Fakultät für Mathematik und Informatik

Lehrgebiet Theoretische Informatik

Prof. Dr. André Schulz

Vergleich der Performanz bei kürzesten-Pfad-Berechnungen zwischen Stand-Alone-Lösungen und Graphdatenbanken

Bachelorarbeit im Studiengang BSc. Informatik

Gundula Swidersky

Matrikelnummer 8913447

16.10.2018

betreut von:

Prof. Dr. André Schulz

Lehrgebiet Theoretische Informatik

FernUniversität in Hagen

Zusammenfassung

Durch die starke Vernetzung und die wachsende Menge an strukturierten und unstrukturierten Daten ergeben sich neue Anforderungen an Datenbanksysteme. Graphdatenbanksysteme (GDBS) bauen auf einem Modell auf, das die Auswertung von Beziehungen in vernetzten Daten im Vergleich zu den klassischen Relationalen Datenbanksystemen (RDBS) wesentlich vereinfacht. Das ihnen zugrunde liegende Graph-basierte Modell bildet Objekte und Beziehungen in Form eines Graphen durch die Definition von Knoten und Kanten direkt im Datenmodell ab. Komplexe Joins, mit denen Beziehungen zwischen Relationen in klassischen, Tabellen-basierten RDBS hergestellt werden, sind für die Auswertung von Beziehungswissen in GDBS nicht nötig. Dadurch ist speziell in diesem Bereich eine höhere Performanz bei Auswertungen mit den GDBS zu erwarten, weswegen diese Systeme immer populärer werden. Die Auswertung von Anfragen in einem GDBS erfolgt je nach Problemstellung mit dem passenden effizienten Graph-Algorithmus.
In dieser Arbeit stellen wir nach der Behandlung von allgemeinen Grundlagen das GDBS anhand von Neo4j vor und erklären grundlegende Funktionalitäten der Anfragesprache Cypher. Anschließend vergleichen wir die Performanz zwischen Neo4j und einer eigenen implementierten Stand-Alone-Lösung anhand des kürzesten-Pfad-Problems. Hierzu haben wir drei ausgewählte Graph-Algorithmen implementiert, die kürzeste Pfade in Graphen suchen: Die bidirektionale Breitensuche für ungewichtete Graphen, den Dijkstra-Algorithmus für Graphen mit nicht-negativen Kantengewichten und den Bellman–Ford-Algorithmus für gewichtete Graphen ohne negativen Zyklus. Für den Vergleich der Performanz haben wir Experimente entworfen, mit denen wir die Laufzeiten der Algorithmen anhand von verschiedenen Eingabegraphen messen und auswerten. Als optionale Aufgabe haben wir den Bellman–Ford-Algorithmus als Neo4j-Erweiterung implementiert, um auch diesen in den Vergleich einzubeziehen. Für jedes Experiment haben wir Hypothesen aufgestellt, die mit verschiedenen Messexperimenten überprüft werden. Als Eingabegraphen haben wir sowohl synthetische Graphen auf Basis des Barabási–Albert-Modells und des Erdös–Rényi-Modells als auch ausgewählte extern aus dem Internet geladene sogenannte Real-World-Graphen verwendet. Für die Generierung und Vorbereitung der Graphen für die Experimente haben wir einen Graphgenerator implementiert. Für eine weitgehende Automatisierung der Messexperimente haben wir weitere Programme entwickelt. Für zusätzliche Vergleiche mit dem Neo4j-Plugin für effiziente Graph-Algorithmen haben wir auch den Dijkstra-Algorithmus als Neo4j-Erweiterung implementiert, da wir bei den Messexperimenten höhere Laufzeiten mit Neo4j gemessen haben. Zu jedem Experiment beschreiben wir den Ablauf und die teilweise überraschenden Ergebnisse. In einem Fazit fassen wir die daraus gewonnenen Erkenntnisse zusammen und geben abschließend einen Ausblick mit Ideen zu weiteren Experimenten.

Inhaltsverzeichnis

1 Einleitung

Durch die starke Vernetzung und die wachsende Menge an strukturierten und unstrukturierten Daten ergeben sich neue Anforderungen an Datenbanksysteme. Speziell, wenn es um stark vernetzte Daten geht, werden Graphdatenbanken immer populärer [8]. Sie werden heute bereits in Unternehmen oder auch in der Wissenschaft zur Pflege und Auswertung von vernetzten Daten eingesetzt.

Viele Probleme lassen sich auf das Modell eines Graphen übertragen. In der Diskussion stellen wir oftmals das Problem mit Abhängigkeiten und Beziehungen, zum Beispiel auf einem Flipchart oder einem Whiteboard, dar. Dabei entsteht bereits ein graphisches Modell, in dem die Knoten und Kanten entsprechend so definiert werden, dass sie den Objekten und Beziehungen der in der Diskussion entstandenen graphischen Darstellung entsprechen. Ein solches Modell lässt sich in der Regel direkt in einer Graphdatenbank abbilden. Wir haben hier also eine einfache Modellierung und einen einfachen Weg, wie das Modell beim Datenbankentwurf in eine Datenbank überführt wird.

Bei konventionellen Relationalen Datenbanksystemen (RDBS), ist der Datenbankentwurfsprozess etwas aufwändiger. Ausgehend vom Entity-Relationship-Modell (ER-Modell), das bei der Diskussion der Anforderungen als graphisches Modell entwickelt wurde, werden Relationen (Tabellen) gebildet und diese einer Normalisierung unterzogen. Dabei werden Schlüssel und Fremdschlüssel definiert, um Beziehungen bei Bedarf über Join-Operationen wieder herstellen zu können. Der etwas aufwändigere Datenbankentwurfsprozess bei RDBS soll die Konsistenz und die Integrität der Daten sicherstellen und ist für OLTP-Anwendungen (OLTP steht für Online Transaction Processing) optimiert worden.

Auswertungen von Beziehungsmustern können daher zu komplexen Join-Operationen mit langdauernden Berechnungen führen und damit zum Flaschenhals für darauf basierende Anwendungen werden. Vukotic et. al. haben hierzu ein Experiment durchgeführt, bei dem Freundschaftsbeziehungen in einem sozialen Netzwerk mit 1 Mio. Personen ausgewertet wurden. Bei diesem Experiment zeigte sich, dass die Join-Operation bei der Auswertung im RDBS bei einer Freundschaftstiefe von Vier bereits mehrere Stunden dauerte, während die Auswertung mit dem Graph-DBS Neo4j nur wenige Sekunden dafür benötigte [37].

Das Graph-DBS Neo4j unterstützt sowohl OLTP als auch OLAP-Anwendungen (OLAP steht für Online Analytical Processing). In Neo4j wird ein Graph nativ gespeichert, d.h. in einer für einen Graphen üblichen Datenstruktur, hier Adjazenzlisten, die eine schnelle Navigation im Graphen ermöglichen. Für eine höhere Performanz werden in Neo4j sowohl Listen für Vorgänger als auch für die Nachfolger eines Knotens geführt. Ein Graph wird hier grundsätzlich als ge-

richteter Eigenschaftsgraph abgebildet, d.h. weitere Informationen werden als Eigenschaften an Knoten und Kanten des Graphen angeheftet. Die Auswertung erfolgt dann über Graph-Algorithmen, wie in der Graphentheorie üblich.

Durch die native Speicherung der Daten als Graph minimiert sich die Anforderung an die logische und physische Datenunabhängigkeit, die man als Vorteil des Einsatzes eines DBS erreicht. Denn die Form des Datenmodells eines Graphen ist bekannt und ändert sich nicht.

Wenn wir Auswertungen basierend auf einem Graphen durchführen und dies üblicherweise mit Algorithmen der Graphentheorie durchgeführt werden kann, warum sollten wir dann überhaupt ein Graph-DBS, wie zum Beispiel Neo4j, verwenden und nicht einfach eine Stand-Alone-Anwendung entwickeln? Schließlich müsste diese ja sogar performanter sein, da mit der Nutzung eines DBS immer ein gewisser Overhead für Verwaltungsaufgaben anfällt, der bei einer Auswertung aber gar nicht nötig wäre. Ab welcher Größe eines Graphen oder unter welchen Umständen ist es sinnvoll, doch mit einem Graph-DBS zu arbeiten?

Diese Fragestellung werden wir in dieser Arbeit mit dem Thema ‚Vergleich der Performanz bei kürzesten-Pfad-Berechnungen zwischen Stand-Alone-Lösungen und Graphdatenbanken' untersuchen und auswerten. Wir beginnen mit wichtigen Grundlagen und der Entwicklung von DBS. Anschließend stellen wir GDBS anhand Neo4j vor und erklären grundlegende Funktionen der Anfragesprache Cypher. Für den Vergleich der Performanz zwischen der Lösung mit Neo4j und einer implementierten Stand-Alone-Lösung haben wir drei Algorithmen für die kürzeste-Pfadsuche ausgewählt, die wir in der Stand-Alone-Lösung implementiert haben.

Für die Durchführung des Vergleichs der Performanz haben wir drei Experimente entworfen und Hypothesen aufgestellt, die in den Experimenten überprüft werden. Die Messungen der Laufzeiten werden sowohl mit synthetisch generierten Graphen als auch mit ausgewählten aus dem Internet geladenen Real-World-Graphen durchgeführt. Die Ergebnisse der Messungen werden anschließend zusammengeführt und ausgewertet. Dabei wird überprüft, ob die jeweilige Hypothese angenommen werden kann oder nicht. In einem Fazit werden die Ergebnisse zusammengefasst und als Ausblick Ideen für weitere Experimente gegeben.

2 Datenbanken - Motivation und Entwicklung

Die in dem Kapitel ‚Datenbanken - Motivation und Entwicklung' enthaltenen Beschreibungen, Definitionen und Abbildungen sind eng angelehnt an Saake, Sattler und Heuer [30] sowie Schneider [32], die wichtige Grundlagen zu DBS beschreiben.

2.1 Dateiverarbeitung vs. Datenbanksystem

In der traditionellen Datenverwaltung nutzt man Dateien, die direkt oder mittels Dateiverwaltungssystem im jeweiligen Betriebssystem erstellt werden. Bei der Entwicklung einer Anwendung werden eine oder mehrere Dateien in einem festgelegten Format erstellt. Die Struktur dieser Dateien ist so aufgebaut, dass diese die Anwendung optimal unterstützt. Dabei kommt es mitunter zu einer redundanten Datenhaltung, da dieselben Daten in einem anderen Zusammenhang bereits in Dateien einer weiteren Anwendung verarbeitet werden, ohne dass der Entwickler davon Kenntnis hat. Änderungen oder Löschungen von Daten führen oft zu Inkonsistenzen oder Fehlern im Datenbestand. Ein Beispiel dafür ist die Anpassung der Adresse eines Kunden, die in verschiedenen Dateien gespeichert ist und dann durch die Anwendung aber nur in einer Datei geändert wird. Die Verwaltung der Daten bei dieser Art der Verarbeitung ist also sehr abhängig von dem jeweiligen Programm, dass diese Daten verwaltet. Schnittstellen zwischen Anwendungen sind wegen Inkonsistenzen oder Redundanzen der Daten sehr fehleranfällig. Bei Änderungen des Formates einer Datei müssen Anwendungen und Schnittstellen entsprechend mit angepasst werden. Dies hat einen hohen Pflegeaufwand der Anwendungen zur Folge. Auch die Auswertung von Daten aus verschiedenen, verteilten Dateien ist problematisch. Um diesen Nachteilen der traditionellen Datenverwaltung entgegen zu wirken, wurde nach einer Lösung gesucht, die die Daten so abstrahiert und modelliert, dass diese unabhängig von verschiedenen Programmen in Form eines gemeinsamen Datenbestandes aufgebaut und verarbeitet werden können. So entstanden die ersten Datenbanksysteme in den 60er Jahren. Bevor wir näher auf die weitere historische Entwicklung und die Architektur von Datenbanksystemen eingehen, werden wir zunächst die wichtigsten Begriffe definieren und erklären.

Datenbank (DB)

Eine Datenbank (DB) ist eine integrierte und strukturierte Sammlung persistenter Daten, die allen Benutzern eines Anwendungsbereichs als gemeinsame und verlässliche Basis aktueller Information dient [32].

Datenbank Management System (DBMS)

Ein Datenbankmanagementsystem (database management system), kurz DBMS, ist ein All-Zweck-Softwaresystem, das den Benutzer bei der Definition, Konstruktion und Manipulation von Datenbanken für verschiedene Anwendungen applikationsneutral und effizient unterstützt [32].

Datenbanksystem (DBS)
Ein Datenbanksystem, kurz DBS, ist die Kombination eines DBMS mit einer Datenbank [30].

Die ersten Datenbanksysteme in den 60er Jahren unterstützten das hierarchische Modell (zum Beispiel Baumstrukturen) oder das Netzwerkmodell, das allgemeine Verknüpfungen zulässt. Beide Modelle sind an Datenstrukturen von kommerziellen Programmiersprachen angelehnt und basieren auf Zeigerstrukturen zwischen Datensätzen. Die Datenmanipulationssprache (DML) war navigierend anhand von Zeigerstrukturen. Bei diesen Modellen war der Grad der Abstraktion zur physischen Speicherung der Daten allerdings noch zu schwach, sodass die Anwendungsprogrammierung bei Änderungen entsprechend angepasst werden musste.

2.2 Datenunabhängigkeit

In den 70er und 80er Jahren folgte die Entwicklung und der Ausbau der relationalen Datenbanksysteme (RDBMS), die durch weitere Abstraktion des Datenmodells, dem sogenannten 3-Ebenen-Modell, eine Trennung zwischen der physischen und der logischen Datenebene realisieren und damit die Datenmanipulationssprache von der Programmiersprache der Anwendungsentwicklung separiert [30].

Das 3-Ebenen-Modell
1975 wurde durch das ANSI/SPARC-Standardisierungskomitee das 3-Ebenen-Modell vorgeschlagen, das eine abstrakte Architektur für ein Datenbanksystem beschreibt und die Grundlage für alle modernen Datenbanksysteme bildet. Ziel ist es dem Benutzer eine abstrakte Sicht auf die von ihm benötigte Daten zu geben und diese Sicht von der konkreten physischen Sicht zu trennen [32].

Das Modell besteht aus drei Abstraktionsebenen, wie in Abb. 1 auf Seite 5 illustriert:

- externe Ebene
- konzeptuelle Ebene
- physische Ebene

Die externe Ebene umfasst eine Anzahl von externen Schemata oder Benutzersichten. Dabei beschreibt ein externes Schema die Datenbanksicht einer Gruppe von Datenbanknutzern, zum Beispiel Vertrieb oder Personalwesen.

Der konzeptuellen Ebene liegt ein konzeptuelles Schema zugrunde, das mittels eines von dem DBMS bereitgestellten Datenmodells die logische Struktur der gesamten Datenbank für eine Gemeinschaft von Benutzern erfasst. Es handelt

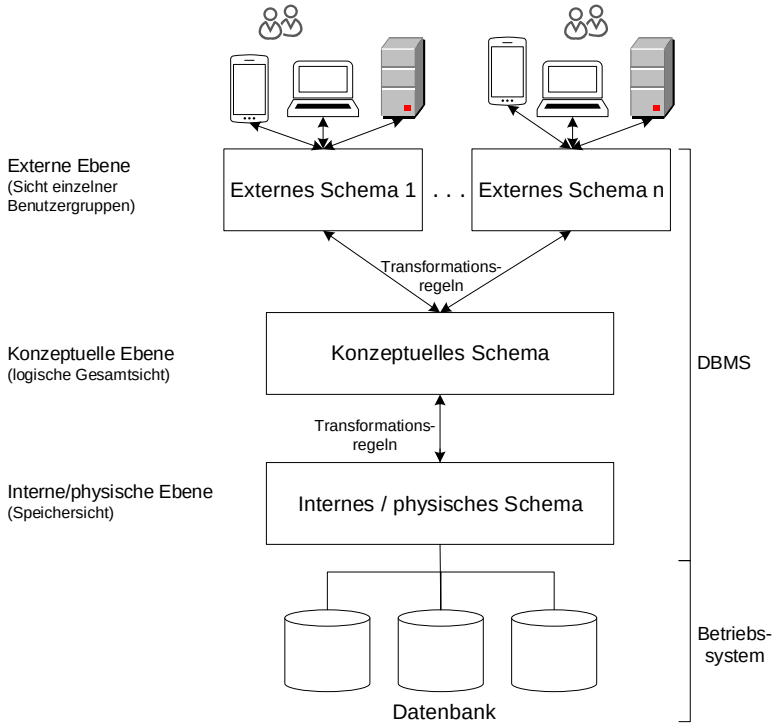


Abbildung 1: 3-Ebenen-Modell eines DBS (in Anlehnung an Schneider [32]).

sich dabei um eine globale Beschreibung der Datenbank, bei der die physischen Speicherstrukturen verborgen bleiben.

Die physische Ebene besteht aus einem physischen, internen Schema, das die physischen, persistenten Speicherstrukturen der Datenbank beschreibt. Dazu gehören der Aufbau der Daten, ihre physische Speicherung und Zugriffspfade.
Transformationsregeln (mappings) definieren die Beziehungen zwischen jeweils aufeinander folgenden Ebenen.

Durch die Abbildung im 3-Ebenen-Modell bleibt die jeweils höhere Ebene unbeeinflusst gegen Änderungen auf niedrigeren Ebenen des Modells. Durch diese Abstraktion ergibt sich eine gewisse Datenunabhängigkeit. Je nachdem, auf welcher Ebene Daten geändert werden, unterscheidet man zwischen der logischen und der physischen Datenunabhängigkeit. Dabei bedeutet die logische Datenunabhängigkeit, dass sich Änderungen des konzeptuellen Schemas nicht auf die externen Schemata auswirken. Physische Datenunabhängigkeit bedeutet, dass sich Änderungen am internen Schema (zum Beispiel Änderung Zugriffsstruktur in B-Baum, Austausch von Algorithmen) nicht auf das konzeptuelle Schema auswirken.
Datenbanksysteme (DBS) ermöglichen die integrierte Speicherung von großen Datenbeständen, auf die mehrere Anwendungen gleichzeitig zugreifen können.

Hierbei garantiert das Prinzip der Datenunabhängigkeit die weitestgehende Unabhängigkeit der Datenrepräsentation von Optimierung und Änderung der Speicherstrukturen. Sie ermöglicht auch eine Reaktion auf Änderungen der Anwendungsanforderungen, ohne die logische Struktur der Daten ändern zu müssen [32].

2.3 Architektur eines Datenbankmanagementsystems

Codd, der 1981 den ACM Turing Award für seine fundamentalen und fortwährenden Beiträge zu Theorie und Praxis von DBMS erhielt, hat 1982 die Anforderungen an ein Datenbankmanagementsystem beschrieben [5]. Er ist auch Urheber des relationalen Datenmodells, das seit den späten 80er Jahren als etabliert gilt. Als wichtigste Ziele, die mit dem relationalen Datenmodell erreicht werden sollen, erachtete Codd die physische und logische Datenunabhängigkeit, die Erleichterung der Kommunikation durch ein gemeinsames Verständnis über die Datenbasis und die Ermöglichung der mengenorientierten Verarbeitung der Daten. Dabei soll es eine klar definierte Verbindung zwischen logischen und physischen Aspekten im DBMS geben [5].

Die von Codd beschriebenen Anforderungen lassen sich verallgemeinern, sodass ein modernes Datenbanksystem – unabhängig vom gewählten Datenmodell – folgende Anforderungen erfüllen sollte [32].

Anforderungen an ein Datenbankmanagementsystem (DBMS):

- Datenunabhängigkeit
- Effizienter Datenzugriff
- Gemeinsame Datenbasis
- Nebenläufiger Datenzugriff
- Fehlende oder kontrollierte Redundanz
- Konsistenz und Integrität der Daten
- Datensicherheit (Schutz vor unautorisiertem Zugriff, Zugriffskontrollen)
- Bereitstellung von Backup- und Recovery-Verfahren
- Stellen von Anfragen mittels einer Anfragesprache (query language)
- Bereitstellung verschiedenartiger Benutzerschnittstellen
- Flexibilität bzgl. Änderungen der Datenbankstruktur oder bei Änderung von Anwendungsprogrammen
- Schnellere Entwicklung von neuen Anwendungen

Mit dem nebenläufigen Zugriff ermöglicht ein DBMS den Mehrbenutzerbetrieb (concurrent access), bei dem mehrere Benutzer gleichzeitig ändernd auf den Datenbestand zugreifen können. Durch das Konzept der Transaktionen, bei dem Datenbankoperationen, die zusammengehören, zu einer Transaktion zusammengefasst werden, stellt das DBMS sicher, dass die Operationen konsistenzerhaltend durchgeführt werden. Dies wird durch das sogenannte ACID-Prinzip sichergestellt.

ACID steht als Abkürzung für **A**tomicity **C**onsistency **I**solation **D**urability mit folgender Bedeutung [30]:

- Atomicity: Die Transaktion wird als Ganzes ausgeführt. Tritt während der Ausführung der Transaktion ein Fehler auf, so wird diese komplett zurückgerollt.
- Consistency: Bei Ausführung der Transaktion werden die Daten von einem konsistenten Zustand wieder in einen konsistenten Zustand überführt.
- Isolation: Änderungen durch die Transaktion werden erst sichtbar, wenn diese erfolgreich ausgeführt wurde.
- Durability: Die Änderungen durch die Transaktion werden in der Datenbank persistiert.

Mit dem Einsatz eines DBMS soll auch die Redundanz von Daten vermieden werden. In einigen besonderen Fällen sollte jedoch eine kontrollierte Redundanz durch das DBMS ermöglicht werden. Bei der kontrollierten Redundanz wird ein ausgewählter Teil der Daten bewusst redundant gehalten, um logische Beziehungen zwischen den Daten zu erhalten oder um eine bessere Performance für oft zusammen angefragte Daten zu ermöglichen. Allerdings ist es für die Konsistenzerhaltung der Daten dann wichtig, dass Änderungen der redundant gehaltenen Daten die Propagation (Verteilung) der Änderungen zu den Kopien unterstützt durch das DBMS schnellstmöglich erfolgt.
Um die Integrität, also die Korrektheit und Vollständigkeit der Daten, zu gewährleisten, sollte ein DBMS Funktionen für die Pflege von Integritätsbedingungen und die automatische Prüfung dieser unterstützen.

Im Gegensatz zur traditionellen Datenorganisation, bei der bei einer Anfrage an einen Datenbestand mittels eines eigenen Anwendungsprogramms gestellt wurde, soll das DBMS eine Abfragesprache (query language) zur Verfügung stellen, die sowohl von Benutzern für Adhoc-Anfragen, als auch von Anwendungsprogrammen eingebettet in die Programmiersprache genutzt werden kann. Eine über verschiedenartig bereitgestellte Benutzerschnittstellen, ob direkt oder über eine Benutzeroberfläche oder durch Nutzung einer Programmierschnittstelle, gestellte Anfrage sollte effizient beantwortet werden. Das DBMS soll Funktionen bereitstellen, die eine Anfrage auf Korrektheit prüfen (lexikalische Analyse und Syntaxanalyse) und diese entsprechend optimieren und effizient ausführen.

Ferner soll die Definition des Datenbestandes mit einer Datendefinitionssprache (data definition languages, kurz DDL) und die Manipulation mit einer Datenmanipulationssprache (data manipulation languages, kurz DML) erfolgen, die von dem DBMS zur Verfügung gestellt wird. Schließlich soll eine gewisse Flexibilität für Änderungen an der Struktur der Datenbank und bei der Erstellung von neuen Anwendungsprogrammen gegeben werden. Bei der Anwendungsentwicklung muss zum Beispiel die Prüfung von Integritätsbedingungen oder die Nebenläufigkeit nicht programmiert werden, da diese bereits durch das DBMS gewährleistet sind.

Aus der Vielzahl der hier beschriebenen Anforderungen wird klar, dass ein DBMS ein komplexes Softwareprodukt ist, dessen grundlegenden Komponenten im Architekturmodell in Abbildung 2 dargestellt werden.

Die DBMS-Software ist hierarchisch in Schichten organisiert (Schichtenarchitektur). Jede dieser Schichten übernimmt bestimmte festgelegte Aufgaben. Dabei kann jede Schicht über eine Schnittstelle auf Objekte und Operationen der ihr direkt unterliegenden Schicht zugreifen. Jede Schicht stellt Schnittstellen für die Kommunikation zwischen den direkt um sie befindlichen Schichten bereit. Interna der jeweiligen Schicht bleiben dabei gekapselt und damit den anderen Schichten verborgen [32]. Dies hat den Vorteil, dass interne Objekte und Operationen der Schicht geändert werden können, ohne dass dies Auswirkungen auf die anderen Schichten des DBMS hat.

Weitere Details zu den einzelnen Schichten eines DBMS sind im Anhang A beschrieben.

2.4 Relationale Datenbanken

Das Relationale Datenmodell

Im relationalen Modell wird ein Modell der realen Welt in Relationen abgebildet. Eine Relation ist eine Teilmenge eines kartesischen Produkts über Mengen von Werten. Ein Element der Relation wird als Tupel bezeichnet. Jede Relation wird als Tabelle dargestellt. Dabei werden Objekte und Beziehungen zwischen diesen in Tabellen modelliert. Jede Tabelle hat einen Namen, zum Beispiel den Entity-Typ, und besteht aus einer Menge von Datensätzen (Zeilen) desselben Typs. Ein Tupel (Zeile) entspricht dabei einer Entity und beschreibt diese über die Kombination der Werte. Anfragen oder Datenmanipulationen werden auf diesen Mengen ausgeführt, d.h. diese sind mengenorientiert.

Definition Relation: [30]
Sind $D_1, D_2, \ldots, D_n$ Mengen von Werten, so ist $R \subseteq D_1 \times D_2 \times \ldots \times D_n$ eine n-stellige Relation über den Mengen (domains) und n ist der Grad (degree) der Relation. Ein Element $r = (d_1, d_2, \ldots, d_n) \in R$, $(d_i \in D_i, i = 1, \ldots, n)$ ist ein Tupel der Relation (n-Tupel).

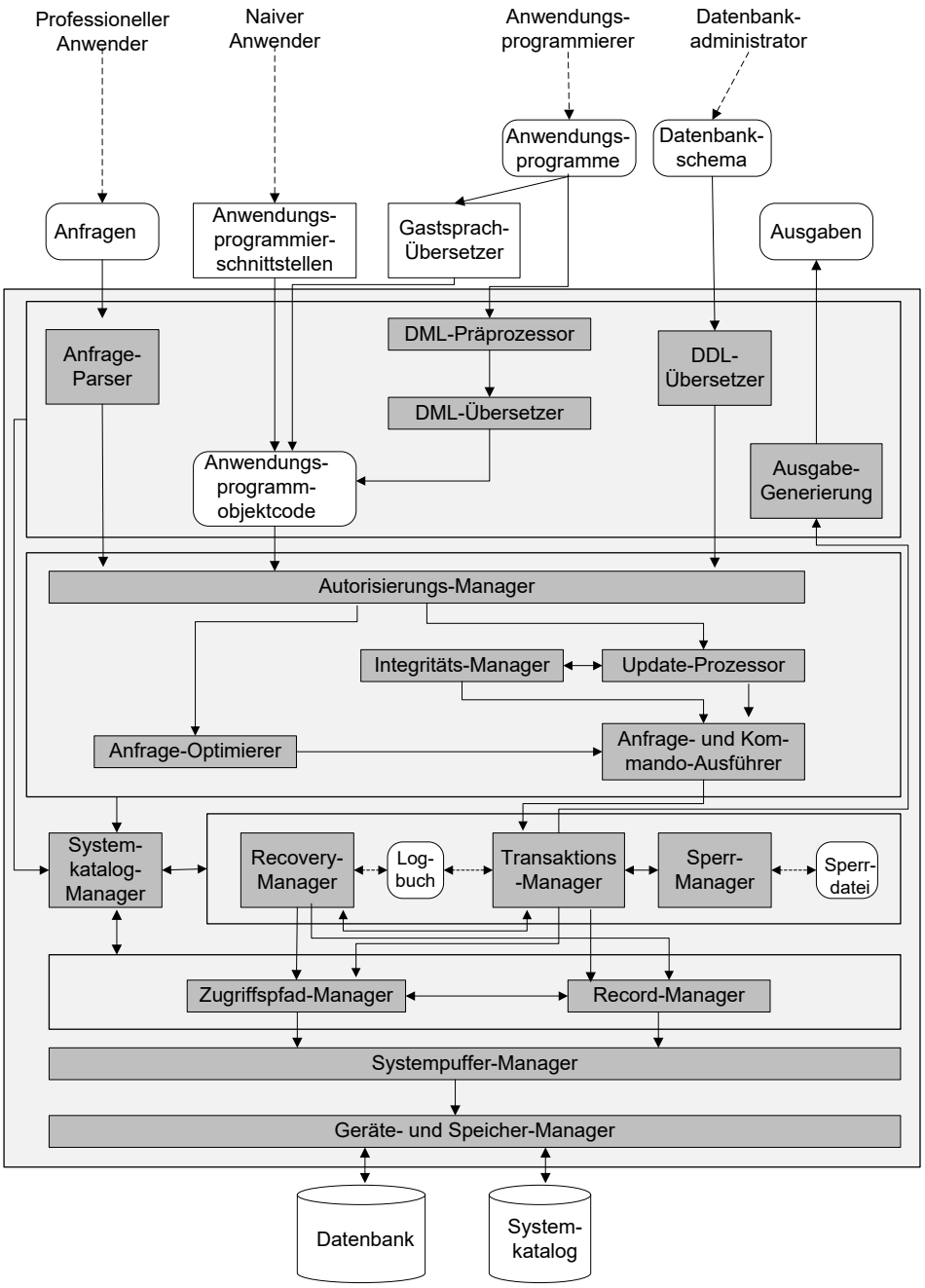


Abbildung 2: Softwarearchitektur eines DBMS (in Anlehnung an Schneider [32]).

Definition Relationenschema: [30]
Ein Relationenschema $R(A_1, \ldots, A_n)$ spezifiziert eine Relation mit Namen R und mit den paarweise verschiedenen Attributen $A_1, \ldots, A_n$. Jedem Attribut A_i ist ein Wertebereich dom(A_i) zugeordnet. Die zu $R(A_1, \ldots, A_n)$ gehörigen Relationen sind also sämtlich Relationen des Typs $R \subseteq$ dom$(A_1) \times$ dom$(A_2) \times \ldots \times$ dom(A_n). In einer Datenbank existiert zu jedem Zeitpunkt genau eine Relation R zum Relationenschema $R(A_1, \ldots, A_n)$.

Jedes Attribut einer Relation muss elementar sein, d.h. es setzt sich nicht aus mehreren Werten zusammen.

Definition Schema der Datenbank: [30]
Die Gesamtheit der Relationenschemata einer Datenbank heißt Schema der Datenbank. Dazu gehören auch die Menge der Integritätsbedingungen.

Das Relationenschema dient als Basis für die Modellierung von relationalen Datenmodellen.

Vom Entity-Relationship-Modell (ERM) bis zur Datenbank
Ein weit verbreitetes Tool zur Modellierung ist das Entity-Relationship-Modell (ERM), das interaktiv anhand der Anforderungen und des abzubildenden Bereichs erstellt wird. Bei der Modellbildung werden Objekte und Beziehungen der realen Welt bzw. aus dem betrachteten Bereich in einem Modell, dem sogenannten Entity-Relationship-Modell, abgebildet. Dabei werden Objekte, sogenannte Entities, oder Beziehungen mit gleichen Eigenschaften zu einem Typ zusammengefasst. Diese werden dann bei der Erstellung des Modells betrachtet. Ein solcher Entity-Typ könnte zum Beispiel STUDENT oder KURS sein, der Teil eines Modells einer Universität ist. Der Entitytyp STUDENT repräsentiert dabei die Menge aller Studenten. Ein entsprechender Beziehungstyp könnte BELEGT sein. In dem Modell würde u.a. ein Beziehungstyp BELEGT(STUDENT, KURS) definiert und graphisch dargestellt werden. Sowohl Entity-Typen als auch Beziehungstypen besitzen Attribute. Die Komplexität der Beziehung wird im Modell angeben, ebenso Integritätsbedingungen. Das in der Diskussion entstandene Modell wird anschließend in ein Relationenschema überführt, das Ausgangsbasis für den Datenbankentwurf ist.

Im weiteren Prozess des Datenbankentwurfs wird das aus dem ER-Modell abgeleitete Relationenschema mittels eines Algorithmus [32] in die sogenannte dritte Normalform (3NF) oder sogar in die sogenannte Boyce-Codd-Normalform (BCNF) überführt. Bei dem Prozess der Normalisierung werden mögliche Redundanzen und Anomalien in den Relationen beseitigt und die Ausgangs-Relationen entsprechend aufgeteilt. Des Weiteren werden Schlüssel und Fremdschlüssel definiert, die benötigt werden, um eine eindeutige Identifizierbarkeit herzustellen und um aufgeteilte Relationen bei Bedarf wieder zusammenführen

zu können (Join-Operation). Nach der Überführung des Relationenschemas in die gewünschte Normalform werden die Tabellen in der Datenbank mittels SQL bzw. einer Datenbeschreibungssprache, kurz DDL, angelegt und dabei Schlüssel und Integritätsbedingungen definiert.

Die Abfragesprache SQL
Die Standard-Abfragesprache der Relationalen Datenbanken ist SQL, weswegen diese oft auch SQL-Datenbanken genannt werden. SQL beinhaltet eine Datendefinitionssprache (DDL) mit der Tabellen angelegt, die Datentypen der Attribute festgelegt werden, Schlüssel und Fremdschlüssel festgelegt werden und auch Indexe zu den Schlüsseln der Tabellen angelegt werden. Auch Integritätsbedingungen werden damit definiert. Der DML-Teil von SQL wird für Anfragen, als auch für die Manipulation der Daten in den Tabellen verwendet. So sind damit neben CRUD-Operationen (CRUD steht für Create Read Update Delete) auch komplexe Anfragen an die Datenbank möglich, bei denen Tabellen mit einander verbunden werden, um mit sogenannten Joins Beziehngen zwischen den abgebildeten Relationen herzustellen. SQL baut auf der Relationenalgebra auf, wurde aber über die Jahre um viele weitere Sprachkonstrukte erweitert, die die Verwaltung und Auswertung von u.a. temporalen Daten, multidimensionalen Daten (Spatial-, Geo-Daten) und objektorientierte Datenstrukturen in Verbindung mit Relationen ermöglicht. Eine Anfrage mit SQL hat folgenden allgemeinen Aufbau [30]:

SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_n$ WHERE $Prädikat(R_1, \ldots, R_n)$;

Dabei stehen die A_i für Attribute, die R_i für Relationen (Tabellennamen) und $Prädikat(R_1, \ldots, R_n)$ für eine Bedingung, die zusammengesetzt sein kann aus mehreren Bedingungen, die logisch miteinander verknüpft sind. Hierüber werden u.a. auch Bedingungen für den Verbund von Relationen (JOIN) angegeben. Die Bedeutung der obigen SQL-Query kann in Umgangssprache folgendermaßen ausgedrückt werden: Selektiere aus dem Kreuzprodukt der Relationen $R_1, \ldots, R_n$ alle Tupel, die die Bedingung $Prädikat\ (R_1, \ldots, R_n)$ erfüllen und projiziere die so entstandene Relation auf die Attribute $A_1, \ldots, A_n$. Die Anfrage ist hierbei mengenorientiert, d.h. es wird beschrieben, welche Menge von Daten als Zielmenge erwartet wird, doch nicht, wie diese Daten ermittelt werden. Diese werden dann vom Datenbankmanagementsystem mittels im DBMS implementierten Algorithmen berechnet und ausgegeben.

Die Funktionalität der Datendefinition mit SQL wird anhand des folgenden Beispiels beschrieben; siehe Listing 1:

Listing 1: Beispiel Datendefinition mit SQL.

```
CREATE TABLE STUDENT_KURS
(KURSNR INTEGER,
MATRIKELNR INTEGER,
BESTANDEN INTEGER,
CONSTRAINT STUDENT_KURS_PK PRIMARY KEY
        (KURSNR, MATRIKELNR),
CONSTRAINT STUDENT_KURS_MATRIKELNR_FK FOREIGN KEY
        (STUDENT) REFERENCES STUDENT(MATRIKELNR),
CONSTRAINT STUDENT_KURS_KURSNR_FK FOREIGN KEY
        (KURSNR) REFERENCES KURS(KURSNR))
```

Diese Anweisung erzeugt eine Tabelle namens STUDENT_KURS mit den drei Spalten KURSNR, MATRIKELNR, BESTANDEN mit dem Wertebereich des einfachen Datentyps INTEGER. Mit CONSTRAINT STUDENT_KURS_PK PRIMARY KEY (KURSNR, MATRIKELNR) wird der Primärschlüssel, bestehend aus der Kombination von KURSNR und MATRIKELNR, gebildet. Er darf daher nicht leer sein und muss auch eindeutig sein.

Mit CONSTRAINT STUDENT_KURS_MATRIKELNR_FK FOREIGN KEY (STUDENT) REFERENCES STUDENT(MATRIKELNR) wird festgelegt, dass MATRIKELNR ein Fremdschlüssel der Relation STUDENT sein soll. Dies bedeutet, dass für die MATRIKELNR nur ein Wert eines Schlüssels der Relation STUDENT zulässig ist. Dies gilt analog für den weiteren Fremdschlüssel, der mit dem weiteren CONSTRAINT in der Anweisung definiert wird. Mit der Definition der Tabelle werden damit auch wichtige Integritätsbedingungen festgelegt, die automatisch durch das DBMS geprüft werden und für Konsistenz der Daten in der Datenbank sorgen.

Über die hier vorgestellte Funktionalität hinaus bietet SQL noch viele weitere Funktionen, wie zum Beispiel Aggregierungsfunktionen oder die Schachtelung von Abfragen, auf die wir hier nicht näher eingehen. Diese werden von Saake et. al. genauer beschrieben [30].

