suhl u. a. 2013).

Der Einsatz von E-Bussen stellt bedingt durch ihre eingeschränkte Reichweite besondere Anforderungen an die Umlaufplanung. Denn während die üblicherweise mit Diesel betriebenen Omnibusse eine so große Reichweite haben, dass sie während eines Betriebstages nur selten neu betankt werden müssen, ist die Reichweite von E-Bussen so gering, dass Aufladungen während eines Betriebstags meist nötig sind. Diese Aufladungen müssen daher berücksichtigt werden, was die Komplexität des zu lösenden Optimierungsproblems erhöht[4]. Diese Problemstellung wird in der Literatur als *Electric Vehicle Scheduling Problem (EVSP)* bezeichnet und wie das VSP weiter in *Single Depot Electric Vehicle Scheduling Problem (SDEVSP)* und *Multi Depot Electric Vehicle Scheduling Problem (MDEVSP)* unterschieden. Mit der Lösung von MDEVSP mithilfe heuristischer Methoden beschäftigen sich u.a. die Forschungsarbeiten Kliewer, Reuer u. a. (2015), Adler (2014) und Kooten Niekerk u. a. (2017).

[3]Zumindest dann nicht, wenn es sich um Umlaufplanungen realistischer Größenordnung handelt.
[4]Ein weiteres, in dieser Arbeit jedoch nicht betrachtetes Problem ist die optimale Standortwahl für die Ladestationen. Die Einrichtung einer Ladestation ist überaus kostspielig, weshalb ihre Anzahl auf das für den reibungslosen Betriebsablauf unbedingt nötige Maß begrenzt sein sollte.

3. Methodische und technische Grundlagen

3.1. Metaheuristiken

Das im Rahmen der vorliegenden Arbeit implementierte System verwendet Metaheuristiken, um Lösungen für das EVSP zu finden. Laut Brockhaus (2005) ist die Heuristik die „Lehre vom methodischen Erkenntnisgewinn mithilfe von Denkmodellen, Analogien und Gedankenexperimenten“. Anders als in der Logik, kommt es in der Heuristik nicht auf die formale Beweisbarkeit neuer Erkenntnisse an. In der Informatik bezeichnet der Begriff „Heuristik“ laut Engesser u. a. (1988) Verfahren zur Lösung komplexer Probleme, die auf Hypothesen und Vermutungen aufbauen und die mit höherer Wahrscheinlichkeit (jedoch ohne Garantie) das Auffinden einer besseren Lösung beschleunigen.

Im OR werden unter dem Begriff „Heuristik“ Verfahren zur Lösung komplexer Optimierungsprobleme verstanden, mit denen durch ein systematisches Vorgehen bei vertretbarem Rechenaufwand möglichst gute Lösungen auch dann gefunden werden können, wenn exakte Lösungsverfahren entweder nicht existieren oder zu aufwändig sind (Werners 2013). Im Gegensatz zu exakten Verfahren kann eine Heuristik also auch NP-schwere Optimierungsprobleme in akzeptabler Zeit lösen. Sie führt jedoch meist nicht zur mathematisch optimalen Lösung, sondern nähert diese nur an. Auch ist es im allgemeinen nicht möglich zu beurteilen, wie groß der Abstand der gefundenen Lösung zur mathematisch optimalen Lösung ist (Domschke u. a. 2007; Ellinger

u. a. 2003; Garey u. a. 1979)[1].

Heuristiken sind i.d.R. problemspezifisch. Sie nutzen also bekannte Eigenschaften eines bestimmten zu lösenden Problems aus. Dagegen beschreiben Metaheuristiken „allgemeine Prinzipien und Schemata zur Entwicklung und Steuerung von heuristischer Verfahren". Metaheuristiken beinhalten eine abstrakte Komponente und erlauben es deshalb, mit einem Algorithmus Lösungen für verschiedene Problemarten zu finden (Bogon 2013, S. 25). Die im Rahmen dieser Arbeit verwendeten und im Folgenden näher beschriebenen Metaheuristiken *Simulated Annealing (SA)* und *Ant Colony Optimization (ACO)* gehören den sogenannten naturanalogen Verfahren an, welche versuchen, bestimmte in der Natur beobachtbare Phänomene nachzubilden (Suhl u. a. 2013, S. 13 und S. 137).

3.1.1. Simulated Annealing

Die Grundidee des SA entstammt der Metallurgie. Bei der Metallgewinnung aus Erz wird flüssiges Metall schnell abgekühlt. In einem abkühlenden, flüssigen Metall sind die Atome bestrebt, ein möglichst niedriges Energieniveau einzunehmen. Die Atome des Metalls ordnen sich daher in Gitterstrukturen an (kristalline Anordnung). Kühlt das Metall zu schnell ab, findet allerdings keine optimale Kristallisation statt und das Endprodukt hat ggf. nicht die gewünschten Materialeigenschaften. Aus physikalischer Sicht sind die Atome im Metall dann teilweise in lokalen Energieminima verblieben. Im weiteren Verarbeitungsprozess wird das Metall deshalb erneut erhitzt und dann langsam abgekühlt, um den Atomen das Auffinden ihrer optimalen Position zu ermöglichen. Diesen Vorgang bezeichnet man im deutschen als *Tempern* und im englischen als *Annealing*.

Die Wahrscheinlichkeit dafür, dass ein Atom von einem niedrigeren Energieniveau E in ein höheres Energieniveau $E + \Delta E$ wechselt lässt sich mit der Boltzmann-Funktion[2] berechnen, in der T für die Temperatur und k für die Boltzmannkonstan-

[1] Allerdings kann für viele Heuristiken eine Worst-case-Abschätzung getroffen werden.
[2] Diese Funktion geht auf den Begründer der theoretischen Thermodynamik, Ludwig Boltsmann zurück.

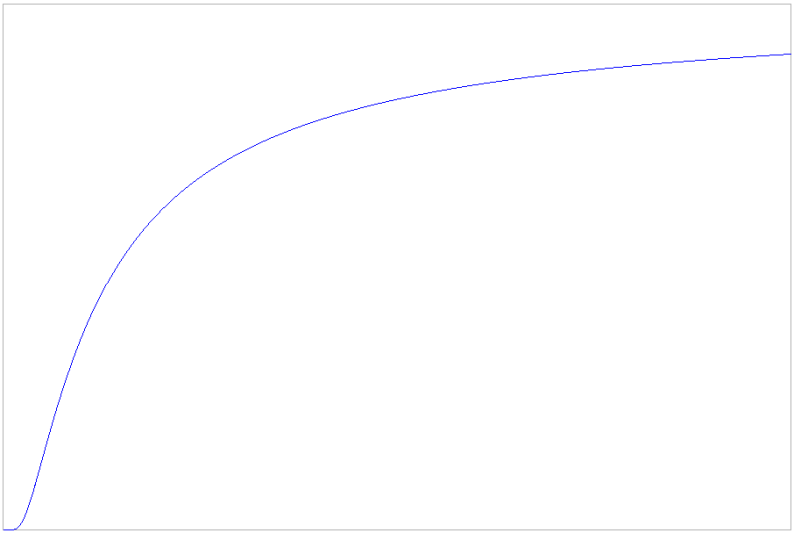


Abb. 3.1: Plot der Boltzmann-Funktion

te[3] steht:

$$p = e^{\frac{-\Delta E}{kT}}$$

Diese Funktion sagt aus, dass ein Atom ein niedriges Energieniveau mit einer gewissen Wahrscheinlichkeit zugunsten eines höheren E. aufgeben und somit ein lokales Energieminimum wieder verlassen kann[4]. Bei niedrigen Temperaturen ist diese Wahrscheinlichkeit aber fast gleich 0. Die Abbildung 3.1 zeigt einen Plot des Funktionsgrafen. Die Temperatur T ist auf der x-Achse aufgetragen; die Wahrscheinlichkeit p auf der y-Achse. Offenbar steigt die Wahrscheinlichkeit also mit steigender Temperatur an. Diesen Umstand macht man sich beim Tempern zunutze, indem man das Material langsam abkühlt. Zunächst wird ein Atom oft in lokalen Energieminima „landen“, die es aufgrund der hohen Temperatur sehr wahrscheinlich auch wieder verlassen wird. Es hat also eine gewisse Chance, durch den Umweg über ein höheres Energieniveau ein niedrigeres E. einzunehmen und so ggf. auch das globale Optimum zu erreichen.

Das SA ist nun eine Metaheuristik, welches das physikalische Prinzip des Temperns auf Optimierungsprobleme überträgt, weshalb es auch den *naturanalogen Verfahren*

[3]Bei der Boltzmannkonstante handelt es sich um eine Naturkonstante.
[4]Die physikalischen Zusammenhänge sind tatsächlich weitaus komplexer als in diesem Abschnitt dargestellt.

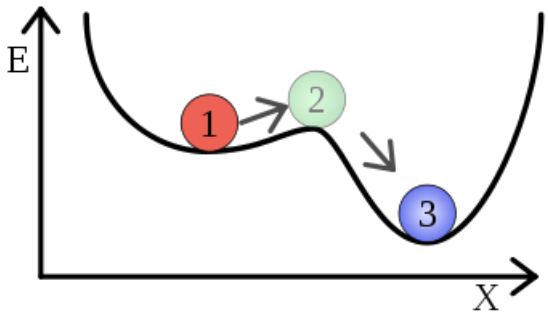


Abb. 3.2: Übergang eines Atoms auf ein niedrigeres Energieniveau

Das Energieniveau des Atoms (1) befindet sich in einem lokalen Minimum. Erst der „Umweg“ über ein höheres Energieniveau (2) führt ins globale Minimum (3). Weil das Atom auch in Zustand (1) bereits ein stabiles Gleichgewicht erreicht hat bedarf es dazu eines Anstoßes in Form von Wärmeenergie.

Bildquelle: ()Wikibooks, Dr. Georg Wiora, https://de.wikibooks.org/wiki/Werkstoffkunde_Metall/_Innerer_Aufbau/_Struktur)

zugerechnet wird[5]. Der prinzipielle Ablauf des SA für ein Minimierungsproblem ist wie folgt:

1. Definiere die Menge der möglichen Lösungen (den Lösungsraum)
2. Definiere eine Energiefunktion, mit der sich das „Energieniveau“ E (also die Güte) einer Lösung berechnen lässt
3. Setze die aktuelle Temperatur T auf den Anfangswert und lege die Mindesttemperatur fest, bei welcher das Verfahren beendet werden soll
4. Wähle per Zufall aus dem Lösungsraum eine Lösung s aus und berechne ihr Energieniveau $E(s)$
5. Wähle per Zufall aus dem Lösungsraum eine „benachbarte“ Lösung s' aus und berechne ihr Energieniveau $E(s')$
6. Berechne $\Delta E = E(s) - E(s')$
7. Falls $\Delta E > 0$ ist: Ersetze s durch s'

[5]Naturanaloge Verfahren sind der Natur nachempfunden, während nicht-naturanaloge Verfahren wie bspw. Tabu-Search ihren Ursprung eher in Mathematik und Logik haben (Bogon 2013, S. 28).

8. Falls $\Delta E <= 0$ ist: Ersetze s durch s' mit der Wahrscheinlichkeit $p = e^{\frac{-\Delta E}{T}}$

9. Reduziere die aktuelle Temperatur, falls die Mindesttemperatur noch nicht erreicht ist

10. Ist das Terminierungskriterium noch nicht erfüllt, so fahre mit 5. fort

Dieser Ablauf lässt sich ohne weiteres für ein Maximierungsproblem adaptieren. Bei geschickter Wahl der Anfangs- sowie der Mindesttemperatur, dem Abkühlungsschema und bei passender Energiefunktion sollte s nun eine gute oder sogar die optimale Lösung enthalten. Wie bei allen Heuristiken gibt es dafür jedoch keine Garantie. In der Literatur wird als Voraussetzung für das Funktionieren des SA eine topologische Struktur des Lösungsraums genannt. Das bedeutet, dass entscheidbar sein muss, ob zwei Lösungen „benachbart“ (im Sinne von „einander ähnlich“) sind oder nicht (Klüver u. a. 2012, S. 87-92).

3.1.2. Ant Colony Optimization

Die Ant Colony Optimization (ACO) ist, wie SA, eine naturanaloge Metaheuristik. Sie wurde von der naturwissenschaftlichen Arbeit von Goss u. a. (1989) inspiriert, in der das Schwarmverhalten der Argentinischen Ameise beschrieben wird und die eine Erklärung für die sogenannte *Schwarmintelligenz* von Insektenschwärmen liefert. Insektenschwärme sind nämlich zu beeindruckenden Optimierungsleistungen fähig, obwohl das einzelne Insekt nur auf im Grunde recht primitive Reizmuster reagieren kann und über keinerlei Intelligenz verfügt. Ameisenkolonien sind bspw. in der Lage, den kürzesten Weg zu einer kilometerweit von ihrem Nest entfernten Futterquelle zu finden. Eine einzelne Ameise kann dies nicht bewerkstelligen.

Wie sich herausstellte sondern Ameisen bei der Futtersuche Pheromone[6] ab, um den Rückweg zum Nest finden zu können. Hat eine Ameise nun Futter gefunden, sondert sie beim Transport zurück ins Nest besonders viel Pheromon ab. Der so

[6] Pheromone sind laut Brockhaus (2005) „...von Tieren (...) produzierte hochwirksame Substanzen, die Stoffwechsel und Verhalten anderer Individuen der gleichen Art in kleinsten Konzentrationen beeinflussen“.

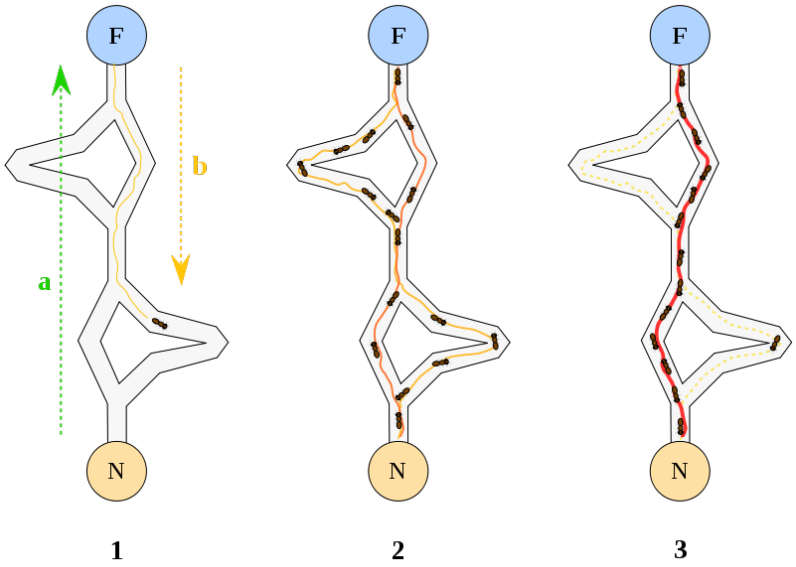


Abb. 3.3: Verhalten einer Ameisenkolonie bei der Futtersuche

(1) Eine einzelne Ameise hat eine Futterquelle entdeckt und sondert deshalb auf dem Rückweg zum Nest ein Pheromon ab. (2) Andere Ameisen bewegen sich auf dem Weg zur Futterquelle eher zufällig, wählen tendenziell aber eher den bereits mit dem Pheromon markierten Weg aus und sondern auf ihrem Rückweg zum Nest ebenfalls das Pheromon ab. (3) Auf dem kürzesten Weg konzentriert sich das Pheromon stärker als auf einem längeren, weil der kürzere Weg in gleicher Zeit von einer größeren Anzahl von Ameisen verwendet wird. Angelockt vom Pheromon benutzen nach einer Weile fast alle Ameisen den (bislang) kürzesten Weg. Nur einige wenige Ameisen weichen ab und finden evtl. zufällig noch kürzere Wege.

(Bildquelle: Wikipedia, https://de.wikipedia.org/wiki/Ameisenalgorithmus)

stärker markierte Weg zieht ihre Artgenossen an, die sich nun ebenfalls an den Futtertransport machen und dabei das Pheromon abgeben. Da der Rückweg auf der schnellsten Strecke weniger Zeit beansprucht sammelt sich darauf auch mehr Pheromon an als auf den längeren Strecken. Immer mehr Ameisen verwenden daher die kürzeste Strecke. Trotzdem verlassen einige Insekten immer wieder den am stärksten markierten Weg und explorieren andere Strecken. So kann der tatsächlich kürzeste Weg gefunden werden. Dieses Vorgehen ist relativ stabil gegen Störungen: Wird der bislang kürzeste Weg blockiert, so finden die Ameisen schnell eine Alternativroute (siehe Bogon (2013), Dorigo u. a. (2004, Abschnitt 1) und Goss u. a. (1989, Abschnitt 2.4).

Diese Erkenntnisse wurden von Dorigo u. a. (2004) in einen Algorithmus zur Lösung kombinatorischer Probleme übertragen, der seither für viele Probleme adaptiert und sehr erfolgreich eingesetzt wurde. Damit ACO angewandt werden kann, muss sich das Problem in eine für ACO geeignete Repräsentation, den *Konstruktionsgraphen* bzw. *Construction graph* überführen lassen. Im folgenden wird dies am Beispiel des Travelling Salesman Problem (TSP) erläutert:

Knoten Es muss eine endliche Menge zu kombinierender Komponenten geben, welche die Knoten des Graphen bilden. Beim TSP sind dies die zu besuchenden Orte.

Kanten Die gerichteten Kanten des Konstruktionsgraphen repräsentieren die Kombinationsmöglichkeiten der Komponenten. Im Falle des TSP sind dies die Wege, die zwischen den Orten existieren.

Pfade Ein Pfad entsteht beim Durchwandern des Konstruktionsgraphen und repräsentiert eine mögliche Lösung des Problems. Beim TSP ist dies eine Rundreise.

Restriktionen Für das Bilden von Pfaden können Einschränkungen existieren. Beim TSP stellt die Regel, dass jeder Ort genau einmal besucht werden muss eine solche Restriktion dar.

Bewertung Jedem Pfad muss ein Wert zugewiesen werden können. Beim TSP ist das die während einer Rundreise zurückgelegte Gesamtstrecke.

Die von realen Ameisen erzeugte Pheromonspur wird dadurch simuliert, dass jeder Kante ein Kantengewicht zugeordnet ist. Ein Pfad entsteht, indem eine „künstliche Ameise“ den Konstruktionsgraphen ausgehend von einem Startknoten exploriert, bis ein vorab definiertes Terminierungskriterium erfüllt ist. Diese Kantengewichte werden während des Generierens neuer Pfade bei der zufälligen Auswahl des nächsten Knotens so berücksichtigt, dass Kanten mit hoher Gewichtung eine größere Chance haben, ausgewählt zu werden als Kanten mit niedrigem Gewicht. Jede „künstliche Ameise“ hat ein Gedächtnis und speichert darin die Reihenfolge der besuchten Knoten. Hat sie einen Pfad beendet, so wird dieser Pfad bewertet und die Ameise kehrt auf ihm zum Startknoten zurück. Dabei hinterlässt sie ihre Pheromonspur, indem sie die jeweiligen Kantengewichte um einen mit reziproker Proportionalität von der Bewertung des Pfads abhängigen Betrag erhöht.

Der ACO-Algorithmus läuft iterativ ab. In jeder Iteration wird eine vorab definierte Anzahl von Pfaden (Lösungen) generiert und bewertet. Anschließend erfolgt die Aktualisierung der Kantengewichte[7] und die nächste Iteration beginnt. Diese Schleife wird solange durchlaufen, bis ein Terminierungskriterium erfüllt ist. Im Laufe des Verfahrens werden die Kantengewichte wie oben beschrieben sukzessive so angepasst, dass sich die meisten Ameisen tendenziell am gerade besten Pfad orientieren. Weil die Kantenauswahl aber dennoch eine zufällige Komponente hat kommt es immer wieder zu Abweichungen und einige Ameisen finden eine bessere Lösung. Daher ist der Algorithmus in der Lage, lokale Extrema wieder zu verlassen[8].

3.2. Parallelisierung

In dieser Arbeit soll ein massiv-paralleles System implementiert werden, das Lösungen für EVSPs findet. Unter dem Begriff *Parallelverarbeitung* (*parallel processing*) versteht man laut Engesser u. a. (1988) die „...gleichzeitige Verarbeitung eines Programms durch mehrere Prozessoren...“. Im Idealfall ergibt sich dabei eine Be-

[7] Die Kantengewichte können vor dem Start des Algorithmus mit einem passenden Initialwert versehen werden.

[8] Jedenfalls dann, wenn die Bewertungsfunktion und die Initialgewichte der Kanten gut gewählt wurden.

schleunigung der Verarbeitung um einen Faktor, welcher der Anzahl der eingesetzten Prozessoren entspricht. Dieser Idealfall tritt dann ein, wenn die einzelnen Prozessoren die ihnen zugewiesenen Teilaufgaben völlig unabhängig voneinander bearbeiten können. Andernfalls müssen Prozessoren die Abarbeitung einer Teilaufgabe unterbrechen, um auf Ergebnisse anderer Prozessoren zu warten. Laut Bengel u. a. (2015) ist die Steigerung der Verarbeitungsgeschwindigkeit eines Programms das Hauptziel der Parallelverarbeitung[9]. Das Erzielen eines Ergebnisses innerhalb einer akzeptablen Zeitspanne steht also im Vordergrund. Kommen mehrere Hundert oder Tausend Prozessoren (bzw. Prozessorkerne) zum Einsatz, spricht man von massiver Parallelverarbeitung (Bengel u. a. 2015, S. 34).

Von der Parallelverarbeitung zu unterscheiden ist die *Verteilte Verarbeitung* (Distributed computing). Letztere befasst sich mit der Koordination vieler (möglicherweise räumlich verteilter) Computer zur Lösung einer gemeinsamen Aufgabe. Die Parallelverarbeitung betrachtet dagegen i.d.R. einen einzelnen Computer (Bengel u. a. 2015).

3.2.1. Prozesse und Threads

Ursprünglich konnten Computer zu einem Zeitpunkt nur ein einzelnes Programm ausführen. Um die Ressourcen des Computers (insbesondere die CPU) möglichst gut auszunutzen wurden die auszuführenden Aufgaben zu Paketen zusammengefasst und in Form sog. *Batch-Jobs* nacheinander abgearbeitet (*Stapelverarbeitung*[10]). Später wurde die direkte Interaktion zwischen Endbenutzer und Computer üblich, was auf Seiten des Betriebssystems einen Mehrprogrammbetrieb (*Multitasking*) erforderte,

[9] Zumindest bei der im Rahmen der vorliegenden Arbeit relevanten *echten Parallelverarbeitung*. Der Zweck der *quasi-parallelen Verarbeitung* ist es dagegen, eine bessere Auslastung eines Prozessors zu erreichen. Das gelingt bspw. durch das Zeitscheibenverfahren, bei dem der Prozessor abwechselnd kurze Abschnitte aus mehreren Programmen ausführt, sodass der Benutzer eines solchen Systems den Eindruck gewinnt, die Programme würden parallel zueinander ausgeführt werden, obwohl dies tatsächlich nicht der Fall ist. Die Verarbeitungsgeschwindigkeit des Systems sinkt durch das ständige Austauschen der Prozessorregister und anderen Verwaltungsaufwand allerdings ab (Ernst u. a. 2015).

[10] Tatsächlich wurden die einzelnen Arbeitspakete auf Lochkarten definiert und dann einem Systemadministrator übergeben. Dieser sammelte die Lochkarten und ließ sie vom Computer dann stapelweise abarbeiten. Die Ergebnisse wurden den Benutzern danach meist in Form von Ausdrucken übergeben.

um das System einerseits für mehrere gleichzeitige Benutzer verwendbar zu machen und um andererseits die Computerressourcen gut ausnutzen zu können (Brause 2017, S. 28).