2.5 Überblick Datenbankmodelle und Marktentwicklung

Die Angaben in diesem Unterkapitel zur Historie und zu den Datenbankmodellen sind angelehnt an die Ausführungen von Meier und Kaufmann [22].

Seit den späten 80er Jahren gelten relationale Datenbanksysteme als etabliert und sind heute immer noch kommerziell am meisten verbreitet. Die relationalen DBS sind durch ihr generalisierendes Modell sehr breit einsetzbar, doch gibt es Anwendungsbereiche und Nischen, bei denen die SQL-Datenbanksysteme an ihre Grenzen stoßen.

Aufgrund solcher Anforderungen entstanden in den 90er Jahren objektorientierte Datenbanksysteme, mit denen sich komplexere Objektstrukturen verwalten lassen. Diese DBS unterstützen deklarative als auch navigierende DML. Allerdings reichten die Fähigkeiten der deklarativen Sprachen nicht aus, um Definition und Abfragen dieser komplexen Datenstrukturen abzubilden. Dies wurde mit einer im DBMS integrierten Programmiersprache basierend auf C++ oder Java ermöglicht.

Im weiteren Verlauf der Geschichte wurden kommerzielle relationale DBS mit objektorientierten Strukturen angereichert, um den Funktionsumfang dieser auf die Verwaltung und Auswertung von komplex strukturierten, sowie räumlichen Daten und Geo-Daten zu erweitern. So entstanden die sogenannten objektrelationalen DBMS (ORDBMS). Dies führte auch zu einer Erweiterung des SQL:2003-Standards. Auch durch die Verbreitung von XML-Daten wurden viele kommerzielle DBMS um Datentypen und Funktionen für die Verwaltung dieser Datenbestände erweitert.

Die moderne vernetzte Welt stellt uns vor das Problem riesige Mengen an sehr unterschiedlich strukturierten Daten zu verwalten und sinnvoll auszuwerten. Diese Anforderungen gehen weit über die Leistungsfähigkeit von konventionellen DBS hinaus, sodass die Entwicklung vom SQL-Standard weg hin zu neuen Datenmodellen geht.

So entstanden die sogenannte NoSQL Datenbanksysteme, wobei mit NoSQL „not only SQL“, also nicht nur SQL, gemeint ist. Die SQL-DBS sind zwar immer noch marktführend, dennoch es gibt ein großes Wachstum in der Welt der NoSQL-Datenbanksysteme. Tabelle 1 auf Seite 14 gibt eine grobe Übersicht über die wichtigsten Datenbankmodelle. Datenbankmodelle legen fest, wie Daten im Datenbanksystem gespeichert und manipuliert werden.

Tabelle 1: **Übersicht Datenbankmodelle.**

Datenbankmodell	**Beschreibung, Merkmale**	**Abfragesprache**	**Einordnung**
Hierarchisches Modell	Verwaltung hierarchischer Daten, zum Beispiel Baumstrukturen	Zeigernavigation angelehnt an Programmiersprache	-
Netzwerkmodell	Generisches Datenmodell mit netzwerkartiger Verknüpfung	Zeigernavigation angelehnt an Programmiersprache	-
Relationenmodell	Daten normiert in Relationen, dargestellt in Tabellen mit Attributen, einfacher Datentypen	SQL	SQL
Objektorientiertes Modell	Abbildung komplex strukturierter Datentypen, Verwendung Typsystem der Programmiersprache (C++, Java)	teilweise SQL und eigene Methoden mit C++ oder Java	NoSQL
Objektrelationales Modell	Erweiterung Relationenmodell mit Konzept der Objektorientierung, komplexe Datentypen in Verbindung mit Relationen	SQL erweitert und ORM-Software (Mapping Klasse zu Tabelle)	NoSQL
Wissensbasiertes bzw. deduktives Modell	Fakten und Regeln, aus denen weitere Fakten hergeleitet werden, unterstützt Rekursion	SQL3, Datalog (regelbasierte Spracherweiterung)	SQL, deduktive DB
Dokument-Datenbanken (Schlüssel-Wert-DB)	Verwaltung von Schlüssel-Wert-Paaren, Wert entspricht Dokument (strukturierte Daten in Datensätzen), schemafrei	Map-Reduce-Verfahren	NoSQL
XML-Modell	Verwaltung von XML-basierten Dokumenten (XML = eXtended Markup Language)	XQuery	NoSQL
RDF-Modell (Resource Description Framework)	Verwaltung Web 2.0 Daten, Tripel (Subjekt, Prädikat, Objekt), auch als Graph auswertbar	SPARQL	NoSQL
SQL/MM-Modell	Multidimensionale Daten (Spatial, Multimedia)	SQL/MM	SQL
Föderirertes Modell (verteiltes Modell)	Ein logisches DB-Schema, Fragmente auf verteilten Rechnern, Anfragen lokal und verteilt	SQL erweitert (Partitionierung, Konsistenzerhaltung)	SQL
Temporales Modell	Relationenmodell mit zeitabhängigen Daten	SQL erweitert (temporal)	SQL
Graphenmodell	Daten u. Beziehungen modelliert als Graph	Cypher, SPARQL	NoSQL

Der Grund für diese Entwicklung ist, dass das SQL-basierte Datenbanksystem, das auf der Relationenalgebra basiert, an seine Grenzen stößt, wenn es um die Erfüllung von erweiterten Anforderungen in neuen Anwendungsgebieten oder die performante Auswertung von großen Datenmengen (Big Data), geht.

Big Data sind große Datenmengen, die zu bestimmten Zwecken analysiert werden sollen, wobei die Charakteristik der Daten mit vier V beschrieben werden kann [31]:

- **Volume**: Riesige Datenmengen werden verwaltet.
- **Velocity**: Die Daten werden nicht nur periodisch aktualisiert, sondern es kommen millisekündlich neue Daten hinzu, die gefiltert und ausgewertet werden müssen.
- **Variety**: Die Daten liegen in verschiedenen Modellen, Formaten und Strukturen vor. Integration und Verarbeitung von heterogenen, strukturierten und unstrukturierten Daten muss von dem System unterstützt werden.
- **Veracity**: Es kommt zum sogenannten Datenrauschen, bei der Daten durch die Integration nicht mehr eindeutig interpretierbar sind (ungenau, mehrdeutig, inkonsistent).

Die bisherigen Systeme waren sehr stark transaktionsorientiert und damit die Transaktionsverarbeitung und die damit verbundene Erhaltung der Konsistenz der Daten von großer Bedeutung. Durch stark wachsende Datenmengen, die durch die Nutzung des Internets und der Digitalisierung mit fortwährender Vernetzung entstehen und auch durch Entwicklungen der Hardware (schnellere Speicher, mehr Parallelität), ergeben sich neue Anforderungen an Datenbanksysteme:

So sollen umfangreiche strukturierte und unstrukturierte Daten in Echtzeit ausgewertet und basierend darauf Entscheidungen zum weiteren Verfahren getroffen werden können. Daher haben nicht-relationale Datenbankkonzepte gegenüber den relationalen an Gewicht gewonnen.

Es entstanden die ersten No-SQL-Datenbanksysteme, wobei No-SQL zunächst für nicht-relationale Ansätze steht, bei denen die Speicherung der Daten nicht in Tabellen erfolgt und diese auch nicht mit SQL abgefragt werden können. Diese DBS unterstützen die hoch skalierbare Verarbeitung in Rechnerclustern, die heterogene Daten mit wechselnden Strukturen akzeptiert und dafür die Zusicherung der Konsistenz der Daten als nachrangig ansieht.

Die Interpretation von No-SQL als ‚Not only SQL' kam erst auf, als eine Reihe von leichtgewichtigen Systemen entstand, die zwar Gemeinsamkeiten mit relationalen Daten und SQL haben, aber einerseits SQL nicht voll abdecken und

dafür andererseits in anderen Bereichen mehr Flexibilität bieten.

Von No-SQL auch als ‚New SQL' sprechen wir bei der neuen Generation der DBMS, die den modernen softwaretechnischen Vorstellungen entspricht, aber ohne die etablierten Vorteile von SQL zu ignorieren.

Die großen Datenbank-Hersteller von RDBMS haben auf die sich ändernden Anforderungen reagiert und neben SQL auch Erweiterungen in ihr DBMS integriert. Zu den Erweiterungen gehören unter andrem die Speicherung und Verarbeitung von Spaltenvektoren (column store), die für eine Verarbeitung im Hauptspeicher besser geeignet sind, oder die Verarbeitung von Spatial-Daten. Insgesamt sind diese sogenannten xRDBMS flexibler in Bezug auf das eingesetzte Schema der Daten [31]. So können unterschiedliche Anforderungen in den Anwendungen, wie in Abbildung 3 gezeigt, im Idealfall mit nur einem xRDBMS bedient werden.

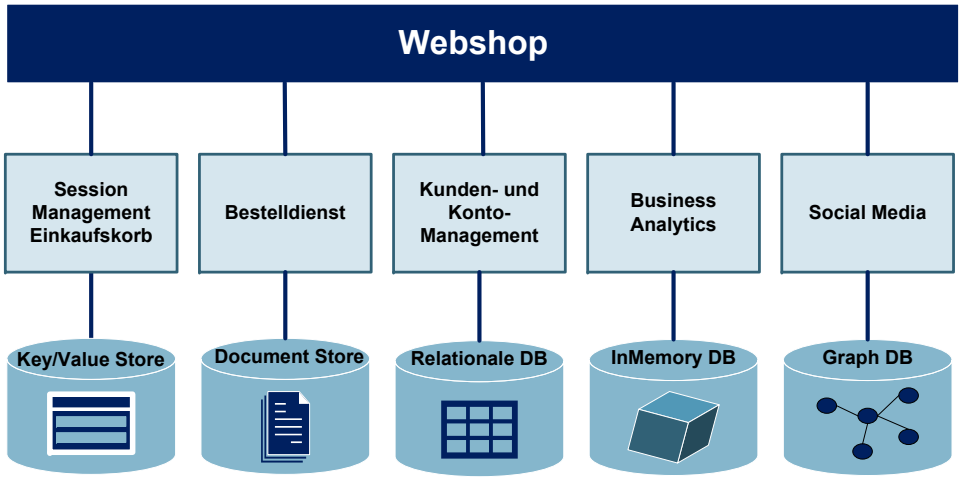


Abbildung 3: Illustration der SQL- und NoSQL-Technologien einer massiv verteilten Web-Anwendung [22].

Aktuelle Auswertungen und Trends zu DBMS werden auf der Internetseite von solidIT veröffentlicht und monatlich aktualisiert. solidIT ist ein österreichisches IT-Beratungsunternehmen mit den Schwerpunkten Softwareentwicklung, Schulung und der Beratung für datenbankorientierte Applikationen.

DB-Engines ist eine Initiative zur Sammlung und Präsentation von Informationen zu Datenbankmanagementsystemen. Neben den relationalen DBMS werden hier auch Systeme und Konzepte aus dem stark wachsenden NoSQL-Sektor betrachtet. Auf der Internetseite gibt es u.a. auch eine Rangliste der verschiedenen DBMS, die nach Kategorien gefiltert werden kann [8].

Das DB-Engines-Ranking zeigt Trends auf, die einen Überblick über die Entwicklung der verschiedenen DBMS geben. Zum Beispiel zeigt die Graphik 4 auf Seite 17 die wachsende Popularität von Graph-DBMS im Vergleich zu anderen DBMS auf. Für die Ermittlung der Trends in diesem Diagramm werden für jede Kategorie seit Januar 2013 die drei besten Systeme jedes Monats herangezogen und deren Popularitätspunkte gemittelt. Der Anfangswert jeder Kategorie wird dabei auf 100 normiert.

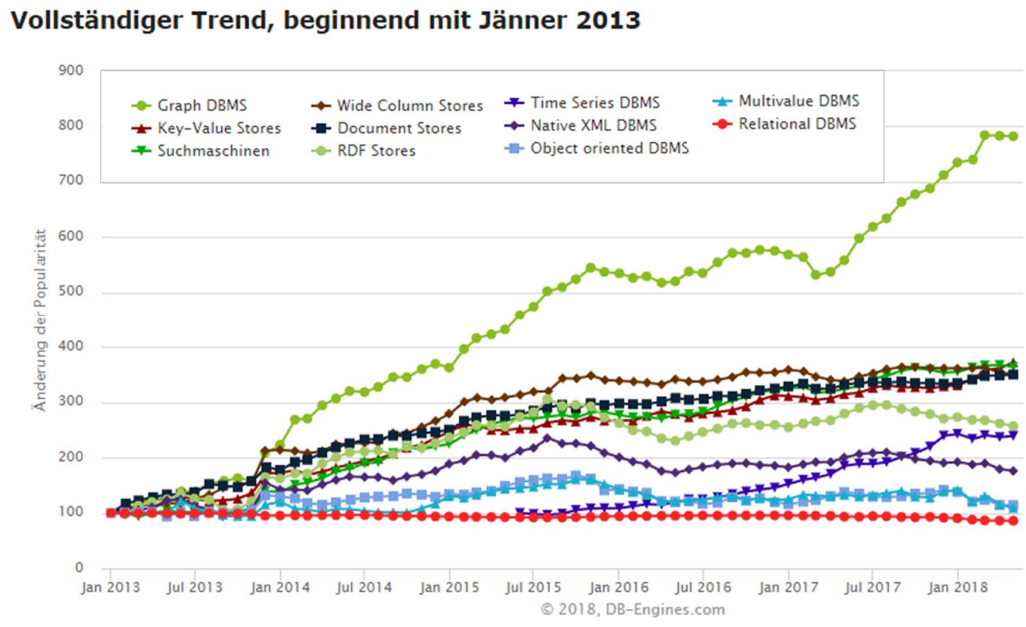


Abbildung 4: Trend der DBMS-Popularität seit Januar 2013 [8].

3 Graphdatenbanken anhand Neo4j

Graphdatenbanken zählen zu den sogenannten NoSQL-Datenbanken. Das ihnen zugrunde liegende Modell ist das Graphenmodell, eine Datenrepräsentation, die insbesondere für die Darstellung von Beziehungen zwischen Objekten geeignet ist. Dabei verstehen wir formal unter einem Graphen G eine Struktur aus Knoten und Kanten $G = (V, E)$, wobei V die Menge der Knoten und $E \subseteq V \times V$ die Menge der Kanten, die Knoten aus V verbinden, bezeichnen. Die Kanten eines Graphen können gerichtet oder ungerichtet sein. Zudem können Kanten auch mit einem Gewicht belegt sein, das je nach Anwendung eine andere Bedeutung haben kann, wie zum Beispiel eine Entfernung zwischen zwei Punkten.

Repräsentation von Graphstrukturen
Die am gebräuchlichsten Repräsentationen eines Graphen sind die Adjazenzmatrix und die Adjazenzliste. Hierbei wird die Kantenbeziehung zwischen zwei Knoten des Graphen in Form einer Datenstruktur gespeichert. Wird zum Beispiel eine Adjazenzmatrix genutzt, um den Graphen abzubilden, so wird in einer $n \times n$ - Matrix (a_{ij}) an Stelle (i, j) eine Eins eingetragen, wenn Knoten i und Knoten j, des Graphen adjazent sind, also direkt miteinander verbunden sind ($0 \leq i, j \leq n$ und $i \neq j$). Sonst wird eine Null eingetragen. Bei gerichteten Graphen berücksichtigt man zudem die Richtung. Bei gewichteten Graphen wird statt der Eins dann das Gewicht der Kante eingetragen. Bei Graphen mit wenigen Kanten bietet es sich an, den Graphen mit einer Adjazenzliste abzubilden. Hier wird je Knoten eine Liste mit seinen direkten Nachfolgern geführt. Auch hier kann man je Listenelement das Gewicht einer Kante mitführen.

Operationen und Anfragen auf Graphen

- Bestimmen des Knotensgrades, das ist die Summe der Anzahl der ausgehenden und eingehenden Kanten.
- Traversierung des Graphen mittels Tiefensuche und Breitensuche.
- Filtern, Extrahieren von Graphdaten mittels einer Anfragesprache.
- Anwendung von Graph-Algorithmen der Graphentheorie.

Typische Grundelemente von Anfragesprachen auf Graphen [31]:

- Filter für Knoten- und Kanteneingenschaften, die Knoten und Kantenmenge anhand von Prädikaten über ihren Eigenschaften liefern,
- Graph Matching zur Extraktion von Knoten, Kanten und Pfaden zwischen Knoten anhand von Mustern,
- die Konstruktion neuer Graphen auf Basis gegebener (d.h. durch andere Anfragekonstrukte ermittelte) Knoten und Kanten.

Bei den Graphdatenbanken unterscheiden wir zwei Modelle, das RDF-Modell (RDF steht für Resource Description Framework) und das Property-Graph-Modell [31]:
Das RDF-Modell besteht aus einer Menge von Tripeln, die Aussagen über Dinge (Web-Ressourcen) repräsentieren. Dabei werden alle Ressourcen über sogenannte Web-Identifikatoren (Uniform Resource Locator - URI) identifiziert. Aussagen beschreiben diese Ressourcen über ihre Eigenschaften (Prädikate) und Ausprägungen. Ein solcher Tripel besteht daher aus `subjekt prädikat objekt`. Dabei können sich mehrere Aussagen durchaus auf das gleiche Subjekt beziehen. Auch Objekte können nicht nur Literale, sondern selbst wieder URIs sein und daher auf andere Subjekte verweisen. Durch dieses Beziehungsgeflecht entstehen dann komplexe Graphen, die dann mittels der Anfragesprache SPARQL, einer vom World Wide Web Consortium (W3C) standardisierten Anfragesprache, ausgewertet werden können.
Beim zweiten Model, dem sogenannten Property-Graph-Modell, werden Knoten und Kanten direkt mit Eigenschaften (Properties) modelliert. Dabei sind Properties hier nicht streng typisiert, sie werden über Name-Wert-Paare abgebildet. Im Vergleich zum RDF führt das Property-Graph-Modell oft zu einer kompakteren Darstellung, insbesondere, wenn auch Kanten eindeutig identifiziert oder mit Eigenschaften versehen werden sollen. Als Anfragesprache wird hier eine deklarative Sprache, wie zum Beispiel Cypher genutzt. Cypher ist im Gegensatz zu SPARQL noch nicht als Standard etabliert. Mit dem OpenCypher-Projekt laufen jedoch Bemühungen diese als Industrie-Stand zu etablieren.

Das Graphenmodell für Datenbanken nutzt damit Definitionen und Algorithmen der Graphentheorie. Zum Beispiel enthalten Graph-DBMS u.a. Algorithmen zur Berechnung von Zusammenhangskomponenten, zur Berechnung von kürzesten Pfaden, zur Ermittlung des nächsten Nachbarn oder der Berechnung von Matchings [22]. Bereits über diese grundlegenden Algorithmen können zum Beispiel Freundschaftsbeziehungen in einem sozialen Netzwerk anhand von Pfadlängen viel einfacher als bei klassischen RDBMS ausgwertet werden.

Typische Anwendungsfälle für den Einsatz von Graph-Datenbanksystemen sind zum Beispiel die folgenden:

- Auswertungen von Sozialen Netzwerken, zum Beispiel Soziogramm, Freundschaftsbeziehungen, Clicken etc.
- Recommendations: Empfehlungen, zum Beispiel basierend auf Vorlieben oder vorherigen Einkäufen
- Zum gezielten Marketing, das ein Netzwerk durchdringen soll
- Auswertungen der vernetzten Infrastruktur eines Rechenzentrums, Darstellung von Abhängigkeiten, Root-Cause-Analysis

- Bestimmen von kürzesten Pfaden, Routenplanung, zum Beispiel finde alle Restaurants in der Nähe des Bahnhofs
- Auswirkungen eines Sicherheitslochs in Netzwerken oder der Auswirkungen eines Virenbefalls
- Verwaltung von (hierarchischen) Stammdaten, inkl. zugehöriger Metadaten

3.1 Das Graphenmodell in Neo4j

Als konkretes Beispiel eines Property-Graph-Datenmodells betrachten wir in dieser Arbeit das Open-Source-Graph-Datenbanksystem Neo4j. Dabei orientieren wir uns an Informationen und der Dokumentation des Herstellers Neo4j, Inc. [34].

In Neo4j wird ein Graph nativ gespeichert, d.h. auch physisch in Form von Knoten und Kanten in Adjazenzlisten abgebildet, die, wie in der Graphentheorie üblich, traversiert und dabei entsprechend bearbeitet und ausgewertet werden können. Das bedeutet, dass das logische und das physische Datenmodell in diesem Fall nahezu beieinander liegen. Dabei müssen Beziehungen der Daten untereinander bei Auswertungen nicht, wie im relationalen Datenbanksystem üblich, durch aufwändige Join-Operationen hergestellt werden. Auswertungen von stark vernetzten Daten, wie zum Beispiel einem sozialen Netzwerk, sind dadurch sehr performant möglich.

Grundsätzlich wird in Neo4j der Graph als gerichteter Graph geführt. Zur Steigerung der Performanz werden in doppelt verketteten Listen sowohl Links zu Vorgänger und Nachfolger des jeweiligen Knotens geführt. Dadurch kann der Graph effizient in beiden Richtung traversiert werden. Durch die Speicherung als Eigenschaftsgraph (property graph) ist es möglich, alle relevanten Informationen in Form von Eigenschaften und Labels an die Knoten und Kanten des Graphen anzuhängen. Die Struktur eines Graphen in Neo4j ist also wie folgt organisiert:

Knoten (nodes):

- Knoten entsprechen Objekten (Entities)
- Knoten sind durch Beziehungen miteinander verbunden
- Knoten können eine oder mehrere Eigenschaften haben
- Knoten können eine oder mehrere Bezeichnungen (Schlüssel-Wert-Paare) haben, je nach Rolle des Knotens im Graphen

Kanten (relationships):

- Kanten verbinden zwei der oder mehr Knoten miteinander (Verbindungen, Hierarchien)
- Die Kanten sind gerichtet
- Durch Kantenverbindungen können auch Zyklen abgebildet werden
- Kanten können eine oder mehrere Eigenschaften haben

Eigenschaften (properties):

- Eigenschaften sind bezeichnete Werte (enum) vom Typ String
- Eigenschaften können mit Integritätsbedingungen versehen werden und auch indexiert werden
- Es können zusammengesetzte Indexe aus mehreren Eigenschaften gebildet werden

Bezeichnungen (labels):

- Bezeichnungen ermöglichen die Gruppierung von Knoten in Mengen
- Ein Knoten kann mehrere Bezeichnungen haben
- Zu Bezeichnungen (Namen) kann ein Index aufgebaut werden, mit dem sich Knoten schnell im Graphen finden lassen
- Entspricht die Bezeichnung bereits einem Schlüssel, zum Beispiel bei nummerierten Knoten, dann ist die Verarbeitung insgesamt schneller

Um den Vorgang der Suche und des Filterns bestimmter Eigenschaften oder Labels performant durchzuführen, können wir zusätzlich zu der nativen Speicherung des Graphen auch Indexe auf Eigenschaften und Labels in Neo4j anlegen. Dies ermöglicht das Filtern von Eigenschaften und Labels, sodass nur relevante Teile des Graphen in die weitere Auswertung einbezogen werden. Allerdings ist dabei auch zu bedenken, dass ein Durchsuchen von Indexen in einigen Fällen evtl. länger dauern kann, als einen Graphen zu traversieren. Dies sollte also beim Datenbankentwurf und der Weiterentwicklung des abgebildeten Modells berücksichtigt werden. Bevor wir näher auf den Entwurfsprozess eingehen, wollen wir zunächst den Aufbau und die Funktionen von Neo4j unter Einbeziehung der in Kapitel 2.3 beschriebenen Anforderungen an ein Datenbanksystem beschreiben.

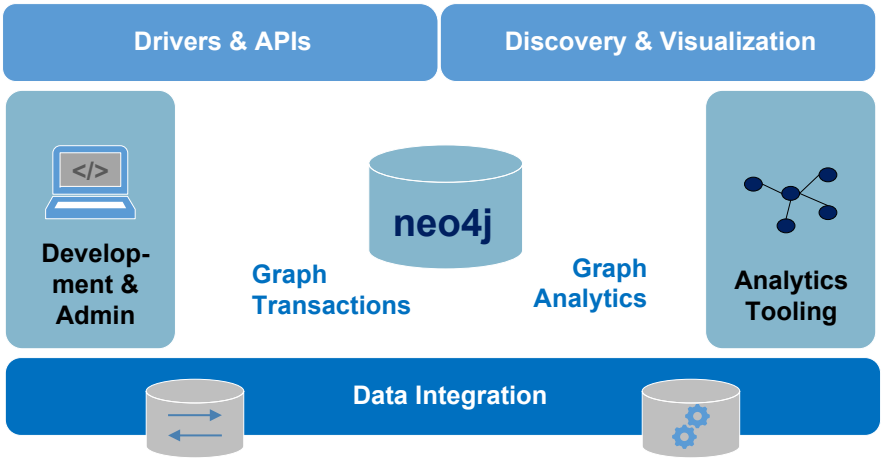


Abbildung 5: Die Neo4j Plattform, in Anlehnung an Neo4j Inc. [34].

3.2 Aufbau und Funktionen von Neo4j

Abbildung 5 zeigt die Neo4j-Plattform mit den bereitgestellten Komponenten, die wir im Folgenden näher beschreiben werden.

Neo4j kann entweder als Desktop-Version mit integrierter Datenbank auf einem Einzelplatz-System (PC oder Laptop) oder als dedizierter Datenbank-Server installiert werden. Mit der Neo4j-Desktop-Software ist es auch möglich, sich mit einer Neo4j-Datenbank zu verbinden, die auf einem separaten Server installiert ist, zuzugreifen, vorausgesetzt dass die Kommunikation in dem genutzten Netzwerk entsprechend eingerichtet ist, inklusive der Freischaltung von genutzten Protokollen und Ports. Die Installation ist auf verschiedenen Plattformen mit Betriebssystemen Windows, MAC OS X und verschiedenen Linux-Derivate möglich. Die genauen Systemvoraussetzungen und Installationsanleitung sind in der Neo4j-Dokumentation [23] beschrieben. Der Download der Software und die Installation von Neo4j, wie wir sie in den Experimenten verwenden, ist im Kapitel 7.2 genau beschrieben.
Neo4j unterstützt sowohl die transaktionale (OLTP) als auch die analytische Verarbeitung (OLAP) des in der Datenbank abgebildeten Graphen. Mit dem Neo4j Browser, der Bestandteil der Neo4j Software ist, können wir Auswertungen sowohl in Form von Tabellen als auch in einer graphischen Darstellung vornehmen. In der graphischen Darstellung wird zunächst ein Ausschnitt gemäß der abgefragten Filterbedingungen angezeigt. Dabei lässt sich die Anzahl der zu zeigenden Knoten und Kanten beschränken. In dem angezeigten Graphen können wir uns mit einem Doppelklick auf einen Knoten oder eine Kante weitere Details zu dem Graphen anzeigen lassen. Waren zuvor direkte Kanten eines Knotens noch nicht sichtbar, so werden diese nun angezeigt. In dem angezeigten Graphen können wir auch Knoten verschieben, um Teile des Graphen übersicht-

licher anzuzeigen. Durch das Markieren mit verschiedenen Farben können Teile des Graphen hervorgehoben werden, um zum Beispiel gefundene Muster besser sichtbar zu machen.

Durch die von Neo4j bereitgestellten Treiber und Schnittstellen können Daten auf verschiedene Weise importiert und exportiert werden oder direkt in einer Anwendung geladen, manipuliert und analysiert werden. Abbildung 6 zeigt eine grobe Übersicht über die Neo4j Architektur.

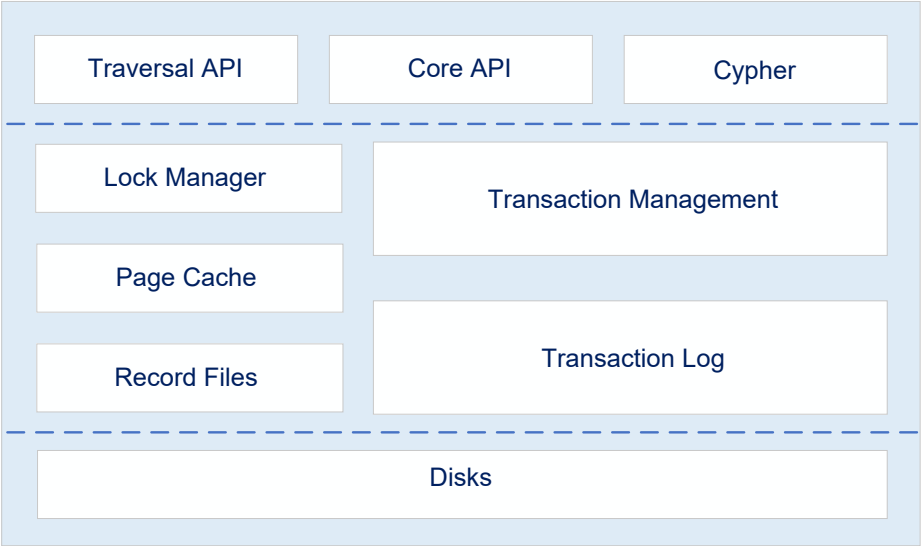


Abbildung 6: Übersicht Neo4j Architektur, in Anlehnung an Robinson [29].

Als Schnittstellen (Application Programming Interface API) zum Benutzer werden hier, die Kernel API, die Core API und Cypher aufgeführt.
Die Kernel-API ist eine Low-Level-Schnittstelle, die auf Event-Handlern basiert. Event-Handler sind Methoden, die beim Eintreten bestimmter Ereignisse ausgeführt werden. Ein Beispiel dafür wäre das beim Löschen einer Beziehung im Graphen, bei der gewisse Prüfungen durchgeführt werden und die Löschung nur erfolgt, wenn diese erfolgreich sind. Die Traversal-API (bzw. das Traversal Framework), ist eine deklarative Java API. Darüber können Bedingungen formuliert werden, die die weitere Navigation des Graphen einschränken, indem zum Beispiel nur ein bestimmter Beziehungstyp berücksichtigt wird. Das Traversieren des Graphen wird also eingeschränkt auf den interessierenden Bereich. Dabei kann auch die Richtung der Navigation vorgegeben werden und festgelegt werden, ob mit Breitensuche oder Tiefensuche im Graphen weiter navigiert wird.

Eine gängigere Schnittstelle ist die sogenannte Java Core-API von Neo4j. Diese vereint gleich mehrere Möglichkeiten mit der Neo4j-Graph-DB zu interagieren

und Abfragen und Manipulationen auf dem Graphen durchzuführen: Die Java OGM-Library (OGM bedeutet Object Graph Mapping), die von Neo4j zur Verfügung gestellt wird, ermöglicht es Objekte und Beziehungen der jeweiligen Anwendungs-Domäne mit dem Neo4j-Graph-Modell zu verknüpfen (mappen). Die Verbindung wird dabei auf Treiber-Ebene verwaltet. Unterstützt von Neo4j werden derzeit HTTP als transaktionaler Endpunkt für Remote-Verbindungen, Embedded für die Nutzung über eine direkte Verbindung mit einer Java-Applikation und BOLT ein nativer Treiber von Neo4j (binäres Protokoll). Java-Anwendungen können mittels JDBC-Treiber die Verbindung zur Neo4j-Graph-DB aufbauen, ähnlich wie es auch beim Zugriff auf relationale Datenbanken üblich ist. Anfragen und auch Manipulationen werden dann mittels der Anfragesprache Cypher durchgeführt.

Der mittlere Bereich der Architektur zeigt uns, dass Transaktionsmanagement und Nebenläufigkeit wichtige Funktionen des Neo4j-Graph-DBMS sind. Tatsächlich werden hier im Gegensatz zu anderen No-SQL-DBMS Transaktion nach dem ACID-Prinzip abgewickelt und dafür entsprechende Sperrmechanismen eingesetzt und entsprechende Log-Dateien geführt. Wie bei relationalen DBMS gibt es auch hier Backup- und Recovery-Verfahren. Neo4j erfüllt die in Kapitel 2.3 aufgeführten Anforderungen, die an ein DBMS gestellt werden. Wenn es um die Anforderungen an die Konsistenz und die Integrität der Daten geht, so gibt es auch im Neo4j-Graph-DBMS strukturelle Integritätsbedingungen, die geprüft werden. So sorgt das DBMS dafür, dass die Knoten und Kanten des Graphen eindeutig identifizierbar sind. Eigenschaften der Knoten und der Kanten haben spezifische Datentypen, deren Wertebereich überprüft wird. Allerdings hängt die Konsistenz und Integrität der Daten auch von dem Entwurfsprozess ab, da bei einem Graph-DBMS nach der Erstellung eines ER-Modells keine Normalisierung, wie beim Relationalen Modell üblich, durchgeführt wird. Wenn beim Entwurf also nicht die richtigen Fragen gestellt werden und nicht ganz klar ist, was genau ausgewertet werden soll, kann bei der Abbildung in den Graphen ein wichtiges Objekt oder eine wichtige Beziehung vergessen werden. Das kann zur Folge haben, dass das zugrunde liegende Modell noch einmal angepasst werden muss, nachdem bereits Daten erfasst worden sind.

Trotz der nativen Abbildung des Graphen ist das Datenmodell universell genug, um eine gewisse Datenunabhängigkeit zu ermöglichen. Dies wird auch durch das flexible Anhängen von Eigenschaften und Labeln, die entsprechend indexiert werden können, unterstützt. Allerdings gibt es auch Einschränkungen, die eine Festlegung beim Entwurf erzwingen: Kanten werden grundsätzlich als gerichtete Kanten in Neo4j angelegt. Aus Performance-Gründen sollen diese aber nur in einer Richtung angelegt werden, selbst, wenn ein Navigieren in der angelegten Richtung vom Modell her nicht möglich ist. Dabei handelt es sich um eine Empfehlung, die unter anderem von Graphgrid, Inc. [27] und in einem technischen Artikel von Bachman [3], gegeben wird. Grundsätzlich kann man in Neo4j im Graphen in allen Richtungen navigieren, egal in welche Richtung die ange-

legte Kante zeigt. Das bedeutet aber auch, dass der ausgewählte Algorithmus, die Navigation in der für das Problem korrekten Richtung unterstützen muss, was derzeit nicht bei allen Algorithmen im Neo4j-DBMS berücksichtigt wird. In vielen Fällen kann es jedoch vermieden werden Kanten in beiden Richtungen anzulegen. Dies sollte beim Datenbankentwurfsprozess entsprechend berücksichtigt werden.

Neo4j verwaltet einen Page Cache, dessen Größe in der Konfiguration festgelegt wird, siehe Kapitel 7.2. Wenn das System neu gestartet wird, so ist dieser zunächst leer. Dadurch kann es passieren, dass die ersten Anfragen etwas länger dauern. Der Page Cache wird durch weitere eingehende Anfragen nach und nach mit den Daten des Graphen gefüllt. Daher wird in der Knowlede Base (KB) von Neo4j [38] empfohlen, nach einem Neustart des Datenbanksystems zunächst ein Aufwärmen des Caches vorzunehmen. In dem KB-Artikel werden hierzu Beispiele gegeben, wie man dies mit Cypher oder auch mit Nutzung der Java Schnittstelle durchführen kann. In der aktuellen Version von Neo4j, die wir für die Experimente im Einsatz haben, gibt es eine APOC Procedure (APOC steht für Awesome Procedures on Cypher), die mit dem Kommando `CALL apoc.warmup.run()` ausgeführt werden kann, um den Graphen in den Cache zu laden. Der Graph wird dann ganz oder teilweise in den Cache geladen, je nachdem , ob er von der Größe in den Cache passt.

Erweiterungsmöglickeiten des Neo4j-Graph-DBMS:

Neo4j bietet standardmäßig nach der Installation von Neo4j Desktop die Installation von zwei Plugins namens `APOC` und `effiziente Graphalgorithmen` auf Tastendruck an. `APOC` sind Stored Procedures, die von Neo4j entwickelt wurden, um gemeinsam genutzte Standardfunktionen bereitzustellen, damit diese nicht individuell von den Nutzern des Neo4j-Graph-DBMS programmiert werden müssen. Das Neo4j Plugin für effiziente Graphalgorithmen ist eine Bibliothek, die effiziente, parallele Versionen von allgemeinen Graph-Algorithmen enthält und in Form von Cypher Procedures bereitstellt. Diese enthält auch den Dijkstra-Algorithmus, dessen Laufzeit wir im Rahmen der Experimente, siehe Kapitel 7, messen und mit der Laufzeit der Stand-Alone-Lösung vergleichen.
Neo4j kann auch durch eigene Plugins erweitert werden. Im Developer-Manuell, das Bestandteil der Neo4j Dokumentation [23] ist, werden die einzelnen Möglichkeiten und das Vorgehen detailliert beschrieben. Dabei ist einem darin beschriebenen Prozess zu folgen, um in der programmierten Erweiterung auf die Datenbankschnittstelle zuzugreifen und um die so erstellten Procedures oder Functions in der Datenbank bekannt zumachen. Die Erweiterungen werden in Form eines Java-Archivs (Datei vom Typ JAR) in das Verzeichnis namens plugins gestellt, das sich im Verzeichnis NEO4J_HOME befindet, wo diese nach Neustart von Neo4j aktiv ist. Im Kapitel 5.4 wird dieser Prozess anhand unserer eigenen Erweiterung in Neo4j detailliert beschrieben.

3.3 Die Anfragesprache Cypher

Wie bereits bereits bei der Beschreibung der Graphenmodelle zu Beginn von Kapitel 3 erwähnt, stellt Neo4j mit Cypher eine deklarative Anfragesprache zur Verfügung, mit der Graphen erstellt, manipuliert und ausgewertet werden können. Wir werden im Folgenden die wichtigsten Sprachkonstrukte kurz beschreiben. Dabei lehnen wir uns im Folgenden an die Beschreibungen in der Neo4j-Dokumentation [23] an. Hier ist unter anderem im Developer Manual eine sehr ausführliche Beschreibung von Cypher enthalten.

Da es bei der Auswertung eines Graphen oft darum geht, einen Subgraphen oder ein Muster (Pattern) zu finden, wird mit der Anfragesprache Cypher eine sogenannte ASCII Art Graph-Pattern-Beschreibung ermöglicht. Mit dem Begriff ‚ASCII Art' ist gemeint, dass ASCII-Zeichen dazu verwendet werden, um ein Bild oder Muster darzustellen. In Cypher werden daher Knoten mit runden Klammern und Kanten mit Linien oder Pfeilen dargestellt. Werden bestimmte Kanten beschrieben, so wird diese Beschreibung in eckigen Klammern angegeben.

Einfache Muster, die aus einer Verbindung von einzelnen Knoten besteht, sind zum Beispiel

```
(:Person)-[:LEBT_IN]->(:Stadt)
```

und

```
(:Stadt)-[:IST_TEIL_VON]->(:Land).
```

Komplexe Beziehungen entstehen, wenn mehrere dieser Muster miteinander kombiniert werden. Wenn wir zum Beispiel wissen wollen, welche Person denn in einem bestimmten Land lebt, dann müssen wir die Musterbeschreibungen entsprechend kombinieren. In unserem Beispiel ergibt sich dadurch folgende Muster-Beschreibung:

```
(:Person)-[:LEBT_IN]->(:Stadt)-[:IST_TEIL_VON]->(:Land)
```

Wie in SQL werden auch hier Schlüsselworte für eine Anfrage verwendet. So beginnt eine Cypher-Klausel in der Regel mit dem Schlüsselwort `MATCH`, das in SQL dem Schlüsselwort `SELECT` entspricht. Mit `WHERE` werden hier ebenso wie in SQL Bedingungen angegeben, nach denen gefiltert werden soll. Mit dem Schlüsselwort `RETURN` wird beschrieben, was als Rückgabe erwartet wird.

Bevor wir die weiteren Cypher-Konstrukte beschreiben, geben wir eine Übersicht über die Syntax der in Cypher verwendeten Muster (Pattern) von Knoten und Kanten. Je nach verwendetem Muster können dabei ein oder mehrere Knoten oder Kanten (Beziehungen) beschrieben werden, nach denen in dem Graphen gesucht werden soll.
Tabelle 2 gibt eine kurze Übersicht über die Syntax der Knoten-Muster. Die Syntax von Kanten-Mustern wird in Tabelle 3 dargestellt.

Tabelle 2: Cypher-Syntax, Übersicht Knoten-Muster.

Knoten-Muster	Bedeutung
()	Ein anonymer Knoten, der für alle möglichen Knoten steht.
(mat)	Irgendein Knoten, der mit einer Variablen namens mat für die Dauer der Anfrage versehen ist.
(:Person)	Ein Knoten vom Typ Person.
(mat:Person)	Ein Knoten vom Typ Person, der mit Variable mat versehen ist.
(mat:Person {name: "Matt"})	Knoten vom Typ Person mit der Eigenschaft name, die den Wert "Matt" hat und mit Variable mat versehen ist.

Tabelle 3: Cypher-Syntax, Übersicht Kanten-Muster.

Kanten-Muster	Bedeutung
–>	Eine gerichtete Kante, die für alle von da ausgehende Kanten steht.
-[lebt]->	Irgeneine ausgehende Kante, die mit Variable namens lebt versehen ist.
-[:LEBT_IN]->	Eine Kante vom Typ LEBT_IN.
-[lebt:LEBT_IN]->	Eine Kante vom Typ LEBT_IN, die mit Variable namens lebt versehen ist.
-[lebt:LEBT_IN {arbeitet: ["Firmaxy"]}]->	Wie in Zeile vorher, nur mit zusätzlicher Eigenschaft.

Die in der Tabelle aufgeführten Muster können wir in einer Anfrage entsprechend kombinieren. Für die Beschreibung der Eigenschaften können auch reguläre Ausdrücke (regular expressions, kurz regex) eingesetzt werden. Diese werde im Developer Manual genauer beschrieben. Für die Vereinfachung von Anfragen können solche Muster auch in Variablen gespeichert werden, wie im Folgenden Beispiel illustriert wird:

```
lebt = (:Person)-[:LEBT_IN]->(:Stadt)
```

Ein Cypher-Kommando kann aus mehreren Anfrage-Klauseln bestehen, die miteinander kombiniert werden. Die Tabellen 4 und 5 geben eine Übersicht über die in den Anfrage-Klauseln verwendeten Schlüsselwörter und deren Bedeutung. Wir unterscheiden hier zwischen lesenden und schreibenden Klauseln. Darüber

hinaus gibt es noch Cypher-Klauseln für Schema-Operationen, zum Beispiel um einen Index auf Labels oder Eigenschaften zum schnellen Auffinden von Startknoten anzulegen.

Tabelle 4: Lesende Cypher-Klauseln nach Kategorie und Bedeutung.

Schlüsselwort	Kategorie	Bedeutung
MATCH	Suche	Suche mit Musterabgleich in der DB.
OPTIONAL MATCH	Suche	Optionaler Musterabgleich, zusätzlich zu MATCH, nutzt Null-Werte für fehlende Muster.
RETURN ... [AS]	Rückgabe	Rückgabewerte, zum Beispiel Knoten oder eine Ergebnistabelle.
UNWIND ... [AS]	Rückgabe	Rückgabe als Sequenz von Zeilen.
UNION	Rückgabe	Verbund der Ergebnisse der Anfrage, Duplikate werden entfernt.
UNION ALL	Rückgabe	Wie UNION, nur mit Duplikaten.
WITH ... [AS]	Piping	Weiterleiten der Ergebnisse zur weiteren Verarbeitung in Klauseln.
WHERE	Filtern	Filtern der Daten von MATCH oder WITH mit Bedingungen.
ORDER BY	Sortieren	Aufsteigendes oder absteigendes Sortieren der Daten.
SKIP	Filtern	Bestimmte Anzahl an Resultaten am Anfang überspringen.
LIMIT	Filtern	Resultat auf eine bestimmte Anzahl limitieren.

Eine besondere Cypher-Anfrage-Klausel ist `CALL ... [YIELD]` da mit dieser in Neo4j installierte Procedures oder Functions aufgerufen werden können. Diese können sowohl lesende als auch manipulierende Operationen beinhalten. Auch der als eigene Neo4j-Procedure implementierte Bellman–Ford-Algorithmus kann so mit Cypher aufgerufen werden. Mit dem Schlüsselwort `YIELD` werden die Rückgabe-Parameter aus einer Procedure ausgewählt, die dann mit `RETURN` zurückgegeben werden.

Als Operatoren können in Cypher-Klauseln sowohl Vergleichsoperatoren, als auch mathematische Operatoren verwendet werden. Zudem können reguläre Ausdrücke für eine allgemeinere Mustererkennung eingesetzt werden.

Tabelle 5: Schreibende Cypher-Klauseln nach Kategorie und Bedeutung.

Schlüsselwort	Bedeutung
CREATE	Erstellen eines Knotens oder einer Kante.
DELETE	Knoten, Kanten oder Pfade entfernen.
DETACH DELETE	Löschen von Knoten, dabei werden alle Beziehungen mit entfernt.
REMOVE	Entfernen einer Eigenschaft eines Labels oder eines Typs.
SET	Knoten-Label oder Eigenschaften von Knoten oder Kanten einen Wert zuweisen.
FOREACH	Update von Daten in einer Liste oder in einem Aggregat.
MERGE	Stellt sicher, dass ein spezifiziertes Muster im Graphen vorhanden ist. Ist es nicht vorhanden, so wird es angelegt.
MERGE ON CREATE	Wie MERGE, wobei angegeben wird, was getan werden soll, wenn das Muster angelegt werden muss.
MERGE ON MATCH	Wie Merge, wobei angegeben wird, was im Fall der Existenz passieren soll.

Cypher stellt zudem eine Vielfalt an Funktionen zur Verfügung:

- Prädikat-Funktionen, wie all(), any() oder exists(),
- mathematische Funktionen, wie sin(), abs() oder exp(),
- skalare Funktionen mit einem Rückgabewert, wie size() oder id(),
- Stringfunktionen, wie rTrim(), substring() oder toUpper(),
- temporale Funktionen, wie date() oder time(),
- spatiale Funktionen, wie distance() oder point() und
- die Möglichkeit benutzereigene Aggregats-Funktionen auszuführen, die zum Beispiel nach `RETURN` aufgerufen werden können.

Nachdem wir die Sprachkonstrukte vorgestellt haben, werden wir die Vorstellung von Cypher mit einem kleinen Beispiel abrunden. Hierzu betrachten wir folgenden Beispielgraphen, der Teil eines sozialen Netzwerkes sein könnte, siehe Abbildung 7.

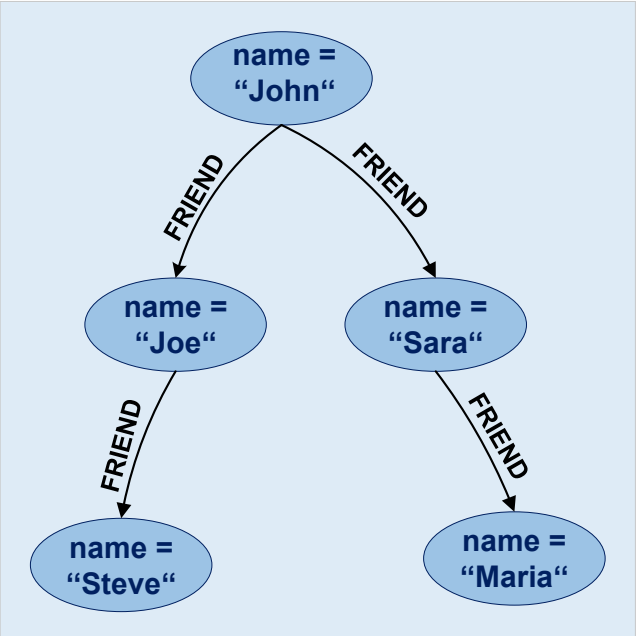


Abbildung 7: Beispielgraph eines Freundschaftsnetzwerks [23].

Wenn wir nun wissen wollen, mit wem John indirekt befreundet ist, durch direkte Freunde seiner Freunde, so können wir dies mit folgender Cypher-Query anfragen:

```
MATCH (john {name: 'John'})-[:FRIEND]->()-[:FRIEND]->(fof)
RETURN john.name, fof.name
```

Das Ergebnis dieser Anfrage ist eine Tabelle mit zwei Zeilen, siehe Tabelle 6.

Tabelle 6: Ergebnis der Anfrage zu Johns indirekten Freunden.

john.name	fof.name
John	Maria
John	Steve

Zum Abschluss sehen wir uns ein Beispiel mit einem regulären Ausdruck an. Wir wollen aus einer Liste mit Namen, diejenigen Benutzer wählen, die eine direkte Freundschaftsbeziehung zu einem Benutzer mit einem Namen haben, der mit dem Buchstaben S beginnt. Als Ergebnis soll der Name des Benutzers und der Name seines Freundes ausgegeben werden. Dies wird mit Cypher wie folgt angefragt:

```
MATCH (user)-[:FRIEND]->(friends)
WHERE user.name IN ['Joe', 'John', 'Sara', 'Maria', 'Steve']
 AND friends.name =~ 'S.*'
RETURN user.name, friends.name
```

Als Ergbenis erhalten wir wieder eine Tabelle mit zwei Zeilen, siehe Tabelle 7.

Tabelle 7: Ergebnis der Anfrage zu Benutzern mit direkten Freunden, deren Name mit S beginnt.

user.name	friends.name
Joe	Steve
John	Sara

Nachdem wir uns mit der Speicherung und der möglichen Auswertung von Daten, die in Form eines Graphen vorliegen, vertraut gemacht haben, werden wir den Entwurfsprozess einer auf einem Graphenmodell basierenden Datenbank nun näher betrachten.

3.4 Der Datenbankentwurf

Im Rahmen des Datenbankentwurfs wird die reale Welt bzw. der interessierende Anwendungsbereich, in ein Modell abgebildet. Zu diesem Zeitpunkt ist das spätere Datenmodell der Datenbank noch nicht festgelegt. Daher kann das Entity-Relationship-Modell als Modellierungswerkzeug auch für den Datenbankentwurf für eine Graphdatenbank angewendet werden, um die Entitäten, die Beziehungen und die Eigenschaften entsprechend abzubilden. Das dadurch entstandene graphisch dargestellte ER-Modell kann ohne aufwändige Normierung, wie sie bei relationalen Datenbanken erforderlich wäre, in das Graphenmodell überführt werden. Die wesentlichen Unterschiede im Datenbankentwurf ausgehend vom ER-Modell wird in Abbildung 8 anhand eines einfachen Beispiels illustriert. Dabei gehen wir davon aus, dass beim Relationenmodell eine Normalisierung vorgenommen wird.

Auch, wenn die Überführung in das Graphenmodell sehr einfach erscheint, sind auch hier einige Regeln zu berücksichtigen, damit die Auswertungen gemäß der ursprünglichen Anforderungen auch möglich sind. Fehler passieren zum Beispiel, wenn relevante Objekte oder Beziehungen bei der Überführung in das Graphenmodell nicht abgebildet werden, da diese bei der Modellierung vergessen wurden. Robinson empfiehlt daher das so abgebildete Graphenmodell mit einigen Abfragen zu testen, um so festzustellen, ob die Ergebnisse der Auswertungen passend sind [29]. Um Fehlern bei der Modellierung bzw. der Überführung des ER-Modells in das Graphenmodell zu verhindern, kann die Überführung mit den von Andreas Meier et. al. beschriebenen Regeln [22] durchgeführt werden.

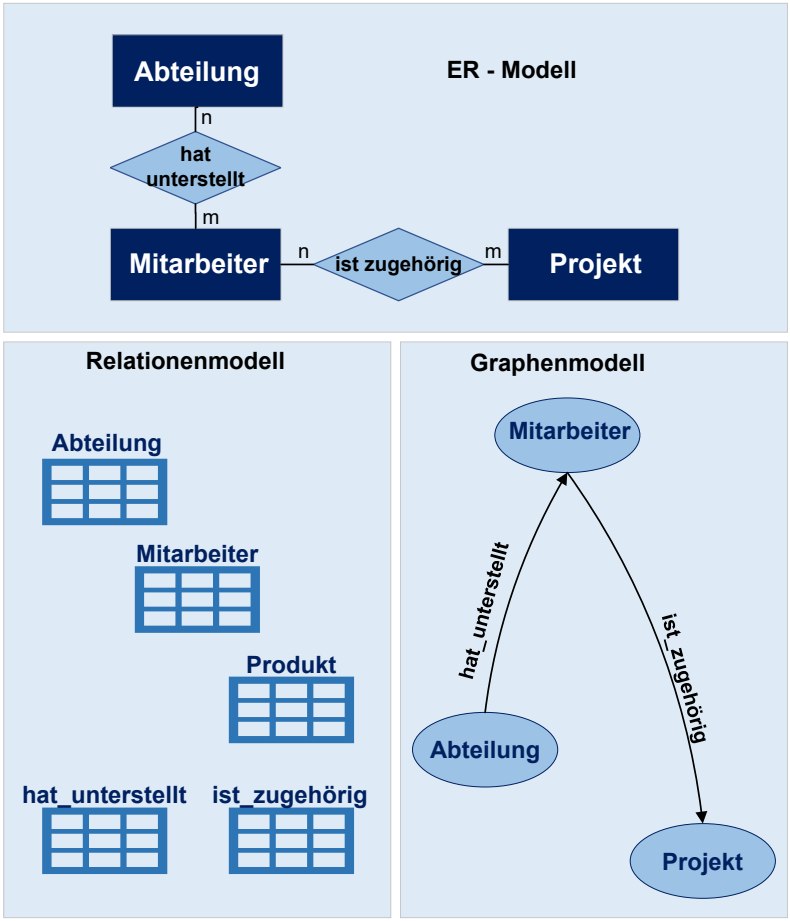


Abbildung 8: Datenmodellierung im Vergleich: RDB und Graph-DB [22].

4 Kürzeste Pfade

4.1 Das kürzeste-Pfad-Problem

Das Problem, in einem Netzwerk den kürzesten, schnellsten oder billigsten Pfad zu finden, ist allgegenwärtig. Ein Netzwerk und auch viele andere Probleme lassen sich leicht als Graph mit Knoten und Kanten modellieren und kodieren. Ein Pfad in einem Graphen ist dann ein Weg von einem gegebenen Startknoten zu einem gegebenen Zielknoten, bei dem kein Knoten doppelt durchlaufen wird. Wenn die Kanten des Graphen gewichtet sind, zum Beispiel mit einer Entfernungsangabe, dann wollen wir nicht nur wissen, ob es einen Pfad zwischen zwei Knoten gibt, sondern welches der kürzeste Pfad ist. In diesem Fall wählen wir also den Pfad, bei dem die Summe der Kantengewichte am kleinsten ist. Je nach Problemstellung kann das Gewicht der Kanten eine andere Bedeutung haben, zum Beispiel können die Kanten mit Kostenangaben belegt werden oder mit Angaben zur Bandbreite. Dabei kann es auch vorkommen, dass Kanten mit negativen Gewichten belegt werden. Wie geht man nun am besten vor, um den kürzesten Pfad in dem gegebenen Graphen zu finden?

Wir betrachten im Folgenden eine eingeschränkte Form des kürzesten-Pfad-Problems, bei der die Eingabe ein Graph $G = (V, E)$ mit einer Gewichtsfunktion $w\colon E \to \mathbb{R}$ ist und der so definierte Graph keinen negativen Zyklus, also keinen Kreis, bei dem die Summe der Kanten negativ ist, enthält. Mit dieser Einschränkung ist die Suche nach dem kürzesten Pfad bzw. nach den kürzesten Pfaden, effizient lösbar. Wenn wir diese Einschränkung nicht vornehmen, so wird das Problem NP-vollständig [14] [16].

Für den Vergleich der Performanz zwischen der Neo4j-Graph-DB und der implementierten Stand-Alone-Lösung beschränken wir den Eingabegraphen auf Graphen, die höchsten eine Kante in jeder Richtung zwischen 2 Knoten haben. Weiterhin werden wir uns auf das kürzeste-Pfade-Problem mit einem Startknoten (Englisch: single-source shortest-path problem, kurz SSSP) fokussieren und zwar mit den folgenden Varianten:

(i) Problem der Berechnung eines kürzesten Pfades von einem gegebenen Startknoten $s \in V$ zu einem gegebenen Zielknoten $t \in V$ in G.

(ii) Problem der Berechnung aller kürzesten Pfade von einem Startknoten $s \in V$ zu jedem anderen Knoten in dem gegebenen Graphen G.

Betrachten wir nun die Algorithmen, die zur Lösung des Problems implementiert wurden und deren Laufzeiten später in den Experimenten untersucht und verglichen wird.

4.2 Algorithmen zur Berechnung des kürzesten Pfades

4.2.1 Bidirektionale Breitensuche

Die bidirektionale Breitensuche berechnet den kürzesten Pfad von einem Startknoten zu einem Zielknoten in einem ungewichteten Graphen. Die Länge des kürzesten Pfades ergibt sich dabei aus der Anzahl der Kanten, die vom Startknoten bis zum Zielknoten mindestens durchlaufen werden müssen. Der Algorithmus baut dabei auf einem wichtigen Basis-Algorithmus zum Durchlaufen eines Graphen auf, nämlich der Breitensuche. Daher werden wir zunächst erklären, wie die Breitensuche in einem Graphen funktioniert und welches Ergebnis wir damit erhalten.

Die Breitensuche (BFS, von engl. Breadth-First-Search) ist ein Algorithmus zum Durchsuchen eines Graphen. Er dient als Basis für viele Algorithmen, wie zum Beispiel den im nächsten Abschnitt beschrieben Dijkstra-Algorithmus. Für einen Graphen $G = (V, E)$ und einen vorgegebenen Startknoten s erforscht die Breitensuche systematisch alle Kanten von G, um alle Knoten, die von s aus erreichbar sind, zu finden. Bei der Breitensuche erfolgt, wie der Name bereits verrät, die Suche zunächst in der Breite. Es werden zuerst alle Knoten besucht, die mit dem aktuellen Knoten direkt verbunden sind und erst danach wird die Suche mit den so neu gefundenen Knoten fortgesetzt. Es werden also alle Knoten besucht, die die Distanz k von s haben, bevor irgendein Knoten mit Distanz $k+1$ besucht wird. Der Algorithmus 1 berechnet damit die Distanz (die kleinste Anzahl von Kanten) von s zu jedem erreichbaren Knoten. In dem dadurch erzeugten Breitensuchbaum ist der einfache Pfad von s nach v ein kürzester Pfad von s nach v in G [6].

Bei dem hier illustrierten Algorithmus Breitensuche wird der Zustand der Knoten während der Bearbeitung mit den Farben Weiß, Grau oder Schwarz markiert. Zu Beginn bekommt jeder Knoten die Farbe Weiß zugewiesen, da zuvor noch kein Knoten des Graphen besucht wurde. Ein neu besuchter Knoten wird mit Grau markiert und ein fertig bearbeiteter Knoten bekommt die Farbe Schwarz zugeordnet. Wenn die Breitensuche beendet ist, sind alle Knoten, die vom Startknoten s aus erreichbar sind, schwarz markiert worden, auch s selbst. Zu jedem Knoten ist die Distanz zu s gespeichert, wobei $s.d = \infty$ gesetzt ist, wenn der Knoten von s nicht erreichbar ist. Außerdem wird zu jedem Knoten der Vorgänger gespeichert, über den dieser von s aus erreicht werden kann. Durch eine Rückverfolgung über den jeweiligen Vorgänger-Knoten (backtracking) ist es möglich, den Pfad von s zu dem Knoten auszugeben. Da das Durchsuchen der Adjazenzlisten im schlechtesten Fall eine Laufzeit von $O(|E|)$ beträgt und die Initialisierung mit $O(|V|)$ hinzukommt, ergibt sich eine worst case Gesamtlaufzeit für die BFS von $O(|V| + |E|)$ [6].

Die bidirektionale Breitensuche baut auf der eben beschriebenen Breitensuche auf. Dabei wird der Graph nach dem kürzesten Pfad zwischen dem Startknoten

Algorithm 1 Breitensuche.

```
1: procedure BFS(G, s)                    ▷ Eingabe: Graph G, Startknoten s
2:     for jeden Knoten u ∈ G.V − {s} do
3:         u.farbe = WEISS
4:         u.d = ∞
5:         u.π = NIL
6:     end for
7:     s.farbe = GRAU
8:     s.d = 0
9:     s.π = NIL
10:    Q = ∅
11:    Q.enqueue(s)
12:    while not Q.isEmpty() do
13:        u = Q.dequeue()
14:        for jeden Knoten v ∈ G.Adj[u] do
15:            if v.farbe == WEISS then
16:                v.farbe = GRAU
17:                v.d = u.d + 1
18:                u.π = u
19:                Q.enqueue(v)
20:            end if
21:        end for
22:        u.farbe = SCHWARZ
23:    end while
24: end procedure
```

und dem Zielknoten von zwei Richtungen aus mit der Breitensuche durchsucht. Eine Breitensuche startet mit dem Startknoten und eine zweite Breitensuche startet mit dem Zielknoten. Dadurch, dass Start- und Zielknoten nicht gleich sind, breiten sich die dabei erzeugten Suchbäume in verschiedenen Richtungen aus. Beim Besuch eines Knotens wird überprüft, ob sich die beiden Suchen getroffen haben. Das ist dann der Fall, wenn es einen Knoten gibt, den beide bereits besucht haben. Der kürzeste Pfad wird dann durch das Verbinden der gefunden Pfade der beiden Suchen gebildet [17].

Durch die bidirektionale Suche kann der Suchraum verkleinert und die Suchtiefe halbiert werden. Wie wir in Abbildung 9 sehen können, trifft man sich bei dieser Herangehensweise auf halbem Weg, vorausgesetzt, dass ein Pfad zwischen Start- und Zielknoten existiert. Außerdem lässt sich gut erkennen, dass bei der Suche nicht alle vorhandenen Knoten besucht werden mussten.
Wenn es allerdings gar kein Pfad zwischen dem Startknoten und dem Zielknoten gibt (Graph nicht zusammenhängend), dann wird eine komplette Breitensuche von beiden Seiten durchgeführt, sodass die Laufzeit im schlechtesten Fall wieder der der Breitensuche entspricht. Durch die Speicherung, welcher Knoten bereits

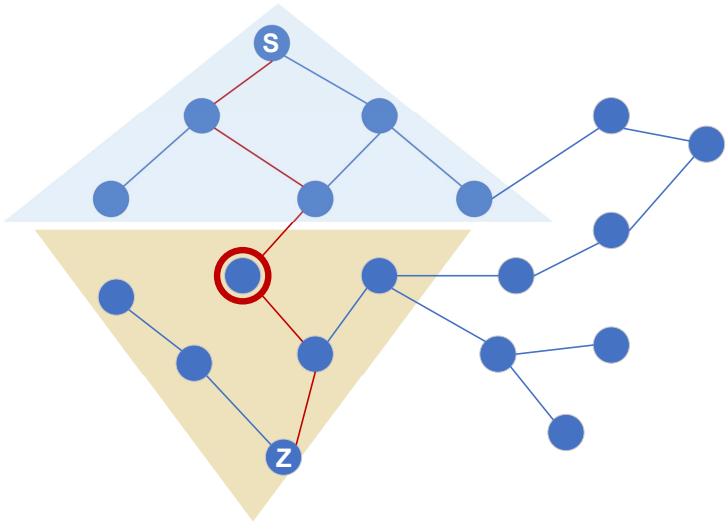


Abbildung 9: Beispiel einer bidirektionale Breitensuche, Aufeinandertreffen im rot markierten Knoten, gefundener Pfad ist rot markiert.

besucht wurde, kann dies während der Suche mit einer zusätzlichen Überprüfung in konstanter Zeit ($O(1)$) innerhalb der bereits vorhandenen Schleifen abgebildet werden. Die Laufzeit wird dadurch also nicht erhöht.

4.2.2 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus löst das kürzeste-Pfade-Problem mit einem Startknoten auf einem gewichteten, gerichteten Graphen $G = (V, E)$ für den Fall, dass alle Kantengewichte größer oder gleich Null sind. Er wurde nach seinem Erfinder Edsger W. Dijstra benannt, der ihn 1959 in einem Artikel veröffentlichte [10]. Der Algorithmus gehört zu den sogenannten Greedy-Algorithmen, da er stets diejenige Entscheidung trifft, die im Moment am besten erscheint. Das heißt, er trifft eine lokal optimale Entscheidung in der Hoffnung, dass diese Entscheidung zu einer global optimalen Lösung führen wird [6]. Die Grundidee des Algorithmus ist es, immer derjenigen Kante zu folgen, die den kürzesten Streckenabschnitt vom Startknoten aus verspricht.

Die in diesem Abschnitt folgenden Algorithmen und Beschreibungen sind angelehnt an Cormen et. al. [6].

Wir setzen also für den hier betrachteten Algorithmus 4 $w(u, v) \geq 0$ für alle Kanten voraus. Während der Ausführung des Dijkstra-Algorithmus wird eine Menge S von Knoten, für die die korrekten Gewichte der kürzesten Pfade vom Startknoten aus bereits bestimmt sind, verwaltet. In jedem Schritt wird derjenige Knoten $u \in V - S$ ausgewählt, der die kleinste Schätzung seines kürzesten Pfades hat und zu S hinzugefügt. Dabei werden alle Kanten, die aus u austreten

relaxiert. Zur Verwaltung der d-Werte (aktuell kürzeste Distanz oder kleinstes Gewicht) wird eine Min-Prioritätswarteschlange Q verwendet, wobei die d-Werte als Schlüssel dienen.

Wir betrachten zunächst zwei allgemeine Prozeduren, die in verschiedenen Algorithmen bei der Suche nach dem kürzesten Pfad als Unterprogramme eingesetzt werden, so auch bei dem hier betrachteten Dijkstra-Algorithmus.

Für jeden Knoten $v \in V$ verwalten wir ein Attribut $v.d$, das eine obere Schranke für das Gewicht des kürzesten Pfades vom Startknoten s nach v speichert. Wir bezeichnen $v.d$ als Schätzung des kürzesten Pfades. Die Initialisierung wird mit der Ausführung der ProZedur InitSingleSource(G, s) 2 durchgeführt und es gilt $v.\pi =$ NIL (kein Vorgänger) für jeden Knoten $v \in V$ sowie $s.d = 0$ und $v.d = \infty$ für $v \in V - \{s\}$. Eine weitere Prozedur namens Relax behandelt die Relaxation, die oft Bestandteil von Algorithmen ist, die den kürzesten Pfad berechnen.

Algorithm 2 Initialize-Single-Source.

```
1: procedure InitSingleSource(G, w, s)
2:     for jeden Knoten v ∈ G.V do
3:         v.d = ∞
4:         v.π = NIL
5:     end for
6:     s.d = 0
7: end procedure
```

Algorithm 3 Kanten-Relaxation.

```
1: procedure Relax(u, v, w)          ▷ Eingabe: Knoten u, v, Gewicht w
2:     if v.d > u.d + w(u, v) then
3:         v.d = u.d + w(u, v)
4:         v.π = u
5:     end if
6: end procedure
```

Der Prozess des Relaxierens einer Kante (u, v) besteht darin, zu testen, ob wir den bisher gefundenen kürzesten Pfad nach v verbessern können, indem wir über u laufen, und, wenn dies der Fall ist, die Attribute $v.d$ und $v.\pi$ entsprechend zu aktualisieren. Ein Relaxationsschritt kann den Wert der Schätzung des kürzesten Pfades verringern und das Vorgängerattribut $v.\pi$ des Knotens v aktualisieren 3.

Die Laufzeit von Dijkstras Algorithmus hängt davon ab, wie die Min-Prioritätswarteschlange implementiert wird. Bevor wir näher darauf eingehen, wollen wir zunächst beschreiben, was eine Min-Prioritätswarteschlange ist und welche Ope-

Algorithm 4 Dijkstra-Algorithmus.

```
1: procedure DIJKSTRA(G, w, s)        ▷ Eingabe: Graph G, Gewicht w,
   Startknoten s
2:     InitSingleSource(G, s)
3:     S = ∅
4:     Q = G.V
5:     while not Q.isEmpty() do
6:         u = Q.extractMin()
7:         S = S ∪ {u}
8:         for jeden Knoten v ∈ G.Adj[u] do
9:             Relax(u, v, w)
10:        end for
11:    end while
12: end procedure
```

rationen diese bereitstellt.

Eine Prioritätswarteschlange ist eine Datenstruktur für die Verwaltung einer Menge Q von Einträgen mit Schlüsseln. Eine Min-Prioritätswarteschlange ist so organisiert, dass dem Element mit dem aktuell kleinsten Schlüssel die höchste Priorität gegeben wird.

Die Min-Prioritätswarteschlange stellt folgende Operationen bereit:

- Q.build($\{e_1, \ldots, e_n\}$): $Q := \{e_1, \ldots, e_n\}$.
- Q.insert(e): $Q := Q \cup \{e\}$.
- Q.minimum(): return $min(Q)$.
- Q.extractMin(): $e := min(Q)$; $Q := Q - \{e\}$; **return** e.
- Q.decreaseKey(e, k): if (key(e) $\leq k$) then key(e) $:= k$.

Wenn wir nun die Schritte des Algorithmus 4 betrachten, so sind die am häufigsten genutzten Operationen deleteMin() und decreaseKey(). Weiterhin wird die Funktion build() oder insert() bei der Initialisierung des Algorithmus genutzt, um einmalig alle Knoten des Graphen in die Warteschlange zu speichern. Die Laufzeiten der aufgeführten Operationen hängen wiederum von der Implementierung der Min-Prioritätswarteschlange ab. Es ist also sinnvoll, eine Min-Prioritätswarteschlange zu wählen, die insbesondere die hauptsächlich von dem Dijkstra-Algorithmus genutzten Warteschlangen-Operationen effizient ausführt.

Die Laufzeit des Dijkstra-Algorithmus lässt sich somit am besten abschätzen, indem wir diese in Abhängigkeit der Warteschlangen-Operationen betrachten.

Cormen et. al. führen die Analyse wie folgt durch [6]: Der Algorithmus ruft sowohl die Prozedur insert() als auch die Prozedur etractMin() für jeden Knoten exakt einmal auf. Da jeder Knoten $u \in V$ einmal zur Menge S hinzugefügt wird, wird jede Kante aus der Adjazenzliste $Adj[u]$ während der Ausführung des Algorithmus genau einmal in der **for**-Schleife betrachtet. Da die Gesamtanzahl der Kanten über alle Adjazenzlisten gleich $|E|$ ist, iteriert diese **for**-Schleife insgesamt $|E|$-mal und somit ruft der Algorithmus decreaseKey() höchstens $|E|$-mal auf. Es ergibt sich somit im worst-case folgende Gesamtlaufzeit für den Dijkstra-Algorithmus:

$$T_{Dijkstra} = O(|E| \cdot T_{dk} + |V| \cdot T_{em}) \qquad (1)$$

wobei T_{dk} und T_{em} für die Laufzeit der decreaseKey()- und der extractMin()-Operationen der implementierten Prioritätswarteschlange stehen.

Wählt man zum Beispiel ein Array, in dem man die Elemente der Warteschlange in ungeordneter Form speichert, wobei der Index des Arrays der Nummerierung der Knoten des Graphen $G = (V, E)$ entspricht, so können insert() und decreasKey() in $O(1)$ Laufzeit durchgeführt werden. Im Gegensatz dazu hat extractMin() im worst-case eine Laufzeit von $O(|V|)$, da wir im schlechtesten Fall das gesamte Array durchsuchen müssen. Dadurch ergibt sich in diesem Fall eine Gesamtlaufzeit von $O(|E| + |V|^2) = O(|V|^2)$.

In der für den Vergleich der Performanz implementierten Stand-Alone-Lösung nutzen wir einen binären Heap als Min-Prioritätswarteschlange. Bei der implementierten Lösung merken wir uns zusätzlich eine Referenz auf den Knoten des Graphen, sodass wir weiterhin einen direkten Zugriff auf diesen haben, wenn der Wert des Schlüssels mit der decreaseKey()-Operation aktualisiert werden muss.

Im Folgenden werden wir daher die Organisation und Arbeitsweise des binären Heaps als Beispiel einer Min-Prioritätswarteschlange ausführlicher betrachten und die sich daraus ergebende worst-case Laufzeit des Dijkstra-Algorithmus angeben.

Binärer Heap als Min-Prioritätswarteschlange

Bei der nun folgenden Vorstellung und Beschreibung des binären Heaps orientieren wir uns an den Ausführungen von Dietzfelbinger, Mehlhorn und Sanders [9]. Die Elemente der Warteschlange sind bei dieser Lösung als linksvollständiger Binärbaum mit n Knoten organisiert, der heapgeordnet ist, d.h. die Knoten des Baumes sind so platziert, dass die Heapbedingung erfüllt ist. Die Heapbedingung verlangt, dass die Schlüssel entlang jedes Weges von einem Blatt bis zur Wurzel des Baumes schwach monoton fallend sein müssen. Dadurch hat insbesondere der Eintrag in der Wurzel des Baumes den minimalen Schlüssel. Für die Realisierung des binären Heaps wird hier ein Array eingesetzt, in dem man einen binären Baum sehr einfach verwalten kann.

Um die n Einträge der Prioritätswarteschlange zu speichern, benutzen wir die ersten n Positionen eines Arrays $h[1..w]$. Dabei ist das Array heapgeordnet, d.h., für j mit $2 \leq j \leq n$ gilt: $h[\lfloor j/2 \rfloor] \leq h[j]$. Dass Zelle $h[1]$ einen minimalen Eintrag enthält, entspricht der Situation bei einem geordneten Array, weswegen die Operation minimum() in $O(1)$ Zeit durchgeführt werden kann. Die weiteren Operationen werden im Folgenden in Pseudocode beschrieben:

Algorithm 5 Binärer Heap - Operation insert().

```
1: procedure INSERT(e)                          ▷ Eingabe: Element e
2:     n := n + 1                                ▷ Es gilt n < w.
3:     h[n] := e
4:     siftUp(n)
5: end procedure
```

Die Einfügeoperation insert(e) setzt den neuen Eintrag e vorläufig an das Ende des Heaps. Danach wird e mit der Prozedur siftUp(n) zu einer passenden Position auf dem Weg von $h[n]$ zur Wurzel verschoben, bis die Heapbedingung wieder erfüllt ist.

Algorithm 6 Binärer Heap - Hilfsmethode siftUp().

```
1: procedure SIFTUP(j)                          ▷ Eingabe: Element j
2:     if j = 1 ∨ h[⌊j/2⌋] ≤ j  then
3:         return
4:     end if
5:     swap(h[j], h[⌊j/2⌋])
6:     siftUp(⌊j/2⌋)
7: end procedure
```

Die Operation extractMin() gibt den Inhalt der Wurzel zurück und ersetzt die Wurzel mit dem Inhalt von Knoten n im Heap. Da $h[n]$ eventuell größer ist als $h[2]$ oder $h[3]$, kann nun die Heapbedingung verletzt sein. Mit dem Aufruf der Prozedur siftDown() wird der neue Eintrag in der Wurzel des Baumes nach unten geschoben, bis die Heapbedingung wieder erfüllt ist.

Mit der Operation decreaseKey(k) wird der Schlüsselwert eines vorgegebenen Elementes auf den Wert k aktualisiert, sofern k kleiner als der aktuelle Schlüssel ist.