Voraussetzung für Multitasking ist, dass das Betriebssystem in der Lage ist, laufende Programme (*Prozesse* oder auch *Tasks*) zu unterbrechen, um die Verarbeitung eines anderen Programms fortzusetzen. Ein unterbrochener Prozess wird in eine Warteschlange eingereiht, um später fortgesetzt zu werden. Zwischenzeitlich kommt ein anderer Prozess zum Zug. Um dieses Umschalten zu ermöglichen, müssen sämtliche relevanten Statusdaten eines Prozesses im sog. *Prozesskontext* gespeichert werden. Der Prozesskontext befindet sich i.d.R. im Hauptspeicher und nimmt bspw. den Inhalt der CPU-Register zum Zeitpunkt des Umschaltens auf. Kommt der Prozess wieder an die Reihe, so wird sein Zustand zum Zeitpunkt des Unterbrechens aus dem Prozesskontext wiederhergestellt. Die Koordination der Prozesse wird von bestimmten Komponenten des Betriebssystems übernommen[11]. Aus Sicht der einzelnen Prozesse wirkt dies so, als würde jeder Prozess auf einer eigenen, dezidierten CPU ausgeführt (Brause 2017, S. 28-30 und Abschnitt 2.2).

Ein Prozesskontext ist recht umfangreich. Es müssen neben den Registern der CPU zahlreiche weitere Zustandsdaten[12] gespeichert werden. Ein Prozesswechsel ist daher teuer in dem Sinne, als dass seine Verarbeitung selbst viele Ressourcen bindet und einige Zeit in Anspruch nimmt. Für die Parallelverarbeitung im oben beschriebenen Sinne ist dieses Konzept daher nicht brauchbar. Alleine die umfangreichen Zugriffe auf den Hauptspeicher beim Lesen und Schreiben der Prozesskontexte würde den Vorteil der Parallelverarbeitung schnell zunichte machen. Moderne Prozessoren arbeiten deutlich schneller als die Bausteine, die den Speicherzugriff realisieren[13].

[11] Der *Scheduler* und der *Dispatcher* sorgen für die Organisation der Warteschlange nach einer festgelegten Strategie. Sie deaktivieren zu bestimmten Zeitpunkten den gerade laufenden Prozess, reihen ihn in die Warteschlange ein und bestimmen, welche Prozess als nächstes aktiviert wird. Dessen Zustand wird wiederhergestellt und die Verarbeitung wird fortgesetzt.

[12] Bspw. Informationen über alle offene Dateien und der logische Zustand aller verwendeten Ein-/Ausgabegeräte.

[13] Seit etwa 2004 ist es kaum noch möglich und auch kaum noch sinnvoll, die Taktgeschwindigkeit von Prozessoren weiter zu erhöhen. Die Zugriffsgeschwindigkeit auf den Hauptspeicher kann bislang nicht im gleichen Maße gesteigert werden wie die Geschwindigkeit der CPU. Das führt dazu, dass die CPU immer stärker durch das Warten auf den Speicher ausgebremst wird. Außerdem führt die Miniaturisierung der CPU dazu, dass es bei einem Prozessortakt von über 5 GHz praktisch nicht mehr möglich ist, die enorme Abwärme des Prozessors abzuführen.

Hier kommt *Mutlithreading* zum Einsatz. Ein *Thread* ist ein unabhängig ausführbares Codestück innerhalb eines Programms (i.d.R. eine Prozedur bzw. eine Funktion), dass innerhalb eines Prozesskontextes mehrfach ausgeführt werden kann. Um zwischen zwei Threads wechseln zu können reicht es aus, einige wenige Prozessorregister zu speichern, weshalb Threads auch als *leichtgewichtige Prozesse* bezeichnet werden. Moderne Prozessoren stellen interne Speicherbereiche mit besonders kurzen Zugriffszeiten und spezielle Maschinenbefehle für die Verwaltung von Threads bereit. Die Threadverwaltung selbst wird meist vom Betriebssystem oder der Laufzeitumgebung des Programms erledigt (siehe Brause (2017, S. 34 ff.) und Bengel u. a. (2015, Abschnitt 2.1.1.2)).

3.2.2. Caching

Der Arbeitsspeicher moderner Personalcomputer arbeitet wesentlich langsamer als die CPU. Das führt dazu, dass die CPU häufig auf den Arbeitsspeicher wartet. Da es zwischen CPU und Speicher nur eine Verbindung (den Speicherbus) gibt, verschärft sich diese Problematik bei der Verwendung von Multicoreprozessoren zusätzlich. Deshalb existiert zwischen CPU und Arbeitsspeicher ein Zwischenspeicher, der aus besonders schnellen Speicherbausteinen aufgebaut ist. Diesen Zwischenspeicher bezeichnet man als *Cache*[14]. Jeden Wert, den die CPU aus dem Arbeitsspeicher liest, legt die Memory Management Unit (MMU)[15] zusätzlich im Cache ab. Genauso wird auch mit jedem zu schreibenden Wert verfahren. Nachfolgende Zugriffe auf dieselbe

Hinzu kommt, dass die Strukturen in Prozessoren bereits so klein sind, dass bei der Entwicklung einer CPU auch quantenmechanische Effekte beachtet werden müssen. Daher fokussiert sich die Prozessorenentwicklung seit 2005 vor allem auf Multicoreprozessoren (siehe Ernst u. a. (2015, S. 241 f.) und Waldrop (2016).

[14] Der Arbeitsspeicher besteht typischerweise aus DRAM-Modulen (*Dynamic Random Memory Access*), die mit einer internen Taktrate von etwa 260 MHz arbeiten oder aus SDRAM-Modulen (*Synchronous Dynamic Random Memory Access*), welche mit der Taktrate des Mainboards arbeiten können, die typischerweise bei 3 bis 4 GHz liegt. Der Cache ist typischerweise zwei- oder dreistufig aufgebaut. Die erste Stufe (der L1-Cache) befindet sich meist direkt auf dem CPU-Chip (dem *Die*) und ist für jeden CPU-Kern einmal vorhanden. Die beiden nachgelagerten Stufen des Caches heißen L2- bzw. L3-Cache. Oft befinden sie sich ebenfalls auf dem Die der CPU oder sie sind zumindest direkt an die CPU angebunden (dann bestehen sie meist aus besonders schnellen SDRAM-Modulen). Der L1-Cache ist relativ klein (typisch sind 32 KB), während der L2- und der L3-Cache jeweils einige Megabyte groß sein können. Sie sind normalerweise nur einmal vorhanden und werden von allen Kernen gemeinsam benutzt.

[15] Die MMU ist das Speichermanagementsystem, das alle Zugriffe auf den Arbeitsspeicher koordiniert.

Speicheradresse können dann ohne Umweg über den Arbeitsspeicher direkt aus dem Cache bedient werden („Cache-Hit“). Nur wenn ein Wert nicht im Cache gefunden werden kann wird er aus dem Hauptspeicher gelesen („Cache-Miss“). Allerdings ist die Größe des Caches sehr begrenzt, sodass die Werte nicht unbegrenzt lange vorgehalten werden können. Die Caching-Strategie entscheidet darüber, welche Werte gecached (also in den Cache eingetragen) werden und wie lange diese Werte dort verbleiben. Dabei macht sich die Caching-Strategy zunutze, dass Programme recht häufig aufeinander folgende Speicherzellen abrufen („Datenlokalität“)[16]. Diese Eigenschaft bezeichnet man als *Datenlokalität* (siehe Bengel u. a. (2015, S. 39 ff.) und Ernst u. a. (2015, S. 246 ff.)).

3.2.3. Multiprozessor- und Multicoreprozessor-Systeme

In einem Multiprozessor-System sind mehrere CPUs verbaut. Dabei verfügt jede CPU über einen eigenen Cache. Alle CPUs teilen sich jedoch die anderen Ressourcen des Computers; insbesondere den Hauptspeicher[17] (Bengel u. a. 2015, Abschnitt 2.1.2).

Ein Multicoreprozessor vereint im Grunde mehrere CPUs auf einen einzigen Chip, denn alle für die eigenständige Ausführung eines Threads benötigten Komponenten sind mehrfach vorhanden. Diese integrierten CPUs werden als *Kerne* (*Cores*) bezeichnet. Auch sie haben jeweils eigene Caches, teilen sich aber den Hauptspeicher (siehe Bengel u. a. (2015, Abschnitt 2.1.2.4) und Ernst u. a. (2015, S. 243 f.)). Üblicherweise haben Multicoreprozessoren heute zwischen zwei und vier Kernen. Aber auch CPUs mit bis zu 24 Kernen sind erhältlich. Solche mit 32 Kernen sind in naher Zukunft zu erwarten[18].

[16] Bspw. werden beim Durchlaufen einer Programmschleife zur Berechnung des Betrages eines Vektors nacheinander die einzelnen Komponenten des Vektors aus dem Speicher ausgelesen. Diese sind mit großer Wahrscheinlichkeit in aufeinander folgenden Speicherstellen abgelegt, weshalb es sinnvoll ist, den Cache schon beim ersten Speicherzugriff mit dem Inhalt aller folgenden Speicherzellen zu füllen. Solche sogenannten Burst-Zugriffe auf den Arbeitsspeicher sind deutlich schneller als einzelne Zugriffe.

[17] Wegen der gemeinsamen Verwendung des Hauptspeichers durch mehrere Prozessoren wird ein solches System als „eng gekoppelt“ bezeichnet. Den gemeinsamen Zugriff auf den Hauptspeicher nennt man *Unified Memory Access (UMA)*.

[18] Siehe https://www.golem.de/news/xeon-e5-2699-v5-intels-skylake-ep-fuer-server-mit-32-cpu-

3.2.4. Programmierung von Multicore-Architekturen

Bereits bei der Erstellung eines Programms muss die parallele Ausführbarkeit besonders berücksichtigt werden. Nur selten lassen sich innerhalb eines Programms Aufgaben identifizieren, die sich tatsächlich in völlig unabhängig voneinander ausführbare Teilaufgaben zerlegen lassen. Wenn dies der Fall ist, spricht man von *nebenläufigen Prozessen* und meint damit, dass es zwischen zwei Aktionssträngen keinen kausalen Zusammenhang gibt. Aber auch dann ist es möglich, dass die Prozesse miteinander um die Nutzung gemeinsamer Ressourcen (bspw. dem Hauptspeicher) konkurrieren. Typischerweise entstehen aber Teilaufgaben, die nur teilweise unabhängig voneinander bearbeitet werden können. Dann spricht man von *kooperierenden Prozessen*. Es sind dann zwei Aktionsstränge an der Lösung ein- und derselben Aufgabe beteiligt und dabei logisch so miteinander verknüpft, dass eine Synchronisation zwischen ihnen erfolgen muss (Bengel u. a. 2015, Abschnitt 2.1.3).

Durch die Nutzung von Multicore-Architekturen verschärft sich auch die Diskrepanz zwischen der Arbeitsgeschwindigkeit des Prozessors und der des Hauptspeichers. Moderne Multicore-Prozessoren enthalten daher mehrstufige Caches. Darunter Caches, die einzelnen Cores zugeordnet sind aber auch Caches, die von mehreren Cores gemeinsam verwendet werden, was die Synchronisation der Cores begünstigt.

3.2.5. Race Conditions, Semaphoren und Deadlocks

Werden Threads mangelhaft synchronisiert, kann das zu Fehlern führen, die nur schwer aufzuspüren sind. Wird die Funktion `setNextElement()` der Klasse `RoundRobin` im folgenden Codebeispiel von mehreren Threads parallel verwendet, so ist es sehr wahrscheinlich, dass es irgendwann zu einer Zugriffsverletzung im Hauptspeicher kommt, weil versucht wird, in ein nicht existierendes 6. Element des Vektors $\vec{a}$ zu schreiben.

```
1 class RoundRobin {
2         int a[5];
```

kernen-aufgetaucht-1611-124637.html (abgerufen am 13.06.2017).

```
3           int currentElement := 0;
4           void setNextElement(int value) {
5                   if(currentElement < 5) {
6                           a[currentElement] := value;
7                           currentElement++;
8                   }
9                   if(currentElement >= 5) {
10                          currentElement := 0;
11                  }
12          }
13  }
```

Denn wenn der aktuelle Wert von `currentElement` 4 ist und ein Thread *A* gerade die Zeile 6 ausgeführt hat kann ein Thread *B* gerade dabei sein, Zeile 5 auszuführen. Thread *B* wird nun als nächstes die Zeile 6 ausführen, weil `currentElement` ja immer noch den Wert 4 hat. Bevor der Thread *B* das jedoch tun kann führt Thread *A* die Zeile 7 aus. Nun hat `currentElement` allerdings den Wert 5 und der Thread *B* wird versuchen, in das Arrayelement `a[5]` zu schreiben. Dieses Arrayelement existiert jedoch nicht. Im besten Fall wird das Programm nun mit einer Fehlermeldung abgebrochen. Es ist aber auch möglich, dass das Laufzeitsystem derartige Fehler nicht bemerkt und es wird eine möglicherweise anderweitig verwendete Speicherstelle unkontrolliert verändert.

Eine solche Situation, in der das Verhalten des Programms von der Reihenfolge abhängt, in der parallel laufende Threads gemeinsam verwendete Codeteile durchlaufen nennt man *Race Conditions*. Race Conditions lassen sich durch die Verwendung von *Semaphoren* vermeiden. Eine bestimmte Semaphore ist systemweit nur einmal verfügbar. Besitzt ein Thread A diese Semaphore, so kann kein anderer Thread B sie erhalten, bis A die Semaphore wieder freigegeben hat. So lässt sich sicher stellen, dass kritische Codeabschnitte wie die Zeilen 5 bis 11 des obigen Beispiels zu jedem Zeitpunkt nur von einem Thread durchlaufen werden.

Das Konzept der Semaphore geht auf Dijkstra (1971) zurück und birgt die Ge-

fahr sogenannter *Deadlock*. Zu einem Deadlock kommt es, wenn zwei Semaphoren benötigt werden, um eine bestimmte Aktion durchzuführen, die beiden Semaphoren aber im Besitz zweier unterschiedlicher Threads sind. Das Philosophenproblem macht dies anschaulich: Fünf Philosophen sitzen an einem Tisch. Auf diesem Tisch befinden sich fünf Teller und fünf Gabeln. Wird ein Philosoph hungrig, so greift er zuerst die Gabel zu seiner linken und dann die zu seiner rechten und beginnt zu essen. Findet er eine Gabel nicht vor, so wartet er, bis sie wieder verfügbar ist. Wenn er satt ist, legt er die beiden Gabeln wieder zurück an ihren Platz. Wenn nun alle fünf Philosophen zur gleichen Zeit hungrig werden, so kommt es zu einem Deadlock, denn alle nehmen die linke Gabel auf und warten dann vergeblich darauf, dass die rechte Gabel wieder an ihrem Platz liegt. Das passiert jedoch nie, da ja auch ihr jeweils rechter Nachbar wartet.

Siehe Brause (2017, Abschnitt 2.3) und Ernst u. a. (2015, Abschnitt 8.3.2).

3.3. General Purpose Computation on Graphics Processing Unit

Die Ausführungen in diesem Abschnitt basieren auf Bengel u. a. (2015, Abschnitt 2.2), Nischwitz u. a. (2011, Kapitel 17), NVIDIA (2017) und Kirk u. a. (2010).

Seit das Konzept einer separaten Erweiterungskarte für die Grafikfähigkeiten eines Personalcomputers 1977 mit dem Apple II eingeführt wurde, haben sich die PC-Technik insgesamt und insbesondere die PC-Grafikhardware in unvorstellbarer Weise weiterentwickelt. Vor allem die Verbreitung von 3D-Computerspielen hat dazu beigetragen (siehe Abschnitt 1.2). 3D-Grafikberechnungen beruhen im wesentlichen auf Matrixoperationen (also auf linearer Algebra). Denn es werden ja dreidimensionale Objekte in eine zweidimensionale Darstellungen überführt[19]. Eine 3D-fähige

[19] Dies stellt eine grobe Vereinfachung dar. Die darzustellenden Objekte sind nicht real, sondern lediglich stark reduzierte Modelle im Speicher des Rechners. Um ihre Darstellung real erscheinen zu lassen, sind sehr aufwändige Berechnungen nötig. Wiederum stark vereinfacht dargestellt, werden sie dazu innerhalb der Grafikhardware i.d.R. aus zahllosen Dreiecken zusammengesetzt, auf welche eine realistisch wirkende Oberflächentextur projiziert wird. Es entsteht dabei

Grafikkarte ist u.a. auf die besonders effiziente Ausführung solcher Operationen spezialisiert.

Zunächst waren die für die 3D-Grafikdarstellung nötigen Routinen fest in den Chips der Grafikkarte „verdrahtet“[20]. Später wurden diese Spezialchips durch einen frei programmierbaren Prozessor-Chip ersetzt und es wurde möglich, die Rechenleistung einer Grafikkarte auch für andere Anwendungen zu verwenden. Diese anderweitige Verwendung wird als *General Purpose Computation on Graphics Processing Unit (GPGPU)* bezeichnet. Dabei kommt der GPU die Rolle eines Coprozessors zu, auf dem hochgradig parallelisierbare Programmteile sehr viel schneller als auf der CPU ausgeführt werden können[21].

3.3.1. CUDA

Anfangs war ein erhebliches technisches Know-how nötig, um die Hardware der Grafikkarte für GPGPU nutzen zu können. Der Programmcode, der auf der GPU ausgeführt werden sollte, musste zunächst durch das für die Grafikprogrammierung gedachte Application Programming Interface (API) hindurchgeschleust werden[22]. Das änderte sich 2007, als der Grafikkartenhersteller NVIDIA die *Compute Unified Device Architecture (CUDA)* vorstellte[23]. CUDA ist ein API, das die Parallelpro-

ein dreidimensionales Modell, das dann für die Bildschirmdarstellung in ein zweidimensionales System überführt wird. Dabei müssen u.a. verdeckte Anteile ausgeblendet und realistische Lichtverhältnisse sowie Schattenwürfe simuliert werden. Diese Vorgänge werden als „Rendering“ bezeichnet (Foley u. a. 1995).

[20] Diese Grafikkarten-Architektur wird als *Fixed Function Pipeline* bezeichnet.

[21] Für gut parallelisierbare Problemlösungen lassen sich auf der Grundlage von GPGPU mithilfe relativ kostengünstiger Grafikkarten sehr leistungsfähige Parallelrechner aufbauen. Tatsächlich bieten manche Hersteller auch auf GPGPU spezialisierte Grafikkarten an.

[22] Oft eingesetzt wurden dazu Open Graphics Library (OpenGL) von der Khronos Group (siehe https://www.opengl.org/) und Direct3D von Microsoft (siehe https://de.wikipedia.org/wiki/Direct3D).

[23] Neben CUDA, das nur auf Grafikkarten des Herstellers NVIDIA nutzbar ist, existieren noch weitere APIs für GPGPU mit Hochsprachen wie C und C++. Diese APIs werden in der vorliegenden Arbeit jedoch nicht verwendet und es wird daher auch nicht näher auf sie eingegangen. Ein Beispiel ist die *Open Computing Language (OpenCL)*. OpenCL ist ein offener Standard für die Programmierung von Parallelrechnern, der auch die Nutzung von GPUs erlaubt, aber weit darüber hinaus geht (siehe https://www.khronos.org/opencl/). Auch DirectCompute von Microsoft erlaubt GPGPU. Etwa zeitgleich mit CUDA erschien das *AMD Stream SDK* des NVIDIA-Konkurrenten AMD, das GPGPU auf AMD-Grafikkarten erleichterte.

grammierung auf der Grafikkarte mithilfe der Programmiersprache C ermöglicht[24], ohne den Umweg über das Grafikinterface gehen zu müssen. Gleichzeitig mit der CUDA-Einführung begann NVIDIA damit, Grafikkarten auszuliefern, auf denen eine spezielle parallele Schnittstelle verbaut war, sodass CUDA unmittelbar auf den Arbeitsspeicher und die GPU der Grafikkarte zugreifen konnte. Eine solche Grafikkarte war und ist Voraussetzung für die Nutzung von CUDA.

Auch wenn APIs wie CUDA es stark vereinfacht haben, GPGPU-Programme zu entwickeln bleibt es doch eine sehr hardwarenahe Tätigkeit. Das bedeutet, dass der Programmierer tief greifende Kenntnisse vom Aufbau und von der Funktionsweise einer Grafikkarte benötigt, um ein effizientes CUDA-Programm entwerfen zu können.

Wie bereits in Abschnitt 1.2 erwähnt bildet eine moderne Grafikkarte innerhalb des PCs eine eigenständige Einheit. Ihre GPU wird bei der GPGPU als Coprozessor der CPU verwendet. CUDA bezeichnet den PC, in welchem sich die Karte befindet daher als *Host* und die Grafikkarte als *Device*. Entsprechend werden der Arbeitsspeicher des PCs als *Host-Memory* und der Arbeitsspeicher der Grafikkarte als *Device-Memory* bezeichnet.

3.3.2. CUDA-Architektur

Eine CUDA-konforme Grafikkarte besteht aus mehreren Prozessoren, die als *SMs* bezeichnet werden[25]. Jeder SM beinhaltet mehrere Caches, einen 64KB großen *Shared Memory* und 64 CUDA-Kerne, die in zwei Blöcke mit jeweils 32 Kernen unterteilt sind. Diese Blöcke heißen Streaming Processor (SP). Jeder SP verfügt über einen *Scheduler* (den sogenannten *Warp-Scheduler*), der festlegt, welche Threads auf den Kernen des SP als nächstes zur Ausführung kommen[26]. Jeder CUDA-Kern beinhal-

[24] Auch C++ kann verwendet werden, wenn auch stark eingeschränkt. Polymorphie ist bspw. nicht möglich, da CUDA keine virtuellen Methoden unterstützt.

[25] Die Mikroarchitektur der GPUs CUDA-konformer Grafikkarten wird ständig weiterentwickelt. Die erste Architektur-Generation trug die Bezeichnung „Tesla“. Aktuell wird die Architektur-Generation „Pascal“ verwendet, die bald von „Volta“ abgelöst werden wird. Die Ausführungen dieses Abschnitts beziehen sich auf eine Ausprägung der „Pascal“-Architektur.

[26] Tatsächlich ist der SP nur ein logisches Konstrukt, welches das Verständnis der Architektur vereinfacht, das aber auf dem Chip der GPU physikalisch nicht existiert.

Abb. 3.4: Blockdiagramm eines SM der „Pascal"-Architektur

(Bildquelle: wccftech, WCCF PTE LTD.,
http://wccftech.com/nvidia-pascal-gp104-gpu-gddr5x-leak/g)

tet wiederum folgende Komponenten:

- Eine Arithmetic Logic Unit (ALU) für Berechnungen mit ganzen Zahlen
- Eine Floating Point Unit (FPU) für Berechnungen mit Fließkommazahlen
- Eine Special Funktion Unit (SFU) für die Durchführung transzendenter Funktionen wie Sinus, Cosinus etc.[27]

3.3.3. Das SIMT-Prinzip

Laut Flynn (1972) lassen sich Prozessorarchitekturen grob in vier Klassen einteilen:

- *Single Instruction Single Data (SISD)*

  Ein Datenstrom wird mit einer sequenziellen Befehlsfolge verarbeitet. Eine

[27] Auch das ist nicht ganz korrekt. Denn tatsächlich teilen sich mehrere Kerne jeweils eine SFU. Für das Verständnis der Architektur ist dieser Umstand aber unerheblich.

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Abb. 3.5: Klassifikation nach Flynn
(Bildquelle: Ernst u. a. (2015))

echte Parallelverarbeitung ist nicht möglich.

- *Multiple Instruction Single Data (MISD)*
  Diese Klasse existiert in der Praxis nicht. Derartige Prozessoren müssen eine Folge unterschiedlicher Befehle zur selben Zeit auf ein- und dasselbe Datum anwenden.

- *Single Instruction Multiple Data (SIMD)*
  Es werden mehrere Datenströme gleichzeitig mit derselben Befehlsfolge verarbeitet. Auf allen beteiligten Prozessoren wird also der gleiche Befehl auf unterschiedliche Daten angewandt. Dies ist das Arbeitsprinzip von Vektorprozessoren

- *Multiple Instruction Multiple Data (MIMD)*
  Es werden mehrere Datenströme mit mehreren unterschiedlichen Befehlsfolgen verarbeitet. Dies ist bspw. das Prinzip eines Multicoreprozessors, bei denen jeder Kern im Grunde eine vollwertige CPU mit eigenem Befehlszähler und Registersatz ist. Es erlaubt die größte Flexibilität, ist aber hardwareseitig mit dem größten Aufwand verbunden.

Moderne Prozessoren kombinieren das SIMD- mit dem MIMD-Prinzip: Sie beinhalten mehrere Kerne, von denen jeder (mindestens) einen Befehlsstrom abarbeiten kann und zusätzlich noch Erweiterungen des Befehlssatzes wie *Streaming SIMD Extensions (SSE)* und *Advanced Vector Extensions (AVX)*, mit denen Vektoroperationen möglich sind[28].