Die Operation build() wird verwendet, um einen Heap mit n gegebenen Elementen aufzubauen. Dieser kann in Zeit $O(nlog(n))$ aufgebaut werden, indem man die Elemente nacheinander einfügt. Eine mögliche Alternative dazu ist der Heap-Aufbau von unten nach oben (bottom-up). Mit der Prozedur siftDown()

Algorithm 7 Binärer Heap - Operation extractMin().

```
1: procedure EXTRACTMIN
2:     result := h[1]
3:     h[1] := h[n]
4:     n := n − 1
5:     siftDown(1)
6:     return result
7: end procedure
```

Algorithm 8 Binärer Heap - Hilfsmethode siftDown().

```
1: procedure SIFTDOWN(j)                          ▷ Eingabe: Element j
2:     if 2j ≤ n then                             ▷ j ist kein Blatt
3:         if 2j + 1 > n ∨ h[2j] ≤ h[2j + 1] then
4:             m := 2j
5:         else
6:             m := 2j + 1
7:         end if
8:         if h[j] > h[m] then
9:             swap(h[j], h[m])
10:            siftDown(m)
11:        end if
12:    end if
13: end procedure
```

kann die Heapbedingung in einem Unterbaum der Höhe $k + 1$ hergestellt werden, wenn diese vorher schon in beiden Unterbäumen der Höhe k gegolten hat. Insbesondere kann man den Vorteil nutzen, dass die Blätter des Baumes zu Beginn bereits die Heapbedingung erfüllen.

Der hier vorgestellte binäre Heap gehört zu den nicht-adressierbaren Prioritätswarteschlangen, da wir bei dieser Datenstruktur nicht mehr direkt auf das Element (bei einem Graphen wären das die Knoten) zugreifen können. Das hat zur Folge, dass speziell bei der decreaseKey()-Operation nach dem Knoten,

Algorithm 9 Binärer Heap - Operation decreaseKey().

```
1: procedure DECREASEKEY(e, k)
2:     m := searchElementPos(e)
3:     if h[m] > k then
4:         h[m] := k
5:         siftUp(m)
6:     end if
7: end procedure
```

Algorithm 10 Binärer Heap - Operation build().

```
1: procedure BUILDHEAPBACKWARDS
2:     for j := ⌊n/2⌋ downto 1 do
3:         siftDown(j)
4:     end for
5: end procedure
```

dessen Schlüssel reduziert werden soll, gesucht werden muss. Im schlechtesten Fall muss hier der gesamte Heap durchsucht werden, was mit einer worst-case Laufzeit von $O(n)$ zu veranschlagen ist. Dieser Aufwand kann mit dem Führen einer zusätzlichen Referenz, die zum Beispiel in einem weiteren Array gespeichert und in der Anwendung mitgeführt wird, auf eine Laufzeit von $O(log(n))$ reduziert werden. Die beschriebenen Prozeduren müssten in diesem Fall noch um die Aktualisierung der Referenz erweitern werden, überall da, wo sich die Position eines Knotens verändert. Die Aktualisierung der Referenz kann in $O(1)$ Zeit durchgeführt werden.

Zusammenfassend können wir folgendes Laufzeitverhalten vermerken: Wenn man einen binären Heap implementiert, benötigt die Erstellung eines leeren Heaps und das Finden des Minimums jeweils konstante Zeit, die Operationen extractMin() und insert() können in logarithmischer Zeit $O(log(n))$ ausgeführt werden, build() und decreaseKey() benötigen lineare Zeit. Durch das Führen einer zusätzlichen Referenz auf den Knoten (auch als Griff auf den Knoten bezeichnet) kann die Laufzeit von decreaseKey auf $O(log(n))$ verbessert werden. Mit dieser Verbesserung erhalten wir nach der obigen Laufzeitformel (1) eine worst-case Gesamtlaufzeit für den Dijkstra-Algorithmus von

$$T_{Dijkstra} = O(|E| \cdot O(log(n)) + |V| \cdot O(log(n))) = O((|E| + |V|) \cdot log(n)).$$

4.2.3 Bellman–Ford-Algorithmus

Der Bellman–Ford-Algorithmus berechnet ebenfalls die kürzesten Pfade von einem gegebenen Startknoten zu allen anderen Knoten im Graphen. Im Unterschied zum Dijkstra-Algorithmus kann er Eingabegraphen mit positiven und negativen Kantengewichten verarbeiten. Solange sichergestellt ist, dass der Graph keinen negativen Zyklus enthält, also einen Zyklus bei dem die Summe der Kantengewichte kleiner als Null ist, findet der Bellman–Ford-Algorithmus den kürzesten Pfad [14]. Dabei kommt er mit einer Standard-Prioritätswarteschlange (Priority Queue), die nach dem FIFO-Prinzip (first in first out) organisiert ist, aus. Eine Sortierung nach der bisher berechneten kürzesten Distanz ist hier nicht erforderlich. Die Prioritätswarteschlange wird in der nun folgenden Beschreibung des Algorithmus mit Q abgekürzt.

In der Beschreibung des Bellman–Ford-Algorithmus 11 gehen wir davon aus,

Algorithm 11 Bellman–Ford-Algorithmus.

```
1: procedure BellmanFord(G, s)  ▷ Eingabe: Graph G, Startknoten s
2:     dist[s] = 0
3:     pred[s] = None
4:     Q.enqueue(s)
5:     while not Q.isEmpty() do
6:         v = Q.dequeue()
7:         for w in Neighborhood(v) do
8:             if dist[w] > dist[v] + length[(v, w)] then
9:                 pred[w] = v
10:                dist[w] = dist[v] + length[(v, w)]
11:                if not Q.contains(w) then
12:                    Q.enqueue(w)
13:                end if
14:            end if
15:        end for
16:    end while
17:    return dist[], pred[]
18: end procedure
```

dass der Eingabegraph in Form einer Adjazenzliste oder einer Adjazenzmatrix vorliegt. Die Laufzeit des Algorithmus beträgt im schlechtesten Fall (worst case) $O(|V||E|)$, wobei mit $|V|$ die Anzahl der Knoten und mit $|E|$ die Anzahl der Kanten gemeint ist. Die Korrektheit und die Laufzeit haben Hochstättler und Schliep bewiesen, weswegen dies hier nicht näher ausgeführt wird [14].

Es gibt auch eine Variante des Bellman–Ford-Algorithmus, in der negative Zyklen erkannt werden. Diese wird in Cormen et. al. detailliert beschrieben [6]. Da wir jedoch in unserer Betrachtung nur Graphen als Eingabe zulassen, die keine neagtive Zyklen enthalten, haben wir auf die Betrachtung dieser Variante verzichtet.

5 Implementierte Lösungen

5.1 Systemvoraussetzungen und Entwicklungsumgebung

Die hier beschriebenen Programme und auch die Experimente wurden in den im Kapitel 7.2 beschriebenen Environments getestet. Die mit der Java Standard Edition 8 (Java SE 8) entwickelten Programme sind zwar durch den Bytecode und die Nutzung der jeweils passenden Java Virtual Machine nahezu systemunabhängig, dennoch gibt es in der Implementierung auch Teile, die auf die genutzten Environments zugeschnitten sind. Ein Beispiel dafür sind die Nutzung von Umgebungsvariablen oder auch die Erstellung von Skripten, die im Betriebssystem Windows als Batch-File und im Betriebsystem Mac OS X als Shell-Skript ausgeführt werden, um gewisse Aktivitäten in den Experimenten, wie zum Beispiel das Starten eines Services oder das Anstoßen eines Datenbank-Imports weitgehend zu automatisieren.

Als Entwicklungsumgebung wurde die NetBeans IDE 8.2 eingesetzt, die man sich kostenlos im Internet von netbeans.org herunterladen kann [25] und die auf zweierlei Weise lizensiert ist, einmal unter der Common Development and Distribution License (CDDL) und zusätzlich der GNU General Public License, Version 2 mit Classpath-Ausnahme.

Die Entwicklung der Java-Programme wurde mit dem Java SE Development Kit 8 der Oracle Corporation [15] durchgeführt. Daher setzen wir voraus, dass das System, auf dem die Software ausgeführt wird, mit einem Java SE Development Kit 8 oder einer wenigstens einer Java Runtime 8 ausgestattet ist.

Für die Arbeit und die Experimente mit der Neo4j-Graph-DB sollte diese ebenfalls in dem Environment installiert sein, in dem die Experimente durchgeführt werden. Es gibt zwar die Möglichkeit die Neo4j-Graph-DB auf einem separaten Server zu installieren, doch für die Experimente wurde die Neo4j-Desktop Software verwendet, bei dem Client-Anwendung und die Neo4j-Graph-DB auf einem System läuft. So sind Experimente auf Einzelsystemen, wie einem PC oder einem Laptop möglich. Informationen zur verwendeten Software und Version werden im Kapitel 7.2 beschrieben.

Für die Erweiterung von Neo4j durch die Erstellung eines Neo4j-Plugins wurde ebenfalls die NetBeans IDE genutzt, da diese mit einem Maven-Plugin ausgestattet ist und somit nach Erstellung eines Maven-Projekts mit entsprechenden Angaben in der Datei `pom.xml` die notwendigen Abhängigkeiten (Neo4j Dependencies) von dem Maven-Repository aus dem Internet geladen werden konnten. Neben den notwendigen Jar-Files (Neo4j Bibliotheken) wird dabei gleichzeitig die passende Maven-Compiler-Version heruntergeladen. Dieser Compiler ist notwendig, um die Erweiterung mit der eigenen Neo4j-Procedure als Jar-File gemäß den Vorgaben von Neo4j Inc. zu paketieren. Der Download wird mit dem ersten Ausführen von ‚Clean and Build' für das Maven-Projekt in der NetBeans

IDE durchgeführt. Die so erzeugte Jar-File muss anschließend in das Verzeichnis NEO4J_HOME/plugins kopiert werden. Nach einem Neustart der Neo4j-DB ist das Plugin aktiv. Weitere Details hierzu werden in Kapitel 5.4 und in der Neo4j Dokumentation [23] beschrieben.

Für die Auswertungen der Messdaten der Experimente wurde R, eine Programmiersprache für statistische Auswertungen und Grafiken, in der Version 3.5.1 eingesetzt [36].
Zusätzlich zu den Ausführungen in den nachfolgenden Kapiteln ist die Software auch im Sourcecode kommentiert. Der Sourcecode, die Jar-Files, die R-Auswertungen und die Experimentdaten sind dieser Arbeit auf einem USB-Stick beigefügt. Im Hauptverzeichnis des USB-Sticks befindet sich eine Datei mit einer Übersicht der Verzeichnisse und Inhalte des USB-Sticks.

5.2 Der Graphgenerator

5.2.1 Beschreibung des Funktionsumfangs

Der Graphgenerator ist eine GUI-basierte Anwendung, mit der ein einzelner oder auch Hunderte von Graphen generiert werden können. Er verfügt über drei Generatoren:

- Barabási–Albert-Generator
- Erdös–Rényi-Generator
- Externer Graph Generator

Das Programm wird in der Konsole aus dem Verzeichnis, in dem sich die Datei GraphGen2018.jar befindet, wie folgt gestartet:

Listing 2: Start des Graphgenerators.

```
> java -Xms12g -Xmx12g -jar GraphGen2018.jar
```

Bei dem Aufruf des Graphgenerators wird die für die Anwendung benötigte JVM Heap-Größe auf 12 GB festgelegt. Dieser Wert ist abhängig von dem Hauptspeicher, den das System (Laptop / PC) zur Verfügung hat und der Aufruf des Programms muss entsprechend darauf angepasst werden. In den meisten Fällen ist es sinnvoll, den initialen Heap (Parameter -Xms) genauso groß zu wählen, wie den maximalen Heap (Parameter -Xmx).

Nach dem Start des Programms wählen wir im Menü einen Generator aus, siehe Abbildung 10.

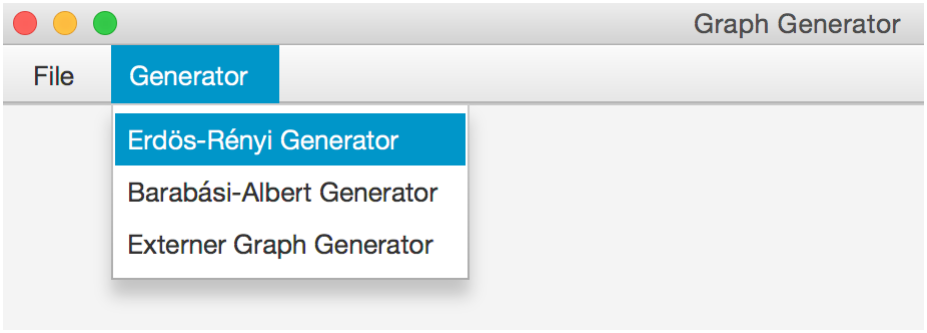


Abbildung 10: Graphgenerator: Auswahl des Generators.

Im dann folgenden Dialog geben wir die Parameter des zu generierenden Graphen ein. Neben der Eingabe der Anzahl der Knoten, ist die Kantenwahrscheinlichkeit einzugeben. Diese sollte bei größeren Graphen entsprechend klein gewählt werden, da bei dem zugrunde liegenden Erdös–Rényi-Modell die Anzahl der Kanten mit einer höheren Kantenwahrscheinlichkeit erheblich ansteigt. Als Abwandlung zum Modell kann festgelegt werden, ob gerichtete oder auch gewichtete Graphen generiert werden sollen.

Da gleich mehrere Graphen generiert werden können, wird der Dateiname dynamisch bestimmt und es erscheint kein Speicherdialog. Dafür ist das Verzeichnis auszuwählen, indem dann die generierten Graphen gespeichert werden. Dabei wird hier das Hauptverzeichnis des Experiments ausgewählt. Hat das Verzeichnis des Experiments zum Beispiel den Namen `Experiment1`, so müssen vor dem Generieren der Graphen einmalig drei Unterverzeichnisse mit Namen ‚input', ‚config' und ‚config_template' angelegt werden. In diese Verzeichnisse werden die vom Graphgenerator ausgegebenen Dateien geschrieben. Dabei wird für jeden generierten Graphen eine Textdatei mit den Daten des Graphen im Verzeichnis ‚input' abgelegt. Außerdem werden Konfigurationsdateien angelegt, die für die Messexperimente später zur Steuerung des Messlaufs verwendet werden. Diese werden gleich mit dem Graphen erstellt, da es zu mühselig wäre, diese später manuell für die Messexperimente zu pflegen.

Beim Generieren von Graphen bis zu 1000 Knoten, wird der generierte Graph auf dem Canvas angezeigt, wobei die Knoten des Graphen als n -Eck ($n =$ Anzahl der Knoten) arrangiert sind. Speziell beim Erdös–Rényi-Modell können wir dadurch schöne Muster erzeugen. Bei einer Kantenwahrscheinlichkeit von Eins wird der vollständige Graph angezeigt, siehe Abb. 12.

Zudem können wir bei der Angabe der Parameter auch auswählen, ob der Graph mit Knotennummern angezeigt werden soll. Zusätzlich zu der automatischen Speicherung der Graphdaten, gibt es die Möglichkeit auch das Bild eines einzeln generierten Graphen im PNG-Format zu speichern.

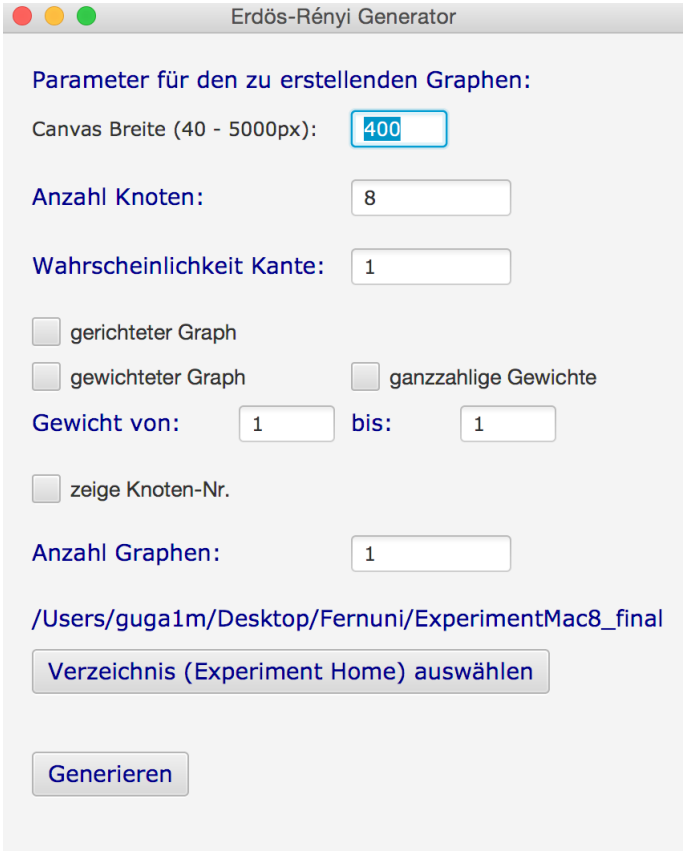


Abbildung 11: Graphgenerator: Parametereingabe.

Bei den Parameterangaben für die Generierung eines Graphen basierend auf dem Barabási–Albert-Modell ist neben der Kantenwahrscheinlichkeit noch ein weiterer Parameter einzugeben, mit dem festgelegt wird, wie viele Knoten der Startgraph besitzen soll. Dieser wird dann automatisch auf Basis des Erdös–Rényi-Modells generiert. Danach wird dann gemäß Barabási–Albert-Modell der Graph Kante für Kante abhängig von den aktuellen Knotengraden in dem Graphen aufgebaut, bis die Anzahl der vorgegebenen Knoten erreicht ist. Abbildung 13 zeigt ein Beispiel eines so generierten Graphen. Außerdem wird mit der Abbildung auch die Möglichkeit der Speicherung des generierten Bildes, siehe markierter Menüeintrag, gezeigt.

Der ‚Externe Graph Generator' wurde nachträglich in den Graphgenerator integriert, um auch extern aus dem Internet geladene Real-World-Graphen für die Experimente nutzbar zu machen. In dem zugehörigen Parameter-Fenster sind daher ein paar Eckdaten zu dem Graphen zu erfassen, bevor dieser aus einer Textdatei geladen werden kann. Neben der Anzahl der Knoten (diese entspricht der maximalen Knotennummer in der Eingabedatei, erhöht um Eins) ist anzugeben, ob es sich um einen gerichteten oder gewichteten Graphen handelt. Auch hier kann die Anzahl der zu generierenden Graphen festgelegt werden.

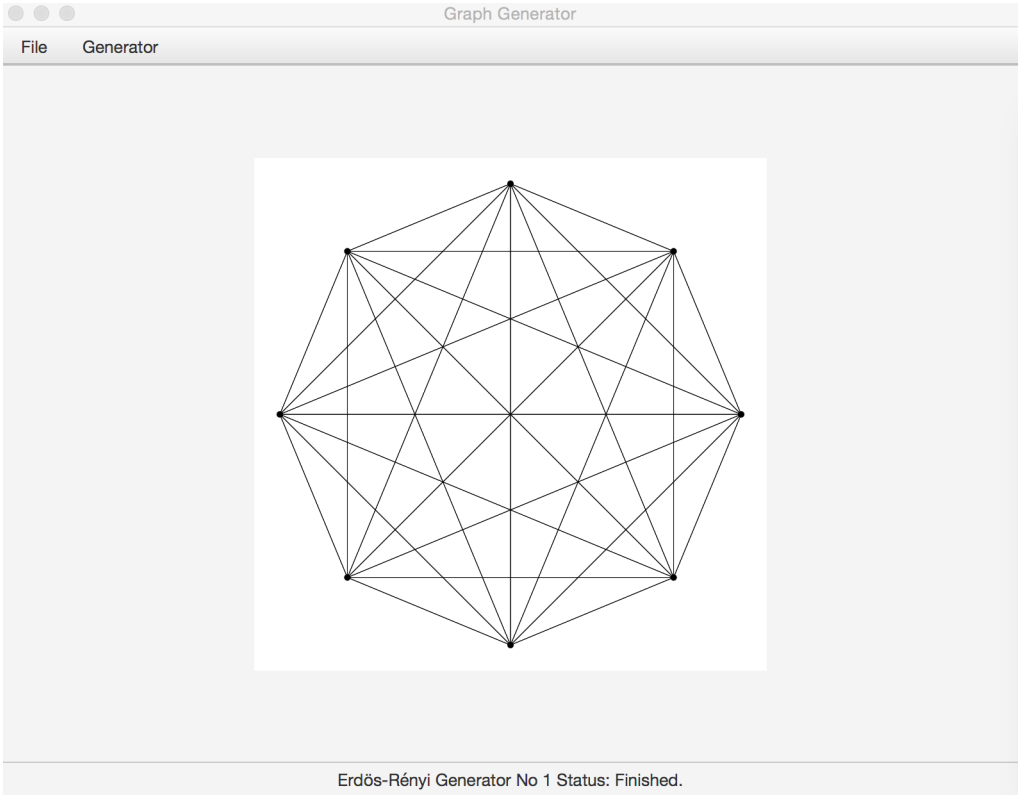


Abbildung 12: Graphgenerator: Beispiel eines mit dem Erdös–Rényi-Generator generierten Graphen.

Anders als bei den anderen Generatoren wird dabei jedoch kein neuer Graph generiert, sondern nur die notwendigen Eingabedateien erzeugt, die für ein Messexperiment erwartet werden. Bei der Bearbeitung der Eingabedatei werden noch ein paar Bereinigungen vorgenommen, die u.a. dafür sorgen, dass es in der damit erzeugten Graphdatei keine Knoten gibt, die auf sich selbst zeigen.

Bei extern geladenen Graphen empfehlen wir die Eingabedateien vor dem Einlesen in den Graphgenerator einer Sichtprüfung zu unterziehen. Hier sollte geprüft werden, ob das Format der Eingabedatei zu dem erwarteten Format passt. Ein Datensatz sollte hier aus ‚FromId ToId' oder ‚FromId ToId Weight' bestehen, wobei die Spalte ‚Weight'nur positive, ganzzahlige Gewichte enthalten sollte. Dies haben wir für Eingabegraphen für die Experimente so festgelegt, um sicher zu stellen, dass wir keinen Eingabegraphen mit negativen Zyklen erhalten. Mit `FromId` ist hier die Id des Startknotens gemeint. `ToId` ist die Id des Zielknotens zu dem eine eingehende Kante von `FromId` existiert. Die Datenfelder können durch ein oder mehrere Leerzeichen oder durch einen Tabulator getrennt (TSP-Format) gespeichert sein. Kommentare werden mit dem Zeichen # zu Beginn einer Zeile eingefügt. Diese werden dann beim Einlesen der Daten des Graphen aus einer Datei ignoriert. Das hier beschriebene Format wird so auch bei der

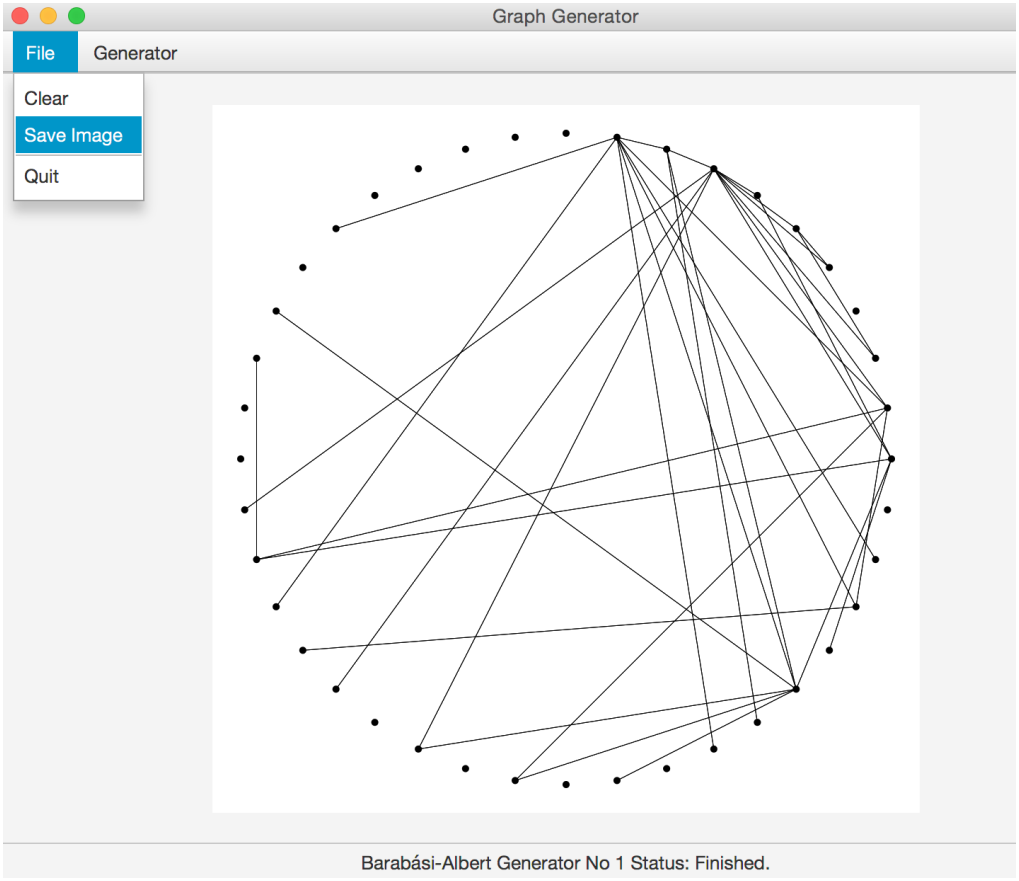


Abbildung 13: Graphgenerator: Beispiel eines mit dem Barabási–Albert-Generator generierten Graphen.

Ausgabe der generierten Graphen verwendet, also bei den Dateien, die in das Unterverzeichnis ‚input' gespeichert werden. In Abbildung 14 sehen wir ein Beispiel einer Graphdatei, wie sie durch den Graphgenerator erzeugt wird.

Bei allen drei Generatoren wird unmittelbar nach der Generierung des Graphen zufällig ein Startknoten bestimmt und zu diesem ein Zielknoten ermittelt und zwar so, dass es einen Pfad von dem Startknoten zu dem Zielknoten in dem Graphen gibt. Durch die zufällig neu gewählten Startknoten in jedem generierten Graphen wollen wir eine gewisse Vielfalt erreichen, damit es bei den Messungen der Laufzeit weniger Auswirkungen durch besonders günstige oder besonders ungünstige Konstellationen durch die Wahl des Startknotens kommt. Das ist auch der Grund, warum wir uns entschieden haben, lieber mehr Graphen zu generieren und dafür etwas weniger Messungen pro Graph durchzuführen, anstatt mit wenigen Graphen möglichst viele Wiederholungen zu testen.

Die Pakete und Klassen des Graphgenerators werden im Anhang B.1 beschrieben.

BA_78_250000_s...w1bis10_dir.txt ×

```
1   # Graph bereitgestellt mit Graph-Generator von Gundula Swidersky.
2   # Bei der Vorbereitung für Laufzeitexperimente wurden unterschiedli
3   # Bei extern eingelesenen Graphen wurden Schleifen von Knoten zu si
4   #
5   # Datum und Zeit der Erstellung: 20180814,02:10:28.259
6   # Generiert auf Basis Barabási-Albert-Modell, Kantenwahrscheinlichk
7   # Graph Typ: 2, gewichteter, gerichteter Graph.
8   # Achtung: Bei ungerichteten Graphen wird nur Kante (i,j) in Datei
9   # Anzahl Knoten gesamt: 250000, Anzahl Kanten: 311442
10  # FromID    ToID   Weight
11  0   5   8
12  0   6   6
13  0   8   5
14  0   9   8
15  0   88   7
16  0   109   8
17  0   170   5
18  0   198   7
19  0   243   8
20  0   445   9
21  0   467   7
22  0   540   2
23  0   616   9
24  0   680   3
25  0   824   5
26  0   842   4
27  0   847   8
28  0   1241   10
29  0   1267   6
30  0   1268   10
31  0   1414   2
32  0   1596   5
33  0   1929   3
34  0   2275   9
35  0   2296   2
36  0   2382   9
37  0   2422   7
38  0   2493   6
39  0   2528   10
40  0   2657   3
```

Abbildung 14: Graphgenerator: Beispiel einer Graphdatei, wie sie durch den Graphgenerator erzeugt und ausgegeben wird.

5.2.2 Verwendete Datenstrukturen

Im Graphgenerator wird der erstellte oder gelesene Graph kodiert in Form einer Adjazenzliste in der Klasse `AdjListGraph` geführt. Die Adjazenzliste ist mit einer `ArrayList` vom Typ `Edge` mit der Dimension Knotenanzahl realisiert. Der Datentyp `Edge` ist in einer separaten Klasse namens `Edge` abgebildet. `Edge` besteht hier aus zwei einfachen Datentypen, nämlich `target` vom Typ Integer und einem generischen, numerischen Datentyp `T` namens `weight`. In `gType` wird der Graphtyp gespeichert. Zudem wird in `directedG` geführt, ob es sich um einen gerichteten Graphen handelt. Die Instanzvariable `amountNodes` speichert die Anzahl der Knoten des Graphen. Der Maximalwert in `maxValue` wird als Maximalwert in Algorithmen genutzt und richtet sich nach dem Datentyp `weight`. Abbildung 15 gibt einen Überblick über die Klassen im Paket `graphs` mit Instanzvariablen und Methoden.

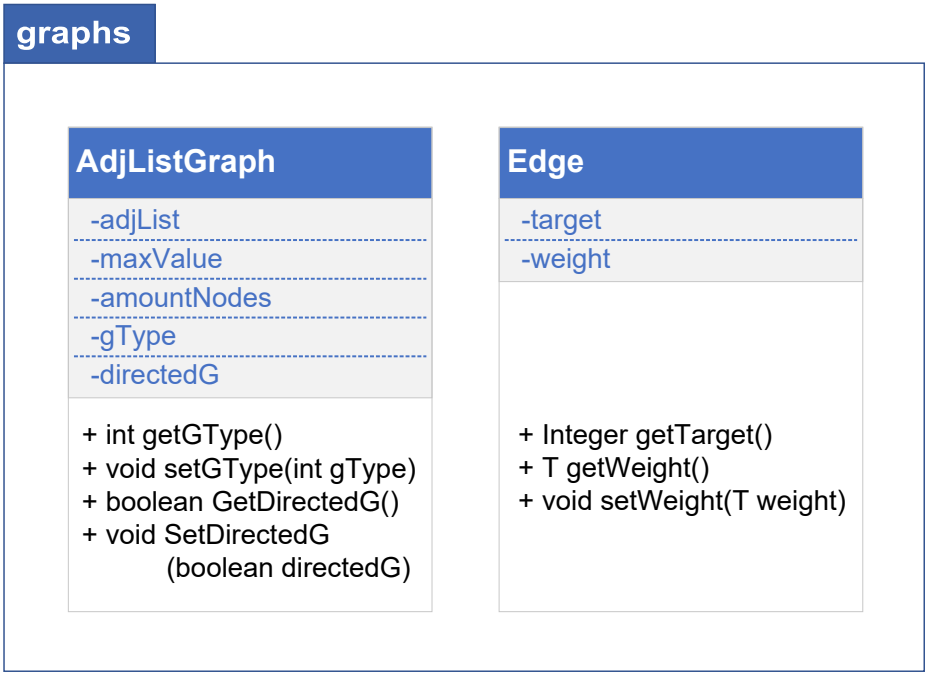


Abbildung 15: Graphgenerator: Datenstrukturen im Paket graphs.

5.3 Implementierte Stand-Alone-Lösung

5.3.1 Beschreibung des Funktionsumfangs

In der Stand-Alone-Lösung wurden die Algorithmen implementiert, die für den Vergleich der Performanz mit der Neo4j-Graph-DB in den Experimenten genutzt werden. Die hier implementierten Algorithmen werden im Kapitel 4.2 beschrieben. Daher werden wir hier nur ergänzend auf die in der Stand-Alone-Lösung implementierten Varianten eingehen.
Mit dem Start des Hauptprogramms mit Namen `MyApp1`, siehe Listing 3, wird auch gleichzeitig der Messlauf der Stand-Alone-Lösung gestartet. Als Eingabe wird die Angabe des Verzeichnisses mit den Konfigurationsdateien erwartet. Diese Dateien dienen nämlich als Steuerung für den Messlauf. Zunächst sollte vor dem Start des Programms die für das Messexperiment notwendige Verzeichnisstruktur auf Betriebssystemebene angelegt werden. Im eventuell noch anzulegenden Hauptverzeichnis für das Experiment, wir nennen es hier `EXPERIMENT_HOME`, müssen folgende Unterverzeichnisse vorhanden sein:

- input
- config
- config_template
- measure

Das Programm wird in der Konsole aus dem Verzeichnis, in dem sich die Datei MyApp1.jar befindet, wie folgt gestartet:

Listing 3: Start der Stand-Alone-Anwendung MyApp1.

```
> java -Xms12g -Xmx12g -jar MyApp1.jar
"EXPERIMENT_HOME/config"
```

Eventuell existiert bereits ein Teil der obigen Verzeichnisse, denn meist werden diese für den Aufruf des Graphgenerators vorher angelegt. Sollten sich die Graphdateien und Konfigurationsdateien noch nicht in den Verzeichnissen `input`, `config` und `config_template` befinden, so müssen diese vor dem Start von `MyApp1` in die entsprechenden Verzeichnisse kopiert werden. Der Aufbau einer Graphdatei wird in Kapitel 5.2.1 beschrieben. Die erwarteten Konfigurationsdateien werden wir im Folgenden erklären.

Die Konfigurationsdateien sollten in der Regel bereits von dem Graphgenerator erzeugt worden sein, wobei zu jedem Graphen ein sogenanntes Template für die Konfigurationsdatei im Unterverzeichnis `config_template` abgelegt wird. Wir erkennen die zugehörige Konfigurationsdatei am Namen, da dieser mit dem Namen der Graphdatei beginnt. Hier wurde lediglich die Endung `config.txt` angehängt.

Im Unterverzeichnis `config` befinden sich dann die Konfigurationsdateien für die Steuerung des Messlaufs. Die Anzahl und ihre Inhalte sind für die Experimente so festgelegt worden, d.h. in der Ausgabe des Graphgenerators entsprechend programmiert worden. Der Grund für dieses Vorgehen ist, dass es sehr mühsam ist, diese Dateien für die vielen Graphen manuell zu pflegen. Dennoch ist es möglich, diese Dateien auch manuell zu erstellen oder zu editieren, wenn das nötig ist.
Im Unterverzeichnis `config` gibt es somit zu jedem Graphen drei Konfigurationsdateien. Die Endung der Dateinamen ist jeweils `config1.txt`, `config2.txt` oder `config3.txt`. Dabei gibt es auch eine Zuordnung zu den Algorithmen, deren Laufzeiten getestet werden sollen. Die Konfigurationsdatei mit Endung `config1.txt` steuert die Ausführung und Messung der bidirektionalen Breitensuche. Die Datei mit Endung `config2.txt` steuert die Berechnung der kürzesten Pfade mit dem Dijkstra-Algorithmus und die Datei mit Endung `config3.txt` steuert die Ausführung und Messung mit dem Bellman–Ford-Algorithmus. Belässt man diese Dateien im Verzeichnis `config`, so werden damit alle drei Algorithmen ausgeführt und dazu Laufzeiten gemessen. Will man zum Beispiel nur die bidirektionale Breitensuche in einem Messexperiment testen, dann sollte man im Verzeichnis `config` nur Dateien mit Endung `config1.txt` bereitstellen, bevor `MyApp1` aufgerufen wird.

Der Satzaufbau einer Konfigurationsdatei ist immer gleich, egal, ob es sich um eine Konfigurationsdatei aus dem Verzeichnis `config_template` oder aus

dem Verzeichnis `config` handelt. Tabelle 8 beschreibt den Satzaufbau einer Konfigurationsdatei.

Tabelle 8: Aufbau einer Konfigurationsdatei.

Nr.	Name des Datenfeldes
1.	Name der Graphdatei (String)
2.	Anzahl Knoten des Graphen (integer)
3.	Graphtyp (einstellige Zahl)
4.	Schalter ob gerichteter Graph (true / false)
5.	Nummer des Algorithmus (einstellige Zahl)
6.	Nummer der zu verwendenden Queue (einstellige Zahl)
7.	Startknoten (integer)
8.	Zielknoten (integer)
9.	Anzahl der Läufe zum Aufwärmen (integer)
10.	Anzahl Wiederholungen des Messlaufs (integer)
11.	Schalter Ausgabe des kürzesten Pfades (true / false)
12.	Schalter ob Statistiken zum Graphen berechnet werden sollen (0 / 1)
13.	Schalter ob Ausgabe der Messdaten (true / false)
14.	Weitere informelle Info zum Graphtyp (integer)
15.	Optionale Experimentinformation (String)

Beispiel des Inhalts einer Konfigurationsdatei

```
EX_out.com-dblp.txt_1_317081.txt,317081,0,false,0,2,197244,1,0,10,false,0,
true,-1,C:/Fernuni/ExperimentWinPCExt
```

Der Name des Graphen wird dort bewusst ohne Pfadangaben hinterlegt, da wir diese Dateien zusammen mit den Dateien der Eingabegraphen kopieren und für weitere Laufzeittests in verschiedenen Environments nutzen können. Jede Konfigurationsdatei enthält nur einen Datensatz. Der modulare Aufbau der verwendeten Dateien für die Experimente hat den Vorteil, dass es recht einfach ist, ein einmal abgebrochenes Experiment fortzusetzen oder auch gezielte Nachmessungen von bestimmten Graphen oder Algorithmen durchzuführen.

Folgende Graphtypen sind hier zugelassen, wobei nur Graphtyp 0, 1 oder 2 für die Experimente eingesetzt werden, siehe Tabelle 9.

Tabelle 10 gibt einen Überblick über die Bedeutung der Nummer der verwendeten Warteschlange in der Konfigurationsdatei.

Tabelle 9: Übersicht Graphtypen.

Graphtyp	Beschreibung
0	ungewichteter ungerichteter Graph
1	ungewichteter gerichteter Graph
2	gewichteter Graph mit ganzzahligen Kantengewichten ≥ 0
3	gewichteter Graph mit Kantengewichten, Typ Double ≥ 0.0
4	gewichteter Graph mit ganzzahligen positiven und negativen Kantengewichten (ohne negativen Zyklus)
5	gewichteter Graph mit positiven und negativen Kantengewichten vom Typ Double (ohne negativen Zyklus)

Tabelle 10: Konfigurationsdatei: Zuordnung Nummer zu Warteschlange.

Queue Nr.	Klasse	Beschreibung
0	StdMinPQ	Einfache Min-Prioritätswarteschlange
1	BinaryHeapPQ	Binärer Heap
2	StdQueue	Standard-Queue (FIFO-Prinzip)

Die in einer Konfigurationsdatei hinterlegten Nummern für die Algorithmen haben folgende Bedeutung, siehe Tabelle 11.

Wird nun `MyApp1` gestartet, so werden die Konfigurationsdateien im Verzeichnis `config` in eine Liste eingelesen und nacheinander abgearbeitet. Der Inhalt einer jeden Konfigurationsdatei wird gelesen und danach der zugehörige Eingabegraph. Dann wird die Ausführung des Algorithmus, wie vorgegeben, angestoßen und die gemessenen Laufzeiten werden in eine Datei ausgegeben. Der Name der Datei wird dynamisch erzuegt und hat die folgende Form: ‚`Messung_Datum_Uhrzeit_NameConfigDatei.txt`‘. Die Dateien werden in einem neu angelegten Unterverzeichnis von `EXPERIMENT_HOME/measure` mit Namen `runxxx` angelegt, wobei `xxx` für Datum und Uhrzeit des Messlaufs steht. Da dieses Verzeichnis auch genutzt wird, um die zu dem Experiment gemessenen Laufzeiten mit Neo4j abzulegen, wird ein Script erstellt, mit dem im Rahmen des Neo4j-Messlaufs eine Environment-Variable auf das Verzeichnis gesetzt wird. Das entsprechende Skript ist dann im Unterverzeichnis `scriptsmac` oder `scriptswin` abgelegt, je nach verwendetem Betriebssystem. Als Statusinformation wird während der Ausführung von `MyApp1` der Inhalt der Konfigurationsdatei, die gerade bearbeitet wird, auf der Konsole angezeigt.

Abbildung 16 zeigt den Aufbau einer Messdatei. Die Pakete und Klassen der Stand-Alone-Lösung werden im Anhang B.2 beschrieben.

Tabelle 11: Übersicht Zuordnung Nummer zu Algorithmus.

Algorithmus Nr.	Methode	Beschreibung
0	doBidirectBFSUndirGraph	Bidirektionale Breitensuche in ungerichteten, ungewichteten Graphen (ignoriert Gewichte)
1	doBidirectBFSDirGraph	Bidirektionale Breitensuche in gerichteten, ungewichteten Graphen, Adj-Liste Vorgänger u. Nachfolger (ignoriert Gew.)
2	doIntDijkstra	Dijkstra für Graphen mit ganzzahligen Gewichten ≥ 0 (ungewichtete Graphen erhalten Gewicht 1).
3	doDoubleDijkstra	Dijkstra für Graphen mit Gewichten, Typ Double ≥ 0.0 (ungewichtete Graphen erhalten Gewicht 1.0).
4	doNoNegCycleIntBellmanFord	Bellman–Ford-Algorithmus mit positiven und negativen ganzzahligen Kantengewichten
5	doNoNegCycleDoubleBellmanFord	Bellman–Ford-Algorithmus mit positiven und negativen Kantengewichten, Typ Double
6	doNoNegCycleBellmanFord	Bellman–Ford-Algorithmus mit positiven und negativen Kantengewichten vom Typ Integer oder Double

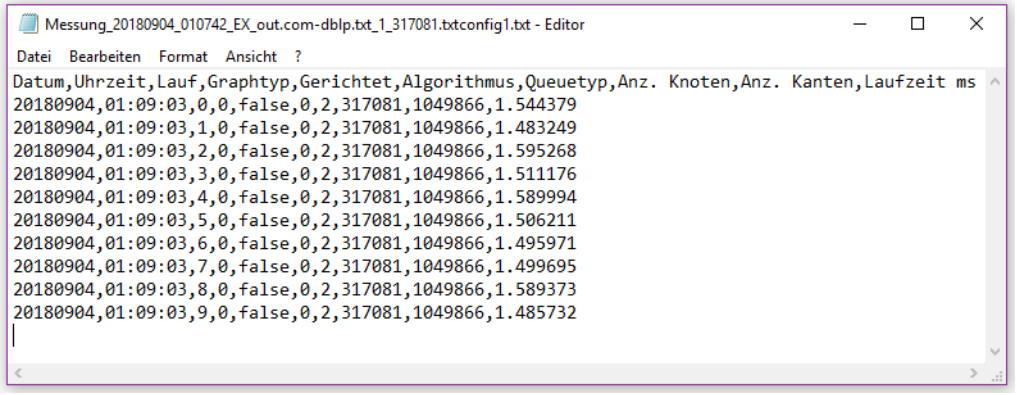


Abbildung 16: Stand-Alone-Lösung: Aufbau einer Messdatei.

5.3.2 Verwendete Datenstrukturen

Der in der Stand-Alone-Lösung verwendete Eingabegraph wird entweder in der Datenstruktur `AdjListGraph` oder `AdjListDirGraph` verwaltet. Letztere wird nur für gerichtete, ungewichtete Graphen im Rahmen der bidirektionalen Breitensuche verwendet und ist eine Erweiterung `AdjListGraph` um eine Adjazenzliste mit den Vorgängerknoten, um effizient auf diese zugreifen zu können.

Abbildung 17 gibt einen Überblick über die Klassen im Paket `graphs` mit Instanzvariablen und Methoden.

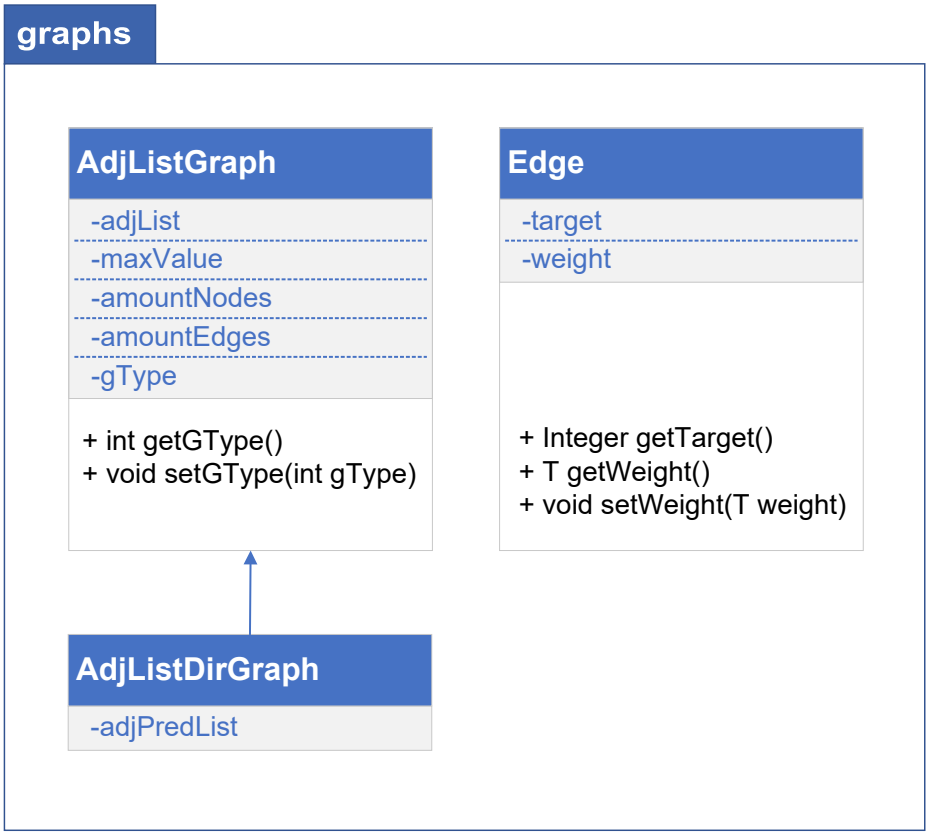


Abbildung 17: Stand-Alone-Lsg: Datenstrukturen im Paket graphs.

Die Datenstruktur `AdjListGraph` ist der in dem GraphGenerator verwendeten sehr ähnlich. So wird hier zusätzlich noch die Anzahl der Kanten abgelegt, die während des Messlaufs ermittelt wird. Dafür wird hier keine Instanz-Variable geführt, ob es sich um einen gerichteten Graphen handelt, da diese Information über die eingelesene Konfigurationsdatei verwaltet wird. Diese geringfügigen Abweichungen haben sich durch die zeitlich getrennte Entwicklung der beiden Programme ergeben. In einer Weiterentwicklung der Lösungen kann man

sicherlich das Paket graphs so gestalten, dass es gemeinsam von beiden Anwendungen genutzt wird. Allerdings sind dann auch beide Lösungen von der gemeinsamen Datenstruktur abhängig.

Weiterhin werden in der Stand-Alone-Lösung verschiedene Typen von Warteschlangen implementiert, die dann in den Algorithmen bei der Berechnung der kürzesten Pfade zum Einsatz kommen. Abbildung 18 gibt einen Überblick über die Klassen im Paket queues mit Instanzvariablen und Methoden.

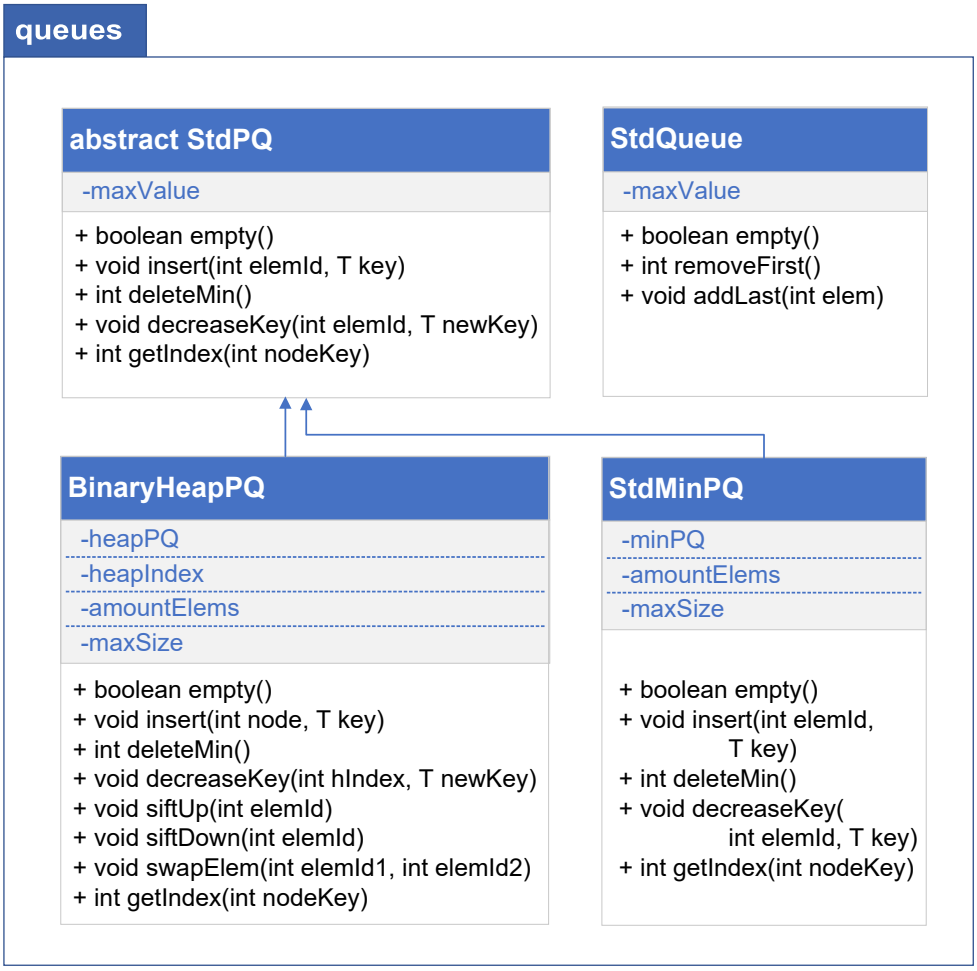


Abbildung 18: Stand-Alone-Lsg: Datenstrukturen im Paket queues.

StdQueue ist eine Standard-Prioritätswarteschlange, die nach dem FIFO-Prinzip organisiert ist und die im Bellman–Ford-Algorithmus eingesetzt wird. StdPQ ist eine abstrakte Klasse, die durch BinaryHeapPQ und StdMinPQ erweitert wird. Wie wir bereits an den Namen der Klassen erkennen können, implementiert die Klasse BinaryHeapPQ einen binären Heap, während die Klasse StdMinPQ eine

Standard-Min-Prioritätswarteschlange implementiert. Die Funktionsweise dieser Warteschlangen wird im Kapitel 4.2.2 ausführlich erklärt.

Wie dort erwähnt, wird in der Stand-Alone-Lösung beim binären Heap eine zusätzliche Referenz auf den Knoten im Heap geführt, um den Schlüsselwert mit der Methode `decreaseKey()` direkt im Zugriff zu haben und dadurch die Laufzeit für die Suche des Knotens im Heap einzusparen.

5.4 Neo4j-Plugins, Eigene Neo4j-Procedure

Neben den Standard-Procedures, zu denen die sogenannten APOC-Procedures und auch die Procedures mit den effizienten Graphalgorithmen gehören, können wir Neo4j auch mit eigenen Procedures erweitern. Eine Arbeitsanleitung dazu finden wir in der Neo4j-Dokumentation [23] und zwar im zur Neo4j-Version passenden Developer Manual.

Wir nutzen diese Erweiterungsmöglichkeit der Neo4j-Graph-DB, um den Bellman–Ford-Algorithmus, der nicht Bestandteil der Neo4j-Graphalgorithmen ist, als eigene Procedure in Neo4j zu implementieren. Dadurch können die Laufzeiten des selben Algorithmus in Neo4j gemessen werden und so die Laufzeiten zwischen Neo4j und der implementierten Stand-Alone-Lösung verglichen werden.

Im Folgenden werden wir die Schritte für die Erweiterung mit einer Procedure in Neo4j anhand des implementierten Algorithmus beschreiben.

Als Erstes müssen wir sicherstellen, dass wir in einer Entwicklungsumgebung arbeiten, mit der das zu erstellende Programm als Jar-File paketiert werden kann. Im Neo4j Developer Manual wird empfohlen, ein Maven-Build-System zu verwenden. Das bedeutet, dass wir jede Entwicklungsumgebung verwenden können, die ein Maven-Plugin installiert hat. Wir verwenden hier die Netbeans IDE, Version 8.2, da bei dieser bereits bei der Installation automatisch ein Maven-Plugin mit installiert wurde.

Nach dem Anlegen eines Maven-Projekts müssen wir die Projektdatei namens `pom.xml`, siehe Auszug in Abbildung 19, mit den Daten pflegen, die wir für unsere Procedure benötigen. Zuerst pflegen wir die Inhalte der Felder `groupId`, `artifactId`, `name` und `description`. Der nächste wichtige Schritt ist, die Abhängigkeiten für den Build zu definieren, da damit die zu verwendende Schnittstelle, die verwendeten Neo4j-Bibliotheken und auch die eingesetzten Versionen definiert werden. Die notwendigen Dateien werden dann automatisch beim Ausführen von ‚Clean and Build' im Projekt vom Maven-Repository aus dem Internet geladen.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.gundula</groupId>
  <artifactId>Neo4jProcOwnSSSPAlgo</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <packaging>jar</packaging>
  <name>Neo4j Own SSSP Procedures</name>
  <description>
      A project with a procedure to run algo BellmanFord with Neo4j Procedure
  </description>

  <properties>
    <neo4j.version>3.4.1</neo4j.version>
  </properties>

  <dependencies>
    <dependency>
      <!-- This gives us the Procedure API our runtime code uses.
           We have a `provided` scope on it, because when this is
           deployed in a Neo4j Instance, the API will be provided
           by Neo4j. If you add non-Neo4j dependencies to this
           project, their scope should normally be `compile` -->
      <groupId>org.neo4j</groupId>
      <artifactId>neo4j</artifactId>
      <version>${neo4j.version}</version>
      <scope>provided</scope>
    </dependency>
```

Abbildung 19: Auszug Datei pom.xml für eigene Neo4j-Procedure.

In der Datei `pom.xml` wird auch das Release des zu verwendenden Maven-Compiler-Plugins angegeben. In Abbildung 20 und 21 werden die erwähnten Teile der Datei `pom.xml` dargestellt.

```xml
    <dependency>
      <!-- Used to send cypher statements to our
      <groupId>org.neo4j.driver</groupId>
      <artifactId>neo4j-java-driver</artifactId>
      <version>1.6.1</version>
      <scope>test</scope>
    </dependency>
```

Abbildung 20: Auszug Datei pom.xml, Abhängigkeiten Java Driver.

Nachdem die in der Datei `pom.xml` spezifizierten Dependencies aus dem Internet geladen wurden, können die entsprechenden Pakete in die eigene Klasse, die für die Procedure angelegt wurde, importiert und gemäß der Neo4j-API-Beschreibung genutzt werden [23]. Abbildung 22 zeigt den Kopf der erstellten Klasse `AlgoIntBellmanFord` mit dem Import der entsprechenden Pakete von `org.neo4j`. Zusätzlich dazu ist in jedem Fall die Klasse `Stream` aus dem Paket `java.util.stream` zu importieren, da das Ergebnis einer Procedure immer ein Stream ist und dafür ein solcher in der Procedure aufgebaut werden muss.

```xml
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <!-- Neo4j Procedures require Java 8 -->
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
```

Abbildung 21: Auszug Datei pom.xml, Maven-Compiler-Plugin.

```java
package org.gundula.bellmanford;

import java.util.ArrayList;
import java.util.stream.Stream;
import org.gundula.queues.StdQueue;
import org.neo4j.graphdb.*;
import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.Node;
import org.neo4j.procedure.*;
```

Abbildung 22: Auszug Klasse AlgoIntBellmanFord, importierte Pakete.

In der Klasse werden zunächst alle Variablen als statische Variablen deklariert und dann die Klassen, die für die Ausgabe genutzt werden, implementiert. In unserem Fall sind dies die Klassen `Output` und `Output2`. `Output` liefert einen Stream mit den im Algorithmus ermitteln Knoten-Ids und Distanzen, die als Ergebnis der Procedure `bellmanford.algoIntBellmanFordStream`, zurückgegeben werden. `Output2` hingegen liefert nur einen Stream mit der gemessenen Laufzeit des Bellman–Ford-Algorithmus als Ergebnis der Procedure `bellmanford.algoIntBellmanFord` zurück.

In der Klasse `AlgoIntBellmanFord` sind also zwei Procedures implementiert, siehe Abbildungen 23 und 24. Diese ist auch ein gutes Beispiel, wie so eine Neo4j-Procedure grundsätzlich aufgebaut ist.

Während der Durchführung der Experimente haben wir interessante Unterschiede beim Vergleich des Dijkstra-Algorithmus festgestellt, weswegen wir zusätzlich für einen weiteren Vergleich auch den Dijkstra-Algorithmus als eigene Procedure in Neo4j implementiert haben. Dieser wurde ebenfalls mit einer Min-Prioritätswarteschlange implementiert, die auch für die Stand-Alone-Lösung eingesetzt wird. Abbildung 25 gibt einen Überblick über die Pakete und Klassen, die als eigene Procedure in Neo4j implementiert wurden.

```
123     @Procedure(value = "bellmanford.algoIntBellmanFordStream", mode=Mode.READ)
124     @Description("Execute Int Variant of Algorithm Bellman-Ford and return "
125             + "result stream")
126
127     public Stream<Output> algoIntBellmanFordStream( @Name("nodeId") long nodeId,
128                                                     @Name("amntNodes") long amntNodes) {
129
130         ArrayList<Output> out = new ArrayList<>();
131         initSP(nodeId, amntNodes);
132
133         calculateSP();
134
135         // Aufbau der Ausgabe mit den Inhalten von minDist[]
136         for (int i=0; i < minDist.length; i++) {
137             out.add(new Output(i,minDist[i]));
138         }
139         // out.add(new Output(tNodeId, tWeight));
140         Output[] outData = new Output[out.size()];
141         out.toArray(outData);
142         Stream<Output> outStream = Stream.of(outData);
143         return outStream;
144     }
```

Abbildung 23: Neo4j-Procedure bellmanford.algoIntBellmanFordStream.

In jeder hier aufgeführten Procedure sind die Methoden `initSP()` und `calculateSP()` implementiert. In der Methode `initSP()` werden die Datenstrukturen initialisiert. Methode `calculateSP()` berechnet die kürzesten Pfade, gemäß Algorithmus, siehe Kapitel 4.2. Die Implementierung entspricht bis auf die Nutzung der Neo4j-Graph-Schnittstelle der Implementierung in der Stand-Alone-Lösung.

Neben der Klasse `AlgoIntBellmanFord` haben wir noch zwei weitere Varianten des Bellman–Ford-Algorthmus implementiert. Die Klasse `AlgoBellmanFord` ist für die Verarbeitung von Kantengewichten im Gleitkommaformat ausgelegt. Sie ist aber nicht für die Experimente eingesetzt worden.

Die Klasse `AlgoIntBellmanFord2` wurde als Alternative zu der Klasse `AlgoIntBellmanFord` implementiert, um zu testen, ob sich Unterschiede in den Laufzeiten ergeben, wenn nicht nur die Id des Knotens in der Prioritätswarteschlange gespeichert werden, sondern der gesamte Knoten mit Referenzen zu den Kanten. Nach einigen Tests zeigte sich jedoch, dass es keine Unterschiede in der Laufzeit gibt, sodass auch diese Klasse nach dem Test nicht mehr für die Experimente eingesetzt wurde.

```
150     public Stream<Output2> algoIntBellmanFord( @Name("nodeId") long nodeId,
151                                                @Name("amntNodes") long amntNodes) {
152         // Beginn der Zeitmessung
153         long[] runTime;
154         runTime = new long[2];
155         runTime[0] = System.nanoTime();
156
157         // Initialisierung
158         Output2 outData;
159         Stream<Output2> outStream;
160         initSP(nodeId, amntNodes);
161
162         // Berechnung
163         calculateSP();
164
165         // Ende der Zeitmessung
166         runTime[1] = System.nanoTime();
167
168         // Aufbau der Ausgabe mit der Laufzeit
169         outData = new Output2((runTime[1] - runTime[0])/1000000);
170         outStream = Stream.of(outData);
171
172         return outStream;
173     }
174
175 }
```

Abbildung 24: Neo4j-Procedure bellmanford.algoIntBellmanFord.

Abbildung 25: Neo4j-Procedure, Übersicht Pakete und Klassen.

Sobald eine Neo4j-Procedure fertiggestellt ist, wird ein Build in dem Maven-Projekt erzeugt. Die dabei erzeugte Jar-File wird in das Verzeichnis `NEO4J_HOME/plugins` gestellt. Das Plugin wird aktiv, sobald der Neo4j-DB-Service neu gestartet wurde.

5.5 Vorbereitung und Durchführung der Messung mit Neo4j

Beschreibung des Funktionsumfangs

In diesem Abschnitt beschreiben wir die Programme und Tools, die implementiert wurden, um den Import der Eingabegraphen in die Neo4j-Graph-DB, die Messung der Laufzeiten der kürzesten-Pfadsuche mit Neo4j durchzuführen und diese mit den Messergebnissen der Stand-Alone-Lösung zusammenzuführen.

Als Erstes wird das Programm `PrepNeo4jGraphs` im Verzeichnis gestartet, in dem sich die Datei `PrepNeo4jGraphs.jar` befindet, siehe Listing 4. Mit `EXPERIMENT_HOME` ist der Pfad zum Hauptverzeichnis des Experiments gemeint, der auch für die Messung der Stand-Alone-Lösung verwendet wurde. Der in Klammern angegebene Parameter `-S` ist optional (Erklärung folgt).

Listing 4: Start des Programms PrepNeo4jGraphs.

```
> java -jar PrepNeo4jGraphs.jar
"EXPERIMENT_HOME/config" [-S]
```

PrepNeo4jGraphs führt folgende Aktionen aus:

- Lesen der Konfigurationsdateien in den Verzeichnissen `config` und `config_template` unterhalb des Verzeichnisses `EXPERIMENT_HOME`.
- Je Konfigurationsdatei im Verzeichnis `config_template`, die es ja nur einmal pro Graphdatei gibt, wird die darin angegebene Graphdatei aus dem Verzeichnis `input` gelesen und in ein Format transferiert, das für den Import in die Neo4j-Graph-DB benötigt wird, siehe Tabelle 12. Für jede Graphdatei werden damit vier Dateien erstellt, die in das Verzeichnis `input` geschrieben werden. Der Import in die Neo4j-Graph-DB erfolgt dann wie von Neo4j Inc. empfohlenen [23]. Dieser wird mit einem Import-Skript im Rahmen des Messlaufs durchgeführt, siehe Listing 5. Beim Import ist zu beachten, dass dieser mit dem passenden Encoding erfolgt, hier UTF-8.
- Lesen der Konfigurationsdateien im Verzeichnis `config` und Erstellen der Skripte, mit denen der Import und der Messlauf mit der Neo4j-Graph-DB im Anschluss durchgeführt wird. Es handelt sich dabei um Skripte, die auf Betriebssystemebene je nach Environment entweder als Shell-Skript oder als Batch-Datei angelegt werden. Diese enthalten unter anderem auch Befehle zum Stoppen und Starten des Neo4j-DB-Services, für die Durchführung des Imports in die Neo4j-Graph-DB und das anschließende Absetzen der vorbereiteten Queries in der Neo4j Cypher-Shell. Alle Skripte werden in einem Sammelskript namens `mergedrun.sh` bzw. `mergedrun.bat` zusammemgefasst, damit der Import und Messlauf mit der Neo4j-Graph-DB mit diesem Sammelskript gestartet werden kann.

Tabelle 12: Dateien für Import eines Graphen in die Neo4j-Graph-DB.

Name der Datei	Beschreibung
EX_out.com-dblp.txt_1_317081.txt	Originale Graphdatei
EX_out.com-dblp.txt_1_317081.txt_nodes_header.csv	Datei mit Überschrift der Knoten des zu importierenden Graphen
EX_out.com-dblp.txt_1_317081.txt_nodes.csv	Datei mit den Knoten des zu importierenden Graphen
EX_out.com-dblp.txt_1_317081.txt_rel_header.csv	Datei mit Überschrift der Kanten des zu importierenden Graphen
EX_out.com-dblp.txt_1_317081.txt_rel.csv	Datei mit den Kanten des zu importierenden Graphen

Listing 5: Beispiel eines Skripts für den Import in die Neo4j-Graph-DB.

```
#!/bin/bash
"$NEO4J_HOME"/bin/neo4j stop
rm -rf "$NEO4J_HOME"/data/databases
rm -f "$NEO4J_HOME"/logs/*.log
GDAT="ER_1_50000_0.0001gew1bis10_dir.txt"
cp "$EXPERIMENT_HOME"/input/"$GDAT"*.csv
 "$NEO4J_HOME"/import
PARAM1="$NEO4J_HOME"/import/"$GDAT"_nodes_header.csv,
 "$NEO4J_HOME"/import/"$GDAT"_nodes.csv
PARAM2="$NEO4J_HOME"/import/"$GDAT"_rel_header.csv,
 "$NEO4J_HOME"/import/"$GDAT"_rel.csv
"$NEO4J_HOME"/bin/neo4j-admin import --nodes:Knoten
 "$PARAM1" --relationships:KANTE "$PARAM2"
 --id-type=ACTUAL --input-encoding="UTF-8"
"$NEO4J_HOME"/bin/neo4j start
sleep 10
```

Wird `PrepNeo4jGraphs` mit dem Parameter `-S` gestartet, so werden nur die oben erwähnten Skripte erzeugt. Dieser Parameter wird also eingesetzt, wenn die Skripte neu generiert werden müssen, aber die Graphen bereits im Neo4j-Import-Format vorliegen.

Direkt im Anschluss kann der Messlauf mit der Neo4j-Graph-DB gestartet werden. Je nach Betriebssystem wird der Start des Messlaufs etwas anders durchgeführt, siehe Listing 6 und 7. Dabei ist anstelle `EXPERIMENT_HOME` der Name des Hauptverzeichnisses des Experiments anzugeben, das bereits für die Messung der Stand-Alone-Lösung verwendet wurde.

Betriebssystem Windows:

Listing 6: Start des Imports und der Messung mit der Neo4j-Graph-DB.

```
> cd EXPERIMENT_HOME/scriptswin
> call setrundir.bat
> call mergedrun.bat
```

Betriebssystem MAC OS X:

Listing 7: Start des Imports und der Messung mit der Neo4j-Graph-DB.

```
> chmod -R 755 EXPERIMENT_HOME/scriptsmac
> cd EXPERIMENT_HOME/scriptsmac
> ./mergedrun.sh
```

Im Windows-Environment kann es vorkommen, dass es zu einem Fehler kommt, wenn der Neo4j-DB-Service beim Aufruf von `mergedrun.bat` nicht gestartet ist. In diesem Fall starten wir zunächst den Neo4j-DB-Service und führen das Skript `mergedrun.bat` erneut aus.

Während des so gestarteten Messlaufs werden einige Meldungen auf der Knosole ausgegeben. So sehen wir u.a., dass der Neo4j-DB-Service gestoppt und gestartet wird oder dass gerade ein Graph in die Datenbank importiert wird. Während der Ausführung der Queries wird jedoch nichts auf der Konsole angezeigt, da die Ausgaben der Neo4j Cypher-Shell in eine Datei umgeleitet werden. Wenn wir trotzdem prüfen wollen, ob der Messlauf Fortschritte macht, können wir uns die Dateien mit den Messdaten im aktuellen Verzeichnis des Messlaufs anzeigen lassen.

Das aktuelle Verzeichnis ist in der Umgebungsvariablen `TESTRUNDIR` gespeichert. Die Verzweigung in das Verzeichnis erfolgt je nach Betriebssystem mit dem Kommando `cd $TESTRUNDIR` bzw. `cd %TESTRUNDIR%`. Dort werden die Dateien des Messlaufs mit der Neo4j-Graph-DB mit Namen `Neo4j_xxx` und `Neo4jb_xxx` hineingeschrieben. Dabei steht `xxx` für den Namen der zugehörigen Konfigurationsdatei.

Die Pakete und Klassen von `PrepNeo4jGraphs` werden im Anhang B.3 beschrieben.

Zusammenführen der Messergebnisse mit Programm PrepResults

Nach der erfolgten Messung der Laufzeiten mit Neo4j werden die Messergebnisse der Stand-Alone-Lösung und die der Neo4j-Graph-DB zu der jeweiligen Konfiguration in einer Datei im CSV-Format zusammengeführt. Die dabei erstellte Datei hat den gleichen Dateinamen wie bei der Messung der Stand-Alone-Lösung, allerdings mit der Endung `_Erg.CSV`.

Die Datenfelder sind durch ein Komma getrennt und mit einer Überschrift in der ersten Zeile versehen. Dieses Dateiformat kann einfach mit verschiedenen Tools wie Tabellenkalkulationsprogrammen oder auch mit statistischen Programmen eingelesen und ausgewertet werden. Tabelle 13 zeigt ein Beispiel der Darstellung der Daten nach einem Import in ein Tabellenkalkulationsprogramm.

Tabelle 13: Beispiel einer zusammengeführte Messdatei.

Datum	Uhrzeit	Lauf	Graph-typ	Ge-rich-tet	Algo-rith-mus	Queue-typ	Anz. Kno-ten	Anz. Kan-ten	Lauf-zeit ms	Neo4j Lauf-zeit ms
20180904	01:09:09	0	0	false	4	2	317081	1049866	144.1	1045
20180904	01:09:09	1	0	false	4	2	317081	1049866	143.3	994
20180904	01:09:09	2	0	false	4	2	317081	1049866	144.6	992
20180904	01:09:09	3	0	false	4	2	317081	1049866	143.2	991
20180904	01:09:09	4	0	false	4	2	317081	1049866	143.0	988
20180904	01:09:09	5	0	false	4	2	317081	1049866	142.7	991
20180904	01:09:10	6	0	false	4	2	317081	1049866	148.2	991
20180904	01:09:10	7	0	false	4	2	317081	1049866	144.0	988
20180904	01:09:10	8	0	false	4	2	317081	1049866	143.4	995
20180904	01:09:10	9	0	false	4	2	317081	1049866	142.9	981

PrepResults wird wie folgt gestartet, siehe Listing 8.

Listing 8: Start des Programms PrepResults.

```
> java -jar PrepResults.jar "TESTRUNDIR"
```

Falls das Programm erneut ausgeführt wird, zum Beispiel nach dem Beheben eines Fehlers, so müssen vorher alle Dateien mit Endung `_Erg.CSV` in dem Verzeichnis mit den zu verarbeitenden Messergebnissen gelöscht werden. In unserem Fall ist das das Verzeichnis, auf das die Umgebungsvariable TESTRUNDIR zeigt.

Die Pakete und Klassen von `PrepResults` werden im Anhang B.4 beschrieben.

5.6 Auswertungen mit der Programmiersprache R

Die mit dem Programm `PrepResults` zusammengeführten Messdaten im Verzeichnis `EXPERIMENT_HOME/measure/runxxx` werden mittels Programmiersprache `R`, einer freien Programmiersprache für statistische Berechnungen und Grafiken, siehe Kapitel 5.1, ausgewertet.

Für die Auswertung wird erwartet, dass die Dateien für die Auswertung im Verzeichnis `results` unterhalb des Verzeichnisses `EXPERIMENT_HOME` bereitgestellt werden. Dies sind die Dateien der zusammengeführten Messergebnisse mit Endung `_Erg.csv`. Der Aufbau einer Messdatei wird in Tabelle 13 auf Seite 69 dargestellt. Falls das Verzeichnis `results` noch nicht existiert, so muss es vorher angelegt werden. Die für die Auswertung vorbereiteten R-Skripte liegen in den farbig markierten Verzeichnissen `CommonRFunc` und `RAuswertung`, siehe Abbildung der Verzeichnisstruktur für ein Messexperiment 26.

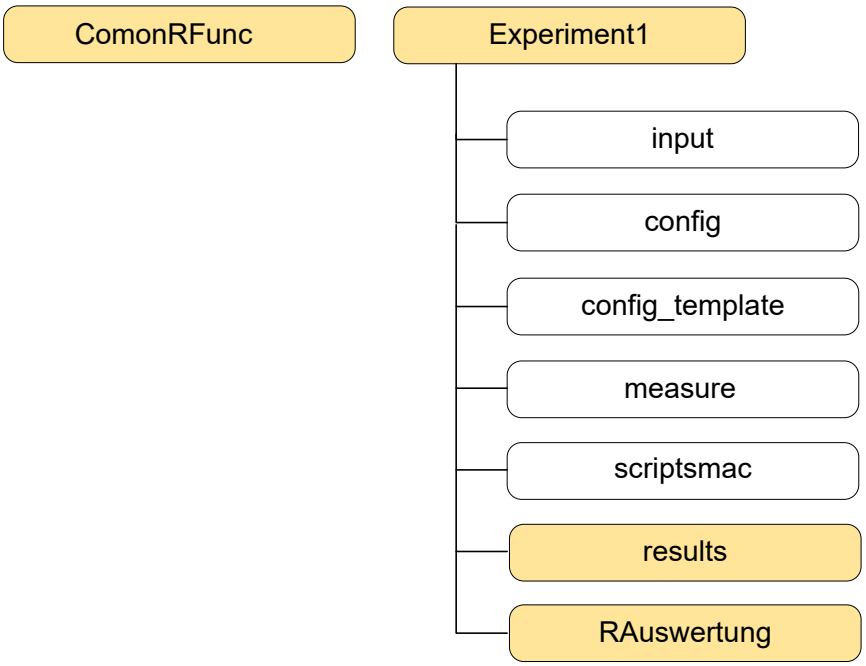


Abbildung 26: Beispiel: Verzeichnisstruktur eines Experiments.

Im Verzeichnis `CommonRFunc`, das sich im selben Verzeichnis wie `EXPERIMENT_HOME` befindet, werden die in den R-Skripten gemeinsam genutzten Funktionen gespeichert. In dem R-Skript namens `ExpFunctions.r` haben wir diese Funktionen zusammengefasst.

Die darin enthaltenen Funktionen lassen sich in vier Kategorien aufteilen:

- Funktionen, die Messdateien lesen und die berechneten Werte pro Messdatei in Form einer Matrix an den Aufrufer zurückgeben. Zu diesen Funktionen gehören `calcmedian`, `calcmean`, `calcmean1` und `readalldata`.

- Funktionen, die obige Matrix als Eingabe erhalten und daraus den Durchschnitt oder Median pro Anzahl Knoten oder nach anderen Kriterien aggregiert berechnen. Zu diesen gehören die Funktionen `calcmediannodes`, `calcmeannodes`, `calcmeannodesedges` und `calcmeanedgedensity`. Alle diese Funktionen erhalten als weiteren Parameter einen Dateinamen der Datei, in der die Tabelle mit den berechneten Daten gespeichert werden soll. Auch hier wird eine Matrix mit den berechneten Daten an den Aufrufer zurückgegeben.

- Hilfsfunktionen, wie `calcdecval`, `calclogscale` und `calctickincrem`. Die Funktion `calcdecval` berechnet die Stelligkeit einer Zahl, `calclogscale` berechnet eine Skala für die Log-basierte Achse und `calctickincrem` berechnet ein Inkrementwert für die Skala der Y-Achse.

- Funktionen, die Plots erstellen, wie zum Beispiel Boxplots oder Barplots oder einfach Plots mit Messdaten und Linien. Dazu gehören die Funktionen mit Namen `myboxplot`, `twoboxplot`, `mybarplot` und `plotmypic`. Neben den üblichen Parametern und Werten für einen Plot wird auch der Name der Pdf-Datei als Parameter mitgegeben, in der der Plot dann gespeichert werden soll.

Je Experiment gibt es ein Auswertungsskript, das im Verzeichnis `RAuswertung` unterhalb von `EXPERIMENT_HOME` gespeichert ist. Nach Ausführung des Skripts werden die Auswertungen des Messexperiments (Tabellen und Plots) im selben Verzeichnis gespeichert. Wollen wir ein neu hinzugekommenes Messexperiment auswerten, so können wir uns das Auswertungsskript eines vorhandenen Messexperiments kopieren und als Ausgangsbasis dafür verwenden.

Das Auswertungsskript ist im Regelfall wie folgt aufgebaut:

- Setzen einer Pfad-Variable auf das aktuelle Arbeitsverzeichnis

- Einbinden der gemeinsamen Funktionen

- Einlesen der Dateinamen der Messdateien getrennt nach Algorithmus

- Setzen von Experiment-spezifischen Variablen, wie Untertitel für Plots mit Angaben zum Graphen oder Angaben zum Environment, indem das Experiment durchgeführt wurde.

- Aufruf der Funktionen zur Auswertung und Erstellung der Plots je Algorithmus

Das Auswertungsskript wird dann mittels R-Console gestartet. Dabei ist zu beachten, dass nach dem Aufruf der R-Console zunächst in das passende Arbeitsverzeichnis mit `setwd()` gewechselt werden muss. Hier geben wir den absoluten Pfad ein, in dem sich das Auswertungsskript befindet, siehe Listing 9 mit einem Beispiel.

Listing 9: Start der Auswertung eines Experiments in der R-Console.