[28] Beispielsweise hat der Prozessor Intel Core i7-7920HQ vier Kerne, von denen jeder zwei Threads parallel ausführen kann („Hyper-Threading“). Außerdem unterstützt er SSE und AVX und ist

GPUs basieren vor allem auf dem SIMD-Prinzip: Sie beinhalten eine große Anzahl von Prozessorkernen, die eine Befehlsfolge auf mehrere Datenobjekte anwenden können[29]. Dieses Prinzip erlaubt es, das Schaltungslayout eines Prozessors deutlich zu vereinfachen, denn es wird nur ein Befehlszähler und ein Befehlsregister benötigt, da alle Kerne zu einem Zeitpunkt denselben Befehl ausführen. Das SIMD-Prinzip hat in seiner reinen Form jedoch den Nachteil, dass es nicht möglich ist, Teile des Datenstroms von der Verarbeitung auszuklammern. Sollen, wie im folgenden Pseudocode dargestellt, bspw. die einzelnen Zeilen der beiden Vektoren $\vec{a}$ und $\vec{b}$ nur dann addiert werden, wenn die entsprechende Zeile in Vektor $\vec{c}$ ungleich Null ist, so lässt sich dies mit einem SIMD-Prozessor nicht ohne weiteres bewerkstelligen, da die bedingte Anweisung in Zeile 5 eine Verzweigung im Ausführungspfad der parallel arbeitenden Kerne nötig macht. Dies wird als Code-Divergenz bezeichnet und muss bei Verwendung eines SIMD-Prozessors aufwändig aufgelöst werden.

```
1 int a[5] = {1, 2, 3, 4, 5};
2 int b[5] = {6, 7, 8, 9, 10};
3 int c[5] = {1, 0, 0, 1, 1};
4 for(int i=0; i<5; i++) {
5         if(c[i]!=0)
6                 result[i]=a[i]+b[i];
7         else
8                 result[i]=0;
9 }
```

Ein SP verfügt daher als zusätzliche Komponente über einen Thread-Scheduler (im CUDA-Jargon *Warp-Scheduler* genannt). Der Warp-Scheduler ermöglicht Code-Divergenz, indem er den Code-Teil, in dem die Divergenz auftritt, mehrfach ausführen lässt: Nämlich einmal für jeden Zweig des Ausführungspfads. Angenommen, jeder Vektorzeile ist bei der Ausführung des obigen Pseudocodes ein GPU-Kern zugeordnet, so wird zunächst die Zeile 6 des Pseudocodes auf den Kernen 0, 3 und 4 ausgeführt, da der Vektor $\vec{c}$ in diesen Zeilen ungleich Null ist. Die Kerne 1 und 2

so in der Lage, einen Befehl auf mehrere Datenobjekte anzuwenden. Zusätzlich enthält er eine GPU des Typs „Intel HD Graphics 630“ (siehe https://ark.intel.com/de/products/97462/Intel-Core-i7-7920HQ-Processor-8M-Cache-up-to-4_10-GHz).

[29] Bspw. könnte jeder Kern einen bestimmten Bildausschnitt bearbeiten.

bleiben dabei inaktiv. Danach erfolgt ein zweiter Durchlauf, in dem die Zeile 8 des Pseudocodes abgearbeitet wird. Dieses Mal arbeiten die Kerne 1 und 2, während alle anderen Kerne inaktiv bleiben[30].

Dieses Prinzip, das von NVIDIA als *Single Instruction Multiple Thread (SIMT)* bezeichnet wird, macht eine GPU wesentlich flexibler als einen SIMD-Prozessor. Auftretende Divergenzen[31] verschlechtern allerdings die Performance, weil Teile des Ausführungspfads sequentiell ausgeführt werden müssen, während jeweils die Kerne, für welche die betreffende Branch-Bedingung nicht erfüllt ist, inaktiv sind. Die Auslastung des SP ist dabei entsprechend schlecht. Das kann sogar soweit gehen, dass von den 32 Kernen eines SP zeitweise nur noch ein einzelner Kern aktiv ist.

3.3.4. Struktur und Ausführung eines CUDA-Programms

Ein CUDA-Programm besteht aus einer Funktion, die als *Kernel* bezeichnet wird[32]. Mit diesem Kernel startet das CUDA-Laufzeitsystem eine große Anzahl von Threads, die dann dem SIMT-Prinzip folgend massiv-parallel ausgeführt werden. Dabei sind die Threads wahlweise als ein-, zwei- oder dreidimensionale Matrix indiziert, damit es innerhalb des Kernels möglich ist, einen Bezug auf die zu bearbeitenden Daten herzustellen. Die genaue Anzahl der zu startenden Threads sowie die Art der Indizierung werden vorab vom Programmierer festgelegt. Der Kernel und sein Aufruf werden dazu in entsprechend erweitertem C bzw. C++ codiert und dann mithilfe eines CUDA-Compilers für die GPU übersetzt.

Threads werden in CUDA also in Form einer Matrix organisiert, die als *Grid* bezeichnet wird. Innerhalb des Grids erfolgt eine weitere Unterteilung in *Blöcke*, deren Größe ebenfalls vom Programmierer festgelegt wird[33]. Ein Thread erhält also eine eindeutige Kennung durch Blocknummer und Index. Diese Daten sind im Kernel

[30] Im CUDA-Jargon wird diese Vorgehensweise als *Conditional Execution* bezeichnet.
[31] Code-Divergenzen werden in CUDA als *Warp Divergence* bezeichnet.
[32] Der Kernel stellt also gewissermaßen das „Hauptprogramm“ dar, aus dem heraus selbstverständlich auch Subroutinen aufgerufen werden können. Der Kernel muss also keineswegs den gesamten Code beinhalten, sondern es ist möglich, den Code zu modularisieren, wie dies in der Softwareentwicklung üblich ist.
[33] Wie wir sehen werden ist auf diese Weise eine optimale Ausnutzung der GPU möglich.

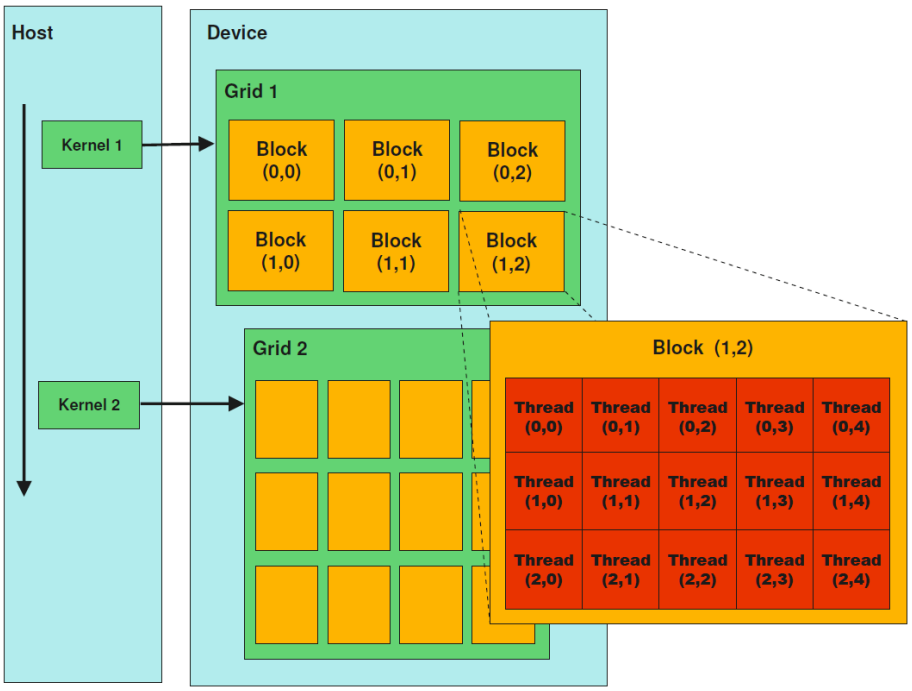


Abb. 3.6: Schematische Darstellung der Thread-Organisation in Grids und Blöcke (Bildquelle: Nischwitz u. a. (2011, S. 497))

über vordefinierte Variablen abrufbar (*blockDim* enthält die Blockgröße, *blockIdx* die Blocknummer und *threadIdx* die Nummer des Threads innerhalb des Blocks):

```
1 int x = blockIdx.x * blockDim.x + threadIdx.x;
2 int y = blockIdx.y * blockDim.y + threadIdx.y;
3 int z = blockIdx.z * blockDim.z + threadIdx.z;
```

Als Beispiel soll hier die Addition großer zweidimensionaler Matrizen dienen. Auf dem Host würde diese Operation in sequentieller Weise durch verschachtelte Schleifen implementiert werden[34]:

```
1 for(int x=0; x<1024; x++) {
2         for(int y=0; y<1024; y++) {
3                 c[x][y] = a[x][y] + b[x][y];
4         }
5 }
```

Es werden auf dem Host also 1.048.576 Additionen sequentiell durchgeführt. Der

[34]Im Beispiel wird unterstellt, dass die Matrizen a, b und c bereits vorher sinnvoll initialisiert wurden. Außerdem wird unterstellt, dass auf dem Host nur ein CPU-Kern verwendet wird.

entsprechende CUDA-Kernel addiert hingegen nur jeweils eine einzelne Matrixzelle[35]:

```
1 __global__ void matrixAddKernel(int a[][], int b[][], int c[][])
2 {
3         int x = blockIdx.x * blockDim.x + threadIdx.x;
4         int y = blockIdx.y * blockDim.y + threadIdx.y;
5
6         c[x][y] = a[x][y] + b[x][y];
7 }
```

Das Schlüsselwort *__global__* teilt dem CUDA-Compiler mit, dass es sich um einen Kernel handelt. Der Aufruf des Kernels erfolgt dann auf dem Host:

```
1 int blockSize = 32;
2 int numberOfBlocks = 1024 / blockSize;
3 dim3 dimGrid(numberOfBlocks, numberOfBlocks);
4 dim3 dimBlock(blockSize, blockSize);
5 matrixAddKernel<<<dimGrid, dimBlock>>>(a, b, c);
```

Dieser Code hat folgende Bedeutung:

- In Zeile 4 wird die Dimension und die Größe des Grids festgelegt[36]. Das Grid soll also zweidimensional sein und $32 \cdot 32$ Blöcke umfassen
- Zeile 3 definiert Dimension und Größe eines Blocks ($32 \cdot 32$ Threads)
- In Zeile 5 erfolgt der Aufruf des Kernels. CUDA startet daraufhin insgesamt 1.048.576 Threads (1.024 Blöcke á 1.024 Threads) und verteilt sie auf die verfügbaren Kerne

Die Threads werden innerhalb der GPU blockweise verarbeitet. Dabei wird jeder Block weiter in Einheiten zu je 32 Threads aufgeteilt. Eine solche Einheit wird in

[35] Um das Beispiel einfach zu halten, ist das Code-Fragment in Pseudocode angegeben. Denn der Kernel wird in C programmiert und der Zugriff auf mehrdimensionale Matrizen erfolgt in C durch Anwendung der sogenannten Pointer-Arithmetik, worunter die Übersichtlichkeit des Beispiels erheblich leiden würde.

[36] *dim3* ist ein in CUDA vordefiniertes Tupel aus drei Integer-Werten.

CUDA als *Warp* bezeichnet. Der Warp-Scheduler verteilt die Warps eines Blocks auf die gerade verfügbaren SPs. Sind alle Warps eines Blocks abgearbeitet, so erfolgt die Verarbeitung des nächsten Blocks. Verfügt die GPU bspw. über 80 SPs können auf diese Weise 80 Warps mit insgesamt 2.560 Threads parallel verarbeitet werden.

CUDA erledigt also die Verteilung der Threads auf die verfügbaren SPs. Die Koordination der Threads, etwa beim Zugriff auf gemeinsam genutzte Speicherbereiche, obliegt jedoch dem Programmierer. Es ist auch möglich, weitere Kernel zu starten, noch bevor die Verarbeitung eines Kernels abgeschlossen ist.

3.3.5. Die CUDA-Speicherhierarchie

Bei der Programmierung für moderne Betriebssysteme wie Microsoft Windows oder Apple macOS muss sich der Programmierer in aller Regel nicht um die Organisation des Arbeitsspeichers kümmern. Das Betriebssystem kümmert sich um das Auslagern von Speicherbereichen auf die Festplatte („paging“), das Caching, das Verlagern häufig benutzter Variablen in freie CPU-Register und um weitere mit der Speicherverwaltung verbundene Aktivitäten[37]. Bei der Programmierung mit CUDA ist die Wahl der passenden Speicherebene weitgehend die Aufgabe des Programmierers.

Vor dem Start eines Kernels müssen alle vom Kernel benötigten Daten zunächst aus dem Host- in den Device-Memory kopiert werden. Zu diesem Zweck existieren spezielle CUDA-Kommandos, die vor dem Kernel aufzurufen sind[38]. Im Kernel sollten alle häufig benutzten Daten aus dem Device-Memory in lokale Variablem kopiert werden, denn der CUDA-Compiler speichert den Inhalt lokaler Variablen in freien GPU-Registern[39]. Das ist deshalb von entscheidender Bedeutung, weil der Zugriff

[37] Dies ist eine vereinfachte Darstellung. Tatsächlich tragen viele weitere Komponenten wie bspw. der beim Programmieren verwendete Compiler und die Laufzeitumgebung zur effizienten Speicherverwaltung bei.

[38] Aktuelle CUDA-Versionen bieten mit einen als *Unified Memory* bezeichneten Speicherbereich auch die Möglichkeit, die benötigten Daten von CUDA automatisch in den Device-Speicher transferieren zu lassen. Allerdings beeinträchtigt das Verwenden von Unified Memory die Performance eines CUDA-Programms. Nach Landaverde u. a. (2014) muss mit einer etwa doppelt so langen Laufzeit gerechnet werden, weshalb im Rahmen der vorliegenden Arbeit auf die Verwendung von Unified Memory verzichtet wurde.

[39] Der Inhalt aller GPU-Register wird auch als der *Execution Context* eines Threads bezeichnet.

auf den Device-Memory vergleichsweise langsam erfolgt. Typischerweise dauert er 100 bis 200 Taktzyklen. Der Inhalt eines Registers steht aber in einem einzigen Taktzyklus zur Verfügung. Leider ist die Anzahl der Register begrenzt. Sind keine freien Register mehr vorhanden werden auch die Inhalte lokaler Variablem im Device-Memory abgelegt[40].

Alle Threads eines Blocks haben Zugriff auf einen gemeinsamen, 48KB großen Speicherbereich, der als *Shared Memory* bezeichnet wird und auf den innerhalb einiger weniger Taktzyklen zugegriffen werden kann[41]. Der Shared Memory wird häufig benutzt, um Speicherzugriffe zu optimieren, indem ein innerhalb des Blocks häufig benutzter Speicherbereich zunächst in den Shared Memory kopiert, dort von den Threads bearbeitet und schließlich in den Device-Memory zurück kopiert wird. In diesem Fall fungiert der Shared Memory als eine Art selbst verwalteter Cache. Damit mehrere Threads zur gleichen Zeit auf den Shared Memory zugreifen können ist dieser in 32 Speicherbänke aufgeteilt. Eine Bank kann zu einer Zeit den Zugriff auf eine Speicherstelle (bzw. einen Speicherbereich) verarbeiten. Greifen also zwei Threads auf ein- und dieselbe Speicherstelle zu, so kann dieser Zugriff sofort bedient werden. Handelt es sich aber um zwei verschiedene Speicherstellen, wird dies als *Bank-Konflikt* bezeichnet und die Zugriffe werden serialisiert.

3.3.6. Coalesced Memory Access

Um die große Latenz des Device-Memory abzumildern wird bei einer Speichertransaktion immer ein 128 Byte großer Cache verwendet und es wird immer ein 128 Byte großer Bereich gelesen bzw. geschrieben. Dies wird als *vereinter Speicherzugriff (coalesced memory access)* bezeichnet. Dadurch werden Zugriffe auf zusammenhängende Speicherbereiche deutlich beschleunigt. Die Threads innerhalb eines Warps sollten daher möglichst auf benachbarte Speicherstellen zugreifen, damit möglichst wenige Speichertransaktionen nötig sind. Verwenden die Threads dagegen weit voneinander entfernte Speicheradressen, so muss der Zugriff auf mehrere Transaktionen

[40] Im CUDA-Jargon wird dies als *Variable Spilling* bezeichnet.
[41] Der Shared Memory wird auch als *Shared Execution Context* bezeichnet.

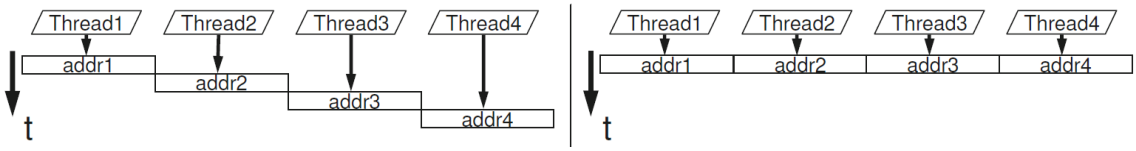


Abb. 3.7: Coalesced Memory Access

Links: non-coalasced access, rechts: coalasced access
(Bildquelle: Nischwitz u. a. (2011, S. 502))

aufgeteilt werden (siehe Abb. 3.7).

3.3.7. Code-Divergenz und Speicherzugriffe

Die negativen Effekte der Code-Divergenz können sich bei Zugriffen auf den Device-Memory noch verschärfen. Wie im Unterabschnitt 3.3.6 dargestellt sollten nach Möglichkeit nur vereinte Speicherzugriffe erfolgen. Wenn eine Speichertransaktion jedoch innerhalb einer bedingten Anweisung erfolgt, so nehmen u.U. nicht alle Threads am Speicherzugriff teil, weil eben nicht alle Threads aktiv sind. Schauen wir dazu zunächst ein Beispiel ohne Code-Divergenz an, in dem die Werte zweier Vektoren um einen bestimmten Wert erhöht werden:

```
1 __global__ void divergingKernel1(int a[], int b[], int value)
2 {
3         int x := blockIdx.x * blockDim.x + threadIdx.x;
4         a[x] := a[x] + value;
5         b[x] := b[x] + value;
6 }
```

Ein Warp besteht, wie bereits ausgeführt, aus 32 Threads und wird von genau einem SP abgearbeitet. Betrachten wir die Aktivitäten des SP beim Bearbeiten des ersten Warps im ersten zu bearbeitenden Block. Die Indexvariable x wird Werte zwischen 0 und 31 annehmen, da die Indizes innerhalb eines Blocks und damit auch innerhalb eines Warps aufeinander folgen. In Zeile 3 erfolgt in allen Threads zunächst ein Lesezugriff auf die jeweilige Speicherstelle von $a[0]$ bis $a[31]$. Das ist in einem vereinten Speicherzugriff (coalesced memory access) möglich und die gelese-

nen Werte werden in den Cache des SP geschrieben[42]. Im Cache wird dann jeweils der Inhalt von *value* hinzuaddiert und das Ergebnis wird in einer weiteren vereinten Speicher-Transaktion wieder in den Device-Memory geschrieben. Damit ist Zeile 3 vollständig abgearbeitet und es wird mit Zeile 4 fortgefahren, wo mit dem Vektor b analog verfahren wird. Der Kernel `divergingKernel1` kommt also mit insgesamt vier Speichertransaktionen pro Warp aus.

Erweitern wir dieses Beispiel nun um eine bedingte Anweisung. Nun sollen die Komponenten der beiden Vektoren a und b für den Thread 0 um *value* erhöht werden. In allen anderen Fällen soll *value* nur dann addiert werden, wenn der Index eine gerade Zahl ist. Andernfalls ist er vom aktuellen Wert zu subtrahieren.

```
1  __global__ void divergingKernel2(int a[], int b[], int value)
2  {
3          int x := blockIdx.x * blockDim.x + threadIdx.x;
4          if( x = 0 )
5          {
6                  a[x] := a[x] + value;
7                  b[x] := b[x] + value;
8          } else {
9                  if( x % 2 = 0 )
10                 {
11                         a[x] := a[x] + value;
12                         b[x] := b[x] + value;
13                 } else {
14                         a[x] := a[x] - value;
15                         b[x] := b[x] - value;
16                 }
17         }
18 }
```

Der Kernel `divergingKernel2` muss wegen der If-Statements in den Zeilen 4 und 9

[42] Denn der Vektor a besteht, wie aus der Definition des Kernels im Beispiel hervorgeht, aus Integer-Werten, von denen jeder auf einer CUDA-konformen GPU genau vier Byte groß ist. 32 solcher Werte passen damit genau in eine Speicher-Transaktion, die immer 128 Byte umfasst (siehe Unterabschnitt 3.3.6).

in drei Durchläufen abgearbeitet werden. Der erste Durchlauf umfasst die Zeilen 6 und 7. An ihm nimmt lediglich der Thread 0 teil und am Ende des Durchlaufs stehen die Werte des Vektors b im Cache. Der zweite Durchlauf umfasst die Zeilen 11 und 12. An ihm nehmen alle Threads mit geradem, von Null verschiedenem Index teil. Da der Cache mit b gefüllt ist muss nun aber zunächst wieder lesend auf den Device-Memory zugegriffen werden, um a zu holen. Der letzte Durchlauf erfolgt schließlich für die Zeilen 14 und 15 mit den restlichen Threads.

Die Verarbeitung in den einzelnen Durchläufen unterscheidet sich im Grunde also nicht von der in `divergingKernel1`, nur das dieses Mal zwar jeweils alle Werte für die Indizes 0 bis 31 gelesen, gecached und wieder in den Speicher geschrieben werden, das aber nur für einen Teil der Werte auch eine Änderung erfolgt. Es ist leicht zu erkennen, dass `divergingKernel2` statt vier insgesamt 12 Speichertransaktionen pro Warp benötigt.

Falls ein Kernel nun viele verschachtelte bedingte Anweisungen enthält und zudem noch zahlreiche nicht-vereinte Speicherzugriffe beinhaltet kann schnell eine Situation entstehen, in der die GPU im wesentlichen damit beschäftigt ist, auf den Device-Memory zu warten.

3.3.8. Anwendungsbereiche

Eine moderne Grafikkarte stellt einerseits zwar eine sehr große kostengünstige Rechenleistung bereit, andererseits bringt ihre Verwendung im Rahmen des GPGPU aber auch gewisse Einschränkungen mit sich. Denn die Grafikhardware ist in erster Linie für ihre originäre Aufgabe, die schnelle Verarbeitung komplexer 3D-Szenen optimiert. Daher ist nicht jedes Problem, das sich grundsätzlich gut Parallelisieren lässt auch für die Lösung mit einem GPGPU-Programm geeignet[43]. Vier Vorbedingungen müssen erfüllt sein, damit eine effizient arbeitende CUDA-Implementierung gelingen kann, die schneller arbeitet als ein Multithreaded-Programm auf der CPU:

[43] Iterative Algorithmen, bei denen jede Iteration auf dem Ergebnis der vorherigen Iteration beruht sind bspw. für die parallele Bearbeitung grundsätzlich eher schlecht geeignet.

1. Es muss ein hinreichend großer Datenbestand vorhanden sein
   Eine GPU kann ihr volles Leistungspotenzial nur dann entfalten, wenn ihre Kerne gut ausgelastet sind.

2. Die auf den Daten durchzuführenden Operationen müssen möglichst unabhängig voneinander sein

3. Die Daten müssen im Device-Memory so angeordnet werden können, das Speichertransaktionen möglich oft als vereinte Speicherzugriffe erfolgen

4. Innerhalb des Kernels sollte es möglichst wenige bedingte Anweisungen geben, um Code-Divergenzen zu vermeiden

4. Parallelisierung der Lösungsansätze für EVSP

Für die Lösung der Planungsaufgabe, die das Erstellen einer optimalen Mehrdepot-Umlaufplanung umfasst, werden in diesem Kapitel zwei Lösungsansätze vorgeschlagen, die auf den Metaheuristiken ACO bzw. SA basieren. Dabei sollen alle auf dem verwendeten Rechner verfügbaren Rechenkerne genutzt werden[1]. Beide Lösungsansätze sind also auf massiv-parallele Verarbeitung ausgelegt. Damit ist die Hoffnung verbunden, dass mit der Anzahl der parallel untersuchten Lösungen die Wahrscheinlichkeit steigt, das globale Optimum im stark zerklüfteten Lösungsraum eines MDEVSPs zu finden.