```
> setwd("C:/ExperimentWinPC6/RAuswertung")
> source("Auswert1ExpWinPC6.r")
```

Die für die Experimente dieser Arbeit zugrundeliegende R-Auswertung wird in dem beigefügten USB-Stick zur Verfügung gestellt.

6 Hypothesen

Nach der Erarbeitung wichtiger Grundlagen zu Graph-DBS, siehe Kapitel 3, und der Betrachtung des kürzesten-Pfad-Problems in Kapitel 4 werden wir den Vergleich der Performanz bei kürzesten-Pfad-Berechnungen zwischen einem Graph-DBS, anhand Neo4j, und einer implementierten Stand-Alone-Lösung mit den folgenden Hypothesen in drei Experimenten überprüfen. Dabei sind jedem Experiment zu untersuchende Implementierungen eines Algorithmus zugeordnet:

6.1 Hypothesen zu Experiment 1: Vergleich anhand bidirektionaler Breitensuche

Hypothese 1.1
Bei großen Graphen, ab einer gewissen Anzahl Knoten oder Kanten des Graphen, ist die Laufzeit mit der Neo4j-Graph-DB durch vermutete Partitionierung und Parallelisierung bei der bidirektionalen Breitensuche geringer als die der implementierten Stand-Alone-Lösung.
Hypothese 1.2
Bei kleineren Graphen fällt der Overhead der Graph-Datenbank mehr in das Gewicht, d.h. die Berechnung mit der Stand-Alone-Lösung ist performanter.

6.2 Hypothesen zu Experiment 2: Vergleich anhand Dijkstra-Algorithmus

Hypothese 2.1
Beim Dijkstra-Algorithmus ist das Ergebnis des Vergleichs der Performanz zwischen der Neo4j-Graph-DB und der Stand-Alone-Lösung von der Wahl der Priority-Queue abhängig.
Hypothese 2.2
Bei kleineren Graphen fällt der Overhead der Graph-Datenbank mehr in das Gewicht, d.h. die Berechnung mit der Stand-Alone-Lösung ist performanter.

6.3 Hypothesen zu Experiment 3: Vergleich anhand Bellman–Ford-Algorithmus

Hypothese 3.1
Beim Bellman–Ford-Algorithmus, der sowohl in der Neo4j-Graph-DB als Plugin als auch in der Stand-Alone-Lösung implementiert ist, werden die Laufzeiten in etwa gleich hoch sein.
Hypothese 3.2
Bei kleineren Graphen fällt der Overhead der Graph-Datenbank mehr in das Gewicht, d.h. die Berechnung mit der Stand-Alone-Lösung ist performanter.

7 Experimente zum Vergleich der Performanz

7.1 Allgemeiner Aufbau und Ablauf

Für den Vergleich der Performanz bei kürzesten-Pfadsuchen zwischen dem Neo4j-Graph-DBS und der implementierten Stand-Alone-Lösung haben wir drei Experimente definiert, wobei jedem Experiment zu untersuchende Implementierungen eines Algorithmus zugeordnet sind:

- Experiment 1: Vergleich anhand der bidirektionalen Breitensuche
- Experiment 2: Vergleich anhand des Dijkstra-Algorithmus
- Experiment 3: Vergleich anhand des Bellman–Ford-Algorithmus

Mit diesen Experimenten werden wir die in Kapitel 6 aufgestellten Hypothesen überprüfen. Für die Messungen der Laufzeiten werden wir verschiedene Messexperimente durchführen. In einem Messexperiment können mehrere Algorithmen getestet werden. Die dabei gesammelten Messdaten werden dann bei der Analyse dem zugehörigen Experiment zugeordnet. Zu einem Experiment kann es also mehrere Messexperimente geben ($n : m$ -Beziehung).

Im Folgenden wird der allgemeine Aufbau und Ablauf der Experimente beschrieben, der für alle drei Experimente gleich gewählt wurde. Um möglichst viele Synergien für die einzelnen Experimente zu nutzen, wurden die Messungen der Algorithmen kombiniert durchgeführt. Das hat den Vorteil, dass der einmal in die Neo4j-Graph-Datenbank importierte Eingabegraph für die Ausführung und Messung mehrerer implementierter Algorithmen genutzt wird.

Messwerte und Vergleichbarkeit

Für den Vergleich der Messungen wird die reine Laufzeit der Algorithmen betrachtet, ohne den Einlesevorgang und auch ohne die Ausgabe der gefundenen Pfade. Ergebnis einer Messung ist also immer die reine Laufzeit des implementierten Algorithmus mit dem gegebenen Eingabegraphen und Startknoten. Bei der bidirektionalen Breitensuche wird zusätzlich noch ein Zielknoten als Parameter übergeben.
Eine wichtige Voraussetzung für einen solchen Vergleich ist, dass die Messungen unter gleichen Bedingungen erfolgen und auch, dass die Messpunkte passend gewählt werden. Weiterhin sollen Störungen von außen vermieden werden, die Messwerte verfälschen können. In unserem Fall sollte also während des Messlaufs keine weitere Anwendung gestartet werden, die nicht Bestandteil des Messexperiments ist. Auch Updates sollten während der Messungen nicht durchgeführt werden. Die Messungen müssen mit der selben Software-Version, den selben Eingabedaten, sowie demselben Start- und Zielknoten durchgeführt werden. Die Daten, die wir bei Abfragen in Neo4j erhalten, sollten so gewählt werden, dass man diese mit den gemessenen Werten der Stand-Alone-Lösung vergleichen kann. Daher haben wir die Möglichkeiten der Ausgabe der Laufzeiten mit Neo4j

erörtert. Zuerst dachten wir daran, die Ausführungszeiten aus einer Neo4j-Log-Datei namens query.log herauszufiltern. Nach ein paar Tests stellte sich jedoch heraus, dass die darin angegebenen Ausführungszeiten nicht genauer sind, als die Ausgaben, die wir mit Ausführung der Query mit der Neo4j Cypher-Shell oder dem Neo4j Browser erhalten. Daher werden wir die Laufzeit, wie folgt, mit Neo4j messen:

- **Bidirektionale Breitensuche:** Die Konsole-Ausgabe der Neo4j Cypher-Shell, und zwar der Wert in Millisekunden, der an Stelle xx in der Ausgabe vom Muster ‚1 row available after xx ms consumed after... ' enthalten ist.
- **Dijkstra-Algorithmus:** Die Ausgabe des Wertes für `evalDuration`, das ist eine Ausgabe der Neo4j-Procedure, die die kürzesten Pfade mit dem Dijkstra-Algorithmus berechnet. Die Ausgabe erfolgt ebenfalls auf der Konsole, da wir die Query mit der Cypher-Shell ausführen.
- **Bellman–Ford-Algorithmus:** Der Wert in `evalDuration`, der als Ergebnis der Ausführung der eigenen Neo4j-Procedure ausgegeben wird.

Die oben aufgeführten, gemessenen Laufzeiten mit Neo4j werden während des Messexperiments von der Konsole in eine Datei umgeleitet. Beim späteren Zusammenführen der Messdaten werden die Messwerte aus diesen Dateien entsprechend herausgefiltert. In der Stand-Alone-Lösung wird die Laufzeit mit der Ausführung des implemtierten Algorithmus gemessen. Hier können wir die Art der Messung selbst einstellen, so, dass sie zu den Messungen mit Neo4j passt. Die gemessene Laufzeit wird dann direkt in Messdateien ausgegeben.

Festlegungen zum Messumfang

Als Eingabegraphen werden sowohl mit dem Graphgenerator synthetisch erzeugte Graphen als auch sogenannte Real-World-Graphen aus dem Internet geladen und für den Import in Neo4j vorbereitet, siehe Beschreibung in Kapitel 7.3. Die auf diese Weise bereitgestellten Eingabegraphen sind für die Ausführung verschiedener Algorithmen geeignet. Damit können Messexperimente direkt nacheinander mit einem bereits in Neo4 importierten Graphen durchgeführt werden. Um sicher zu gehen, dass in den Eingabegraphen keine negativen Zyklen enthalten sind, werden wir nur ungewichtete Graphen oder Graphen mit ganzzahligen, positiven Gewichten für die Experimente einsetzen.
Handelt es sich bei dem importierten Graphen um einen gewichteten Graphen, so wird das Gewicht bei der Ausführung der bidirektionalen Breitensuche ignoriert. Umgekehrt wird einem Graphen ein Kantengewicht von Eins zugeordnet, wenn dieser nicht gewichtet ist und der kürzeste Pfad mit dem Dijkstra- oder dem Bellman–Ford-Algorithmus berechnet werden soll.

Der Startknoten des Eingabegraphen wird im Rahmen der Vorbereitung der Experimente durch den Graphgenerator zufällig bestimmt und zwar so, dass sichergestellt ist, dass ein Pfad zu einem anderen Knoten im Graphen existiert.

Obwohl in der Stand-Alone-Lösung noch weitere Varianten der Algorithmen implementiert sind, haben wir die Algorithmus-Varianten gewählt, die mit ungewichteten Graphen oder mit Graphen mit ganzzahligen Gewichten arbeiten, das sind die in der Tabelle 11 mit Nr. 0, 1, 2 und 4 aufgeführten Implementierungen. Die Auswahl der implementierten Algorithmen in Kombination mit den ausgewählten Eingabegraphen reicht uns aus, um die für die Experimente aufgestellten Hypothesen zu überprüfen.
Für die Durchführung der Experimente mit Neo4j werden die in Neo4j standardmäßig implementierte bidirektionale Breitensuche, der Dijkstra-Algorithmus aus dem Neo4j-Standard-Plugin für effiziente Graph-Algorithmen und zwei in Eigenentwicklung als Neo4j-Plugin implementierte Procedures mit dem Dijkstra-Algorithmus und dem Bellman–Ford-Algorithmus in den Messlauf eingebunden.

Steuerung der Messungen durch Konfigurationsdateien

Zur Steuerung der Durchführung der Messungen zu dem jeweiligen Experiment werden Konfigurationsdateien verwendet. Diese enthalten u.a. den Namen der Datei des Eingabegraphen, die Anzahl der Knoten des Graphen, Informationen darüber, ob der Graph gerichtet oder gewichtet ist, Angabe des Startknotens, Angabe des durchzuführenden Algorithmus und der Anzahl der Messläufe, die durchgeführt werden sollen. Um die Erstellung dieser Konfigurationsdateien zu vereinfachen, werden diese automatisch durch den Graphgenerator erstellt. Sie können jedoch im Nachhinein angepasst oder im Ausnahmefall manuell erstellt werden. Der Aufbau einer Konfigurationsdatei ist in Tabelle 8 beschrieben. Die Namen der für die Experimente verwendeten Dateien sind derart gestaltet, dass man erkennen kann, welcher Graph mit welcher Konfigurationsdatei zusammenpasst und auch, welche Messdaten dazu gehören. Dabei wurde auf absolute Pfad-Angaben in den Dateien verzichtet, damit Eingabegraphen und vorbereitete Messexperimente auch auf anderen Systemumgebungen genutzt werden können.

Import von Eingabegraphen in die Neo4j-Graph-DB

Für jeden in den Messexperimenten verwendeten Eingabegraphen wird ein Bulk-Import der Daten in Neo4j durchgeführt. Dazu werden die in Textdateien vorliegenden Eingabegraphen mit dem Programm PrepNeo4jGraphs.jar in ein Format überführt, das in Neo4j importiert werden kann, siehe Kapitel 5.5. Entsprechende Skripte für den Import werden dabei automatisch erstellt. Diese sorgen u.a. auch dafür, dass die Daten kodiert im UTF-8-Format importiert werden. Die Wahl der Kodierung ist wichtig, insbesondere auch bei Messungen in Systemen mit unterschiedlichen Betriebssystemen. Wird die Kodierung nicht berücksichtigt, so werden Knotennummern bei größeren Graphen in Neo4j beim Import nicht mehr korrekt zugeordnet, was dann zu Fehlern bei der Ausführung der Neo4j-Procedures führt.

Der Neo4j-Bulk-Import setzt allerdings voraus, dass die Graph-Datenbank in Neo4j noch nicht existiert. Diese wird dann mit dem Import des Graphen neu angelegt. Das hat den Vorteil, dass die Messungen nicht auf bereits vorhandenen Statistiken oder in den Cache geladenen Daten aufsetzen und damit wiederum gleichartige Bedingungen für die Messungen der Laufzeiten vorliegen. Ein Nachteil dabei ist allerdings auch, dass für die Durchführung der Experimente eine bereits existierende Graphdatenbank in Neo4j zunächst heruntergefahren und dann gelöscht werden muss. Nach dem Import des Eingabegraphen muss die Graph-Datenbank gestartet werden, was insgesamt eine gewisse Zeit in Anspruch nimmt. Erst dann können die Messungen der Laufzeiten gestartet werden. So kann es passieren, dass die Durchführung eines Experiments je nach Anzahl der Eingabegraphen durchaus mehrere Stunden oder gar Tage dauern kann.

Information zu Messdateien

Die Messungen der Laufzeiten der Stand-Alone-Lösung werden während des Messlaufs als Textdateien im CSV-Format in ein neu erzeugtes Unterverzeichnis namens `runxxx`, wobei xxx für das Erstellungsdatum steht, geschrieben. In dem danach gestarteten Messlauf mit Neo4j werden die Ausgaben der Laufzeiten aus der Konsole in eine Datei umgeleitet und als Textdateien im selben Unterverzeichnis gespeichert. Die aus den Messläufen gesammelten Daten werden dann mit einem Programm (PrepResults.jar) jeweils in einer Datei im CSV-Format zusammengeführt. Die weitere Auswertung der Daten erfolgt dann mittels `R`, einer Programmiersprache für statistische Auswertungen, siehe Kapitel 5.6. Die dafür vorbereiteten R-Skripte können bei neuen Messläufen wiederverwendet oder um neue Auswertungen ergänzt werden.

Ablauf eines Messexperiments

Der Ablauf eines Messexperiments wird in Abb. 27 dargestellt.

Der erste Schritt nach der Vorbereitung der Environments, siehe Kapitel 7.2, ist das Anlegen der Verzeichnisstruktur für ein Messexperiment. Welche Verzeichnisse benötigt werden und wie die Programme oder die Skripte gestartet werden, erklären wir im Kapitel 5. Der nächste Schritt ist die für das Experiment gewählten Graphtypen und Graphgrößen zu generieren oder auch Graphen aus Datasets im Internet herunterzuladen. Nachdem die Eingabegraphen für die Messexperimente in Form von Textdateien vorliegen, werden diese für den Import in Neo4j vorbereitet. Dabei werden auch Skripte für die automatisierte Messung von Laufzeiten mit Neo4j erstellt.

Nach den Vorbereitungsarbeiten beginnt die Messphase. Zunächst starten wir die Stand-Alone-Lösung, mit der die Messungen der implementierten Algorithmen durchgeführt werden. Im Anschluss daran starten wir die Skripte und damit das Messen der Laufzeiten in Neo4j. Die gesammelten Messdateien werden anschließend in CSV-Dateien zusammengeführt. Zum Schluss werden die Messda-

ten mittels R-Skripten ausgewertet. Die Auswertungen und Ergebnisse werden in den Folgekapiteln, ab Kapitel 7.4, zu jedem Experiment beschrieben.

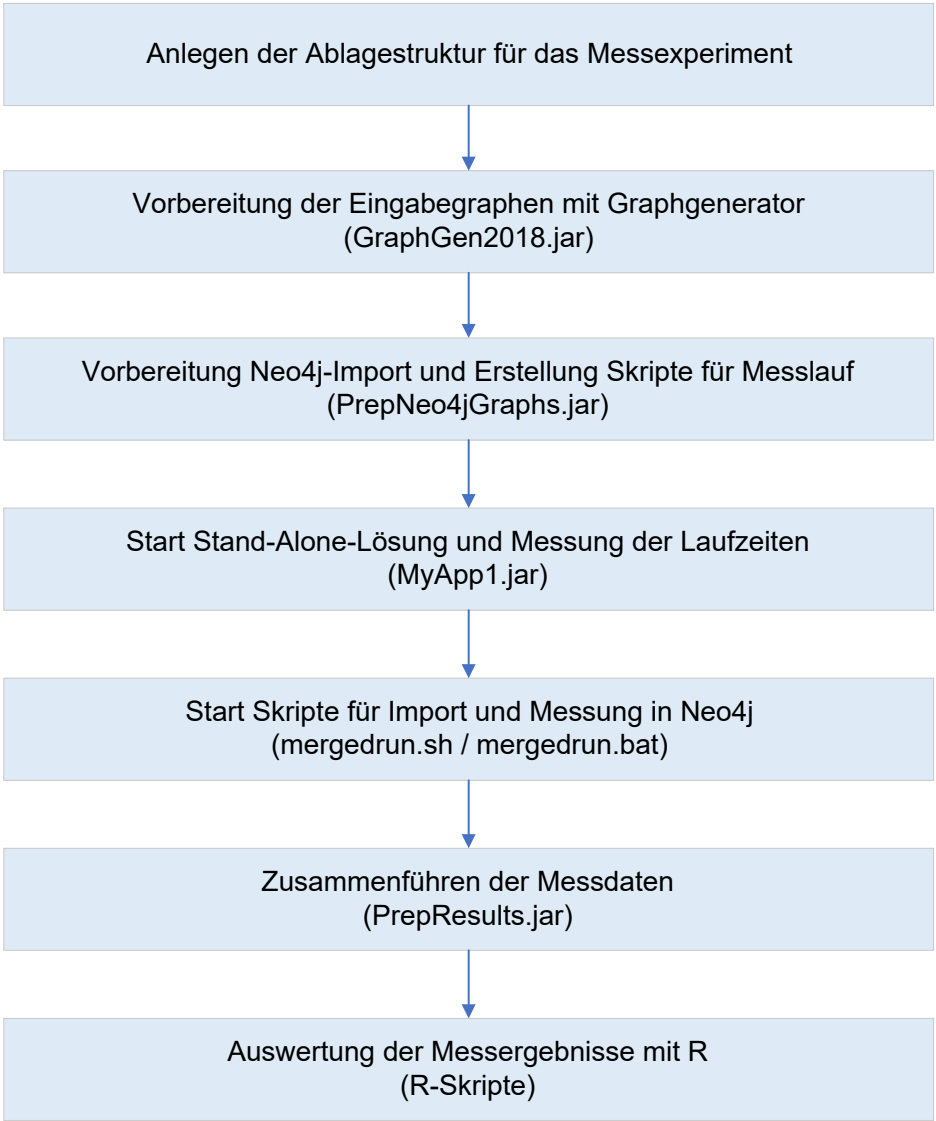


Abbildung 27: Schritte für Vorbereitung u. Durchführung eines Experiments.

7.2 Verwendete Systemumgebungen

Die Experimente werden in verschiedenen Environments mit unterschiedlicher Ausstattung durchgeführt, siehe Angaben in Tabelle 14.

Tabelle 14: Übersicht Environments für die Experimente.

Komponente	Env A	Env B	Enc C
Betriebssystem	Mac OS Yosemite (10.10.5), 64 bit	Windows 10 Enterprise, 64 bit	Windows Server 2008 Datacenter, R2 SP1, 64 bit
Hauptspeicher	16 GB DDR3, 1600 MHz	32 GB DDR4, 2133 MHz	128 GB DDR3, 1600 MHz
Prozessor	Intel Core i7 , 3840 QM 2.8 GHz, 4 CPU Kerne	Intel Core i7 5820K 3.3 GHz, 6 CPU Kerne	2-mal Intel Xeon E5-2690 2.9 GHz, je 8 CPU Kerne
Plattenspeicher	768 GB SSD	500 GB SSD, 4 TB Festplatte SATA III	2-mal 1 TB SSD, 3 TB Festplatten SAS-2, RAID 5

Auf jedem dieser Environments haben wir Neo4j Desktop mit der darin enthaltenen Enterprise Version 3.4.1 installiert. Diese Software wird auf der Internetseite von Neo4j, Inc. zum Download angeboten und ist für Entwickler kostenlos nutzbar [11]. In Env C mussten wir überdies ein Update des Powershell-Releases durchführen, um die Systemvoraussetzungen von Neo4j zu erfüllen, da für die Administration von Neo4j in Windows-Umgebungen Powershell-Skripte eingesetzt werden. Diese werden auch für die Experimente genutzt, um zum Beispiel den Stopp und den Start des Datenbanksystems oder einen Datenbankimport mittels ‚neo4j admin-import' auszuführen.

Nach der Installation von Neo4j sind in jedem Environment noch folgende Konfigurationen durchzuführen:

- Einmaliges Aufrufen von Neo4j Desktop, Anlegen einer neuen Datenbank namens Graph1 und Vergeben des Kennwortes, hier jily5ever. Dieses Kennwort wird für die Experimente genutzt und ist fest in den Programmen hinterlegt, die die Skripte für den Import und die Messung in Neo4j erstellen

- In den Einstellungen von Neo4j Desktop das Verzeichnis zu den Daten von Neo4j eintragen, zum Beispiel E:/neo4j/data, siehe Beispiel in Abbildung 28

- Einmalig die im Neo4j Desktop angebotenen Plugins installieren, und zwar APOC (allgemeine Neo4j Standard-Procedures) und das Plugin für die effizienten Graph-Algorithmen
- Setzen der Environment-Variablen NEO4J_HOME und NEO4J_CONF, wie im Neo4j Operations Guide beschrieben [23]
- In den Environments mit Windows-Betriebssystem das einmalige Registrieren von Neo4j als Windows-Service, wie im Neo4j Operations Guide beschrieben[23]
- Bestimmen der Neo4j-Heap- und -Cache-Größe und Eintragen der von Neo4j vorgeschlagenen Werte in die Datei neo4j.conf im Konfigurationsverzeichnis NEO4J_CONF 10
- Kopieren der eigenen Neo4j-Procedures in das Verzeichnis plugins, das sich unterhalb von NEO4J_HOME befindet (Datei mit Namen Neo4jProcOwnSSSPAlgo-1.0.0-SNAPSHOT.jar)
- Eintragen der Namen der Erweiterungen in Datei neo4j.conf 11

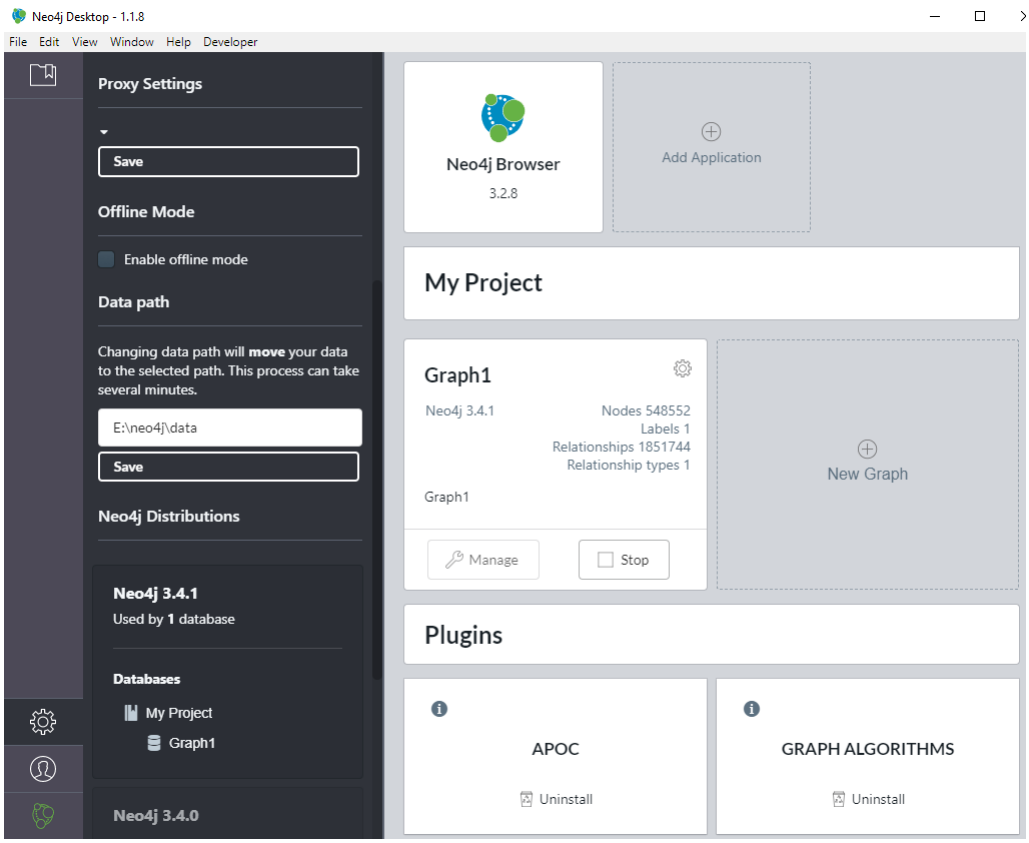


Abbildung 28: Erste Einstellungen im Neo4j Desktop.

Nach Eingabe des folgenden Kommandos in der Konsole erhalten wir die von Neo4j vorgeschlagenen Werte zur Konfiguration von Heap-Größe und Cache-Größe. Diese sind je Environment verschieden und werden dann in der Datei neo4j.conf entsprechend angepasst. Nach einem Neustart von Neo4j sind diese wirksam. Werden diese Werte nicht angepasst, so kann es passieren, dass der Aufruf der Neo4j Procedure, die den kürzesten Pfad mit dem Dijkstra-Algorithmus berechnet, gleich mit einem Fehler (java.lang.OutOfMemoryError) abbricht.

Listing 10: Neo4j Kommandos zur Speicherempfehlung und -belegung.

```
> %NEO4J_HOME%/bin/neo4j-admin memrec
```

Wir erhalten folgende Ausgaben in der Konsole in dem jeweiligen Environment:

Env A:

```
dbms.memory.heap.initial_size=5g
dbms.memory.heap.max_size=5g
dbms.memory.pagecache.size=7g
```

Env B:

```
dbms.memory.heap.initial_size=12200m
dbms.memory.heap.max_size=12200m
dbms.memory.pagecache.size=12200m
```

Env C:

```
dbms.memory.heap.initial_size=31700m
dbms.memory.heap.max_size=31700m
dbms.memory.pagecache.size=82900m
```

Zusätzlich sind mit dem folgenden Eintrag in die Datei neo4j.conf noch die Namen für die Neo4j-Erweiterungen zu erlauben. Dieser Eintrag wurde in jedem Environment durchgeführt:

Listing 11: Neo4j Namen der Erweiterungen eintragen in Datei neo4j.conf.

```
dbms.security.procedures.unrestricted=
    apoc.*,algo.*,bellmanford.*,dijkstra.*
```

7.3 Auswahl der Eingabegraphen

Für die Experimente wurde eine Auswahl von verschieden großen synthetisch erzeugten Zufallsgraphen und aus dem Internet geladenen Real-World-Graphen getroffen. Bei den synthetisch generierten Graphen variiert die Größe der Graphen von 50000 Knoten bis 1 Mio. Knoten. Weiterhin werden sowohl gerichtete als auch ungerichtete Graphen erzeugt, die zudem noch gewichtet sein können.

Generierte Graphen basierend auf dem Erdös–Rényi-Modell

Basierend auf einer ausgewählten Variante des Erdös–Rényi-Modells (kurz ER-Modell) [13] wurden Graphen mit einer vorgegebenen Kantenwahrscheinlichkeit und der Anzahl der gewünschten Knoten generiert. Dabei ist die in der Eingabe gewählte Kantenwahrscheinlichkeit für jede Kante im Graphen gleich. Der Graphgenerator erlaubt, neben der Kantenwahrscheinlichkeit von 0.5, als Abwandlung zum ER-Modell die Eingabe von unterschiedlichen Kantenwahrscheinlichkeiten, sowie die Erzeugung von gerichteten Graphen und zudem die Belegung der Kanten mit Gewichten. Im Folgenden nennen wir die so erzeugten Graphen ER-Graphen oder ER-basierte Graphen.

Generierte Graphen basierend auf dem Barabási–Albert-Modell

Weiterhin wurden mit dem Graphgenerator auch Graphen basierend auf dem Barabási–Albert-Modell (kurz BA-Modell)[4][1] generiert. Dabei wird ausgehend von einem gegebenen Startgraphen, der aus mindestens 2 Knoten besteht, bei jeder Iteration ein neuer Knoten zum Graphen hinzugefügt. Für den neu hinzugekommenen Knoten wird nun mittels einer Kantenwahrscheinlichkeit die proportional zu dem jeweiligen, aktuellen Knotengrad ist, berechnet, ob zwischen dem neu hinzu gekommenen Knoten und einem bereits vorhandenen Knoten eine Kante gebildet wird. Der Vorgang wird solange wiederholt, bis der Graph die vorgegebene Anzahl an Knoten erreicht hat.
Als Variante zum BA-Modell erlaubt der Graphgenerator auch hier die Vorgabe der Anzahl der Startknoten, die Eingabe einer Kantenwahrscheinlichkeit für den Startgraphen und die Auswahl, ob der zu generierende Graph gerichtet sein soll oder die erzeugten Kanten mit einem Gewicht belegt werden sollen. Der Startgraph, auf dem der zu generierende BA-Graph aufbaut, wird durch den Graphgenerator im Rahmen der Initialisierung basierend auf dem ER-Modell automatisch generiert.

Auswahl der aus dem Internet geladenen Real-World-Graphen

Einige der für die Experimente eingesetzten Real-World-Graphen wurden von KONECT (the Koblenz Network Collection) [35] heruntergeladen, einem Projekt, das Daten von vielen unterschiedlichen Netzwerken in Form von Graphen, teilweise mit Zusatzinformationen, gesammelt hat. Unter anderem handelt es sich dabei um Graph-Daten von sozialen Netzwerken, Autoren-Netzwerken, Verlinkungen, Kommunikationsnetzwerken und physischen Netzwerken.
Eine weitere Quelle für die extern geladenen Graphen ist die Webseite der SNAP Group der Stanford University, die im Rahmen des Stanford Network Analysis Project [19] ebenfalls eine Sammlung von Graph-basierten Netzwerken vorhält. Darin enthalten sind u.a. Daten von sozialen Netzwerken, Internet-Netzwerken, Straßennetzwerken, sowie Netzwerken der Zusammenarbeit und der Kommunikation.

Tabelle 15 gibt eine Übersicht der Graphen, die von den angegebenen Quellen heruntergeladen und in den Experimenten eingesetzt wurden.

Tabelle 15: Übersicht der für die Experimente aus dem Internet geladenen Real-World-Graphen.

Name Datei	Anzahl Knoten	Anzahl Kanten	gerichtet	gewichtet	Quellen	Beschreibung
as-skitter.txt	1696416	11095298	nein	nein	[18]	Internet-Topologie-Graph erstellt mit täglicher Routenverfolgung in 2005
com-amazon.ungraph.txt	548552	925872	nein	nein	[40]	Vernetzung von Käufen auf der Amazon-Webseite
out.com-dblp.txt	317081	1049866	nein	nein	[7][41]	DBLP Co-Autoren-Netzwerk
out.opsahl-collaboration.txt	22016	58594	nein	nein	[2][26]	Netzwerk, das Links zwischen Autoren und Veröffentlichungen in arXiv von 1995 bis 1999 enthält.
out.p2p-Gnutella31.txt	62587	147892	ja	nein	[12][28]	Netzwerk Gnutella Hosts, 2002
out.web-Stanford.txt	281904	2312497	ja	nein	[33][20]	Gerichtetes Netzwerk von Hyperlinks zwischen Webseiten der Stanford University
out.wikisigned-k2.txt	138593	740106	ja	ja	[39][21]	Interaktionen von Benutzern auf Wikipedia-Seite

7.4 Experiment 1: Vergleich der bidirektionalen Breitensuche

7.4.1 Aufbau und Durchführung

In diesem Experiment liegt der Fokus auf den ungewichteten Graphen bzw. auf Graphen, deren Gewicht bei der Durchführung des Experiments ignoriert wird. In verschiedenen Messexperimenten werden gemäß des in Kapitel 7.1 beschriebenen Aufbaus die bidirektionale Breitensuche sowohl mit der Graph-DB Neo4j, als auch mit der eigenentwickelten Stand-Alone-Lösung getestet.

Die zu diesem Experiment gehörenden Konfigurationen und Messdateien können über den Namensteil **config1** zugeordnet werden. Speziell für die Tests mit der Neo4j-Graph-DB wird eine Query in der folgenden Form für den Test der bidirektionalen Breitensuche ausgewertet, siehe Listing 12:

Listing 12: Cypher-Query für die Messung der bidirektionalen Breitensuche.

```
MATCH (n:Knoten), (o:Knoten), p=shortestPath((n)-[*]->(o))
 WHERE id(n)=197244 AND id(o)=1 RETURN length(p)
```

Die bei der Query verwendeten Ids der Knoten ändern sich je Query, da der Test mit wechselnden Start- und Endknoten passend zu den Eingabegraphen durchgeführt wird. Die zu verwendenden Start- und Zielknoten wurden im Rahmen der Vorbereitung, wie im Kapitel 7.1 beschrieben, ermittelt.

Im Rahmen der Auswertung der Messergebnisse werden die in Kapitel 6.1 aufgestellten Hypothesen überprüft.

7.4.2 Ergebnisse und Auswertung

Im Folgenden werden die Ergebnisse der verschiedenen Tests zu Experiment 1 ausgewertet und graphisch dargestellt.

Vergleich der Messwerte BA-Graphen, wachsende Knotenzahl

Wir betrachten zunächst Messungen in Environment B (Env B), siehe Tabelle 14. In diesem Environment ist Neo4j auf einer separaten 2 TB Festplatte mit SATA III-Anschluss installiert. Da Betriebssystem selbst und die gesammelten Messdaten befinden sich auf einem 500 GB Solid State Drive (SSD). Bei der Betrachtung der gemessenen Laufzeiten fällt auf, dass die gedachte Messkurve keinen gleichmäßigen Verlauf hat. Durch eine Sichtung der einzelnen Messwerte in den Dateien stellen wir fest, dass wir in einzelnen Fällen erheblich höhere Laufzeiten gemessen haben, trotz Wiederholungsmessung mit dem selben Graphen und Startpunkt. Daher haben wir für den betroffenen Bereich eine Nachmessung

durchgeführt, bei der es interessanterweise nicht mehr zu solchen Abweichungen gekommen ist. Da zu dem ersten Messzeitpunkt die oben angegebene Festplatte fehlerhaft war, und diese später aus diesem Grund ausgetauscht werden musste, liegt die Vermutung nahe, dass diese Messabweichungen dadurch verursacht worden sind. In Abb. 29 sehen wir den Unterschied der Messungen vor und nach der Korrektur. Nach der Nachmessung erhalten wir einen wesentlich glatteren Verlauf der gedachten Messkurve.

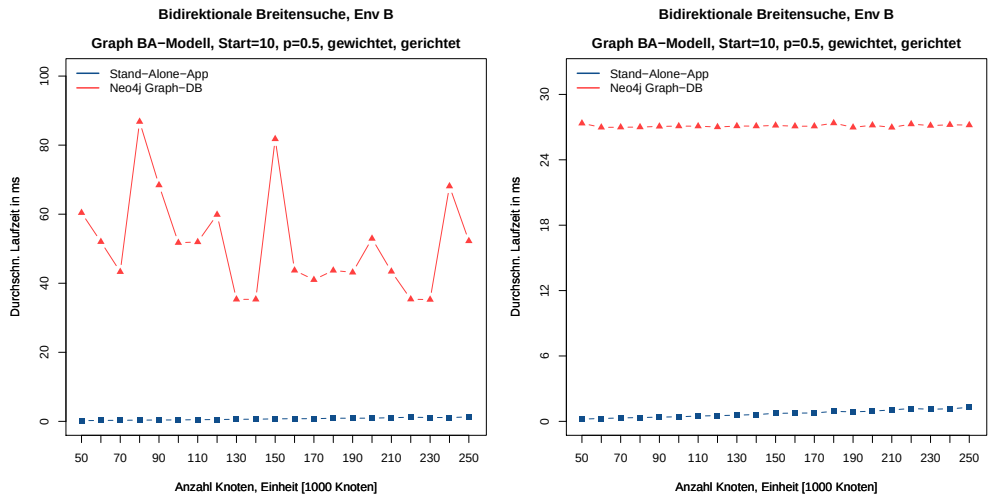


Abbildung 29: Bidirektionale Breitensuche, links: Plot mit Messabweichungen, rechts: Plot mit bereinigten Messdaten.

Dieses Beispiel zeigt, dass es es bei solchen Experimenten wichtig ist, die Messdaten zu überprüfen, um eventuelle Fehler zu korrigieren, da sonst Beobachtungen verfälscht, Hinweise übersehen oder gar falsche Schlüsse aus den Messungen gezogen werden können. Mit ein paar Prüfungen, die Ausreißer erkennen, können wir solche Messfehler minimieren. Neben der Nutzung von statistischen Funktionen, ist es allerdings auch empfehlenswert, mit ein paar Vorabtests zu ermitteln, was denn als Normalwert erwartet wird. Speziell bei Messungen der Laufzeiten der bidirektionalen Breitensuche mit Neo4j würde sonst immer die erste Messung nach dem Import eines neuen Graphen als Ausreißer zählen, da diese im Vergleich zu den übrigen Messungen erheblich höher ist. In den zur Auswertung der Messdaten eingesetzten R-Skripten werden daher Prüfungen durchgeführt, die Abweichungen von Messwerten, die über einem definierten Schwellwert liegen, erkennen. Diese werden als Information bei der Auswertung auf der R-Console ausgegeben und sollten überprüft werden.

Wie bereits erwähnt, fällt bei der Betrachtung der Messwerte der bidirektionalen Breitensuche mit Neo4j auf, dass die Messwerte beim ersten Messlauf immer deutlich höher sind. Ab der zweiten Ausführung sind die gemessenen Laufzeiten dafür sehr niedrig, nämlich fast immer 0 oder 1 Millisekunde. Nur vereinzelt nach einigen Durchläufen kommt es vor, dass die Laufzeit auch einmalig etwa

16 Millisekunden beträgt und dann beim nächsten Durchlauf wieder bei 0 oder 1 ms liegt. Abbildung 30 stellt die gemessenen Laufzeiten der Lösungen ohne den ersten Messwert dar.

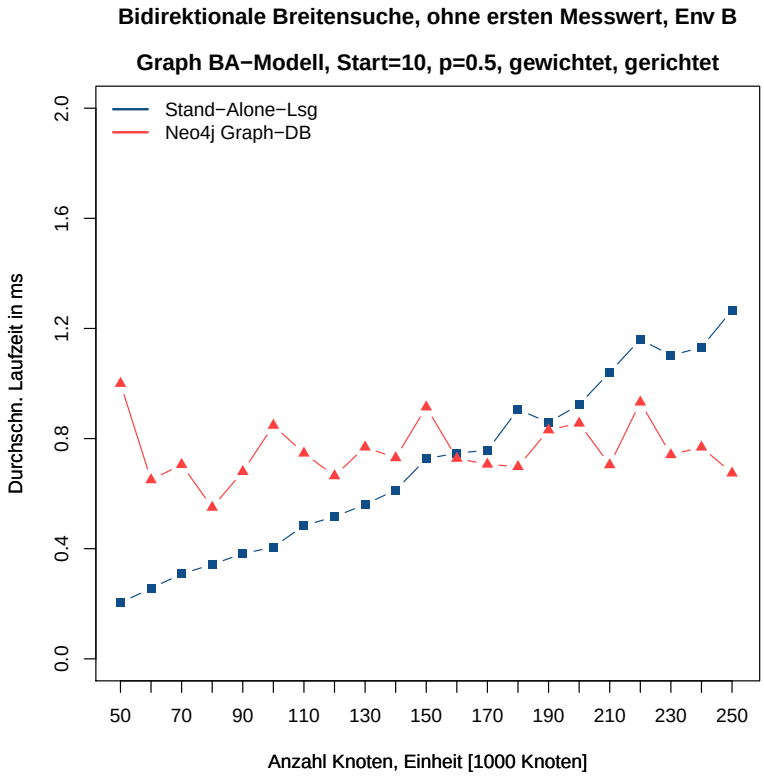


Abbildung 30: Bidirektionale Breitensuche, durchschnittliche Laufzeiten ohne den ersten Messwert.

Wir vermuten, dass bei der ersten Durchführung der bidirektionalen Breitensuche, also direkt nach dem Import des Graphen in die Neo4j-Graph-DB, der Graph in den Cache geladen wird und dadurch die Suche nach dem kürzesten Pfad bei Folgeabfragen entsprechend schneller ausgeführt werden kann. Weiterhin beobachten wir, dass von Zeit zu Zeit nach ein paar ähnlichen Anfragen an den selben Graphen in Neo4j die Laufzeit einmalig etwas höher (etwa 16 ms) ausfällt, aber bei der nächsten Messung dann gleich wieder 0 oder 1 ms beträgt. Auch dieses Verhalten wäre typisch für die Nutzung eines Caches, da es sein kann, dass ein Teil der Daten im Cache inzwischen wieder verdrängt worden ist und bei der Bedienung der Anfrage erneut in den Cache geladen werden muss. Bei der weiteren Analyse haben wir eine Empfehlung von Neo4j, Inc. gefunden, die besagt, dass wir die Performance nach einem Kaltstart der Neo4j-DB steigern können, indem wir durch vorgeschlagene Abfragen bewirken, dass der Graph, je nach Größe, ganz oder teilweise in den Cache geladen wird [38]. Unsere Vermutungen werden dadurch bestätigt.

Weiterhin beobachten wir bei dem Vergleich der Laufzeiten ohne den ersten Messwert, dass sich die bidirektionale Breitensuche eher konstant verhält, während die Messwerte der Laufzeit bei der Stand-Alone-Lösung mit wachsender Knotenzahl ansteigen. So würde bei Folgeanfragen bzw. ähnlichen Anfragen die bidirektionale Breitensuche in Neo4j etwa ab einer Graphgröße von 160 T Knoten kürze Laufzeiten haben als die Stand-Alone-Lösung.

Auswirkung der Anzahl der Messwerte auf die Höhe der Laufzeiten

In unseren ersten Messexperimenten haben wir die Laufzeiten von nur wenigen Graphen, dafür aber mit einer sehr hohen Anzahl an Wiederholungen, zu messen. Dabei fiel uns auf, dass die Messdaten hohe Schwankungen aufwiesen, je nach dem, ob bei dem jeweiligen Graphen ein günstiger Startknoten für die Suche verwendet wurde oder nicht. Um eine gute Mischung von Graphdaten für die Laufzeitmessung zu erhalten, haben wir uns entschlossen, die bis dahin gesammelten Messdaten zu verwerfen und die neuen Messexperimente mit vielen verschiedenen Graphen durchzuführen. Der jeweilige Startknoten für die Suche wird zufällig pro Eingabegraph festgelegt. Je mehr Graphen getestet werden, desto größer ist die Chance, eine ausgewogene Mischung an unterschiedlichen Konstellationen zu bekommen und damit auch ein ausgewogeneres Messergebnis. So wurden insbesondere Graphen im Bereich von 50 T bis 250 T Knoten mit einer Schrittweite von 10 T Knoten je 100 generierte Graphen getestet und pro Graph dann 10 sich wiederholende Anfragen. Die Anzahl der wiederholten Anfragen pro Graph wurde also etwas reduziert, um insbesondere die Durchführungszeiten der Experimente insgesamt in einem zeitlich machbaren Rahmen zu halten. Allerdings ist hierbei zu beachten, das der Durchschnitt der gemessenen Laufzeit etwas höher ausfällt, wenn nur 10 Wiederholungen gemessen werden, statt wie zuvor 50, 100 oder gar 1000 Wiederholungen. Insbesondere ergibt sich eine höhere durchschnittliche Laufzeit bei der bidirektionalen Breitensuche, da ja die erste Anfrage immer eine deutlich höhere Laufzeit hat. So wird die durchschnittliche Laufzeit mit einer höheren Anzahl an Wiederholungen wesentlich niedriger, wie in Abbildung 31 illustriert wird.
Hier wurden jeweils 100 verschiedenen Graphen zu den angegebenen Knotenzahlen gemessen, einmal mit 10 Wiederholungen und einmal mit 50 Wiederholungen. Allerdings zeigt sich damit auch, dass der charakteristische Verlauf der gedachten Messkurve, die sich aus den Messwerten ergibt, nicht davon beeinflusst wird, sodass die weiteren Messungen mit 10 Wiederholungen in jedem der Experimente durchgeführt wird.

Vergleich der Messwerte, 100 T bis 1 Mio. Knoten

Bei der Betrachtung von BA-Graphen mit 100 T bis 1 Mio. Knoten sind alle gemessenen Laufzeiten der bidirektionalen Breitensuche mit Neo4j höher als die der Stand-Alone-Lösung. Betrachten wir jedoch wieder die Messungen ohne den ersten Messwert, so liegt die bidirektionale Breitensuche in Neo4j ab etwa 200 T Knoten vorne.

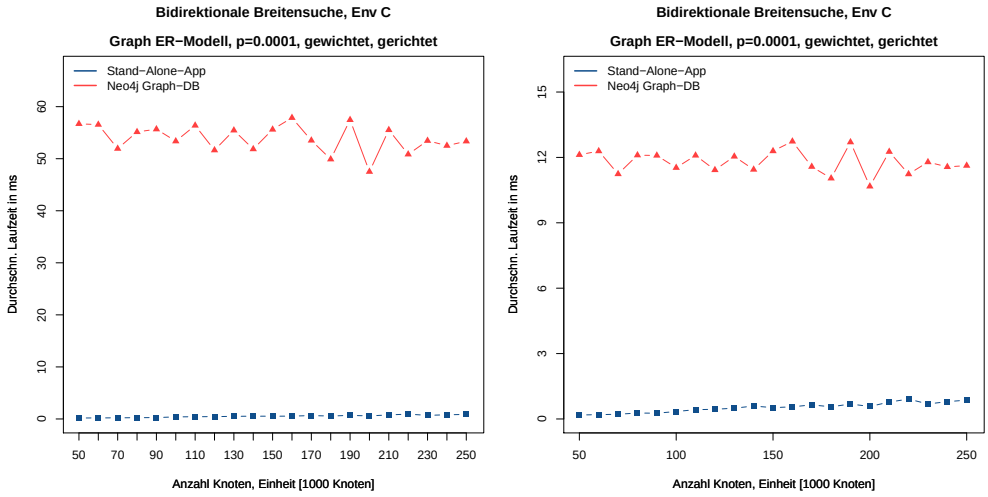


Abbildung 31: Bidirektionale Breitensuche, links: Plot mit 10 Wiederholungen, rechts: Plot mit 50 Wiederholungen.

Weiterhin vermuten wir, dass insbesondere bei großen Graphen eine parallele Verarbeitung mit Neo4j zu kürzeren Laufzeiten führt. In Neo4j kann man die Parallelisierung erlauben und entsprechend passend zum Environment eine gewisse Anzahl von Threads in der Konfiguration von Neo4j (in Datei neo4j.conf) einstellen. Ein Versuch mit dem dafür konfigurierten Env C (Server mit 128 GB RAM und 2 Prozessoren mit je 8 CPU-Kernen) erreichten wir trotz der in der Neo4j-Konfiguration eingestellten Parallelisierung mit 16 Threads keine schnellere Laufzeit. Ein Grund dafür könnte sein, dass die in den Experimenten genutzten Graphen aufgrund ihrer Größe auch bei den kleineren Environments, die wir für die Experimente nutzen, bereits ganz in den Cache passen und deswegen kein Performance-Gewinn mehr erzielt werden kann.

Vergleich ER-Graphen mit BA-Graphen

Der Vergleich der Laufzeiten von ER-basierten Graphen mit den BA-basierten Graphen zeigt einen ähnlichen Verlauf der Messungen mit geringen Abweichungen der Messwerte im gemessenen Bereich von 100 T Knoten bis 1 Mio. Knoten, siehe Abbildung 32 auf Seite 90. Trotz einer etwas höheren Kantenzahl bei den ER-Graphen liegen die Messwerte bei der bidirektionalen Breitensuche recht nah beieinander, obwohl damit auch verschiedene Graphen gemessen wurden. Insgesamt zeigt sich die bidirektionale Breitensuche in Neo4j eher gleichbleibend bei wachsender Knotenzahl.

Die verwendeten Eingabegraphen haben in den meisten Fällen eine Kantenzahl, die größer oder gleich der Anzahl der Knoten ist. Damit sind die auszuwertenden Graphen mit hoher Wahrscheinlichkeit zusammenhängend. Weiterhin wurde zu einem zufällig gewählten Startknoten ein Zielknoten in dem Graphen ermittelt,

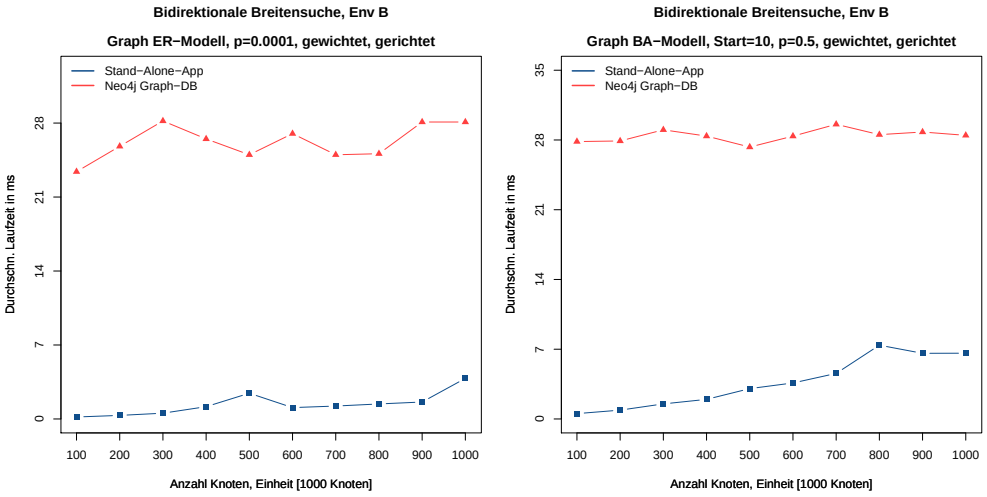


Abbildung 32: Vergleich Laufzeiten ER-Graphen mit BA-Graphen.

zu dem ein Pfad existiert. Somit haben wir dafür gesorgt, dass bei der bidirektionalen Breitensuche auch immer ein kürzester Pfad gefunden wird. Damit kann der schlechteste Fall, dass von beiden Seiten der komplette Breitensuchbaum durchlaufen werden muss, nicht eintreten.

Betrachtung extern geladener Graphen

Als weiteren Vergleich betrachten wir die Messung der Laufzeiten anhand ausgewählter externer, zuvor aus dem Internet geladenen, Real-World-Graphen, gemäß der Übersicht in Kapitel 7.3. Insgesamt wurden sieben verschieden große Graphen für die Messung der bidirektionalen Breitensuche ausgewählt. Wie wir in Abbildung 33 sehen, ist das Verhalten bei den extern geladenen Graphen sehr ähnlich. Auch hier haben wir bei der ersten Anfrage eine höhere Laufzeit und dann wieder eine sehr geringe Laufzeit bei wiederholten, ähnlichen Anfragen von 0 oder 1 ms.

Verhalten bei wachsender Kantendichte und fester Knotenanzahl

Bei der bidirektionalen Breitensuche gibt es keine ersichtlichen Auswirkungen durch die wachsende Kantenzahl bei fester Knotenzahl.

Unterschiede zwischen gerichteten und ungerichteten Graphen

Als Nächstes vergleichen wir die Laufzeiten von gerichteten und ungerichteten Graphen. Da grundsätzlich in dem gemessenen Bereich die Stand-Alone-Lösung geringere Laufzeiten hat als die bidirektionale Breitensuche, betrachten wir zur genaueren Analyse die gemessenen Laufzeiten ohne den ersten Messwert, siehe Abbildung 34. Hier zeigt sich, dass die Stand-Alone-Lösung noch etwas besser bei den gerichteten Graphen abschneidet. Bei wiederholten bzw. ähnlichen Anfragen liegt diese bis zu einer Graph-Größe von etwa 200 T Knoten vorne.

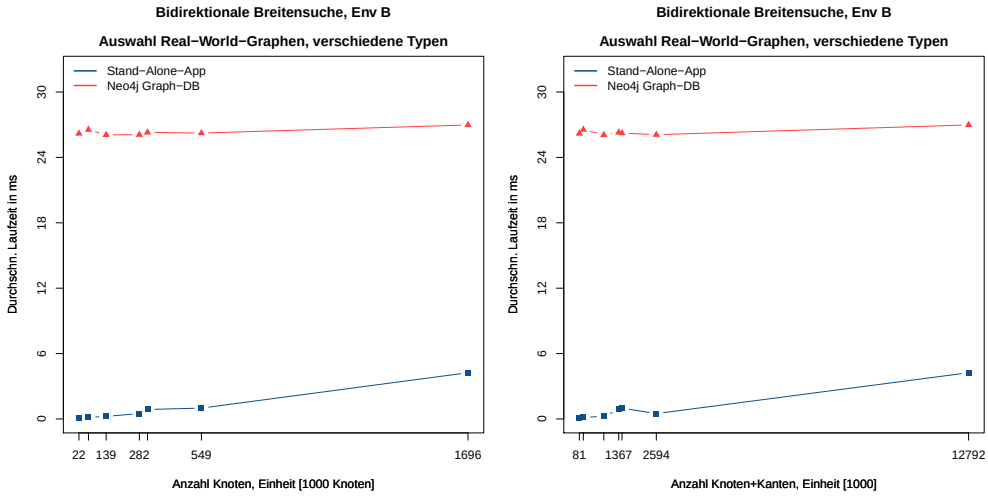


Abbildung 33: Bidirektionale Breitensuche, Betrachtung extern geladener Graphen, Bereich 22 T Knoten bis 1.7 Mio. Knoten.

Das liegt wahrscheinlich daran, dass im Fall der Auswertung eines gerichteten Graphen bei der Stand-Alone-Lösung zusätzlich zu der Adjazenzliste mit den Nachfolgern auch eine mit den Vorgängern geführt wird. Dadurch kann die bidirektionale Breitensuche bei gerichteten Graphen etwas performanter durchgeführt werden.

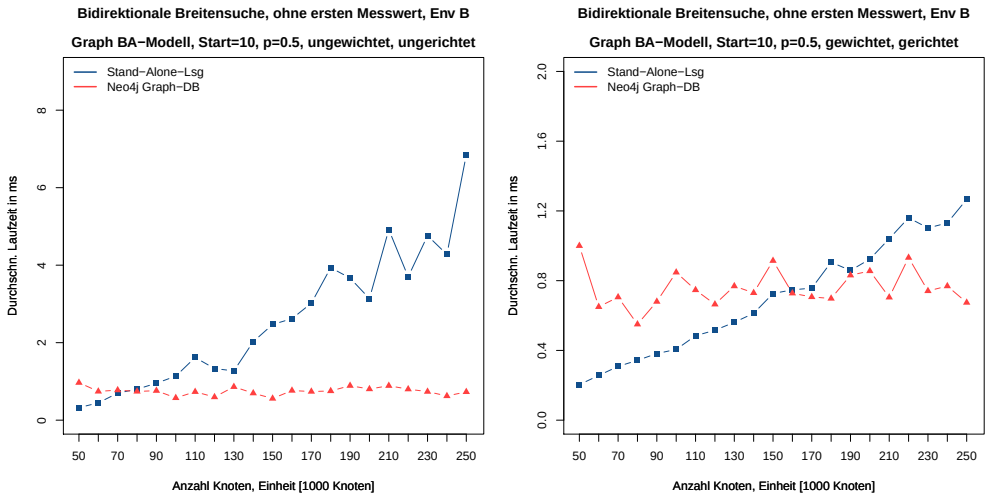


Abbildung 34: Bidirektionale Breitensuche, Unterschiede zwischen ungerichteten und gerichteten Graphen.

Betrachtung der Auswirkungen des Environments

Vergleichen wir die in Abbildung 35 dargestellten Messwerte mit denen in Abbildung 32, so fällt uns auf, dass die gemessenen Laufzeitwerte in Env A geringfügig schlechter sind als in Env B, was bei einem Environment mit etwas mehr RAM, 32 GB im Vergleich mit 16 GB, zu erwarten ist.

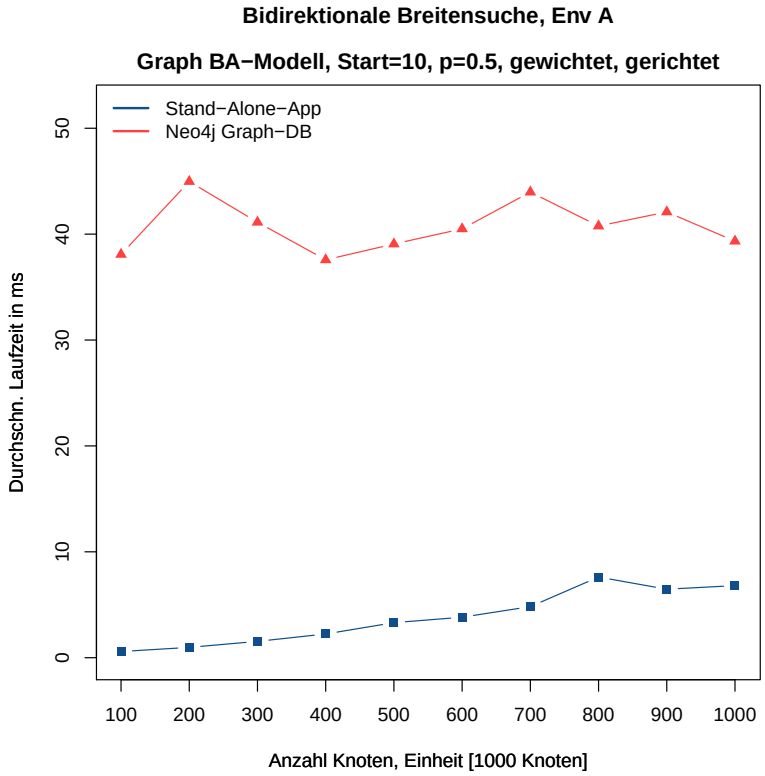


Abbildung 35: Bidirektionale Breitensuche, Vergleich der Messungen in unterschiedlichen Environments, hier mit Env A.

Ein weiterer Vergleich mit Env C, einem Windows Server Environment, zeigt überraschenderweise, dass die Laufzeiten der bidirektionalen Breitensuche in Neo4j bei den Graphen, die im Env C gemessen wurden, etwas schlechter sind, als die, die in Env A gemessenen. Dies tritt trotz erheblich mehr Hauptspeicher, nämlich 128 GB RAM im Vergleich zu 16 GB RAM auf. Zudem war Neo4j gemäß der Empfehlungen im Performance-Guide eingestellt [23]. Dazu haben wir, neben den in Kapitel 7.1 durchgeführten Einstellungen, eine Parallelisierung mit bis zu 16 Threads erlaubt (Eintrag in neo4j.conf). Wir vermuten, dass die etwas schlechteren Laufzeiten durch einem gewissen Server-Overhead verursacht werden, bedingt durch weitere Komponenten und Dienste, die auf dem Server laufen. Möglich wäre auch, dass bei der Größe des Graphen kein weiterer Performanzgewinn durch Parallelisierung erreicht werden kann, da dieser vollständig in den Cache geladen werden kann.

Die Nutzung des Caches in Neo4j bringt sicherlich bis zu einer gewissen Größe des Graphen einen Vorteil. Allerdings wäre es auch interessant zu sehen, wie sich die Messwerte verändern, wenn der Graph groß genug ist und nicht mehr ganz in den Cache passt. Dieses Phenomen konnten wir bei der Größe der hier untersuchten Graphen nicht beobachten.

Anhand der Analyse der gemessenen Laufzeiten und den daraus gewonnen Erkenntnissen können wir folgende Aussagen hinsichtlich der in Kapitel 6.1 aufgestellten Hypothesen machen:

Entgegen unserer Erwartungen können wir Hypothese 1.1 nicht direkt bestätigen. Die Laufzeiten der Stand-Alone-Lösung sind in dem gemessenen Bereich durchweg geringer als die mit Neo4j gemessenen. Selbst bei wiederholten Messungen, nachdem der Graph in den Cache geladen worden ist, hat die Stand-Alone-Lösung kürzere Laufzeiten bei Graphen bis zu 160 T Knoten. Gemäß der Empfehlung von Neo4j Inc. sollte ein Warm-Up als einmalige Aktion nach einem Kalt-Start (oder nach Import in eine neue DB) durchgeführt werden, um den Graphen ganz oder teilweise in den Cache zu laden [38]. Wenn dies grundsätzlich noch vor der ersten Anfrage an die Neo4j-Graph-DB durchgeführt wird, so kann Hypothese 1.1 in dem betrachteten Messbereich angenommen werden. Denn dann ist die bidirektionale Breitensuche in dem getesteten Messbereich bei Graphen größer als 160 T Knoten performanter als die Stand-ALone-Lösung.

Die Hypothese 1.2 können wir insofern bestätigen, dass wenn hier ebenfalls ein Warm-Up des Caches gemäß der Empfehlung von Neo4j Inc. durchgeführt wird, wir einen Schwellwert angeben können, nämlich eine Graphgröße von 160 T Knoten, ab dem die bidirektionale Breitensuche schneller als die Stand-Alone-Lösung ist.

7.5 Experiment 2: Vergleich des Dijkstra-Algorithmus

7.5.1 Aufbau und Durchführung

In diesem Experiment liegt der Fokus auf den gewichteten Graphen, wobei wir nur Graphen mit nicht-negativen, ganzzahligen Gewichten in den Messexperimenten verwenden. Graphen, die ungewichtet sind, ordnen wir bei der Berechnung ein Kantengewicht von Eins zu.

In verschiedenen Messexperimenten wird der Dijkstra-Algorithmus zur Berechnung der kürzesten Pfade von einem Startknoten zu allen anderen Knoten im Graphen gemäß des in Kapitel 7.1 beschriebenen Aufbaus getestet. Speziell bei diesem Experiment werden wir sogar die Laufzeiten von drei Lösungen miteinander vergleichen: Die in Neo4j mit den effizienten Graphalgorithmen implementierte Version des Dijkstra-Algorithmus, eine als Neo4j-Plugin entwickelte Erweiterung und die implementierte Stand-Alone-Lösung. Dabei ist die Implementierung der Erweiterung in Neo4j der Implementierung der Stand-Alone-Lösung sehr ähnlich. Wir verwenden denselben Algorithmus und auch dieselbe Min-Prioritätswarteschlange, nutzen allerdings die Neo4j-API, um auf den in der Datenbank befindlichen Graphen zuzugreifen.

Die zu diesem Experiment gehörenden Konfigurationen und Messdateien können über den Namensteil **config2** zugeordnet werden. Speziell für die Tests mit der Neo4j-Graph-DB werden zwei unterschiedliche Queries für den Test des Dijkstra-Algorithmus ausgewertet, siehe Listing 13 und 14:

Listing 13: Cypher-Query für die Messung des Neo4j-Dijkstra-Algorithmus.

```
MATCH (n:Knoten) WHERE id(n)=17586 CALL algo.shortestPaths(
 n, 'weight', {write:false}) YIELD evalDuration
 RETURN evalDuration
```

Listing 14: Cypher-Query für die Messung der eigenen Neo4j-Erweiterung.

```
CALL dijkstra.algoIntDijkstra(17586,317081)
 YIELD evalDuration RETURN evalDuration
```

Mit der Query in Listing 13 wird zunächst der Startknoten mit der übergebenen Id im Graphen gesucht und dann der Dijkstra-Algorithmus mit der Cypher-Klausel `CALL` aufgerufen. Dieser ist Bestandteil der effizienten Graph-Algorithmen, die Neo4j Inc. als Standard-Plugin für das Neo4j-Graph-DBMS im Rahmen der GNU General Public License zur Verfügung stellt. In unseren Test-Environments wurde das zum installierten Neo4j-Software-Release passende Plugin, Release 3.4.0.0, installiert. Da wir zu einem gegebenen Startknoten den kürzesten Pfad zu allen anderen Knoten im Graphen ermitteln und testen

wollen, ist der Algorithmus mit dem Namen **algo.shortestPaths** unsere Wahl. Dieser ist als Neo4j-Procedure implementiert und es gibt zwei Varianten, die man hier ausführen kann: Eine Variante, die einen Stream mit den Knoten und Distanzen zurückliefert und eine Variante, mit der wir uns nur die Ausführungszeit als Ergebnis liefern lassen können. Letztere Variante ist damit die, die wir für die Messung der Laufzeiten benötigen. Beim Aufruf des Algorithmus ist neben dem Startknoten auch der Name der Eigenschaft mitzugeben, die das Gewicht der Kante enthält. In unserem Fall ist das die Eigenschaft ‚weight'. Außerdem wird mit der Eigenschaft {write:false} festgelegt, dass der gefundene Pfad nicht als Eigenschaft in den Graphen zurückgeschrieben wird. Der Defaultwert für `write` ist nämlich `true`.

Die zweite Query, die in Listing 14 beschrieben ist, ruft mit `CALL` unsere eigene in Neo4j implementierte Procedure auf. Diese erwartet zwei Eingabeparameter: Die Id des Startknotens und die Anzahl der Knoten, die der Graph insgesamt hat. Auch hier wird die Zeit bei der Ausführung in Millisekunden gemessen und als Ergebnis der Query zurückgegeben.

Die Knoten-Ids, die bei einer Query verwendeten werden, ändern sich pro Query, da der Test mit wechselnden Startknoten passend zu den Eingabegraphen durchgeführt wird. Die Startknoten werden im Rahmen der Vorbereitung, wie im Kapitel 7.1 beschrieben, ermittelt und in der zugehörigen Konfigurationsdatei, mit der die Messung gesteuert wird, abgelegt.

Im Rahmen der Auswertung der Messergebnisse werden die in Kapitel 6.2 aufgestellten Hypothesen überprüft.

7.5.2 Ergebnisse und Auswertung

Die zusammengeführten Messergebnisse der Laufzeiten der einzelnen Lösungen haben wir mit dafür vorbereiteten Skripten in der Programmiersprache R, wie im Kapitel 5.6 beschrieben, ausgewertet. Dabei wurden verschiedene Plots zu jedem durchgeführten Messexperiment erzeugt. Insgesamt haben wir 14 Messexperimente mit verschieden großen, unterschiedlichen Graphtypen in drei verschiedenen Environments durchgeführt. Im folgenden Abschnitt werden wir ausgewählte Plots und Erkenntnisse daraus vorstellen.

Betrachtung BA-Graphen bei wachsender Knotenanzahl

Wir beginnen mit der Betrachtung von Graphen, die basierend auf dem Barabási–Albert-Modell generiert wurden. In Abbildung 36 sind je zwei der implementierten Lösungen zusammen in einem Diagramm dargestellt. Dabei werden Graphen mit wachsenden Knotenzahlen im Bereich von 100 T bis zu 1 Mio. Knoten als Eingabegraphen für die Berechnung der kürzesten Pfade eingesetzt.

Auffällig bei der Betrachtung des Verlaufs der gemessenen Laufzeiten ist, dass die Laufzeiten der Neo4j-Lösung deutlich höher sind im Vergleich zur eigenen Neo4j-Procedure, während diese wiederum höher sind als die mit der Stand-Alone-Lösung gemessenen Laufzeiten. Weiterhin beobachten wir, dass der Unterschied speziell bei der Neo4j-Lösung mit wachsender Knotenzahl des Graphen mitwächst. Daher vermuten wir, dass es sich bei dieser Beobachtung wahrscheinlich nicht nur um einen Overhead handeln könnte, der beim Einsatz eines Datenbanksystems in der Regel durch Verwaltungsaufgaben anfällt.

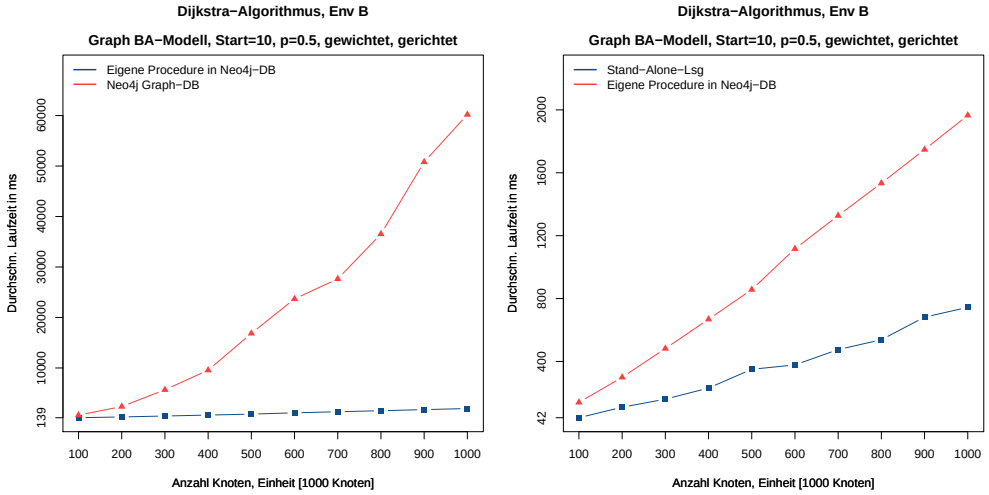
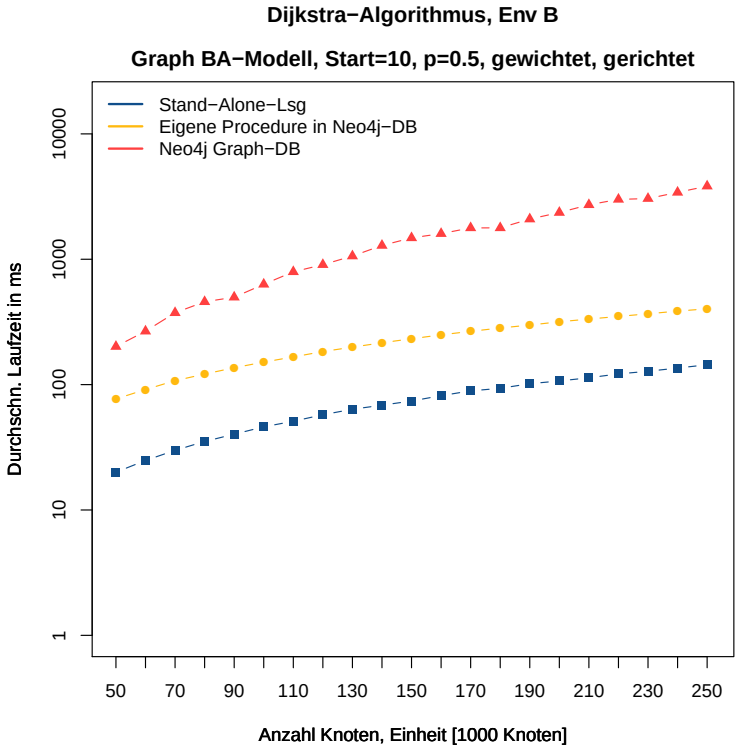
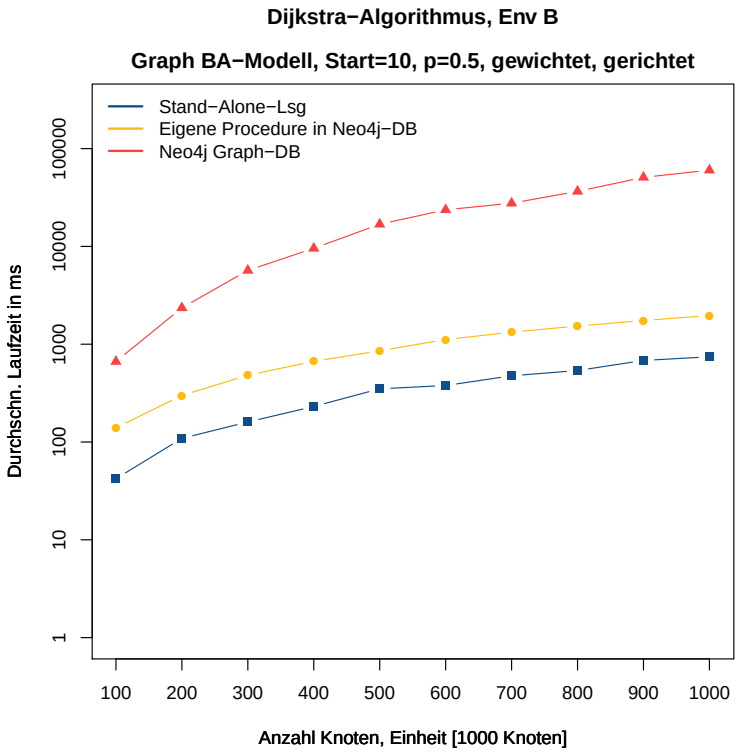


Abbildung 36: Dijkstra-Algorithmus, Vergleich der einzelnen Lösungen auf Basis BA-Graphen bei wachsender Knotenzahl.

Als Nächstes betrachten wir die Entwicklung der Laufzeiten mit wachsender Knotenzahl von allen drei Lösungen in einer Grafik. Dabei nutzen wir eine logarithmische Skala, da die Unterschiede in den Laufzeiten zwischen den einzelnen Lösungen sonst zu groß sind und wir keinen Trend daraus erkennen können. Bei der Betrachtung des Plots in Abbildung 37 bemerken wir, dass auch hier die Laufzeiten bei der Neo4j-Lösung etwas mehr ansteigen als die eigene in Neo4j implementierte Procedure. Im Gegensatz dazu deutet der Vergleich der Laufzeiten der Stand-Alone-Lösung mit denen der eigenen Neo4j-Procedure darauf hin, dass es sich bei der Differenz dieser beiden Lösungen tatsächlich um einen Unterschied durch einen Datenbank-Overhead handeln könnte.



(a) 50 T bis 250 T Knoten



(b) 100 T bis 1 Mio. Knoten

Abbildung 37: Dijkstra-Algorithmus, Vergleich der Lösungen bei BA-Graphen mit wachsender Knotenzahl.

Da der Quellcode des Neo4j-Plugins für effiziente Graphalgorithmen im Rahmen der Lizenz offen eingesehen und gemäß der Lizenzbestimmungen auch geändert werden darf, haben wir uns diesen aus der von Neo4j Inc. auf GitHub bereitgestellten Software-Bibliothek heruntergeladen und gesichtet [24]. Für die Messexperimente haben wir die Version 3.4.0 eingesetzt, aber auch Tests mit der neueren Version 3.4.4 duchgeführt. Ein Vergleich mit dem Quellcode ergibt, dass die selbe Min-Prioritätswarteschlange bei beiden Version und auch in der derzeit neuesten Version 3.4.7 mit dem Dijkstra-Algorithmus eingesetzt wird. Diese ist in der Klasse IntPriorityQueue.java implementiert und ist in folgendem Verzeichnis zu finden:

```
neo4j-graph-algorithms-3.4.0.0/core/src/main/java/org/
neo4j/graphalgo/core/utils/queue
```

In der Klasse `IntPriorityQueue.java`, die einen (Min-)Heap implementiert, wird unter anderem eine Methode namens `update(int element)` aufgerufen, bei der der Schlüsselwert eines Elements, hier die bisher ermittelte kürzeste Distanz vom Startknoten aus, geändert werden soll. Das entspricht in unserem Fall, bei der Nutzung als Min-Heap, der in Kapitel 4.2.2 beschriebenen decreaseKey()-Operation. Die Laufzeit dieser Methode hängt davon ab, wie schnell das zu ändernde Element im Heap gefunden wird, und von den Schritten, die notwendig sind, um die Heap-Bedingung wieder herzustellen. Die Suche nach dem Element wird in der Methode findElementPosition(element) durchgeführt, wo in einer Schleife nach dem Element im Heap gesucht wird. Wie wir gelernt haben, beträgt die Laufzeit einer Suche in einer ungeordneten Menge im worst-case $O(n)$, wobei n für die Anzahl der Elemente steht. In unserem Fall entspricht das der Anzahl der Knoten des Graphen, da beim Dijkstra-Algorithmus alle Knoten in die Min-Prioritätswarteschlange gestellt werden. Auch insgesamt erhalten wir eine Laufzeit von $O(n)$ im worst-case, da die erneuten Herstellung der Heap-Bedingung in einer worst-case Laufzeit von $O(log(n))$ erfolgt. Abbildung 38 zeigt den Auszug aus dem Quellcode, den wir hier beschrieben haben.

Doch warum ist die Laufzeit der gerade beschriebenen Methode ein möglicher Grund, warum speziell bei der Neo4j-Lösung die gemessenen Laufzeiten stärker ansteigen als bei den anderen beiden Lösungen?

Die Antwort erhalten wir, wenn wir diese Operation in der eigenentwickelten Lösung betrachten. Wie in Kapitel 5.3.1 beschrieben wird, führen wir eine zusätzliche Referenz auf ein Element im Heap. Dadurch ist es möglich, ohne Suche direkt in $O(1)$ Zeit auf das Element im Heap mit dem Schlüsselwert, der verringert werden soll, zuzugreifen. Durch die Realisierung des Heap in einem Array ist grundsätzlich ein direkter Zugriff auf die Array-Elemente möglich. Wenn man sich also gemerkt hat, an welcher Position das Element in dem Heap gespeichert wurde, so kann man die gesamte Laufzeit der Operation auf im worst-case $O(log(n))$ verkürzen, nämlich auf die Zeit, die benötigt wird, um die Heap-Bedingung wieder herzustellen.