Beide Ansätze produzieren ausschließlich gültige Lösungen. Das bedeutet, dass alle erzeugten Lösungen die folgenden Restriktionen einhalten[2]:

- Jeder Umlauf endet in dem Depot, in dem er begonnen hat[3]
- Jeder Umlauf enthält ausschließlich Servicefahrten, die kompatibel zu ihrem Vorgänger und zu ihrem Nachfolger innerhalb des Umlaufs sind
- Die Batteriekapazität eines Fahrzeugs lässt zu jedem Zeitpunkt die Rückfahrt ins Start-Depot zu. Die Batteriekapazität sinkt niemals auf einen Wert kleiner

[1] Wahlweise die Kerne der GPU oder die der CPU. Das auf SA basierende Lösungsverfahren wurde allerdings nur für die CPU implementiert.

[2] Für beide Lösungsansätze lassen sich leicht weitere, praxisrelevante Restriktionen implementieren, wir bspw. eine maximale Zeitdauer für Umläufe oder Rüstzeiten, die an Endhaltestellen einzuplanen sind, damit die nächste Servicefahrt vorbereitet werden kann.

[3] Ein Fahrzeug kann auch während eines Umlaufs ins Depot zurückkehren, um von dort aus weitere Servicefahrten zu absolvieren.

oder gleich Null

- Die Servicefahrten innerhalb eines Umlaufs überschneiden sich zeitlich nicht. Ist eine Verbindungsfahrt nötig, so reicht der Zeitraum zwischen den Servicefahrten dazu aus
- Die jeder Servicefahrt zugeordnete Fahrzeugtypgruppe wird berücksichtigt
- Jede Servicefahrt wird genau einmal durchgeführt

4.1. Parallelisierung des Simulated Annealing

Die im Folgenden beschriebene iterative Vorgehensweise für die Lösungssuche mit SA wird auf allen verfügbaren Kernen unabhängig voneinander durchgeführt, um eine vorab definierte Anzahl separater Lösungen (die „Population“) zu erzeugen. Nach jeder Iteration wird aus der Population die Lösung mit den geringsten Gesamtkosten ausgewählt. Diese Lösung wird gespeichert, falls ihre Gesamtkosten geringer sind als die der nach der vorangegangenen Iteration gespeicherten Lösung.

Jede Lösung der Population wird wie folgt erzeugt:

1. **Erzeugen einer gültigen Startlösung**
   Die Startlösung wird entweder durch zufällige Zuordnung aller Servicefahrten zu Umläufen oder durch Anwendung des in Unterabschnitt 4.1.1 beschriebenen Greedy-Verfahrens erzeugt. Ergebnis dieses Schrittes ist ein vorläufiger Umlaufplan, der alle gegebenen Restriktionen einhält.

2. **Temperatur initialisieren**
   Die aktuelle Temperatur wird auf die Starttemperatur festgelegt.

3. **Modifizieren („mutieren“) der Lösung**
   Die im Schritt 1 erzeugte Lösung (das „Original“) wird durch Anwendung eines oder mehrerer zufällig ausgewählter Mutations-Operatoren modifiziert. Das Original wird also in eine Lösung aus seiner Nachbarschaft überführt. Die

Operatoren werden im Unterabschnitt 4.1.2 erläutert.

4. **Entscheiden, ob die Mutation das Original ersetzt**
   Dazu werden die Gesamtkosten der Mutation ermitteln und mit denen des Originals verglichen. Sind die Gesamtkosten der Mutation geringer als die des Originals, ersetzt die Mutation das Original. Andernfalls erfolgt eine Zufallsentscheidung mit einer Wahrscheinlichkeit, die mittels der Boltzmann-Funktion errechnet wird (dabei wird die aktuelle Temperatur in die Berechnung einbezogen; siehe Unterabschnitt 3.1.1).

5. **Temperatur inkrementieren**
   Die aktuelle Temperatur wird gemäß des vorab festgelegten Temperaturprofils verringert.

6. **Kriterien für Terminierung prüfen**
   Das Verfahren wird beendet, wenn ein definiertes Kriterium erfüllt ist[4]. Andernfalls wird mit Schritt 3 fortgefahren.

4.1.1. Erzeugen einer Startlösung

Startlösungen können auf eine von zwei Arten erzeugt werden:

1. Durch rein zufälliges Auswählen eines passenden Umlaufs:

   a) Wähle aus der Menge der noch nicht verplanten Servicefahrten eine Servicefahrt aus

   b) Erzeuge einen neuen (leeren) Umlauf und ordne diesem die soeben ausgewählte Servicefahrt zu

   c) Wähle aus der Menge der noch nicht verplanten Servicefahrten eine Servicefahrt aus. Das Verfahren ist beendet, wenn keine solche Servicefahrt mehr vorhanden ist

[4] Kriterien für die Terminierung können sein ein Zeitraum, eine Anzahl von Iterationen oder ein vom Benutzer des Systems hervorgerufenes Ereignis wie ein Tastendruck.

d) Prüfe für jeden Umlauf, ob die soeben ausgewählte Servicefahrt unter Einhaltung aller gegebenen Restriktionen darin aufgenommen werden kann. Fahre mit Schritt 1b fort, falls kein solcher Umlauf vorhanden ist

e) Wähle unter den in Schritt 1d gewählten Umläufen einen Umlauf zufällig aus und ordne diesem die Servicefahrt zu

f) Fahre mit Schritt 1c fort

2. Mit einem aus Adler (2014, S. 103 f.) entnommenen Greedy-Verfahren[5]:

a) Wähle aus der Menge der noch nicht verplanten Servicefahrten eine Servicefahrt aus

b) Erzeuge einen neuen (leeren) Umlauf und ordne diesem die soeben ausgewählte Servicefahrt zu

c) Wähle aus der Menge der noch nicht verplanten Servicefahrten eine Servicefahrt aus. Das Verfahren ist beendet, wenn keine solche Servicefahrt mehr vorhanden ist

d) Trage alle Umläufe zusammen, welche die soeben ausgewählte Servicefahrt aufnehmen könnten. Falls kein solcher Umlauf zu finden ist, fahre mit Schritt 2b fort

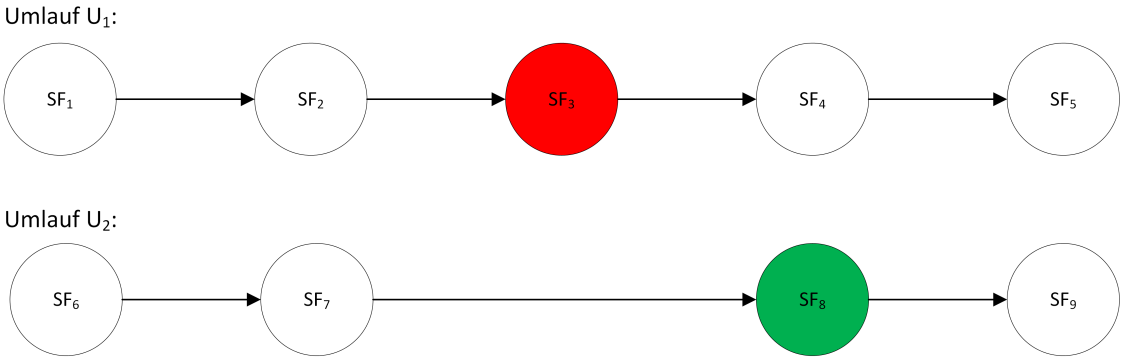
e) Wähle aus den in Schritt 2d zusammen getragenen Umläufen denjenigen aus, bei dem das Hinzufügen der Servicefahrt die geringsten Gesamtkosten verursachen würde und ordne die Servicefahrt diesem Umlauf zu

f) Fahre mit Schritt 2c fort

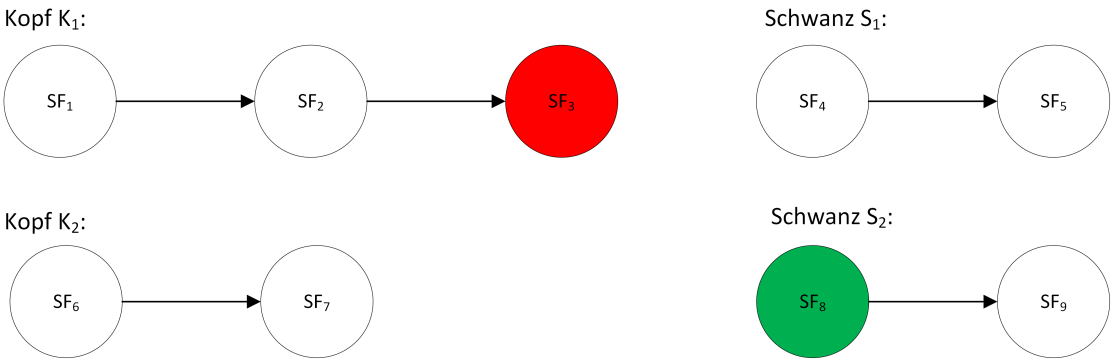
4.1.2. Modifizieren einer Lösung

Für die Erzeugung von Mutationen werden die folgenden Operatoren implementiert:

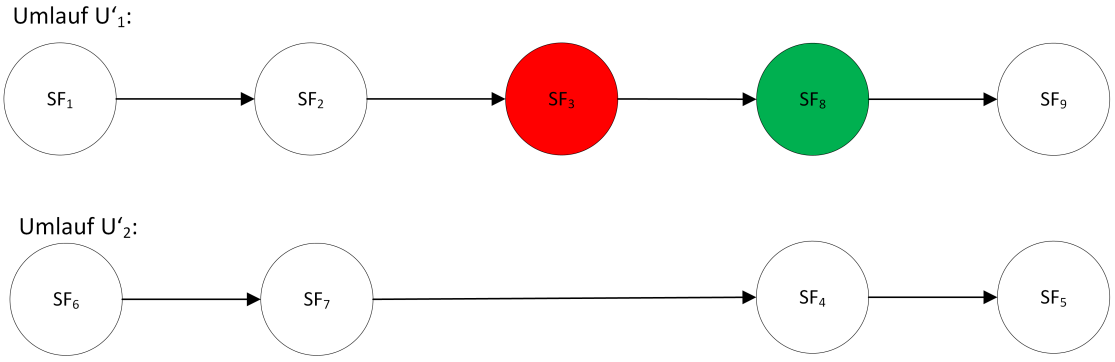
[5] Welches Verfahren zum Einsatz kommt wird vor Programmstart konfigurativ festgelegt (siehe Abschnitt B.3.



Schritt 1: Zufällige Auswahl einer Servicefahrt (hier: SF_3) und einer dazu kompatiblen Servicefahrt (hier SF_8).



Schritt 2: Aufteilen der Umläufe



Schritt 3: Crossover durchführen

Abb. 4.1: Funktionsweise des Crossover-Operators

One-Point-Crossover Bei diesem Operator handelt es sich um eine Abwandlung des in Schöneburg u. a. (1994, S. 198 f.) beschriebenen 'one-point-crossover'. Der grundsätzliche Ablauf ist in Abb. 4.1 dargestellt. Zunächst wird eine zufällig bestimmte Servicefahrt SF_1 ausgewählt. Der sie enthaltende Umlauf U_1 wird in zwei Teile gesplittet: Einen Kopf K_1, der als letzte Fahrt SF_1 beinhaltet und einen Schwanz S_1, der mit der Servicefahrt SF_1' beginnt, welche ursprünglich direkt auf SF_1 folgte. Sodann werden aus der Menge aller Servicefahrten diejenigen Fahrten ausgewählt, die zu SF_1 kompatibel sind. Daraus wird per Zufall eine Fahrt SF_2 bestimmt. Der SF_2 enthaltende Umlauf U_2 wird ebenfalls in einen Kopf K_2 und einen Schwanz S_2 aufgeteilt wird, wobei S_2 als erste Fahrt SF_2 enthält und K_2 mit der Fahrt SF_2' endet, die ursprünglich unmittelbar vor SF_2 durchgeführt wurde. Es werden nun durch Kombination von K_1 mit S_2 und von K_2 mit S_1 zwei neue Umläufe U_1' und U_2' gebildet, sofern dabei alle Restriktionen eingehalten werden können.

Move Servicetrip Eine zufällig ausgewählte Servicefahrt SF wird aus dem Umlauf U_1, dem sie zugeordnet ist, entfernt und in den Umlauf U_2 verschoben. U_2 wird dabei entweder per Zufall oder auf Basis der Gesamtkosten aus der Menge der Umläufe ausgewählt, welche SF aufnehmen können[6]. Bei der Auswahl nach Kosten wird also der Umlauf gewählt, bei dem das Hinzufügen von SF zu den geringsten Gesamtkosten führt (Greedy-Verfahren).

Delete Roundtrip (Random) Ein zufällig ausgewählter Umlauf wird aus der Planung entfernt. Die darin enthaltenen Servicefahrten werden nach den für den Operator 'Move Servicetrip' beschriebenen Verfahren in andere Umläufe eingefügt.

Delete Roundtrip (Worst Cost Relation) Es wird der Umlauf aus der Planung entfernt, der das schlechteste Verhältnis zwischen Gesamtkosten und den nur von den Servicefahrten verursachten Kosten aufweist[7]. Auch hierbei werden die

[6] Welche der beiden Varianten verwendet wird kann konfigurativ festgelegt werden (siehe Abschnitt B.3).

[7] Dieses Kostenverhältnis ist an den in Schnieder (2015, S. 114) definierten Umlaufwirkungsgrad angelehnt und wird ermittelt, indem die durch den Umlauf verursachten Gesamtkosten berechnet werden (inklusive der Anschaffungskosten des eingesetzten Fahrzeugs). Außerdem werden die zeit- und die streckenabhängigen Kosten aller Servicefahrten aufsummiert (exklusive der

frei gewordenen Servicefahrten verteilt, wie für 'Move Servicetrip' beschrieben.

Delete Roundtrip (Least Servicetrips) Es wird der Umlauf aus der Planung entfernt, der die geringste Anzahl von Servicefahrten beinhaltet. Die Verteilung der frei gewordenen Servicefahrten erfolgt wie gehabt.

4.2. Parallelisierung der Ant Colony Optimization

Wie der auf SA basierende Lösungsansatz arbeitet auch der auf ACO beruhende Ansatz iterativ

1. Probleminstanz in einen Konstruktionsgraphen überführen
2. Parallele Exploration des Konstruktionsgraphen, um Population von Lösungen zu erzeugen
3. Bewerten der einzelnen Lösungen
4. Vergleichen der bewerteten Lösungen und Auswahl der besten Lösung aus der in Schritt 2 generierten Population
5. Kantengewichte aktualisieren (Pheromonspuren erzeugen)
6. Falls Terminierungskriterium nicht erfüllt: Weiter mit Schritt 2

Im Folgenden werden die Schritte, die allesamt parallelisierbar sind, im Einzelnen erläutert.

4.2.1. Konstruktionsgraphen erzeugen

Die zu lösende Probleminstanz muss für die Anwendung der ACO in genau einen Konstruktionsgraphen überführt werden (siehe Unterabschnitt 3.1.2), der beim Er-

Anschaffungskosten) und in Relation zu den Gesamtkosten gesetzt. Das Ergebnis ist ein Wert der >= 1 ist, wobei der optimale Wert 1 praktisch nicht erreichbar ist. Im Gegensatz zum Umlaufwirkungsgrad werden also die Anschaffungskosten des eingesetzten Fahrzeugs einbezogen.

zeugen von Lösungen und dem Aktualisieren der Kantengewichte parallel genutzt wird. Dazu werden zunächst alle benötigten Knoten erzeugt[8]:

Wurzelknoten (RootNode) Der Wurzelknoten dient als Ausgangspunkt bei der Erzeugung neuer Lösungen. Es wird genau ein solcher Knoten erzeugt.

Fahrzeugtyp-Depot-Knoten (VehicleTypeDepotNode) Für jede mögliche Kombination aus einem Depot und einem Fahrzeugtyps wird ein VehicleTypeDepotNode erzeugt. Er repräsentiert die Ausrückfahrt eines Fahrzeugs mit einem bestimmten Fahrzeugtyps aus einem bestimmten Depot[9].

Servicefahrt-Knoten (ServiceTripNode) Für jede Servicefahrt der Probleminstanz wird eine ServiceTripNode erzeugt. Sie repräsentiert die Durchführung der Servicefahrt.

Ladestations-Knoten (ChargingStationNode) Für jede Haltestelle der Probleminstanz, an welcher das Aufladen von Fahrzeugen möglich ist, wird eine ChargingStationNode erzeugt. Sie steht für einen Aufladevorgang an einer konkreten Ladestation[10].

Sodann werden die Kanten des Graphen erzeugt. Da die Anzahl der Kanten bei umfangreichen Probleminstanzen sehr groß sein kann ist es sinnvoll, diesen Schritt zu parallelisieren. Das ist deshalb gut möglich, weil der Konstruktionsgraph sowohl beim Erzeugen der Pfade (Lösungen) als auch beim erneuten Durchlaufen der Pfade zur Aktualisierung der Kantengewichte immer nur in eine Richtung durchwandert wird. Es muss immer nur möglich sein, von einem Knoten zum nächsten zu wechseln, nie muss zum vorher besuchten Knoten zurückgekehrt werden (mehr dazu in Unterabschnitt 4.2.2 und in Unterabschnitt 4.2.5). Daher kann jeder Knoten beim

[8] Dieser Schritt kann zwar parallelisiert werden, aber da die Erzeugung der Knoten selbst bei großen Probleminstanzen sehr schnell vonstatten geht, wäre der Nutzen der Parallelverarbeitung eher gering. Zudem werden die Knoten für jede Probleminstanz nur einmal beim Start des Lösungsverfahren erzeugt, weshalb dies sequentiell geschieht.

[9] Und zusätzlich auch zwischenzeitliche Depot-Aufenthalte des Fahrzeugs während eines Umlaufs.

[10] Auch Depots werden dabei als Haltestellen angesehen. Daher wird für jedes Depot auch eine ChargingStationNode erzeugt, um die Aufladevorgänge zu repräsentieren, die während eines Umlaufs in einem Depot stattfinden. Der Aufladevorgang, der nach Abschluss eines Umlaufs im Depot durchgeführt wird um das Fahrzeug für den nächsten Betriebstag vorzubereiten wird im Konstruktionsgraphen jedoch nicht berücksichtigt, da dies weder notwendig noch sinnvoll ist. Er wird lediglich beim Berechnen der Gesamtkosten eines Umlaufs berücksichtigt.

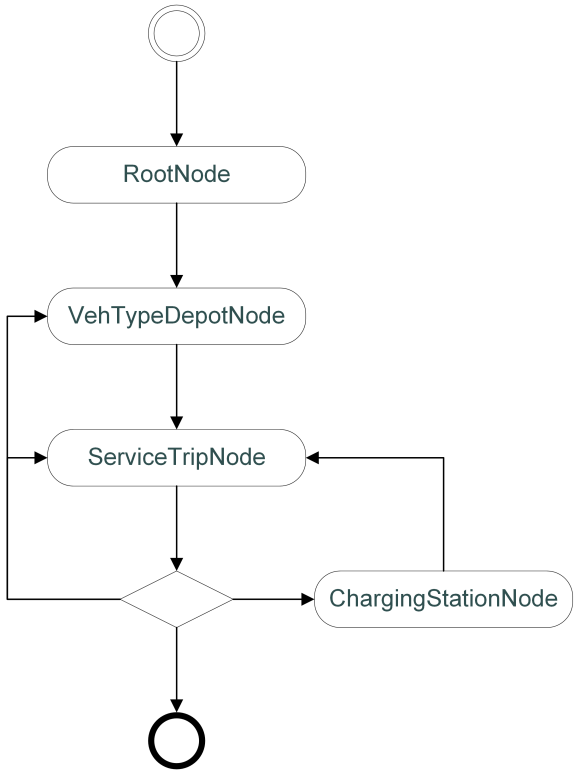


Abb. 4.2: Mögliche Verknüpfungen im Konstruktionsgraphen der ACO

Erzeugen aller von ihm ausgehenden Kanten separat betrachtet werden.

Die Abb. 4.2 zeigt alle im Graphen möglichen Transitionen. Entsprechend werden für jeden Knoten die im folgenden aufgelisteten Kanten erzeugt:

- **RootNode:** Erzeuge eine Kante zu jeder VehicleTypeDepotNode.
- **VehicleTypeDepotNode:** Erzeuge eine Kante zu jeder ServiceTripNode, die mit dem der VehicleTypeDepotNode zugeordneten Fahrzeugtypen durchgeführt werden kann[11] und deren Starthaltestelle vom Depot aus erreichbar ist.
- **ServiceTripNode:** Erzeuge eine Kante zu...
  - jeder ServiceTripNode, die eine kompatible Servicefahrt repräsentiert.
  - jeder ChargingStationNode, die für eine Ladestation steht, welche von der Endhaltestelle der Servicefahrt aus erreichbar ist[12].

[11] Das ist dann der Fall, wenn der Fahrzeugtyp in der Fahrzeugtypgruppe enthalten ist, die der Servicefahrt zugeordnet ist.

[12] Entweder, weil die beiden Haltestellen identisch sind (die Ladestation sich also an der Endhaltestelle befindet), oder weil es eine Verbindungsfahrt gibt, die von der Endhaltestelle zur

– jeder VehicleTypeDepotNode, deren zugeordneter Fahrzeugtyp zur Fahrzeugtypgruppe der Servicefahrt passt und deren Depot von der Endhaltestelle der Servicefahrt aus erreichbar ist.

- **ChargingStationNode:** Erzeuge eine Kante zu jeder ServiceTripNode, deren Starthaltestelle von der Ladestation aus erreichbar ist.

In Abb. 4.3 ist ein (simplifiziertes) Beispiel für einen Konstruktionsgraphen zu sehen. Der Konstruktionsgraph einer realen Probleminstanz ist natürlich erheblich umfangreicher und komplexer[13].

4.2.2. Lösungen erzeugen

Eine neue Generation von Lösungen wird erzeugt, indem der Graph ausgehend von der RootNode durchlaufen wird. Dabei erfolgt die Auswahl des nächsten Knotens per Zufall aus der Menge der ausgehenden Kanten E_{Out} des aktuellen Knotens. Die Wahrscheinlichkeit dafür, dass die Kannte e_k ausgewählt wird, hängt dabei vom Kantengewicht w_k ab und beträgt

$$p_k = \frac{w_k}{(\sum_{i=1}^{n} w_i)} \text{ mit } n \in \mathbb{N} \text{ und } k \leq n.$$

Durch den Umstand, dass jede Servicefahrt in einer Lösung genau einmal verplant werden muss, ist der Konstruktionsgraph frei von Zyklen. Weil die einzelnen Lösungen voneinander unabhängig sind, können sie in parallelen Threads erzeugt werden. Dabei wird für jede einzelne Lösung wie folgt eine Menge von Umläufen generiert:

1. Wähle eine der ausgehenden Kanten der RootNode. Mache die VehicleTypeDepotNode, auf welche diese Kante verweist, zum aktuellen Knoten.

2. Erzeuge einen neuen Umlauf. Das Startdepot und der Fahrzeugtyp des Umlaufs

Ladestation führt.

[13] Für die größte im Rahmen dieser Arbeit verwendete Probleminstanz (`sample_real_10710_SF_140_stoppoints.txt`, siehe **??**) wird ein Konstruktionsgraph mit 10.744 Knoten und 44.842.065 Kanten erzeugt.

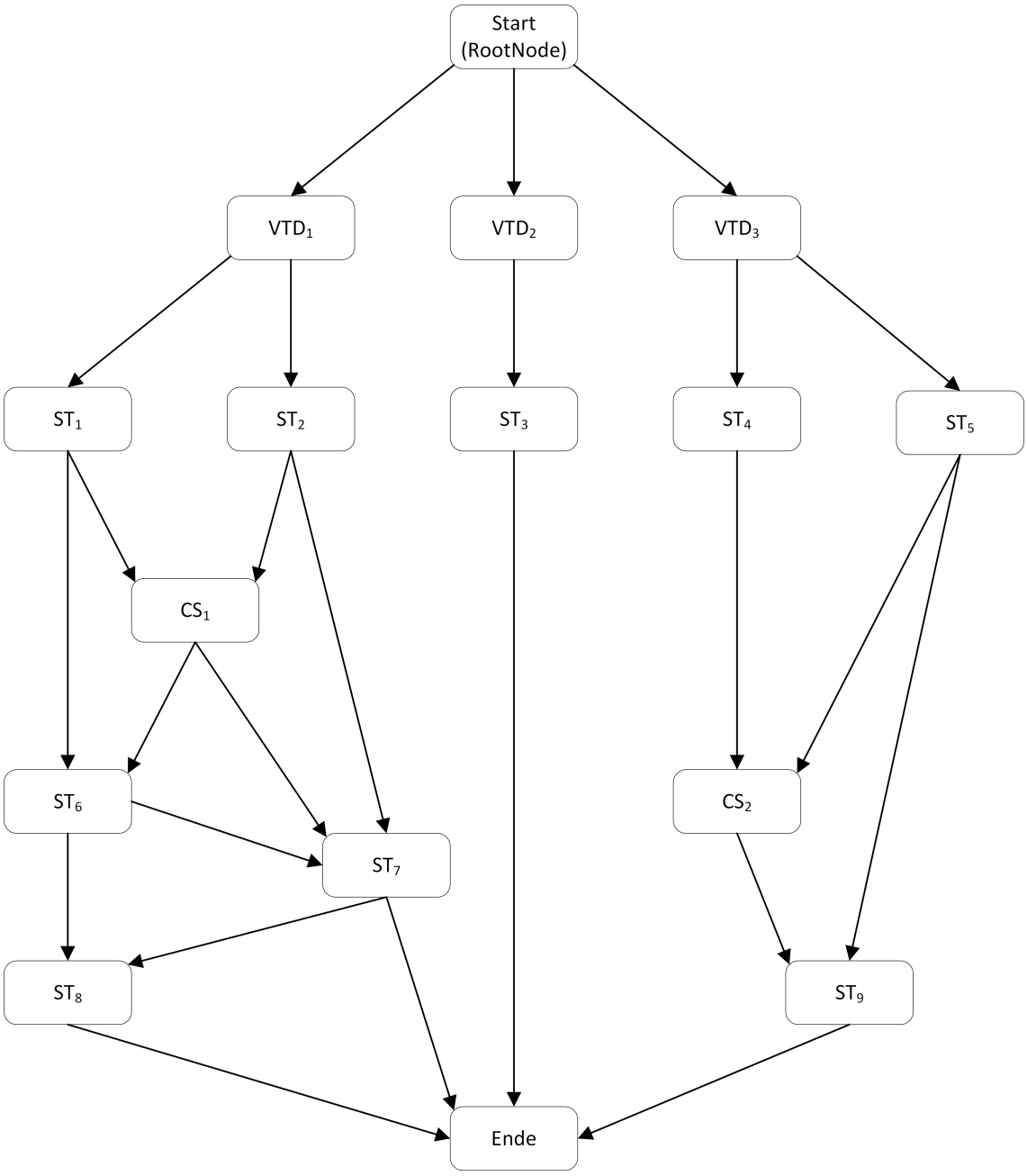


Abb. 4.3: Beispiel eines ACO-Graphen

Es ist nur eine Auswahl der Kanten eingetragen, damit die Übersichtlichkeit gegeben bleibt. Tatsächlich existieren noch Kanten von jeder ServiceTripNode zu jeder VehicleTypeDepotNode sowie Kanten von den ServiceTripNodes zu fast allen ChargingTripNodes und vice versa. Nicht alle Kanten sind zu jedem Zeitpunkt auch aktiv. Bspw. werden Kanten, die zu bereits durchgeführten Servicefahrten führen deaktiviert. ST = Servicefahrt (Service-Trip), CS = Ladestation (Charging-Station-Node), VTD = Depot/Fahrzeugtyp (Vehicletype-Depot-Node)

werden durch die VehicleTypeDepotNode definiert.

3. Initialisiere die Batteriekapazität des Fahrzeugs, mit dem der aktuelle Umlauf durchgeführt wird

4. Bilde für den aktuellen Knoten die Menge der auswählbaren Kanten. Dies sind alle Kanten, für die folgende Bedingungen zutreffen:

   - Falls der Zielknoten eine ServiceTripNode ist, darf die Servicefahrt noch nicht verplant sein

   - Falls der Zielknoten eine ChargingStationNode ist, muss die aktuelle Batteriekapazität des Fahrzeugs unterhalb einer definierten Grenze (der „Ladeschwelle“) liegen[14]

   - Nach Durchführung der durch den Zielknoten definierten Aktivität muss die Batteriekapazität noch ausreichen, um zumindest zum Depot zurückkehren zu können[15].

5. Falls die Menge der auswählbaren Kanten nicht leer ist wird mit dem Schritt 6 fortgefahren. Andernfalls muss der aktuelle Umlauf beendet werden. Das weitere Vorgehen richtet sich dann danach, ob noch unverplante Servicefahrten vorhanden sind oder nicht[16]:

   - Wenn noch weitere Servicefahrten einzuplanen sind: Schließe den aktuellen Umlauf ab und mache die RootNode zum aktuellen Knoten. Fahre dann mit dem Schritt 1 fort.

   - Wenn bereits alle Servicefahrten verplant sind ist die Lösung fertiggestellt und es kann mit der in Unterabschnitt 4.2.3 beschriebenen Bewertung fortgefahren werden.

[14] Auf diese Weise wird verhindert, dass Umläufe entstehen, die unnötige Aufenthalte an Ladestationen enthalten.

[15] Dabei muss selbstverständlich auch der Energieverbrauch für evtl. nötige Verbindungsfahrten berücksichtigt werden. So wird verhindert, dass ein Fahrzeug während der Durchführung eines Umlaufs an einer Haltestelle strandet, weil die Batterie zu stark entladen ist.

[16] Zweckmäßigerweise kann eine Liste der noch zu verplanenden Servicefahrten gepflegt werden.

6. Wähle aus der Menge der auswählbaren Kanten eine Kante aus. Der Zielknoten dieser Kante wird zum aktuellen Knoten. Füge diesen Knoten zum aktuellen Umlauf hinzu und aktualisiere die Batteriekapazität[17]. Fahre mit Schritt 4 fort.

Jede so generierte Lösung enthält für jede zu absolvierende Servicefahrt eine ServiceTripNode. Alle Depotaufenthalte und alle Aufenthalte an Ladestationen sind in den Umläufen enthalten. Leerfahrten sind im Pfad allerdings nicht explizit modelliert, da diese jederzeit aus der Abfolge der Knoten im Umlauf rekonstruiert werden können[18].

4.2.3. Lösungen bewerten

Eine Lösung wird bewertet, indem jeder ihrer Umläufe von vorne nach hinten durchlaufen wird. Dabei werden die folgenden Teilkosten ermittelt:

- Die zeitabhängigen Kosten der einzelnen Umläufe. Dazu wird jeweils bestimmt, wann ein Umlauf beginnt und wie lange er dauert (vom Beginn der Ausrückfahrt bis zum Ende der abschließenden Einrückfahrt).
- Die Anschaffungskosten für die benötigten Fahrzeuge.
- Die streckenabhängigen Kosten jedes einzelnen besuchten Knotens[19].
- Die durch Aufladungen entstehenden Kosten (einschließlich der Aufladung, die am Ende des Betriebstags im Depot stattfindet, um das Fahrzeug für den nächsten Betriebstag vorzubereiten).

Durch Summieren dieser Teilkosten ergeben sich die Gesamtkosten der Lösung. Zu-

[17] Bei einer ServiceTripNode: Subtrahieren des Verbrauchs für eine etwaige Verbindungsfahrt und für die Durchführung der Servicefahrt. Bei einer VehicleTypeDepotNode: Subtrahieren des Verbrauchs für eine etwaige Einrückfahrt. Bei einer ChargingStationNode: Batteriekapazität auf fahrzeugspezifische Maximalkapazität setzen.

[18] Auf diese Weise wird Speicherplatz eingespart und die Anzahl der Zugriffe auf den Speicher wird reduziert, was insbesondere für das Erstellen von Lösungen auf der GPU relevant ist.

[19] Bei einer ServiceTripNode setzt sich die Strecke bspw. aus der während der Servicefahrt zurückgelegten Strecke und der bei einer etwaigen vorhergehenden Verbindungsfahrt zurückgelegten Strecke zusammen. Bei einer ChargingStationNode entstehen streckenabhängige Kosten durch Leerfahrten, die von der Endhaltestelle der vorangegangenen Servicefahrt zur Ladestation und dann weiter zur Starthaltestelle der folgenden Servicefahrt führen.

sätzlich werden weitere Kennzahlen wie die Anzahl der Umläufe und das in Unterabschnitt 4.1.2 bereits beschriebene Kostenverhältnis jedes einzelnen Umlaufs ermittelt. Die Gesamtkosten und alle Kennzahlen werden gespeichert, damit sie beim Auswählen der besten Lösung und beim Aktualisieren der Kantengewichte verwendet werden können.

4.2.4. Beste Lösung auswählen

Die beste Lösung aus der gerade erzeugten und bewerteten Generation ist die mit den geringsten Gesamtkosten. Sie wird markiert und als einzige Lösung in die nächste Generation übernommen[20]. So bleibt die beste bisher gefundene Lösung erhalten.

4.2.5. Kantengewichte aktualsieren

Zu jeder ermittelten Lösung S_k wird für jede verwendete Kante i eine neue Pheromonspur $T_{i,k}$ berechnet. $T_{i,k}$ hängt von drei Parametern ab, deren Gewichtung konfigurierbar ist, und die jeweils Werte zwischen 0 und 1 annehmen können:

- Von den **Gesamtkosten** der Lösung (C_k). Zur Normierung auf das Intervall [0;1] wird dabei das Verhältnis von C_k zu den Gesamtkosten der besten (C_{best}) und der schlechtesten bislang gefundenen Lösung (C_{worst}) berechnet:

$$T_k^C = \frac{C_{worst} - C_k}{C_{worst} - C_{best}} \tag{4.1}$$

- Von der **Anzahl der Umläufe** (V_k)[21]. Auch hier findet die Normierung auf das Intervall [0;1] durch den Bezug auf die beste (V_{best}) und die schlechteste (V_{worst}) bislang gefundene Lösung statt:

$$T_k^V = \frac{V_{worst} - V_k}{V_{worst} - V_{best}} \tag{4.2}$$

[20]Per Konfigurationsparameter kann auch festgelegt werden, dass dies nicht geschieht.
[21]Die Anzahl der Umläufe entspricht der Anzahl der eingesetzten Fahrzeuge.

- Vom **Umlaufkostenverhältnis** des Umlaufs i, in dem die Kante verwendet wird ($CCR_{i,k}$, „Circulation Cost Ratio“). Zur Normierung wird $CCR_{i,k}$ in Relation zum besten (CCR_{best}) und zum schlechtesten (CCR_{worst}) bislang gefundenen Umlaufkostenverhältnis gebracht:

$$T_{i,k}^{CCR} = \frac{CCR_{worst} - CCR_{i,k}}{CCR_{worst} - CCR_{best}} \tag{4.3}$$

Mit α, β und γ als Gewichtungsfaktoren für die einzelnen Parameter ergibt sich für $T_{i,k}$:

$$T_{i,k} = \alpha \cdot T_k^C + \beta \cdot T_k^V + \gamma \cdot T_{i,k}^{CCR} \tag{4.4}$$

Die Aktualisierung der Kantengewichte erfolgt in drei Schritten:

1. **Abschwächen der bereits vorhandenen Spuren**

   Alle Kantengewichte werden mit dem Abschwächungsfaktor f (Fading Factor, $0 \leq f \leq 1$) multipliziert. Das sorgt dafür, dass einmal ausgebrachte Spuren nicht unbegrenzt lange existieren. Dadurch steigt die Chance, dass sich neue Lösungen mit geringen Gesamtkosten durchsetzten können (Dorigo u. a. 2004). Dieser Schritt kann für alle Kanten durchgeführt parallel werden.

2. **Berechnen der neuen Pheromonspuren**

   Jede Lösung der Population hinterlässt eine neue Pheromonspur. Diese Spuren werden berechnet und akkumuliert. Dazu werden die Umläufe der Lösungen erneut durchlaufen, die jeweilige Pheromonspur $T_{i,k}$ wird gemäß Gleichung 4.4 berechnet und für die einzelnen Kanten aufaddiert. Es ergibt sich die Summe der neuen Pheromonspuren für jede Kante eines Umlaufs:

$$T_i = \sum_{k=1}^{n} T_{i,k} \tag{4.5}$$

   Dieser Schritt kann nur teilweise parallel erfolgen, denn die Additionsoperationen für die einzelnen Kanten müssen serialisiert werden, um Race Conditions

zu vermeiden.

3. **Aktualisieren der Kantengewichte**

   Die im vorangegangenen Schritt berechneten Pheromonspuren werden zu den Kantengewichten hinzuaddiert. Das kann für alle Kanten parallel geschehen.

Durch die Faktoren f (Fading), α (Gewichtung der Gesamtkosten), β (Gewichtung der Umlaufanzahl) und γ (Gewichtung des Umlaufkostenverhältnisses) ist eine flexible Steuerung der Berechnung neuer Pheromonspuren möglich[22].

[22]Leider reichte die für diese Arbeit verfügbare Bearbeitungszeit für das Experimentieren mit diesen Faktoren nicht mehr aus.

5. Softwaretechnische Umsetzung

Bei dem im Rahmen der vorliegenden Arbeit erstellten Programm handelt es sich um eine Konsolenanwendung für Microsoft Windows, die Umlaufplanungsprobleme für E-Busse (SDEVSP und MDEVSP) lösen kann[1]. Die Daten der zu lösenden Probleminstanz werden aus einer Datei (dem sog. „Sample-File“, siehe Unterabschnitt 6.1.1) eingelesen, in eine programminterne Darstellung überführt und schließlich durch Anwendung eines der beiden in Kapitel 4 vorgestellten Ansätze gelöst.

Um den Speicherverbrauch des Programms (vor allem im GPU-Betrieb) zu reduzieren gelten für die zu lösende Probleminstanz die folgenden Einschränkungen[2]:

- maximal 10.710 Servicefahrten
- maximal 10.100 Verbindungsfahrten
- maximal 207 Haltestelleen
- maximal 70 Depots
- maximal 74 Linien
- maximal 9 Fahrzeugtypen
- maximal 9 Fahrzeugtypgruppen

Darüber hinaus gelten die folgenden Rahmenbedingungen:

[1] Die Anwendung löst auch konventionelle Umlaufplanungsprobleme (SDVSP und MDVSP), wenn die Konfiguration entsprechend angepasst wird. Siehe dazu Abschnitt B.3.

[2] Durch das Ändern der betreffenden Konstanten im Sourcecode (`Evsp.CudaSolver\EvspLimits.h`) und Neukompilieren des Programms können die Einschränkungen verändert werden.

- Servicefahrten beginnen und enden an regulären Haltestellen, nicht jedoch an Depots (ein Depot ist also keine reguläre Haltestelle; es sind Ein- und Ausrückfahrten nötig)
- Ein Umlauf endet immer in dem Depot, in dem er begonnen hat
- Die Linienzugehörigkeit von Servicefahrten wird nicht berücksichtigt
- Die Anzahl der in einem Depot für die Fahrzeuge verfügbaren Abstellplätze wird nicht berücksichtigt. Es wird davon ausgegangen, dass immer ausreichend viele Abstellplätze verfügbar sind
- Eine evtl. nötige Rüstzeit wird nicht berücksichtigt. Sollte also Zeit nötig sein, damit ein Fahrer vor Fahrtantritt gewisse Vorbereitungen treffen kann, so wird dies vom Programm ignoriert
- Die Kosten für die Aufladung, die nach Abschluss eines Umlaufs im Depot erfolgt, um das Fahrzeug auf den nächsten Betriebstag vorzubereiten, werden den Gesamtkosten des Umlaufs zugeschlagen
- Die Fehlerbehandlung des Programms ist einfach aufgebaut. Daher führen Fehler in den Sample-Files, etwaiger Speichermangel oder evtl. noch vorhandene Programmierfehler meist zum Programmabbruch. Es erfolgt i.d.R. eine kurze Textmeldung im Konsolenfenster, die auf mögliche Fehlerursachen hinweist

5.1. Verwendete Tools, Libraries und Frameworks

Die Entwicklung fand auf dem in Unterabschnitt 6.1.2 beschriebenen Testrechner statt. Dabei wurden die unten aufgeführten Werkzeuge und Bibliotheken verwendet.

Entwicklungsumgebung	**Microsoft Visual Studio Enterprise 2015** *Version 14.0.25431.01 Update 3*
Erweiterungen der Entwicklungsumgebung	**InstallShield LimitedEdition** Ermöglicht das Erstellen von Installationsroutinen mit Visual Studio. Das Setup-Projekt wurde mit InstallShield erstellt.
	NVIDIA Nsight Visual Studio Edition *Version 5.2.0.16321* Nsight ist Bestandteil von CUDA 8.0 und stellt einen Debugger sowie einen Profiler für CUDA bereit.
Frameworks	**NVIDIA CUDA 8** *Version 8.0.61* CUDA ist ein Framework, dass die Programmiersprache C um Konzepte und APIs für GPGPU erweitert (siehe Unterabschnitt 3.3.1). https://developer.nvidia.com/cuda-downloads
Libraries	**Standard Template Library (STL)** Die STL ist eine C++-Bibliothek, die Bestandteil des C++-Standards ist. Sie stellt zahlreiche Klassen und Funktionen bereit. Bspw. Container-Klassen wie die im Programm häufig verwendete Klasse `Vector`. https://msdn.microsoft.com/de-de/library/cscc687y.aspx
	Boost *Version 1.64.0* Boost ist eine umfangreiche, auf der STL aufbauende C++-Bibliothek, aus der verschiedene String-Funktionen verwendet werden. http://www.boost.org/

5.2. Programmstruktur

Das Programm wurde in den Programmiersprachen C und C++ erstellt. Soweit möglich wurden dabei die Paradigmen der Objektorientierten Programmierung angewendet[3]. Das Programm besteht aus 114 Klassen mit insgesamt 17.980 Zeilen Sourcecode[4], die auf vier Module verteilt sind[5]:

EVSP.Console enthält das Hauptprogramm (die Funktion `main`) sowie die Klassen für das Einlesen der Sample-Files und für das Handling der Programm-Konfiguration. Das Modul enthält weiterhin die Funktionalität für die Verarbeitung der Benutzereingaben, für eine rudimentäre Fehlerbehandlung und für den Start des Solver.

EVSP.Model stellt für jede in den Sample-Files vorkommende Entität eine Modellklasse bereit. Diese Modellklassen kapseln die Fachlichkeit der betreffenden Entität. Das Hauptprogramm erzeugt während des Einlesens von Sample-Files Instanzen dieser Modellklassen.

EVSP.CudaSolver Dieses Modul stellt die eigentliche Funktionalität der Anwendung bereit. Es ist die Implementierung der beiden in Kapitel 4 vorgestellten Lösungsansätze. Also insgesamt drei Solver (je einer für ACO auf der CPU und auf der GPU sowie ein Solver für SA auf der CPU) sowie alle damit direkt zusammenhängenden Klassen wie Container-Klassen für CUDA und für CUDA optimierte Modellklassen.

[3] Leider ist es im Rahmen dieser Arbeit nicht möglich, näher auf die Konzepte der Objektorientierung einzugehen. Sie werden hier als bekannt vorausgesetzt. Die Wikipedia-Artikel https://de.wikipedia.org/wiki/Objektorientierung und https://de.wikipedia.org/wiki/Objektorientierte_Programmierung bieten einen recht guten Überblick, der für das Verständnis der Ausführungen in diesem Kapitel ausreichen sollte.

[4] Die Dokumentation des Programms besteht aus zusätzlichen 3.854 Kommentarzeilen. Die Anzahl der Codezeilen wurde mit dem Open-Source-Programm *cloc* ermittelt, das unter der URL https://github.com/AlDanial/cloc abrufbar ist. Außerdem ist `cloc.exe` auch auf der CD enthalten. Ursprünglich umfasste das Programm auch einen in C# geschriebenen Anteil, der eine GUI zur Verfügung stellen sollte. Aus Zeitmangel konnte dieser Teil jedoch nicht rechtzeitig fertig gestellt werden und wurde daher aus der eingereichten Version der Arbeit wieder entfernt. Es ist möglich, dass in Kommentarzeilen noch auf diesen Teil Bezug genommen wird.

[5] Außerdem existiert noch das Modul `EVSP.CudaSolver.UnitTest`, das eine Reihe von automatisierten Tests (Unit-Tests) für verschiedene Klassen des Moduls `EVSP.CudaSolver` enthält. Das Module `EVSP.CudaSolver.UnitTest` dient jedoch nur dem Test und wird nicht Bestandteil des Executables. Daher ist es in der Aufstellung nicht enthalten.

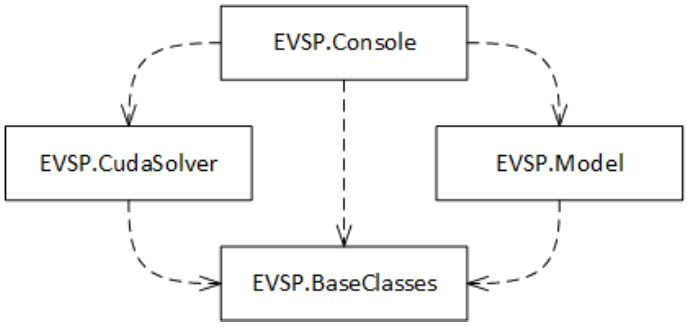


Abb. 5.1: Die Module und ihre Abhängigkeiten

EVSP.BaseClasses enthält Klassen, die grundlegende Funktionen bereit stellen und die in mehreren anderen Modulen benötigt werden. Das sind bspw. Datentypen für Zeitpunkte, Zeiträume, Geldbeträge u.s.w., aber auch Klassen für die Verwaltung der Konfiguration und der Liste bekannter Sample-Files.

Abb. 5.1 zeigt die Abhängigkeitsbeziehungen der Module. Aus dem Modul `EVSP.Console` wird das Executable `EVSP.exe`[6] erzeugt. Die drei anderen Module werden in Library-Files übersetzt und statisch zum Executable hinzu gelinkt[7]. Die der Arbeit beiliegende CD enthält neben den Executables selbstverständlich auch den vollständigen kommentierten Sourcecoude sowie alle von der Entwicklungsumgebung benötigten Files (siehe Anhang A).

5.3. Die Besonderheiten der CUDA-Entwicklung

Bei der Entwicklung eines CUDA-Programms sind einige Besonderheiten zu beachten, von denen die wichtigsten in den folgenden Unterabschnitten erläutert werden.

[6] Wenn das Programm für ein 64bit-Windows übersetzt wird heißt das Executable `EVSP64.exe`.

[7] Diese Vorgehensweise führt zu einer einzelnen ausführbaren Datei. Beim dynamischen Linken wären neben dem Executable `EVSP.exe` noch die drei Dynamic Link Libraries (DLL) `EVSP.Model.dll`, `EVSP.CudaSolver.dll` und `EVSP.BaseClasses.dll` entstanden, was im vorliegenden Kontext keinen Vorteil bietet.

5.3.1. Parameterübergabe an einen Kernel

Alle Daten, die in einem Kernel verwendet werden, müssen als Aufrufparameter des Kernels übergeben werden. Entweder by-value oder als Pointer. Die Übergabe als Referenz ist nicht möglich. Die Aufrufparameter dürfen allerdings eine Gesamtgröße von 4KB nicht überschreiten. Deshalb werden die Daten meist im Heap des Host-Memories erzeugt und dann in den Device-Memory kopiert. Dem Kernel wird der Pointer auf die kopierten Daten übergeben.

Wenn der Kernel beendet ist, müssen die Daten aus dem Device-Memory wieder zurück in den Host-Memory kopiert werden. Aktuelle CUDA-Versionen bieten zwar eine Technik, die als Unified Memory bezeichnet wird und die das Kopieren aus dem Host- in den Device-Memory und vice versa automatisiert, doch leidet nach Landaverde u. a. (2014) die Performance dabei erheblich. Daher wurde im Programm auf die Verwendung von Unified Memory verzichtet[8].

5.3.2. Erzeugen von Datenstrukturen

Klassen und Datenstrukturen im Allgemeinen können auch im Kernel erzeugt werden. Der benötigte Speicher wird im Device-Memory reserviert und kann dann im Kernel verwendet werden. Damit sind jedoch zwei wesentliche Nachteile verbunden:

1. Das Reservieren von Speicher dauert auf der GPU (also im Kernel) länger als das Reservieren des Speichers auf der CPU und das anschließende Kopieren in den Device-Memory

2. Der vom Kernel reservierte Speicher kann nicht in den Host-Memory kopiert werden. Nachdem der Kernel beendet wurde gibt es also keine Möglichkeit, vom Host aus auf die Daten zuzugreifen[9]

Daher werden alle Datenstrukturen i.d.R. vor dem Aufruf des Kernels im Host-

[8] Die Methoden, die das Kopieren erledigen, heißen im Programm i.d.R. `copyToHost` bzw. `copyToDevice`.

[9] Das ist so nicht ganz korrekt, denn es ist zwar möglich, aber mit unverhältnismäßig hohem Aufwand verbunden.

Memory erzeugt und dann in den Device-Memory kopiert. Das bedeutet aber, dass der Speicherbedarf schon vor dem Kernel-Start bekannt sein muss. Dadurch lässt sich der Speichers nicht flexibel nutzen und große Speicherbereiche werden zwar allokiert, aber nie verwendet.

5.3.3. Virtuelle Methoden

Wie bereits erwähnt ist objektorientierte Programmierung unter CUDA nur eingeschränkt möglich. Der CUDA-Compiler unterstützt zwar Klassen und auch die Vererbung und das Überschreiben von Methoden, aber Polymorphie ist praktisch unmöglich. Denn in CUDA können keine virtuellen Methoden verwendet werden. Der Grund dafür ist, dass in C++ für jedes Objekt einer Klasse im Speicher eine Tabelle mit den Sprungzielen der virtuellen Methoden abgelegt wird (Virtual Method Table, VMT). Nun befindet sich das Objekt aber zunächst im Host-Memory und muss vor dem Start eines Kernels in den Device-Memory kopiert werden. Die Sprungziele in der VMT verweisen aber nach wie vor in den Host-Memory, auf den die GPU keinen Zugriff hat.

5.3.4. STL und Boost

Weder die Standard Template Library (STL) noch die Bibliothek Boost können in einem CUDA-Kernel verwendet werden. Alle benötigten Klassen müssen daher selbst entwickelt werden. Das verursacht einen erheblichen Aufwand. Zudem ist es sehr fehleranfällig und geht zulasten der Lesbarkeit und der Wartbarkeit des Codes.

5.3.5. CUDA-Tools, Turnaround-Zeit

Die in CUDA enthaltenen Tools für das Entwickeln und das Debuggen von CUDA-Programmen verhalten sich nicht immer wie erwartet. Sie reagieren zuweilen instabil oder produzieren nicht nachvollziehbare Ergebnisse. Es kommt durchaus auch zu Abstürzen. All dies macht die Entwicklung für CUDA zu einer recht zeitraubenden

Angelegenheit. Erschwerend kommt hinzu, dass die Turnaround-Zeit[10] vor allem im Debug-Mode des Compilers[11] sehr lang ist.

5.4. Die Klassen und ihr Zusammenwirken

Dieser Abschnitt erläutert, auf welche Weise die Klassen zusammenwirken. Dabei wird nur auf die wesentlichen Klassen und Zusammenhänge eingegangen, um dem Leser die Orientierung im Sourcecode zu erleichtern. Detailliertere Informationen finden sich in den Sourcecode-Kommentaren. Sofern nicht anders angegeben befinden sich alle hier erwähnten Klassen im Modul `EVSP.CudaSolver`.

Der Programmablauf gliedert sich in sieben Schritte:

1. **Ermitteln der Konfiguration** im Hauptprogramm[12] durch Auswertung der Kommandozeile und Einlesen des Config-Files[13].

2. **Einlesen des Sample-Files** im Hauptprogramm durch die Klasse `TextFileReader` aus dem Modul `EVSP.Console`. Dabei wird der Inhalt des Files in Instanzen der Modellkassen `EVSP.Model` überführt[14].

3. **Konvertieren der Modellklassen-Instanzen** in eine für die Solver optimierte programminterne Darstellung. Die Konvertierung und das Management

[10] Als Turnaround-Zeit bezeichnet man in der Softwareentwicklung den Zeitraum, der typischerweise vergeht, bis ein Programmierer die Auswirkungen einer Änderung am Code testen kann. Sie setzt sich aus der Zeit für das Compilieren und der Zeit für das Starten des Programms zusammen.

[11] Im Debug-Mode erzeugt ein Compiler zusätzlichen Maschinencode, um die Fehlersuche in einem Debugger zu erleichtern. Außerdem werden sonst durchgeführte Optimierungen unterlassen. Das macht das erzeugte Programm i.d.R. deutlich langsamer. Beim CUDA-Compiler scheint dieser Effekt jedoch außergewöhnlich stark ausgeprägt zu sein.

[12] Das Hauptprogramm befindet sich im File `Main.cpp` im Verzeichnis `EVSP.Console`.

[13] Die Konfiguration wird beim Einlesen des Sample-Files noch um die dort enthaltenen Parameterwerte ergänzt und teilweise auch dadurch verändert.

[14] Sollten die Haltestellennamen im Sample-File anonymisiert sein werden sie mithilfe der Klasse `RandomValuePicker` durch zufällig ausgewählte Straßennamen ersetzt, um die Lesbarkeit der später ausgegebenen Lösung zu verbessern. Diese Option wird mit dem Konfigurationsparameter `use_random_streetnames` gesteuert. Die Straßennamen stammen aus dem File `strassen_osm.txt` (Quelle: OpenStreetMap, http://www.openstreetmap.org/ bzw. http://www.datendieter.de/item/Liste_von_deutschen_Strassennamen_.csv; Config-Parameter `streetfile`).

der Modellobjekte ist in der Klasse `CuProblem` implementiert. Sie enthält Instanzen der Klassen `CuEmptyTrips`, `CuServiceTrips`, `CuStops`, `CuVehicleTypes` und `CuVehicleTypeGroups`.

4. **Parametrisieren und instanziieren des Solvers** in Abhängigkeit von dem Lösungsansatz, den der Benutzer gewählt hat. Dieser Schritt wird von der Klasse `SolverFabric` erledigt. Es wird entweder ein Objekt der Klasse `CuAcoSolver` (für ACO) oder eines der Klasse `CuSaSolver` (für SA) erzeugt und mit allen benötigten Daten versorgt. Für ACO beinhaltet dieser Schritt auch das Erzeugen des Konstruktionsgraphen (Klasse `CuConstructionGraph`).

5. **Starten des Solvers** durch das Hauptprogramm. Der im vorangegangenen Schritt erzeugte Solver generiert solange neue Lösungen, bis eines der in der Konfiguration festgelegten Terminierungskriterien erreicht ist. Dabei werden die Gesamtkosten und die Fahrzeuganzahl der besten gefundenen Lösungen fortlaufend auf der Konsole ausgegeben und in eine Ergebnisdatei geschrieben[15].

6. **Konvertieren der optimalen Lösung** aus der vom Solver verwendeten programminternen Darstellung in ein von Menschen lesbares Format. Dieser Schritt umfasst bspw. das „Übersetzen“ der einzelnen in den Umläufen enthaltenen Aktionen.

7. **Ausgabe der optimalen Lösung** in eine Ergebnis-Datei.

Aus zeitlichen Gründen konnten die Schritte 6 und 7 leider nicht mehr implementiert werden. Sie sind im Rahmen der vorliegenden Arbeit nicht notwendig[16]. Für ein in der Praxis einsetzbares Programm wären sie allerdings unabdingbar, machen sie doch die erzeugte Umlaufplanung erst für das ÖPNV-Unternehmen verfügbar. Deshalb wurden sie hier mit aufgeführt.

[15] Diese Datei wird in einem Unterverzeichnis des Arbeitsverzeichnisses mit dem Namen `result` erzeugt. Die Datei wird im CSV-Format gespeichert und kann mit Microsoft Excel geöffnet werden. Der Dateiname lautet `result.csv`. Sollte bereits eine Datei mit diesem Namen vorhanden sein, so wird dem Dateinamen eine fortlaufende Zahl angehängt.

[16] Zur Kontrolle der Lösungen wird die Ausgabe nicht benötigt, denn deren Validität wird programmintern von Check-Routinen geprüft. Bei Probleminstanzen mit mehreren Hundert Servicefahrten wären „manuelle“ Kontrollen auch nur stichprobenartig möglich.

6. Ergebnisse und Erkenntnisse

In diesem Kapitel werden die von der Konsolenanwendung EVSP erzeugten Lösungen ausgewertet und interpretiert. Auf dieser Grundlage erfolgt dann eine Einschätzung darüber...

- ob die im Kapitel 4 vorgestellten Lösungsansätze für MDEVSP geeignet sind,
- inwieweit eine massiv-parallele Implementierung der Lösungsansätze auf einer GPU sinnvoll ist und
- welche Qualität die erzeugten Lösungen haben.

6.1. Numerische Ergebnisse

6.1.1. Probleminstanzen

Es standen die folgenden Probleminstanzen zur Verfügung:

PI867 Definiert im Sample-File `sample_real_867_SF_207_stoppoints.txt`
1 Depot, 206 Haltestellen, 867 Servicefahrten, 3725 Leerfahrten, 1 Fahrzeugtyp, 1 Fahrzeugtypgruppe

PI867LC Definiert im Sample-File `sample_lowcap_867_SF_207_stoppoints.txt`[1]
Geringere Batteriekapazitäten. Ansonsten identisch mit PI867.

[1] Dieses Sample-File wurde aus dem File `sample_real_867_SF_207_stoppoints.txt` erzeugt, indem die Batteriekapazitäten aller Fahrzeugtypen verringert wurden. Es dient dazu, Umlaufpläne mit möglichst vielen Aufladungen zu provozieren.

LACKNER Definiert im Sample-File `Lackner_Set1.txt`[2]

1 Depot, 100 Haltestellen, 1.135 Servicefahrten, 10.100 Leerfahrten, 9 Fahrzeugtypen, 2 Fahrzeugtypgruppen

PI2633 Definiert im Sample-File `sample_real_2633_SF_67_stoppoints.txt`

2 Depots, 65 Haltestellen, 2.633 Servicefahrten, 301 Leerfahrten, 9 Fahrzeugtypen, 2 Fahrzeugtypgruppen

PI10710 Definiert im Sample-File `sample_real_10710_SF_140_stoppoints.txt`

2 Depots, 138 Haltestellen, 10.710 Servicefahrten, 9.142 Leerfahrten, 9 Fahrzeugtypen, 2 Fahrzeugtypgruppen

Die Probleminstanzen PI867LC, PI10710 und LACKNER wurden während der Entwicklung der Anwendung zu Testzwecken verwendet. Für die in diesem Kapitel beschriebenen Versuche wurden nur die Probleminstanzen PI867 und PI2633 benutzt. Alle o.g. Files befinden sich auf der CD im Verzeichnis `\bin\data`. Nach dem Start des Programms erscheinen sie in einem Auswahlmenü[3]. Wie im Abschnitt B.3 erläutert, wurde allen Sample-Files ein Konfigurationsabschnitt vorangestellt, mit dem der Programmablauf beeinflusst wird. Ansonsten wurden die Files nicht verändert[4].

Ignorierte Attribute

Die folgenden Attribute, die in den Sample-Files vorkommen, werden vom Programm ignoriert:

- `ID`[5]

[2] Dieses Sample-File musste vorab korrigiert werden, da nicht alle Fahrzeugtypen einer Fahrzeugtypgruppe zugewiesen waren. Da das Programm auf eine solche Zuordnung angewiesen ist wurden die Fahrzeugtypen 3 bis 9 der Fahrzeugtypgruppe 1 zugeordnet.

[3] Alternativ kann mit der Kommandozeilenoption `INPUTFILE=<filename>` ein beliebiges Sample-File eingelesen werden.

[4] Von den beschriebenen Korrekturen an LACKNER abgesehen.

[5] Bei fast allen in den Sample-Files enthaltenen Entitäten werden Ids angegeben. Diese Ids werden vom Programm nicht verwendet, denn es vergibt eigene Ids. Das ist deshalb notwendig, weil die Daten meist in Vektoren gespeichert werden, deren Indizes auch zur Identifizierung der Daten benutzt werden. Diese Vorgehensweise ermöglicht eine effizientere Verwaltung der Daten. Die in den Sample-Files werden programmintern auch nicht gespeichert. Das wäre zwar möglich, würde aber zu zusätzlichem Verwaltungsaufwand und zusätzlichem Speicherverbrauch führen.

- `FromTime` und `ToTime` bei Leerfahrten. Sind im Datensatz mehrere Verbindungsfahrten mit gleicher Start- und Endhaltestelle enthalten, so wird die mit der kürzesten Dauer gewählt (bei gleicher Dauer die mit der geringeren Distanz)

- `MaxShiftBackwardSeconds`, `MaxShiftForwardSeconds` und `MinAheadTime` bei Servicefahrten

- `VehCharacteristic`, `VehClass`, `CurbWeightKg`, `Capacity` bei Fahrzeugtypen

- Bei Fahrzeugtypen wird nicht zwischen `slowRechargingTime` und `fastRechargingTime` unterschieden. Sind beide Attribute angegeben, so wird `fastRechargingTime` übernommen und `slowRechargingTime` wird ignoriert

- `VehicleCapacityForCharging` bei Haltestellen. Ggf. wird allerdings jede Haltestelle, bei der hier eine von Null verschiedene Zahl eingetragen ist als Ladestation erkannt (siehe Abschnitt B.3)

- `MinLayoverTime` bei Servicefahrten

- `ChargingSystem`: Ladesysteme werden nicht unterschieden. Es wird davon ausgegangen, dass jede Ladestation für alle Fahrzeugtypen geeignet ist

Umgang mit fehlenden bzw. fehlerhaften Leerfahrten

Die in diesen Dateien definierten Probleminstanzen enthalten Servicefahrten, die nicht an alle Depots angebunden sind. Das bedeutet, dass Ausrückfahrten von manchen Depots zu den Starthaltestellen der betroffenen Servicefahrten fehlen bzw. das Einrückfahrten für die Endhaltestellen der betroffenen Servicefahrten nicht vorhanden sind[6]. Die im Programm verwendeten Algorithmen setzen jedoch voraus, das ein Umlauf jederzeit durch eine Fahrt ins Depot unterbrochen bzw. beendet werden kann. Daher werden die fehlenden Leerfahrten bei Programmstart automatisch

[6]Dies lässt sich bspw. im Sample-File `sample_real_867_SF_207_stoppoints.txt` anhand der Servicefahrt mit der Id 65587 überprüfen. Diese Fahrt beginnt an der Haltestelle 8890. Es existiert jedoch weder eine Leerfahrt, die im Depot mit der Id 1 startet und die an der Haltestelle 8890 endet, noch existiert eine Leerfahrt für den umgekehrten Weg.

ergänzt[7]. Für Fahrtdauer und Fahrtstrecke werden dabei jeweils Durchschnittswerte verwendet. Darüber hinaus existieren Leerfahrten mit ungültigen Angaben für Fahrtdauer und Fahrtstrecke, was ebenfalls beim Programmstart durch das Setzen von Durchschnittswerten korrigiert wird.

6.1.2. Ausstattung des Testrechners

Der Rechner, auf dem die in den folgenden Abschnitten beschriebenen Versuche durchgeführt wurden, ist wie folgt ausgestattet:

Betriebssystem	Microsoft Windows 10 Education *Version 10.0 Build 15063*
CPU	Intel Core i7-6700K *4.00GHz Taktrate* *4 physische Kerne (8 logische Kerne)*
Hauptspeicher	16 GB
Grafikkarte	NVIDIA GeForce GTX 1070 *Treiber-Version: 382.05* *GPU Familie: GP104-A* *Device-Memory: 8 GB*
Festplatte	SSD, 1TB
CUDA-Version	8.0.61

6.1.3. Testaufbau und Testergebnisse

Es wurden 8 Versuche mit der Probleminstanz PI867 und 6 Versuche mit der Probleminstanz PI2633 durchgeführt und ausgewertet:

- PI867 gelöst mit ACO auf der CPU mit den Populationsgrößen 192, 1920 und 9600

[7] Diese automatische Ergänzung lässt sich in per Konfiguration abschalten (siehe Abschnitt B.3). Das führt jedoch dazu, dass Busse während der Planung mitunter an Haltestellen „stranden“, weil das Startdepot des Umlaufs von der Endhaltestelle der zuletzt durchgeführten Servicefahrt aus nicht erreichbar ist.

- PI867 gelöst mit ACO auf der GPU mit den Populationsgrößen 1920 und 9600[8]
- PI867 gelöst mit SA auf der CPU mit den Populationsgrößen 10, 100 und 1000
- PI2633 gelöst mit ACO auf der CPU mit den Populationsgrößen 10, 100 und 1000
- PI2633 gelöst mit SA auf der CPU mit den Populationsgrößen 10, 100 und 1000

Um vergleichbare Ergebnisse zu erhalten, wurde für alle Versuche dieselbe Konfigurationsdatei verwendet[9]. Die von der Anwendung erzeugten Ergebnis-Files befinden sich im Verzeichnis `\bin\result`. Die für die Verwendung in diesem Dokument aufbereiteten Files sind unter `\doc\data` abgelegt.

Ergebnisse für Simulated Annealing

Alle Tests wurden auf der CPU durchgeführt[10]. Sie Tests wurden beendet, wenn keine wesentlich besseren Ergebnisse mehr erwartet wurden. Die folgende Tabelle zeigt die erreichten Ergebnisse:

Instanz	**Population**	**#Fahrzeuge**	**Kosten**	**Laufzeit**
PI867	10	86	8.705.872	259 Min.
PI867	100	87	8.805.699	650 Min.
PI867	1000	87	8.808.613	682 Min.
PI2633	10	360	36.406.942	243 Min.
PI2633	100	394	39.859.745	394 Min.
PI2633	1000	423	42.794.281	2229 Min

[8] Da die Grafikkarte des Testrechners über 1920 Kerne verfügt macht ein Versuch mit nur 192 Threads keinen Sinn.

[9] Die Ergebnis-Files enthalten jeweils auch Angaben zu den Konfigurationseinstellungen. Die Optimierung der Konfigurationsparameter war nicht Gegenstand der Versuche. Obwohl im Vorfeld intensiv nach guten Konfigurationen für die beiden Lösungsansätze ACO und SA gesucht wurde ist zu vermuten, dass die jeweils optimale Konfiguration noch nicht gefunden wurde. Dazu wäre eine systematische Suche mit zahlreichen Tests nötig gewesen, die den Zeitrahmen der Arbeit gesprengt hätte.

[10] SA konnte aus Zeitgründen nicht mehr für die GPU implementiert werden.

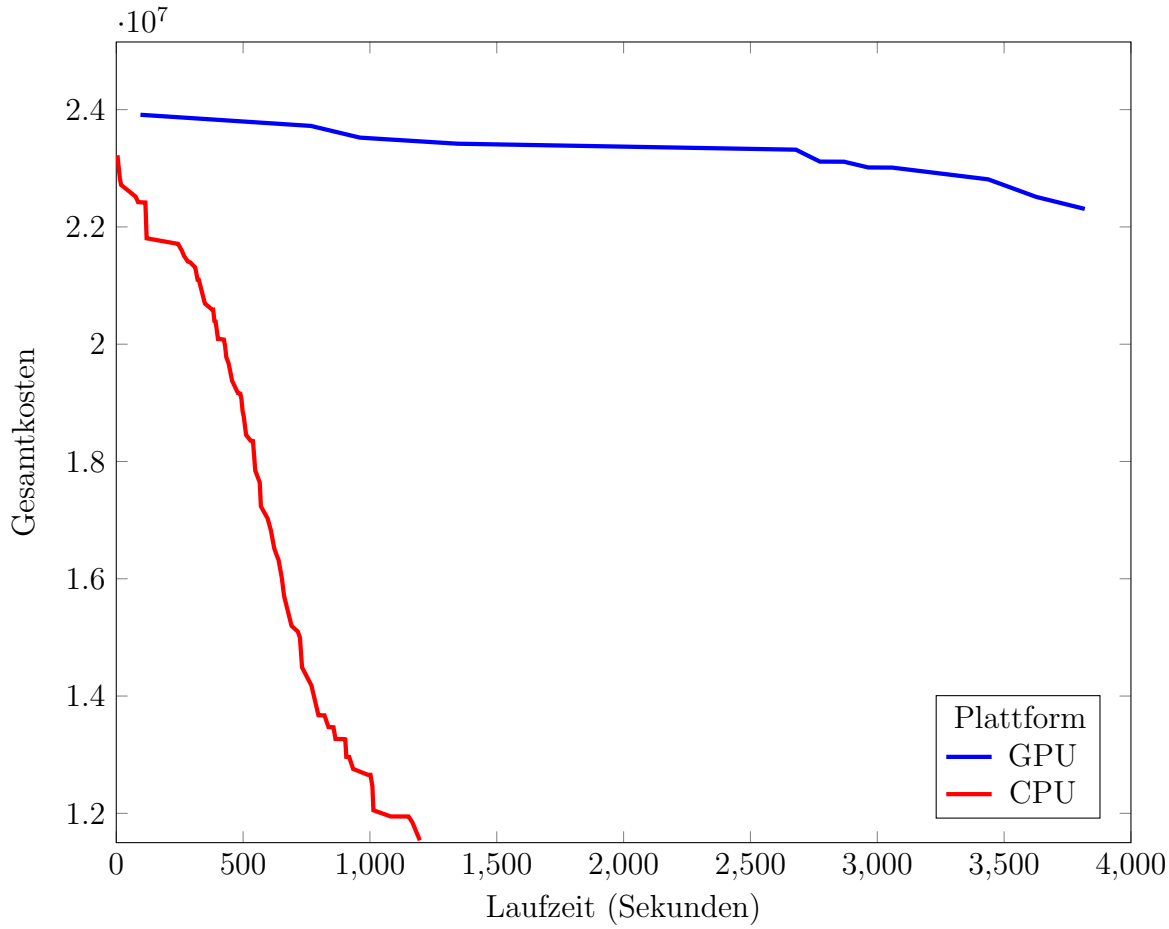


Abb. 6.1: ACO mit 867 Servicefahrten. Vergleich GPU/CPU (Populationsgröße=1920)

Da die Anzahl der eingesetzten Fahrzeuge den größten Einfluss auf die Gesamtkosten einer Lösung hat, zeigt die folgende Tabelle die Zwischenergebnisse zum Zeitpunkt des Erreichens von 87 bzw. 423 Fahrzeugen:

Instanz	**Population**	**#Fahrzeuge**	**Kosten**	**Laufzeit**
PI867	10	87	8.808.719	28 Min.
PI867	100	87	8.812.433	23 Min.
PI867	1000	87	8.809.393	534 Min.
PI2633	10	423	42.802.021	90 Min.
PI2633	100	423	42.810.650	201 Min.
PI2633	1000	423	42.807.061	2220 Min.

Ergebnisse für Ant Colony Optimization

Nachfolgend die besten erreichten Ergebnisse für ACO:

Instanz	Population	CPU	GPU	#Fzg.	Kosten	Laufzeit
PI867	192	X		119	12.147.694	2 Min.
PI867	1920	X		113	11.540.135	20 Min.
PI867	1920		X	219	22.307.096	64 Min.
PI867	9600	X		119	12.153.748	112 Min.
PI867	9600		X	235	23.927.908	6 Min. 28s
PI2633	10	X		326	33.319.458	2 Min.
PI2633	100	X		315	32.216.213	5 Min.
PI2633	1000	X		304	31.090.972	47 Min.

In der Abb. 6.1 wird die ungefähr um den Faktor 10 schlechtere Performance der GPU im Vergleich zur CPU deutlich. Die beiden Tests mit der Probleminstanz PI867 auf der GPU wurden wegen der schlechten Performance vorzeitig abgebrochen, da ihr Fortsetzen keinen Erkenntnisgewinn erwarten ließ[11].

6.1.4. Testauswertung

Die Größe der Population scheint bei keinem der beiden Lösungsansätze einen nennenswerten Einfluss auf die Lösungsqualität zu haben. Denn andernfalls müssten die Tests mit großen Populationen nach n Iterationen deutlich bessere Lösungen liefern als die Tests mit kleinen Populationen. Das ist jedoch hier weder für ACO noch für SA der Fall (siehe Abb. 6.2, Abb. 6.3 und Abb. 6.4). Alle Testergebnisse sind sich bzgl. des Zusammenhangs zwischen Iterationsanzahl und Lösungsqualität sehr ähnlich[12]. Da sich die Größe der Population bei den Tests aber jeweils um den

[11] Der Test mit der Populationsgröße 9600 hätte voraussichtlich wohl mindestens 22 Stunden lang laufen müssen, bis ein Ergebnis mit rund 120 Fahrzeugen erreicht worden wäre.

[12] Allerdings scheint es für ACO zumindest beim Lösen der Probleminstanz PI2633 einen etwas stärkeren Zusammenhang zwischen Populationsgröße und Lösungsqualität zu geben. Das konnte aus Zeitgründen jedoch nicht mehr näher untersucht werden.

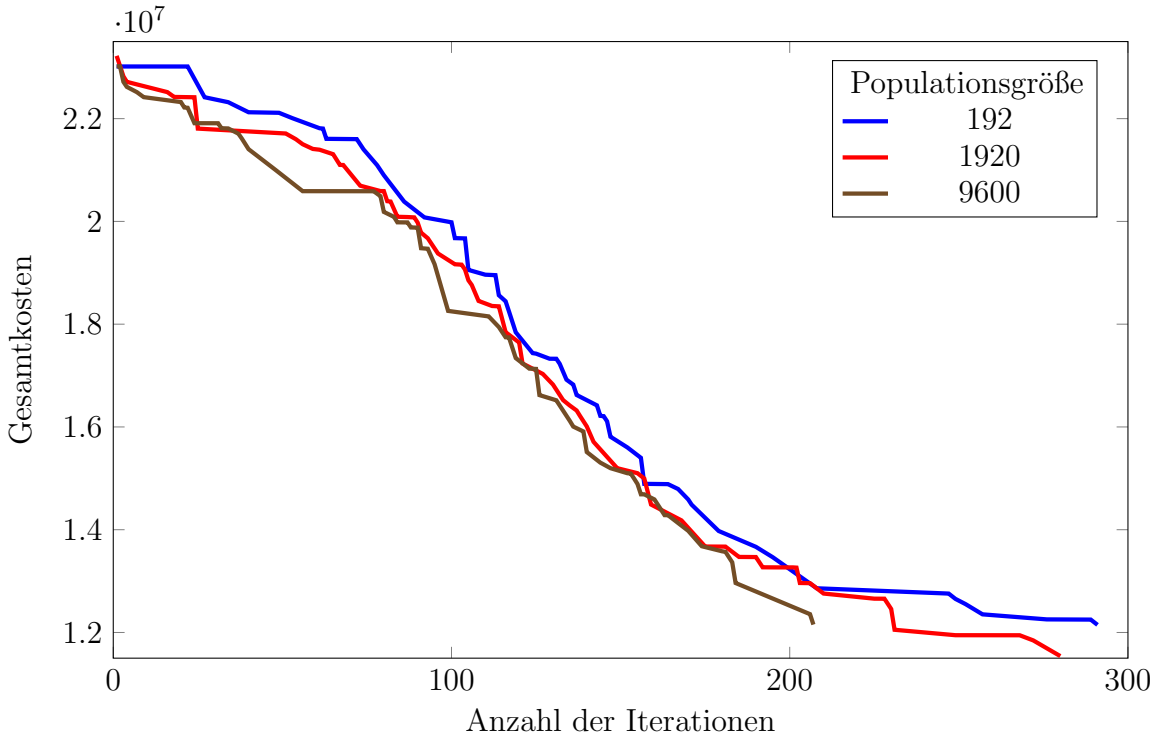


Abb. 6.2: ACO mit 867 Servicefahrten auf der CPU

Zufallsbedingt kann auch ein Test mit kleiner Population zwischenzeitlich durchaus auch bessere Lösungen liefern als einer mit großer Population. Aber insgesamt sollten doch größere Unterschiede erkennbar sein, wenn die Populationsgröße die Lösungsqualität beeinflusst.

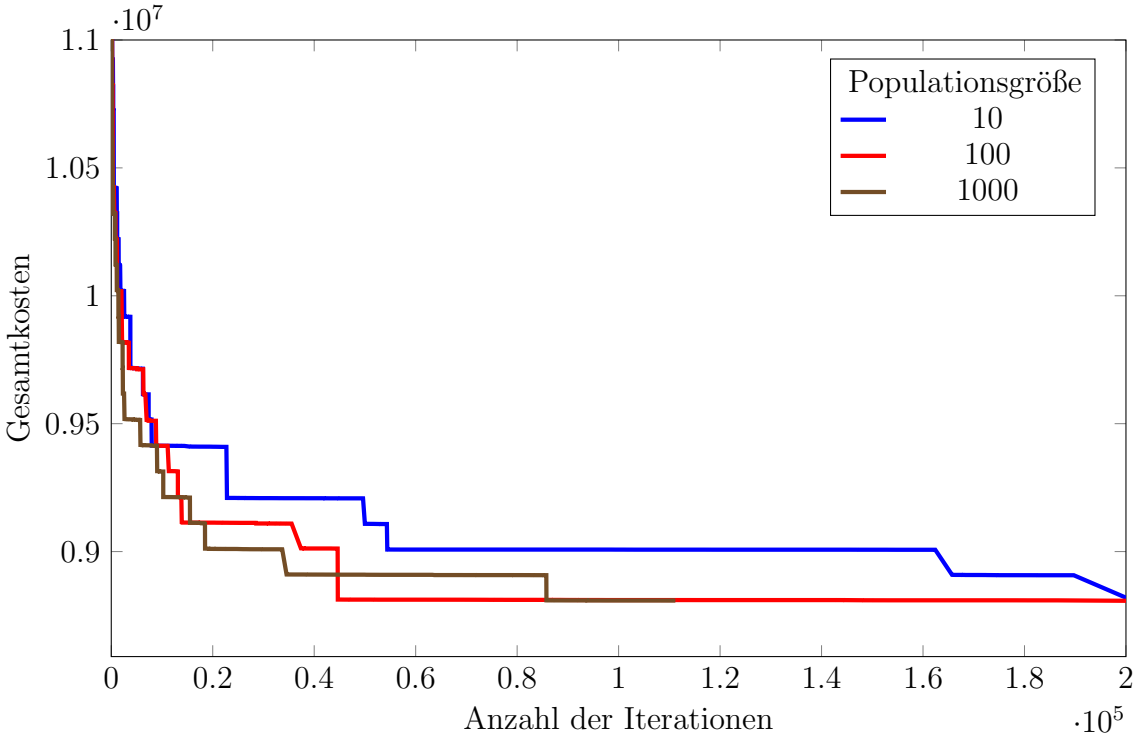


Abb. 6.3: SA mit 867 Servicefahrten auf der CPU

Faktor 10^{13} unterscheiden, müsste ein deutlicher Unterschied zu sehen sein, wenn eine größere Population zu besseren Ergebnissen führen würde.

Wie zu erwarten hat die Populationsgröße bei den Tests auf der CPU einen überaus negativen Einfluss auf die Performance[14] (siehe Abb. 6.5, Abb. 6.6 und Abb. 6.7). Das lässt vermuten, dass es nicht sinnvoll ist, die beiden Lösungsansätze ACO und SA massiv-parallel zu implementieren.

Im direkten Vergleich der beiden Lösungsansätze SA und ACO fällt auf, dass die Lösungsqualität bei ACO sehr stark von der Probleminstanz abzuhängen scheint. Während ACO bei PI867 deutlich schlechter abschneidet als SA, verhält es sich bei PI2366 genau umgekehrt (siehe Abb. 6.8). Der Grund dafür konnte nicht abschließend geklärt werden. Es ist möglich, dass der Lösungsraum bei PI2633 weniger stark zerklüftet ist als bei PI867. Eine weitere Möglichkeit ist, dass die Konfigurations-

[13] Nur zwischen den Tests zu PI867 mit den Populationsgrößen 1920 und 9600 ist der Faktor 5. Aber auch bei diesem Faktor sollte sich ein deutlicher Unterschied zeigen, was jedoch nicht der Fall ist.

[14] Da die Tests auf der GPU abgebrochen wurden kann zum Einfluss der Populationsgröße auf die Performace der GPU keine Aussage gemacht werden. Frühere Tests haben aber gezeigt, dass der Zusammenhang zwischen Populationsgröße und Performance auf der GPU stärker ausgeprägt ist als zu erwarten gewesen wäre.

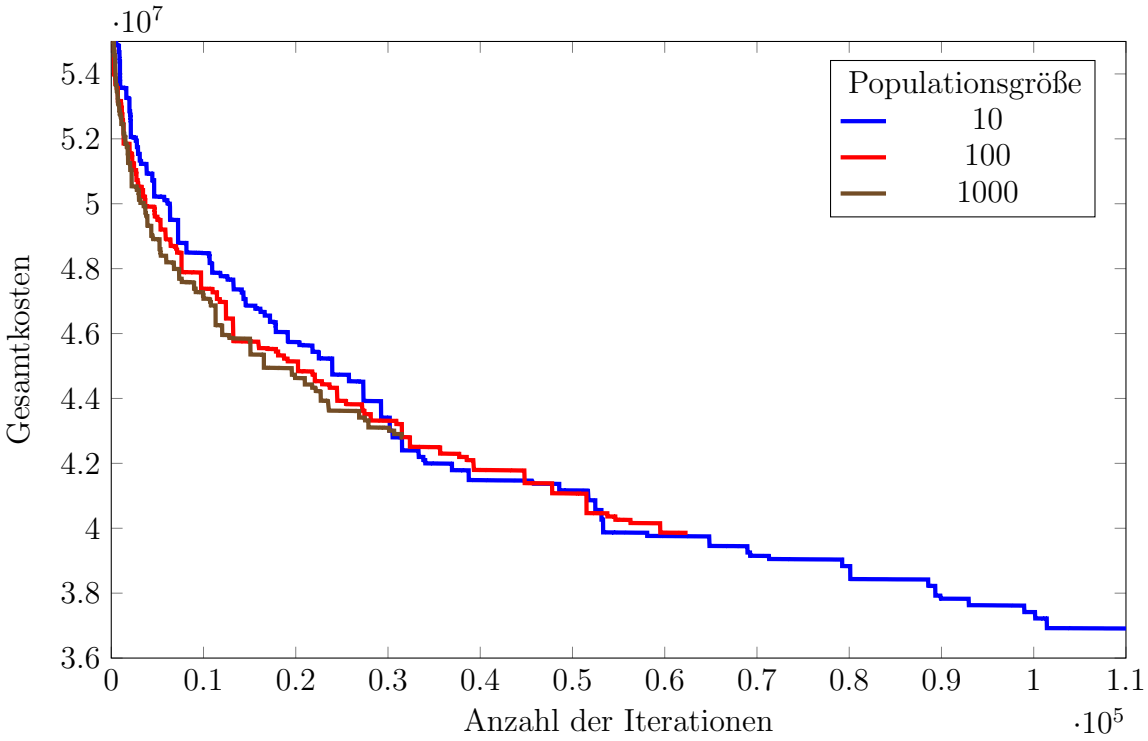


Abb. 6.4: SA mit 2633 Servicefahrten auf der CPU

Es ist recht deutlich zu sehen, dass es keinen signifikanten Zusammenhang zwischen der Populationsgröße und der Lösungsqualität gibt.

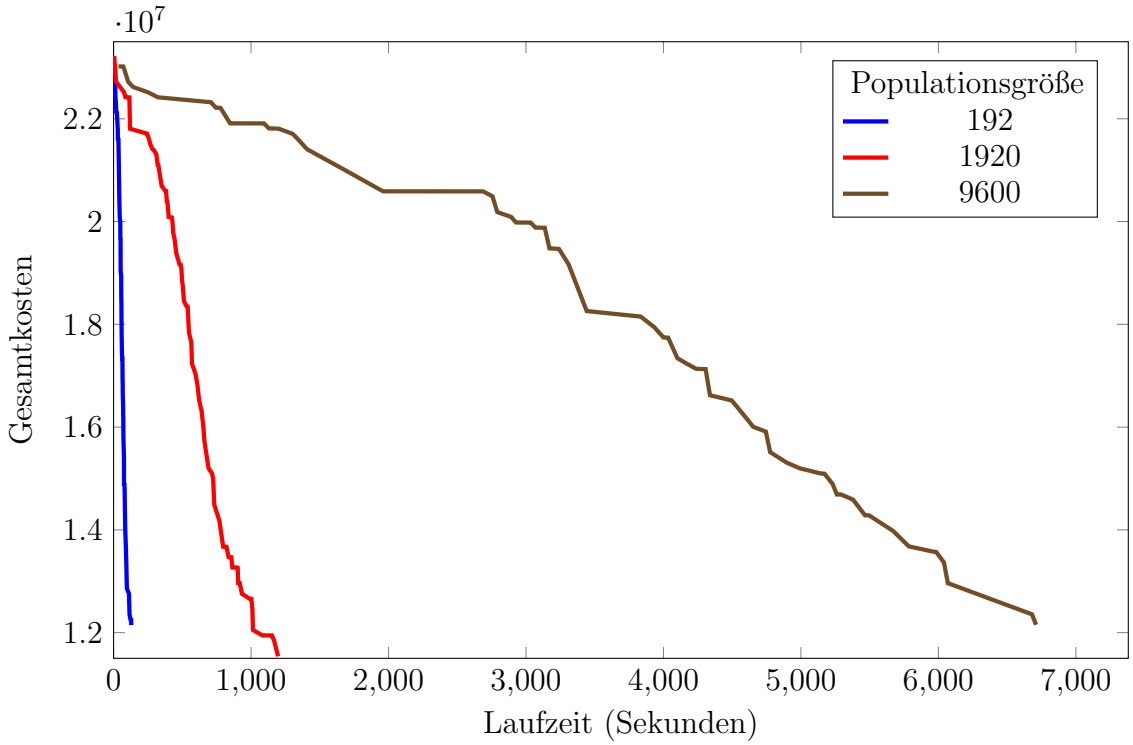


Abb. 6.5: ACO mit 867 Servicefahrten auf der CPU (nach Laufzeit)

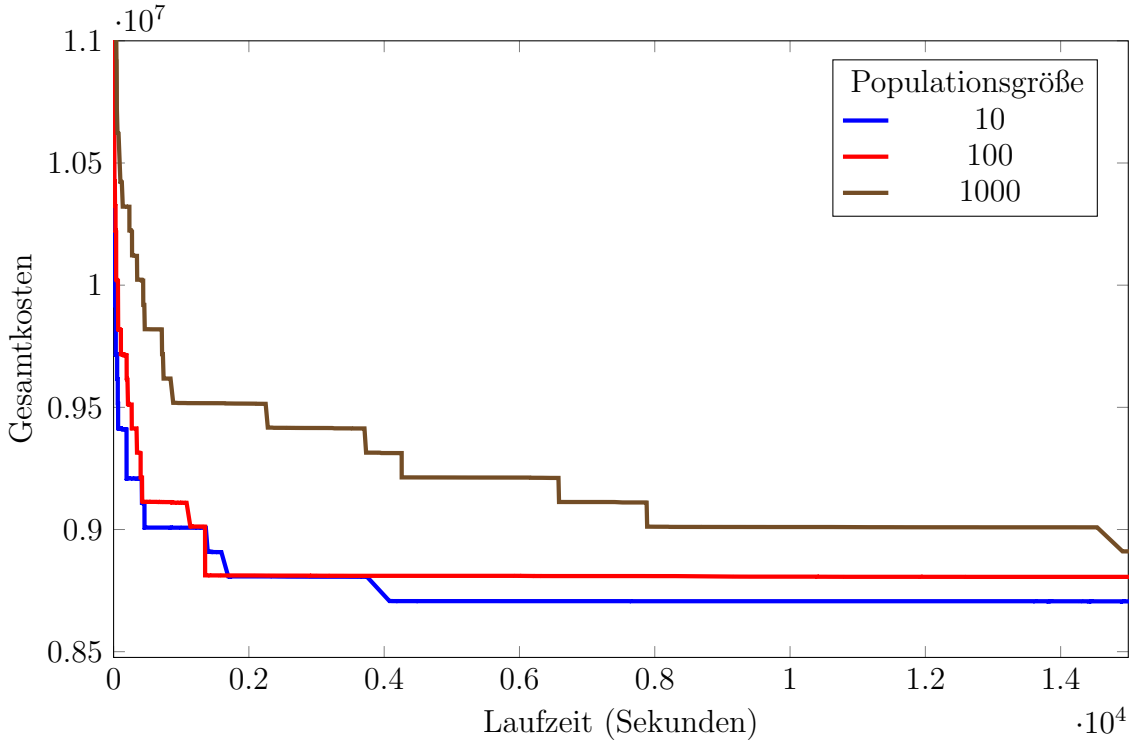


Abb. 6.6: SA mit 867 Servicefahrten auf der CPU (nach Laufzeit)

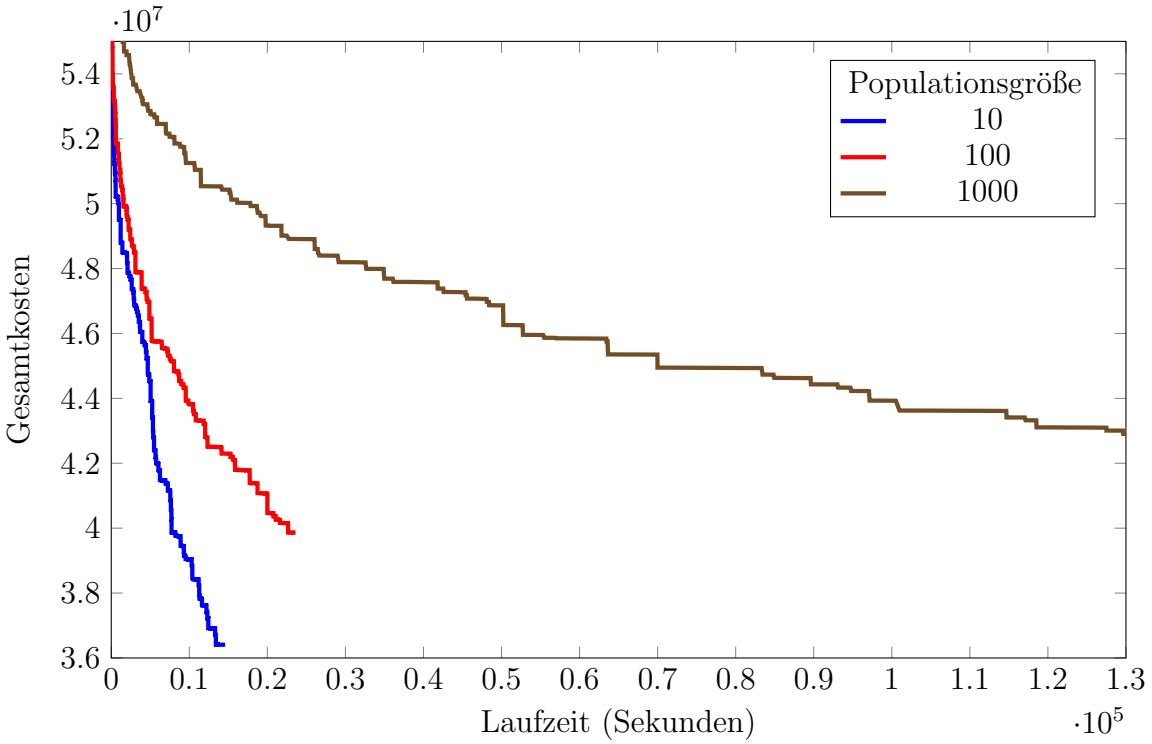


Abb. 6.7: SA mit 2633 Servicefahrten auf der CPU (nach Laufzeit)

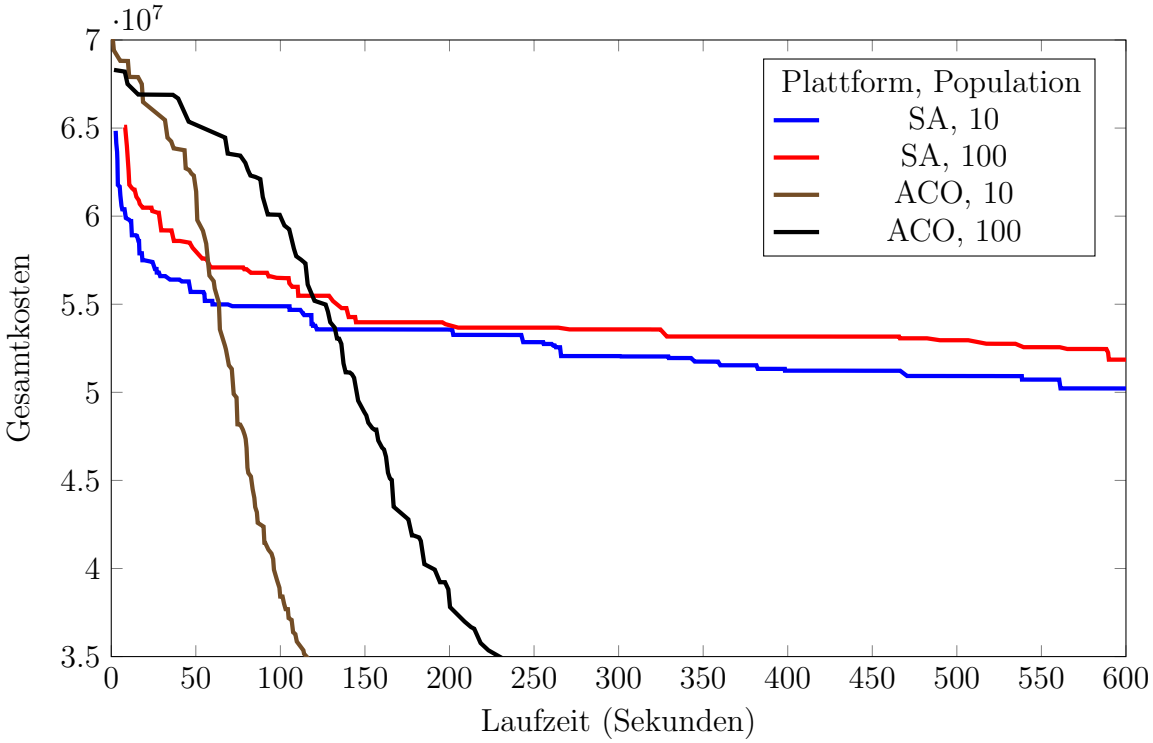


Abb. 6.8: Vergleich SA/ACO (2633 Servicefahrten)

parameter für ACO besser zu PI2366 als zu PI867 passen. Nicht auszuschließen ist auch ein noch unentdeckter Programmfehler.

Die von ACO in den ersten Iterationen gefundenen Lösungen haben i.d.R. rund doppelt so hohe Gesamtkosten wie die Startlösungen von SA. Das lässt sich dadurch erklären, dass die Startlösungen bei SA letztlich durch das Auffüllen von Umläufen erzeugt werden. Ein neuer Umlauf entsteht nur dann, wenn kein passender bestehender Umlauf gefunden werden kann. Bei ACO ist es dagegen viel wahrscheinlicher, dass beim Durchlaufen des Konstruktionsgraphen anfangs viele sehr kurze Umläufe entstehen[15]. Bei der Probleminstanz PI2366 kann ACO diesen anfänglichen Nachteil bald kompensieren und kommt dann sehr schnell zu guten Lösungen. Für PI867 gilt dies jedoch nicht. Vielmehr scheint ACO hier früh bei Lösungen zu stagnieren, die SA bereits in den ersten Iterationen gefunden hat. Möglicherweise kann ACO einmal gefundene lokale Minima nur schwer wieder verlassen.

Die schlechte Performance des ACO-Ansatzes auf der GPU zeigt, dass es nicht

[15]Das geschieht, obwohl die Gewichte der Kanten, die zu zeitlich später stattfindenden Servicefahrten führen, einen geringeren Initialwert erhalten als solche zu früher stattfindenden Servicefahrten.

gelungen ist, das Verfahren so zu implementieren, dass es gut mit CUDA ausgeführt werden kann. Das hat drei Gründe:

1. Der für das Durchwandern des Konstruktionsgraphen nötige Code ist komplex und enthält viele sehr stark verschachtelte bedingte Anweisungen. Dies ist unvermeidlich, da zahlreiche Restriktionen einzuhalten sind[16]. Es führt aber zu ausgeprägter Code-Divergenz (siehe Unterabschnitt 3.3.3 und Unterabschnitt 3.3.8).

2. Es ist nicht möglich, die benötigten Daten im Device-Memory so anzuordnen, dass Coalesced Accesses möglich sind. Zum einen ist der Umfang der Daten groß und zum anderen ist der Datenzugriff häufig vom Zufall abhängig. Auch die Nutzung des Shared Memory ist praktisch ausgeschlossen, da er für die benötigten Datenmengen zu klein ist (siehe Unterabschnitt 3.3.5).

3. Die implementierten Kernel sind meist sehr umfangreich. Das führt zu Variable Spilling (also zum Auslagern lokaler Variablen in den Device-Memory) und dazu, dass es den Warp-Schedulern erschwert wird, für eine gute Auslastung der Kerne zu sorgen.

Während sich der dritte Grund mit einigem Programmieraufwand beheben ließe erscheint es als sehr zweifelhaft, dass auch die Code-Divergenz vermieden und die Datenanordnung im Speicher optimiert werden können. Alle drei angeführten Gründe treffen auch auf SA zu, weshalb die Testergebnisse zur GPU-Performance auch auf SA übertragen werden können.

6.2. Gewonnene Erkenntnisse

- Die beiden Lösungsansätze SA und ACO eignen sich nicht für die massivparallele Implementierung. Sie sind aber gut für die parallele Verarbeitung auf einer Multicore-CPU geeignet

[16] Servicefahrten dürfen sich zeitlich nicht überschneiden. Die Batteriekapazität muss überwacht werden. Die Notwendigkeit von Verbindungsfahrten ist zu prüfen u.s.w.

- Die Performance und die Lösungsqualität bei ACO scheint stark von der zu lösenden Probleminstanz abzuhängen
- Mit einiger Wahrscheinlichkeit lassen sich die Performance und die Lösungsqualität beider Lösungsansätze noch steigern, indem die Konfigurationsparameter angepasst werden

7. Fazit und Ausblick

Um zu klären, inwieweit eine massiv-parallele Implementierung der Metaheuristiken ACO und SA auf einer GPU mithilfe von CUDA zur Lösung von MDEVSP geeignet ist, wurden zwei auf den Metaheuristiken ACO und SA basierende Lösungsansätze konzeptioniert und in Form einer Konsolenanwendung implementiert. Mit dieser Konsolenanwendung wurden sodann zwei unterschiedlich komplexe Probleminstanzen des MDEVSP gelöst. Dabei wurden die Populationsgröße, der angewendete Lösungsansatz und die Plattform (CPU/GPU) variiert.

Im Ergebnis lässt sich feststellen, dass sich beide Lösungsansätze nicht für die Umsetzung für CUDA eignen. Es wird kaum gelingen, die Metaheuristiken ACO und SA so zu implementieren, dass ein performanter Betrieb auf einer GPU möglich ist. Die Gründe dafür sind vor allem, dass die Komplexität des MDEVSP viele zum Teil tief verschachtelte konditionale Anweisungen (IF-THEN-ELSE) erforderlich macht und das eine für die Speicherhierarchie einer GPU geeignete Organisation der verwendeten Daten unmöglich ist, weil zufälliges Auswählen aus einem großen Datenbestand ein wesentliches Merkmal beider Heuristiken ist.

Leider reichte die für diese Masterarbeit vorgesehene Bearbeitungszeit nicht aus, um alle Aspekte der untersuchten Fragestellung ausreichend zu beleuchten. Folgende Punkte sollten in künftigen Arbeiten weiter untersucht werden:

- Es sollte weiter untersucht werden, wie sich die Populationsgrößte auf die Lösungsqualität bei der ACO auswirkt bzw. ob es Probleminstanzen mit bestimmten Eigenschaften gibt, welche die Lösbarkeit mit der ACO beeinflussen. Falls dem so ist sollte untersucht werden, welche Eigenschaften das sind.

- Es sollten zusätzliche systematische Tests durchgeführt werden, um die Auswirkungen zufälliger Schwankungen zu minimieren.

- Es sollte systematisch untersucht werden, wie sich Änderungen der Konfigurationsparameter auf die Lösungsqualität und die Performance auswirken, um so eine möglichst optimale Parameterbelegung zu erhalten.

- Umbau der auf CUDA basierenden Implementierung, sodass kleinere Kernel entstehen. Wird die GPU-Performance dadurch wesentlich besser?

- Statt die GPU dafür zu verwenden, die betreffende Metaheuristik vollständig zu implementieren kann sie auch als eine Art Coprozessor für solche Teilaufgaben eingesetzt werden, für die sie gut geeignet ist. Auf diese Weise könnte sie die CPU wesentlich entlasten.

- Lineares statt wie derzeit implementiert prozentuales Senken der Temperatur bei SA. Steigert dies die Lösungsqualität?

Literatur

Adler, J. (2014). „Routing and Scheduling of Electric and Alternative-Fuel Vehicles“. Dissertation. Arizona State University. URL: http://repository.asu.edu/attachments/134788/content/Adler_asu_0010E_13619.pdf.

AFP-Meldung (2017). „EU-Kommission rügt Deutschland wegen starker Luftverschmutzung“. In: *Der Tagesspiegel*. URL: http://www.tagesspiegel.de/wirtschaft/stickstoffdioxid-eu-kommission-ruegt-deutschland-wegen-starker-luftverschmutzung/19395584.html (besucht am 11. 06. 2017).

Becker, J. (2017). „Pro Elektrobusse: Wer ein Zeichen setzen will, steigt jetzt um“. In: *Süddeutsche Zeitung*. URL: http://www.sueddeutsche.de/auto/e-fahrzeuge-im-oepnv-pro-elektrobusse-wer-ein-zeichen-setzen-will-steigt-jetzt-um-1.3460626 (besucht am 11. 06. 2017).

Bengel, Günther u. a. (2015). *Masterkurs Parallele und Verteilte Systeme*. Wiesbaden: Springer.

Blume, Christian (1991). *Einführung in die Informatik für Naturwissenschaftler und Ingenieure*. 2. Aufl. Hanser Verlag. ISBN: 3-446-15710-7.

BMVI, Bundesministerium für Verkehr und digitale Infrastruktur (2016). *Projektübersicht 2015/16 Hybrid- und Elektrobus-Projekte in Deutschland*. Berlin. URL: http://www.xn--starterset-elektromobilitt-4hc.de/content/1-Bausteine/5-OEPNV/2016-projektuebersicht-20152016-hybrid-und-elektrobusprojekte-in-deutschland.pdf (besucht am 12. 06. 2017).

Bogon, Tjorben (2013). *Agentenbasierte Schwarmintelligenz*. Springer Vieweg. ISBN: 978-3-658-02291-4.

Brause, Rüdiger (2017). *Betriebssysteme*. Springer.

Breitinger, M. (2017). „Die Kommunen werden mit den Kosten alleingelassen“. In: *Zeit Online*. URL: http://www.zeit.de/mobilitaet/2017-02/oeffentlicher-nahverkehr-staedtische-verkehrsbetriebe-finanzierung-probleme-klage (besucht am 11. 06. 2017).

Breitinger, M. u. a. (2017). „Wo der Nahverkehr sein Geld wert ist“. In: *Zeit Online*. URL: http://www.zeit.de/mobilitaet/2017-02/bus-bahn-oeffentlicher-nahverkehr-studie-grafiken/komplettansicht (besucht am 11. 06. 2017).

Brockhaus (2005). *Der Brockhaus in zehn Bänden*. Mannheim: F.A. Brockhaus. ISBN: 9783765324505.

Brockhaus-Enzyklopädie online (2014). Bibliographisches Institut & F. A. Brockhaus AG. URL: https://duisburg-essen-ub.brockhaus.de/enzyklopaedie.

Bruge, P., U. Guida und M. Van den Bergh (2016). *ZeEUS eBus Report: An overview of electric buses in Europe*. Brüssel. URL: http://zeeus.eu/uploads/publications/documents/zeeus-ebus-report-internet.pdf (besucht am 11. 06. 2017).

Bünnagel, K. (2012). „Brennstoffzellenbusse im harten Linieneinsatz“. In: *BusMagazin*. URL: http://www.busmagazin.de/fileadmin/user_upload/Busmagazin/Fachbeitraege/2012_06_Brennstoffzellenbus.pdf (besucht am 12. 06. 2017).

Dijkstra, E. W. (1971). „Hierarchical ordering of sequential processes“. In: *Acta Informatica*. Bd. 1, S. 115–138. DOI: 10.1007/BF00289519. URL: https://rd.springer.com/content/pdf/10.1007%2FBF00289519.pdf.

Domschke, Wolfgang und Andreas Drexl (2007). *Einführung in Operations Research*. Berlin: Springer.

Dorigo, Marco und Thomas Stützle (2004). *Ant Colony Optimization*. Massachusetts Institute of Technology.

DPA-Meldung (2014). „Autofahren ist out, Smartphones werden wichtiger“. In: *Frankfurter Allgemeine Zeitung*. URL: http://www.faz.net/aktuell/technik-motor/auto-verkehr/fuehrerschein-kein-statussymbol-autofahren-ist-out-smartphones-werden-wichtiger-13346242.html (besucht am 11. 06. 2017).

Ellinger, Th., G. Beuermann und R. Leisten (2003). *Operations Research*. 6. Aufl. Berlin: Springer. ISBN: 3-540-00477-7.

Engesser, Hermann, Volker Claus und Andreas Schwill (1988). *Informatik - ein Sachlexikon für Studium und Praxis*. Mannheim: Dudenverlag. ISBN: 3411024216.

Ernst, Hartmut, Jochen Schmidt und Gerd Beneken (2015). *Grundkurs Informatik*. Springer. ISBN: 978-3-658-01627-2.

Flynn, Michael J. (1972). „Some Computer Organizations and Their Effectiveness“. In: *IEEE Transactions on Computers*. Bd. C-21. IEEE, S. 948–960.

Foley, J.D. u. a. (1995). *Computer Graphics: Principles and Practice*. Reading: Addison-Wesley.

Garey, Michael R. und David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman & Co. ISBN: 0-7167-1045-5.

Goss, S. u. a. (1989). „Self-organized Shortcuts in the Argentine Ant“. In: *Naturwissenschaften* (76), S. 579–581.

Grah, A. (2017). „Elektrobusse in Deutschland: Zu teuer, zu unzuverlässig, zu aufwendig“. In: *Spiegel Online*. URL: http://www.spiegel.de/auto/aktuell/elektrobusse-in-deutschland-zu-teuer-zu-unzuverlaessig-zu-aufwendig-a-1138185.html (besucht am 12. 06. 2017).

Kamann, M. (2016). „In vielen Städten drohen Fahrverbote wegen Feinstaub“. In: *Die Welt*. URL: https://www.welt.de/politik/deutschland/article150881212/In-vielen-Staedten-drohen-Fahrverbote-wegen-Feinstaub.html (besucht am 11. 06. 2017).

Kirk, D. und W. Hwu (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Burlington: Elsevier. ISBN: 978-0-12-381472-2.

Kliewer, N. (2005). „Optimierung des Fahrzeugeinsatzes im öffentlichen Personennahverkehr: Modelle, Methoden und praktische Anwendungen“. Dissertation. Universität Paderborn.

Kliewer, N., T. Mellouli und L. Suhl (2006). „A time–space network based exact optimization model for multi-depot bus scheduling“. In: *European Journal of Operational Research*. Bd. 175, S. 1616–1627.

Kliewer, N., J. Reuer und L. Wolbeck (2015). „The Electric Vehicle Scheduling Problem“. In: *CASPT2015 Conference Proceedings*. URL: http://www.caspt.org/proceedings/paper93.pdf (besucht am 12. 06. 2017).

Klüver, Christina, Jürgen Klüver und Jörn Schmidt (2012). *Modellierung komplexer Prozesse durch naturanaloge Verfahren*. Vieweg+Teubner Verlag. ISBN: 978-3-8348-2509-4. URL: http://www.ebook.de/de/product/19454800/christina_kluever_juergen_kluever_joern_schmidt_modellierung_komplexer_prozesse_durch_naturanaloge_verfahren.html.

Kooten Niekerk, M. E. van, J. M. van den Akker und J. A. Hoogeveen (2017). „Scheduling electric vehicles“. In: *Public Transport* 9.1, S. 155–176. ISSN: 1613-7159. URL: http://dx.doi.org/10.1007/s12469-017-0164-0.

Kröhnert, S. u. a. (2011). *Die Zukunft der Dörfer: Zwischen Stabilität und demografischem Niedergang*. Studie. Berlin-Institut für Bevölkerung und Entwicklung. URL: http://www.berlin-institut.org/fileadmin/user_upload/Doerfer_2011/Die_Zukunft_der_Doerfer_Webversion.pdf (besucht am 11. 06. 2017).

Landaverde, Raphael u. a. (2014). „An investigation of Unified Memory Access performance in CUDA“. In: *High Performance Extreme Computing Conference (HPEC)*. IEEE. ISBN: 978-1-4799-6234-1. DOI: 10.1109/HPEC.2014.7040988.

Luftschadstoffe im Überblick (2017). URL: http://www.umweltbundesamt.de/themen/luft/luftschadstoffe-im-ueberblick (besucht am 12. 06. 2017).

Nikowitz, M. (2016). *Advanced Hybrid and Electric Vehicles*. Hrsg. von M. Nikowitz. Wien: Springer. ISBN: 978-3-319-26304-5.

Nischwitz, A. u. a. (2011). *Computergrafik und Bildverarbeitung*. Wiesbaden: Vieweg+Teubner Verlag.

NVIDIA (2017). *CUDA C Programming Guide*. Hrsg. von NVIDIA. Version 8.0. URL: http:%5C%5Cdocs.nvidia.com/cuda/cuda-c-programming-guide/ (besucht am 21. 06. 2017).

Schnieder, L. (2015). *Betriebsplanung im öffentlichen Personennahverkehr - Ziele, Methoden, Konzepte*. Springer.

Schöneburg, E., F. Heinzmann und S. Feddersen (1994). *Genetische Algorithmen und Evolutionsstrategien: Eine Einführung in Theorie und Praxis der simulierten Evolution*. 1. Aufl. Bonn: Addison-Wesley.

Schwarzer, C. (2016). „Warten auf den Elektrobus“. In: *Zeit Online*. URL: http://www.zeit.de/mobilitaet/2016-10/elektromobilitaet-linienbusse-stadtverkehr-vorteile (besucht am 11. 06. 2017).

Schwenn, K. (2016). „Elektrobusse sollen das innerstädtische Klima retten“. In: *Frankfurter Allgemeine Zeitung*. URL: http://www.faz.net/aktuell/wirtschaft/neue-mobilitaet/neue-mobilitaet-elektrobusse-sollen-das-innerstaedtische-klima-retten-14411471.html (besucht am 11. 06. 2017).

Suhl, Leene und Taïeb Mellouli (2013). *Optimierungssysteme: Modelle, Verfahren, Software, Anwendungen*. Berlin: Springer. ISBN: 978-3-642-38937-5.

VDV, Verband Deutscher Verkehrsunternehmen (2016). *Statistik 2015*. (Besucht am 11. 06. 2017).

Waldrop, M. Mitchell (2016). „Mehr als Moore“. In: *Spektrum der Wissenschaft*. URL: http://www.spektrum.de/news/mehr-als-moore/1405206.

Werners, Brigitte (2013). *Grundlagen des Operations Research*. 3. Aufl. Berlin: Springer.

Anhang A.

Inhalt der CD

Verzeichnis	**Inhalt**
\	Das Wurzelverzeichnis enthält alle von der Entwicklungsumgebung benötigten Solution-Files. Insbesondere `EVSP.sln` und `Setup.sln`. Außerdem das Programm `cloc.exe`, mit dem die Lines of code ermittelt wurden.
`bin`	Enthält die Executables `EVSP.exe` und `EVSP64.exe` sowie in den Unterverzeichnissen `data` und `result` die Probleminstanzen und die Testergebnisse.
`doc`	Enthält das vorliegende Dokument im PDF-Format sowie die Latex-Sourcen, mit denen das Dokument erstellt wurde. Außerdem in diversen Unterverzeichnissen alle referenzierten Dateien.
`EVSP.BaseClasses`	Enthält den Sourcecode des Moduls `EVSP.BaseClasses`.
`EVSP.Console`	Enthält den Sourcecode des Moduls `EVSP.Console`.
`Evsp.CudaSolver`	Enthält den Sourcecode des Moduls `Evsp.CudaSolver`.
`Evsp.CudaSolver.UnitTest`	Enthält den Sourcecode des Moduls `Evsp.CudaSolver.UnitTest`.
`EVSP.Model`	Enthält den Sourcecode des Moduls `EVSP.Model`.
`install`	Enthält die Installations-Programme für EVSP und die Laufzeitumgebung.
`Setup`	Enthält die vom Setup-Projekt für die Erzeugung des Installations-Programms benötigten Daten.

Anhang B.

Bedienung des Programms

Bei dem im Rahmen der Arbeit erstellten Programm handelt es sich um eine Konsolenanwendung für das Betriebssystem Microsoft Windows. Das Programm verfügt nicht über eine grafische Benutzeroberfläche und wird daher in einem Kommandozeilenfenster gestartet[1]. Die Bedienung erfolgt also ausschließlich über Tastatureingaben und die Ausgaben des Programms erscheinen in Textform innerhalb des Kommandozeilenfensters. Außerdem wird bei jedem Programmlauf eine Datei im Comma-separated values (CSV)-Format[2] erzeugt, um die Datenauswertung mit einer Tabellenkalkulation wie Microsoft Excel zu erleichtern.

B.1. Systemvoraussetzungen

Die in der folgenden Tabelle aufgeführten Systemvoraussetzungen müssen mindestens erfüllt sein, um das Programm ausführen zu können.

[1] Das Vorhaben, eine grafische Benutzeroberfläche für das Programm zu entwickeln musste aus zeitlichen Gründen leider aufgegeben werden.

[2] Ein Dateiformat für den Austausch simpel strukturierter Daten. Die Daten werden hier für die Weiterverarbeitung in einer Tabellenkalkulation zeilenweise in einem Textfile gespeichert. Die Unterteilung in Spalten erfolgt innerhalb der Zeilen durch Semikola.

Betriebssystem	Microsoft Windows 7[3] *Windows 10 wird unterstützt. Die CD enthält zwei Versionen des Programms. Die 32-Bit-Version kann maximal 8 GB Hauptspeicher nutzen und ist auf jedem unterstützten Windows-System lauffähig. Die 64-Bit-Version nutzt den gesamten Arbeitsspeicher aus, kann aber nur auf 64-Bit-Windows-Systemen ausgeführt werden.*
Arbeitsspeicher	Gemäß Mindestanforderung des Betriebssystems. Mind. 1 GB. *Der Speicherbedarf des Programms hängt von der Größe der zu lösenden Probleminstanz und von der Anzahl der gleichzeitig betrachteten Lösungen ab und kann daher erheblich über 1 GB. liegen.*
CPU	Gemäß Mindestanforderung des Betriebssystems. *Das Programm ist auf Multithreading ausgelegt. Optimal ist daher ein Multicore-Prozessor mit einer Taktrate von mindestens 3 GHz.*
Grafikkarte	Gemäß Mindestanforderung des Betriebssystems. *Das Lösen von Probleminstanzen auf der GPU ist allerdings nur mit einer CUDA-fähigen Grafikkarte möglich. Dies sind alle aktuellen Grafikkarten mit NVIDIA-Chipsätzen der Reihen GeForce, Ion, Quadro und Tesla. Diese Grafikkarten unterscheiden sich stark in der Leistungsfähigkeit und dem verfügbaren Arbeitsspeicher. Die Größe der in akzeptabler Zeit lösbaren Probleminstanz ist daher möglicherweise beschränkt.*
CUDA	Um den ACO-Solver verwenden zu können ist es erforderlich, CUDA 8.0 zu installieren. Das ist auch dann der Fall, wenn die Lösung auf der CPU erfolgen soll. Da das CUDA-Installationsprogramm zur verwendeten Windows-Version passen muss ist es leider nicht möglich, die CUDA in den Setup-Prozess zu integrieren. CUDA kann unter https://developer.nvidia.com/cuda-downloads heruntergeladen und später problemlos wieder vom Rechner entfernt werden.

B.2. Starten des Programms

Die ausführbaren Dateien für beide Varianten des Programms EVSP befinden sich auf der beiliegenden CD im Verzeichnis `\bin`:

- `EVSP.exe` ist die 32-Bit-Variante des Programms. Sie ist auf allen unterstützten

Windows-Systemen lauffähig

- `EVSP64.exe` ist die 64-Bit-Variante des Programms, die nur auf 64-Bit-Windows-Systemen funktioniert

Beide Varianten lassen sich durch einen Doppelklick starten. Alternativ ist es auch möglich, zunächst ein Kommandozeilenfenster zu öffnen und das Programm dann durch Eingabe des Dateinamens zu starten[4]. In diesem Fall ist die Übergabe von Kommandozeilenparametern möglich (siehe dazu Abschnitt B.3). Das Programm kann zwar auch direkt von der CD aus gestartet werden[5], der Programmstart lässt sich jedoch beschleunigen, indem das Verzeichnis `\bin` vorher auf die Festplatte kopiert wird.

Möglicherweise erscheint beim ersten Start eine Fehlermeldung, weil auf dem Rechner die benötigte Laufzeitumgebung noch nicht zur Verfügung steht. Dann empfiehlt es sich, das Programm mit einem der Setup-Programme im Verzeichnis `\install` zu installieren[6]:

- `setup.exe` installiert die 32-Bit-Variante des Programms. Sie ist auf allen unterstützten Windows-Systemen lauffähig

- `setup64.exe` installiert die 64-Bit-Variante des Programms, die nur auf 64-Bit-Windows-Systemen funktioniert

Alternativ kann die Laufzeitumgebung auch mit einem der beiden Programme `vc_redist.x64.exe` bzw. `vc_redist.x32.exe` aus `\install\ISSetupPrerequisites` installiert werden.

B.3. Konfiguration

Das Programm bietet zahlreiche Konfigurationsparameter an, mit denen sich viele Teilaspekte des Programmverhaltens kontrollieren lassen. Die Parameter können beim Aufruf des Programms als Kommandozeilenparameter angegeben oder in ein Config-File eingetragen werden[7]. Auch die Angabe im Sample-File ist möglich, wobei einem Konfigurationsparameter dort ein Fragezeichen vorangestellt sein muss. Im Config-File und in der Kommandozeile darf kein führendes Zeichen verwendet werden.

[4] Ggf. muss zuvor in das Verzeichnis `bin` gewechselt werden.
[5] Beim Start von der CD kann die Funktionalität jedoch eingeschränkt sein.
[6] Selbstverständlich ist später die vollständige Deinstallation des Programms mithilfe der Windows-Systemsteuerung möglich.
[7] Ein Config-File muss immer den Namen `config.ini` haben und im selben Verzeichnis wie das Executable stehen. Auf der CD befindet es sich im Verzeichnis `bin`.

Eine Zeile, die mit einem Asterix-Zeichen (*) beginnt, wird als Kommentar betrachtet und ignoriert. Da derselbe Parameter durchaus mehrfach angegeben werden kann, erfolgt die Auswertung der Konfiguration nach Priorität:

1. Kommandozeile (höchste Priorität)
2. Sample-File
3. Config-File
4. Default-Wert[8] (niedrigste Priorität)

In den auf der CD gespeicherten Sample-Files sowie im Config-File sind alle Konfigurationsparameter mit Kommentaren versehen, die über dir Funktion und mögliche Werte Auskunft geben. Sollten bei Programmstart wichtige Angaben fehlen, zu denen es keine Default-Werte gibt, so erscheint eine Bildschirmmaske, in welcher der Benutzer eine entsprechende Auswahl treffen kann.

[8]Default-Werte sind fest im Sourcecode hinterlegt.

Eidesstattliche Erklärung

Ich versichere an Eides statt durch meine Unterschrift, dass ich die vorliegende Masterarbeit mit dem Titel „Massiv-parallele Implementierung einer Heuristik zur Lösung des Electric Vehicle Scheduling Problems mit CUDA“ selbständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Die Arbeit hat in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum	Wolfgang Bongartz