```
162     /**
163      * Updates the heap because the cost of an element has changed,
164      * Cost is linear with the size of the queue.
165      */
        public final void update(int element) {
167         int pos = findElementPosition(element);
168         if (pos != 0) {
169             if (!upHeap(pos) && pos < size) {
170                 downHeap(pos);
171             }
172         }
173     }
174
175     public final void set(int element, double cost) {
176         if (addCost(element, cost)) {
177             update(element);
178         } else {
179             add(element);
180         }
181     }
182
183     private int findElementPosition(int element) {
184         final int limit = size + 1;
185         final int[] data = heap;
186         int i = 1;
187         for (; i <= limit - 4; i += 4) {
188             if (data[i] == element) return i;
189             if (data[i + 1] == element) return i + 1;
190             if (data[i + 2] == element) return i + 2;
191             if (data[i + 3] == element) return i + 3;
192         }
193         for (; i < limit; ++i) {
194             if (data[i] == element) return i;
195         }
196         return 0;
197     }
```

Abbildung 38: Dijkstra-Algorithmus, Fundstelle in der Priority Queue der Neo4j-Lösung.

Die Implementierung der Min-Prioritätswarteschlange ist somit ein Grund für die höheren Laufzeiten der Neo4j-Lösung. Doch ist das der einzige Grund, warum höhere Laufzeiten bei der Neo4j-Lösung gemessen werden? Gibt es eventuell noch weitere Einflussfaktoren, die eine höhere Laufzeit verursachen?

Betrachten wir die Unterschiede der Laufzeiten in den Plots in Abb. 37, so bemerken wir, dass es zwischen der eigenen Neo4j-Procedure und der Stand-Alone-Lösung auch eine, zwar etwas langsamer wachsende, Differenz der Laufzeiten gibt. Da sowohl die Stand-Alone-Lösung als auch die eigene Neo4j-Procedure dieselbe Implementierung der Min-Prioritätswarteschlange nutzen, bestärkt das unsere Vermutung, dass dieser Unterschied in den Laufzeiten durch einen weiteren Faktor, wie etwa ein Datenbank-Overhead verursacht wird.

Vergleich der Lösungen mit ER-Graphen und wachsender Knotenzahl

Für die weitere Untersuchung betrachten wir daher noch weitere Auswertungen der Laufzeiten. Als nächstes untersuchen wir die gemessenen Laufzeiten der auf dem Erdös–Rényi-Modell basierenden Graphen (kurz ER-Modell bzw. ER-Graphen), siehe Abbildung 39 und 40.

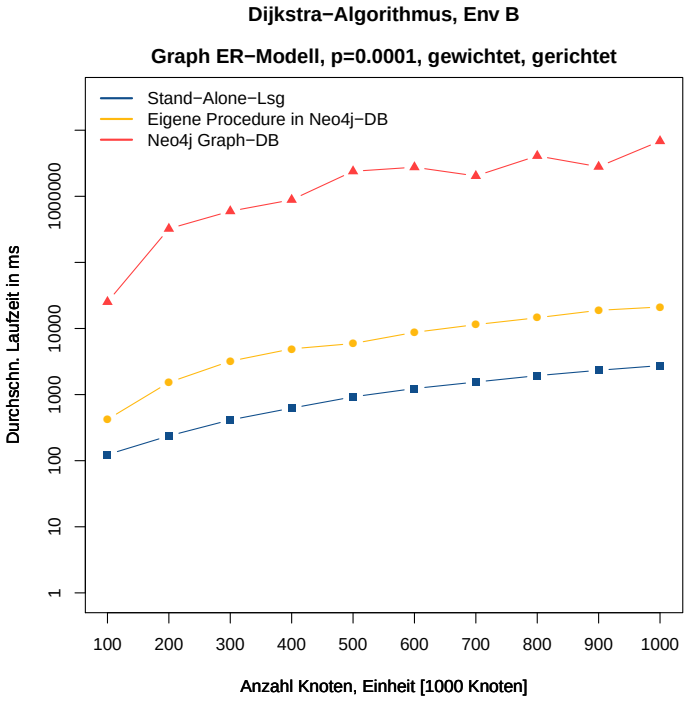


Abbildung 39: Dijkstra-Algorithmus, Vergleich der Lösungen bei ER-Graphen mit wachsender Knotenzahl.

Die hier untersuchten ER-Graphen wurden mit einer Kantenwahrscheinlichkeit von 0.0001 generiert. Das hat zur Folge, dass die meisten dieser Graphen eine höhere Kantendichte aufweisen, als die zuvor betrachteten BA-Graphen. Dies kann sich auch bei den Laufzeiten der implementierten Algorithmen bemerkbar machen. Tatsächlich beobachten wir ein noch etwas stärkeres Ansteigen der Laufzeiten der Neo4j-Lösung im Vergleich zu der eigenen Neo4j-Procedure. Die Vermutung liegt nahe, dass es sich also um einen Faktor bzw. Overhead handeln muss, der von der Anzahl der Kanten abhängt. Im Kapitel 4.2 beschreiben wir den Dijkstra-Algorithmus und geben in einer Formel die worst-case Laufzeit in Abhängigkeit der Operationen, die am häufigsten bei der Ausführung des Dijkstra-Algorithmus vorgenommen werden, an. Dazu gehört auch die decreaseKey()-Operation, die mit der in Abbildung 38 illustrierten Methode im worst-case $|E|$ mal ausgeführt wird, wobei $|E|$ für die Anzahl der Kanten des Graphen steht. Die etwas höhere lineare Laufzeit im worst-case bei der Neo4j-Lösung trägt damit zu einer entsprechend höheren Gesamtlaufzeit im Vergleich zu den beiden anderen Lösungen bei.

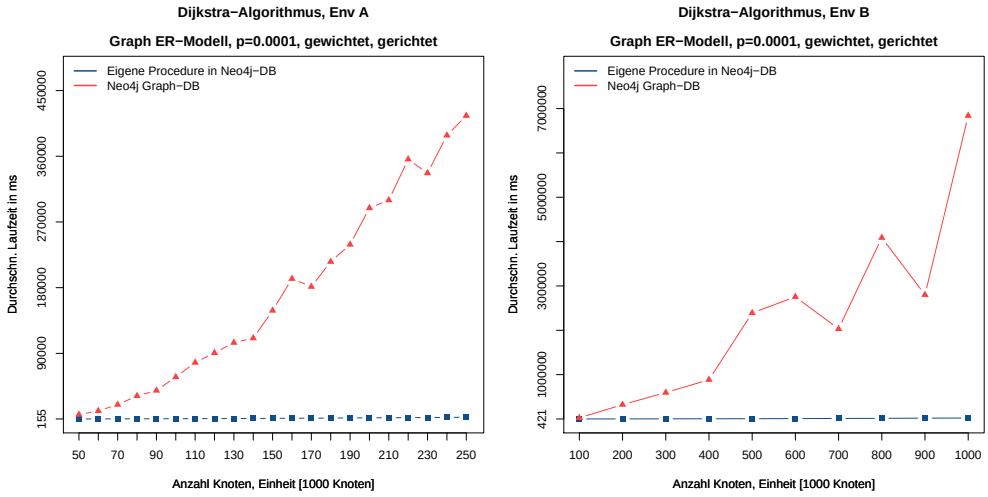


Abbildung 40: Dijkstra-Algorithmus, Vergleich der einzelnen Lösungen auf Basis ER-Graphen bei wachsender Knotenzahl.

Betrachtung des Verhaltens bei wachsender Kantenzahl

Bei der nächsten Messung gehen wir daher noch einen Schritt weiter und vergleichen das Laufzeitverhalten der Lösungen bei fester Knotenanzahl von 50 T Knoten und wachsender Kantenzahl. Dafür haben wir Eingabegraphen auf Basis des ER-Modells mit wachsenden Kantenwahrscheinlichkeiten generiert. Bei einem Graphen mit 50 T Knoten kann so durchaus eine zweistellige Millionenzahl an Kanten erzeugt werden. Entsprechend lange dauert dann auch die Messung der Laufzeiten für das Messexperiment. Die ausgewerteten Plots werden in Abbildung 41 gezeigt.

Besonders interessant ist hier, dass in dem Plot, bei dem der Faktor $f = Kanten/Knoten$ größer als 25 ist, die Laufzeiten der eigenen Neo4j-Procedure stärker anwachsen als bei der Neo4j-Lösung. Ist der Faktor f größer als 200, so nähern sich die beiden Lösungen aneinander an. Wenn wir zurückdenken an die Betrachtung der Element-Suche, die im worst-case in linearer Laufzeit ausgeführt wird und die von der Anzahl der Knoten des Graphen abhängt, so wird klar, dass sich diese Größe bei fester Knotenzahl wie eine Konstante verhält. Betrachten wir die Formel, die wir zur Berechnung der Laufzeit in Kapitel 4.2.2 eingesetzt haben, so ergibt sich, wenn man die 50 T Knoten als feste Größe in diese Formel einsetzt, eine worst-case Laufzeit von $O(|E|)$. Das erklärt auch, warum sich die Laufzeit der Lösungen bei wachsenden Kanten unter dieser Bedingung aneinander annähern. Dennoch wird bei der Betrachtung der Plots auch sichtbar, dass es trotzdem noch einen Unterschied in der Höhe der Laufzeiten gibt, was die Vermutung erhärtet, dass ein zusätzlicher Datenbank-Overhead ein weiterer Grund für die höhere Laufzeit mit Neo4j ist.

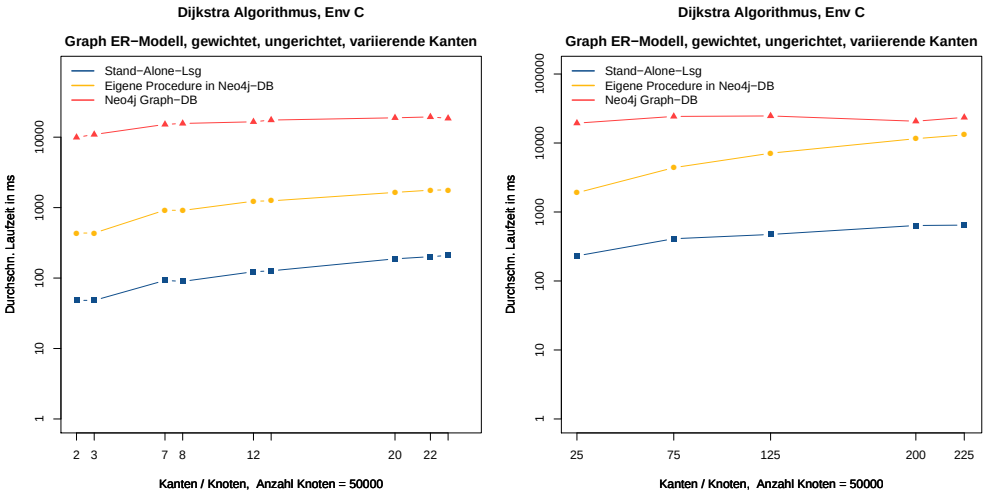
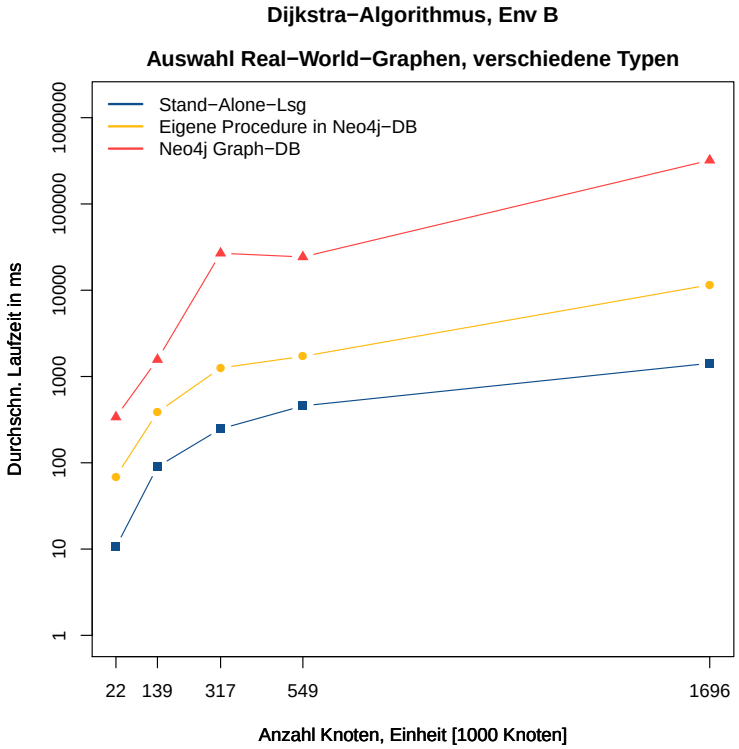


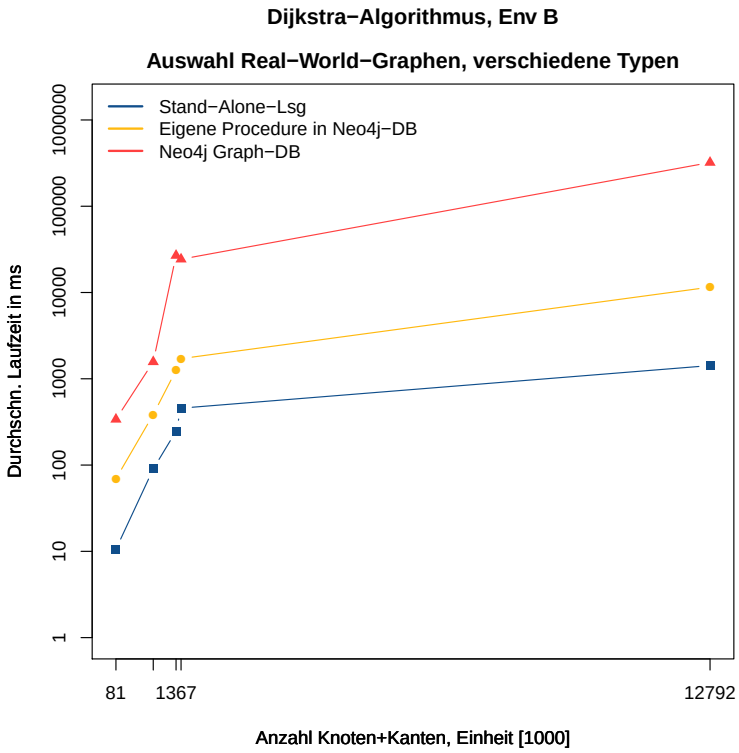
Abbildung 41: Dijkstra-Algorithmus, Vergleich der Lösungen bei wachsender Kanten-Dichte.

Bisher haben wir das Verhalten der Laufzeiten nur mit synthetisch erzeugten Graphen untersucht. In einem weiteren Messexperiment haben wir die Laufzeiten der Lösungen mit ausgewählten, aus dem Internet geladenen, sogenannten Real-World-Graphen, gemessen und ausgewertet. Tabelle 15 gibt eine Übersicht über die ausgewählten Graphen. Vergleichen wir das Laufzeitverhalten der drei gemessenen Lösungen, die in Abb. 42 zusammen auf einer gemeinsamen Skala dargestellt sind, mit den bisherigen Auswertungen, so sehen wir einen ähnlichen charakteristischen Verlauf der gemessenen Laufzeiten.
Dies ist auch in anderen Plots der Auswertung zu dieser Gruppe von Graphen ersichtlich, die wir aus Gründen der Übersichtlichkeit hier nicht darstellen. Diese können Bedarf auf dem dieser Arbeit beigelegten USB-Stick gesichtet werden. Insgesamt bestätigt uns dies, dass wir mit unseren bisherigen Vermutungen richtig liegen.

Darüber hinaus haben wir auch bei diesem Experiment Laufzeitmessungen in unterschiedlichen Environments miteinander verglichen. Das Verhalten war hier ähnlich wie bei der bidirektionalen Breitensuche, weswegen wir hier nicht näher darauf eingehen.



(a) Betrachtung nach Knotenzahl



(b) Betrachtung Summe Knoten u. Kanten

Abbildung 42: Dijkstra-Algorithmus, Vergleich der Lösungen mit extern geladenen Graphen.

Anhand der Analyse der gemessenen Laufzeiten und den daraus gewonnen Erkenntnissen können wir folgende Aussagen hinsichtlich der in Kapitel 6.2 aufgestellten Hypothesen machen:

Es konnte definitiv eine Auswirkung der Wahl der Min-Prioritätswarteschlange, die mit dem Dijkstra-Algorthmus eingesetzt wird, nachgewiesen werden. Aus diesem Grund gilt die Hypothese 6.2 als angenommen.

Weiterhin können wir nach der Analyse auch bestätigten, dass es zusätzlich noch einen weiteren Faktor gibt, der dazu führt, dass die Laufzeiten mit Neo4j etwas höher sind, als mit der implementierten Stand-Alone-Lösung. Obwohl wir mit der eigenen Neo4j-Procedure denselben Algorithmus, inkl. derselben Min-Prioritätswarteschlange, implementiert haben, wie in der Stand-Alone-Lösung, waren die Laufzeiten mit Neo4j etwas höher. Dies bestätigt unsere Vermutung, dass es sich hierbei um einen Datenbank-Overhead handelt. Schließlich fällt ein gewisser Grundaufwand für die Verwaltung in einem DBMS an, der nötig für die Erfüllung der in Kapitel 2.3 beschriebenen Anforderungen an ein Datenbanksystem ist.

Allerdings waren die gemessenen Laufzeiten bei der Stand-Alone-Lösung bei den für die Messungen eingesetzten Eingabegraphen im Messbereich stets geringer als bei den Lösungen mit Neo4j. Dadurch können wir keinen Schwellwert für die Größe eines Graphen angeben, bis zu dem der Datenbank-Overhead sich tatsächlich so auswirkt, dass die Stand-Alone-Lösung dadurch performanter ist. Damit gilt die Hypothese 2.2, siehe Kapitel 6.2, in dem mit den Messexperimenten abgedeckten Messbereich als nicht bestätigt. Die Vermutung ist allerdings, dass die getesten Graphen ggf. noch nicht groß genug waren, um diesen eigentlich erwarteten Effekt messen zu können.

7.6 Experiment 3: Vergleich des Bellman–Ford-Algorithmus

7.6.1 Aufbau und Durchführung

In diesem Experiment liegt der Fokus auf den gewichteten Graphen, wobei hier grundsätzlich auch negative Kantengewichte erlaubt sind. Um jedoch Seiteneffekte durch mögliche negative Zyklen in den Eingabegraphen zu vermeiden, setzen wir hier nur Graphen mit positiven, ganzzahligen Gewichten ein. Dabei wird bei ungewichteten Graphen das Kantengewicht standardmäßig mit Eins vorbelegt.

In verschiedenen Messexperimenten wird der Bellman–Ford-Algorithmus zur Berechnung der kürzesten Pfade von einem Startknoten zu allen anderen Knoten im Graphen gemäß des in Kapitel 7.1 beschriebenen Aufbaus getestet. Speziell bei diesem Experiment werden die Laufzeiten einer eigenentwickelten Neo4j-Erweiterung mit der implementierten Stand-Alone-Lösung verglichen. Dabei ist die Implementierung der Neo4j-Erweiterung der Implementierung der Stand-Alone-Lösung sehr ähnlich. Wir verwenden denselben Algorithmus und auch dieselbe Standard-Prioritätswarteschlange. Allerdings greifen wir dann in der Procedure über die Neo4j-API auf den Graphen in der Neo4j-DB zu.

Die zu diesem Experiment gehörenden Konfigurationen und Messdateien können über den Namensteil **config3** zugeordnet werden. Speziell für die Tests mit der Neo4j-Graph-DB wird eine Query in der folgenden Form für die Ausführung des Bellman–Ford-Algorithmus ausgeführt, siehe Listing 15:

Listing 15: Cypher-Query für die Messung des Bellman–Ford-Algorithmus.

```
CALL bellmanford.algoIntBellmanFord(278634,317081)
 YIELD evalDuration RETURN evalDuration
```

Die zweite Query, die in Listing 15 beschrieben ist, ruft mit CALL unsere eigene in Neo4j implementierte Procedure auf. Diese erwartet zwei Eingabeparameter: Die Id des Startknotens und die Anzahl der Knoten, die der Graph insgesamt hat. Auch hier wird die Zeit bei der Ausführung in Millisekunden gemessen und als Ergebnis der Query zurückgegeben.

Die in der Query verwendeten Knoten-Ids ändern sich bei jeder Query, da der Test mit wechselnden Startknoten passend zu den Eingabegraphen durchgeführt wird. Diese wurden im Rahmen der Vorbereitung, wie im Kapitel 7.1 beschrieben, ermittelt und in Konfigurationsdateien zur Steuerung der Messexperimente gespeichert.

Im Rahmen der Auswertung der Messergebnisse werden die in Kapitel 6.3 aufgestellten Hypothesen überprüft.

7.6.2 Ergebnisse und Auswertung

Da der Bellman–Ford-Algorithmus nicht Bestandteil der Neo4j-Algorithmen-Bibliothek ist und auch nicht im Neo4j-Plugin für effiziente Graphalgorithmen enthalten ist, wurde dieser als Neo4j-Erweiterung implementiert. Nähere Details zur Implementierung werden in Kapitel 5.4 beschrieben. Durch die Implementierung mit demselben Algorithmus und derselben Prioritätswarteschlange erwarten wir ein in etwa gleiches Verhalten der beiden Lösungen bei der kürzesten-Pfad-Suche. Abweichungen sollte es hierbei lediglich durch die Nutzung der Neo4j-API, oder durch die Verwaltung mit dem Neo4j-DBMS geben. Zumindest erwarten wir ab einem bestimmten Schwellwert auch eine Angleichung der Laufzeiten zwischen der eigenen Neo4j-Procedure und der Stand-Alone-Lösung.

Als Erstes werden wir uns das Verhalten und auch die Streuung der Messdaten in einem kombinierten Boxplot ansehen, siehe Abb. 43.

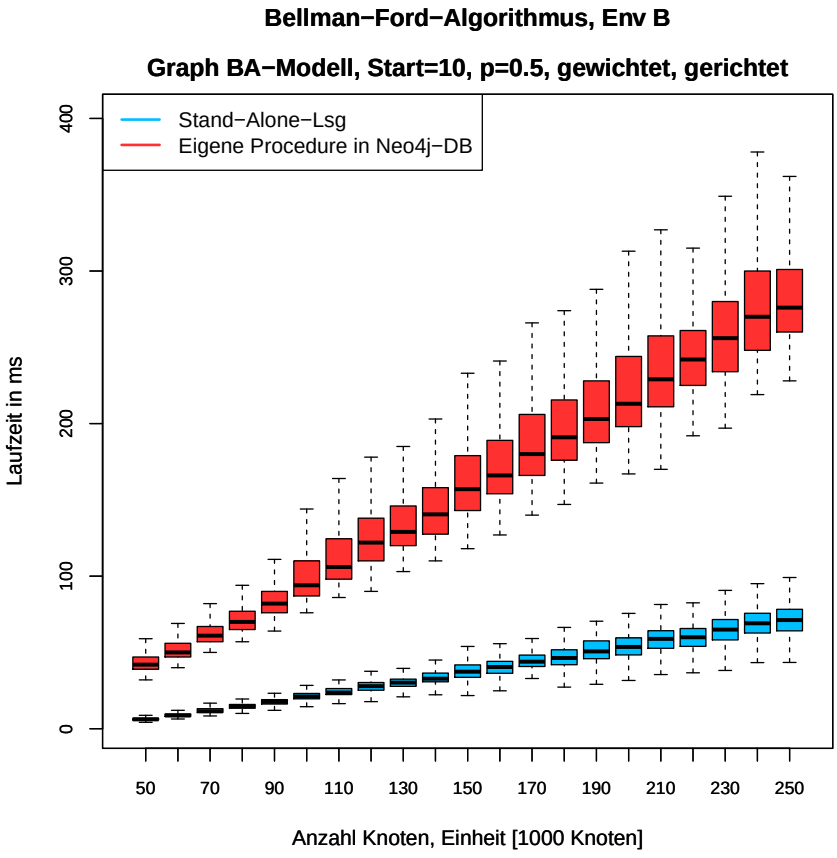


Abbildung 43: Bellman–Ford Algorithmus, Boxplot, $quantile(x, c(1,3)/4)$.

Die Darstellung als kombinierter Boxplot haben wir gewählt, um die Messwerte anhand der selben Skala anzuzeigen. Dadurch lassen sich die Unterschiede der

Messdaten präziser darstellen und wir können diese besser miteinander vergleichen. Erwartungsgemäß sieht der Verlauf des Medians in beiden Plots ähnlich aus. Wir beobachten, dass die Streuung der gemessenen Werte mit wachsender Knotenzahl etwas größer wird, wobei die Messwerte bei der eigenen Neo4j-Procedure eine deutlich höhere Streuung aufweisen. Bei größeren Knotenzahlen liegt der Median auch mehr mittig zwischen dem unteren und dem oberen Quartil des jew. Boxplots. Betrachten wir die gedachte Messkurve, die durch die Verbindung der Median-Werte entsteht, so verlaufen die beiden Messkurven wie zwei Geraden, die eine unterschiedliche Steigung haben. Insgesamt gewinnen wir dadurch den Eindruck, dass der Unterschied der gemessenen Daten durch einen Datenbank-Overhead verursacht werden könnte.

Zur weiteren Analyse betrachten wir nun die Laufzeiten der beiden Lösungen für ER-basierte Graphen in dem selben Messbereich. Die hier betrachten Graphen wurden zuvor mit dem Graphgenerator mit einer Kantenwahrscheinlichkeit von $p = 0.0001$ generiert. Diese Kantenwahrscheinlichkeit sorgt dafür, dass diese Graphen etwas dichter als die bisher betrachteten BA-Graphen sind. Wir betrachten zunächst den graphischen Verlauf der Messwerte, siehe Abbildung 44. Danach ermitteln wir das Verhältnis der Laufzeiten von Neo4j zur Stand-Alone-Lösung, um zu sehen, wie sich der daraus resultierende Faktor bei wachsender Knotenzahl verhält.

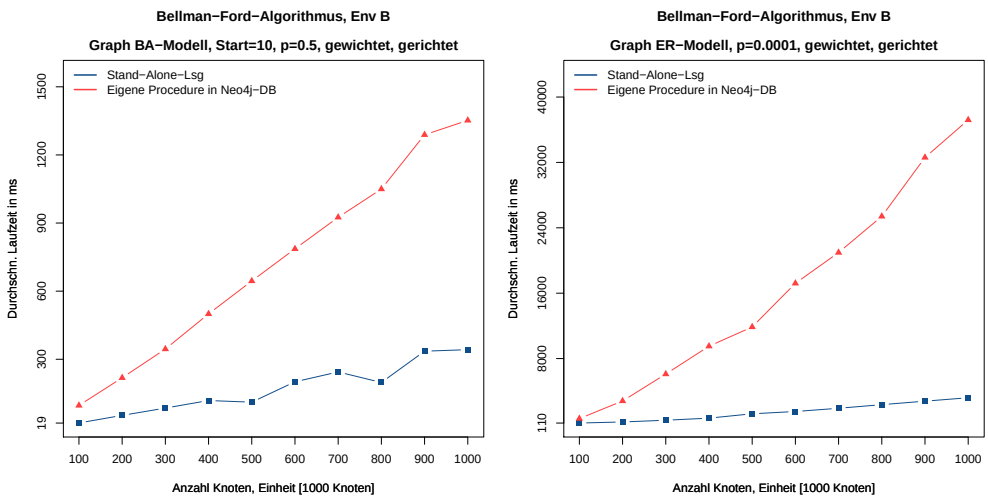


Abbildung 44: Bellman–Ford-Algorithmus, Lösungsvergleich anhand BA- und ER-Graphen mit wachsender Knotenzahl.

Durch die Betrachtung des Verhältnisses der Laufzeiten in den Tabellen 16 und 17, können wir sehen, dass sich der Faktor bei wachsender Knotenzahl relativ konstant verhält, was die Vermutung erhärtet, dass die Laufzeitunterschiede durch einen Datenbank-Overhead verursacht werden.

Tabelle 16: Messwerte Laufzeit BA-Graphen, gewichtet, gerichtet, wachsende Knotenzahl, Env B.

Anzahl Knoten	Durchschn. Anz. Kanten	Durchschn. Laufzeit Stand-Alone [ms]	Durchschn. Laufzeit Neo4j [ms]	Faktor Laufzeit Neo4j / Stand-Alone
100000	125012	21	104	4
110000	137499	24	115	4
120000	150038	27	129	4
130000	162540	30	138	4
140000	175083	33	148	4
150000	187492	38	165	4
160000	200005	40	176	4
170000	212475	45	190	4
180000	225032	45	199	4
190000	237576	51	212	4
200000	250029	53	224	4
210000	262588	58	237	4
220000	275059	60	247	4
230000	287460	64	261	4
240000	300074	69	278	4
250000	312482	70	286	4
300000	375252	85	345	4
400000	499924	118	500	4
500000	625029	111	645	5
600000	749875	201	786	3
700000	874925	244	925	3
800000	999907	198	1050	5
900000	1125405	336	1289	3
1000000	1250258	342	1352	3

Tabelle 17: Messung Laufzeit ER-Graphen, wachsende Knotenzahl, Env B.

Anzahl Knoten	Durchschn. Anz. Kanten	Durchschn. Laufzeit Stand-Alone [ms]	Durchschn. Laufzeit Neo4j [ms]	Kanten / Knoten	Faktor Laufzeit Neo4j / Stand-Alone
100000	624742	110	653	6	5
200000	2500110	245	2812	13	11
300000	5627605	456	6103	19	13
400000	9999073	712	9511	25	13
500000	15622489	1245	11867	31	9
600000	22492359	1524	17208	37	11
700000	30627216	1920	20970	44	10
800000	40002382	2361	25399	50	10
900000	50627134	2789	32604	56	11
1000000	62509472	3208	37198	63	11

Insgesamt beobachten wir, dass der Faktor trotz wachsender Knotenzahl in etwa gleich bleibt. Er ist jedoch bei den ER-Graphen etwas höher als bei den BA-Graphen, sodass wir einen Zusammenhang mit der Kantenanzahl vermuten.

In der folgenden Abbildung werden die Laufzeiten von ER-Graphen dargestellt, die mit wachsender Kantenzahl und bei fester Knotenzahl gemessen wurden. Die Darstellung erfolgt anhand eines Faktors, der sich aus dem Verhältnis von Kanten- zu Knotenanzahl des Graphen ergibt. Die Eingabegraphen haben wir bei der Vorbereitung dieses Messexperiments mit verschiedenen Kantenwahrscheinlichkeiten generiert. Bei der Betrachtung der rechten Graphik mit dem Kanten- zu Knotenverhältnis von 25 bis 225 beobachten wir, dass die Messwerte mit der wachsenden Zahl der Kanten anwachsen, sich jedoch ein ähnlicher Verlauf zeigt wie bei der Betrachtung der gemessenen Laufzeiten der BA-Graphen, siehe Abb. 44. Auch das Verhältnisses der gemessenen Laufzeiten ist wie bei den vorherigen Betrachtungen bis auf einen in etwa konstant bleibenden Faktor gleich.

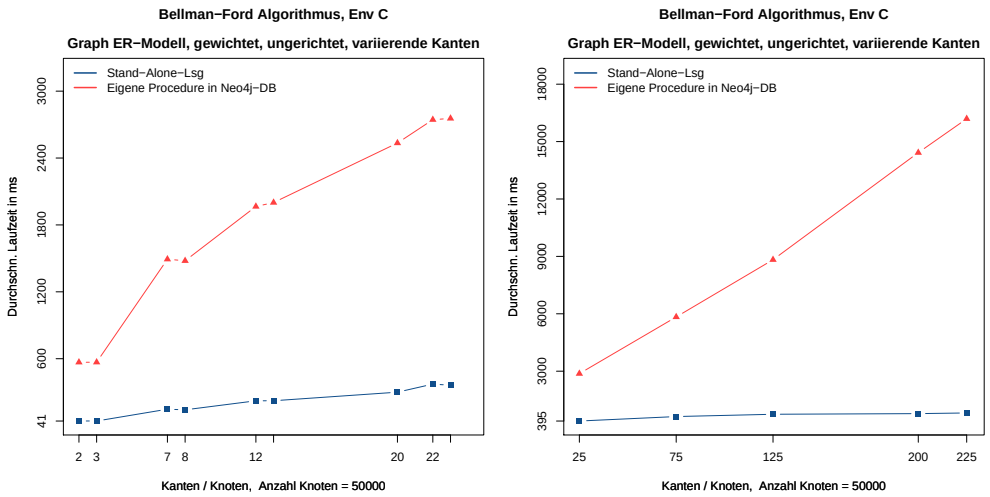


Abbildung 45: Bellman–Ford-Algorithmus, Lösungsvergleich anhand ER-Graphen mit 50000 Knoten wachsender Kantenzahl.

Als weiteren Vergleich betrachten wir das Verhalten anhand von extern aus dem Internet geladenen Graphen, siehe Abbildung 46. Welche Graphen hier als Eingabegraphen eingesetzt werden, wird in Tabelle 15 beschrieben. Das Verhalten der Laufzeiten dieser Gruppe von Graphen ist ähnlich. Auch hier haben wir ein in etwa gleich bleibendes Verhältnis der gemessenen Laufzeiten der Lösungen zueinander. Das erhärtet unsere Vermutung, dass der Unterschied der Laufzeiten durch einen Datenbank-Overhead verursacht wird.

Weitere Analysen beim Vergleich von gerichteten und ungerichteten Graphen und von Messungen in verschiedenen Environments brachten bei diesem Experiment keine neuen Erkenntnisse, daher gehen wir hier nicht näher darauf ein.

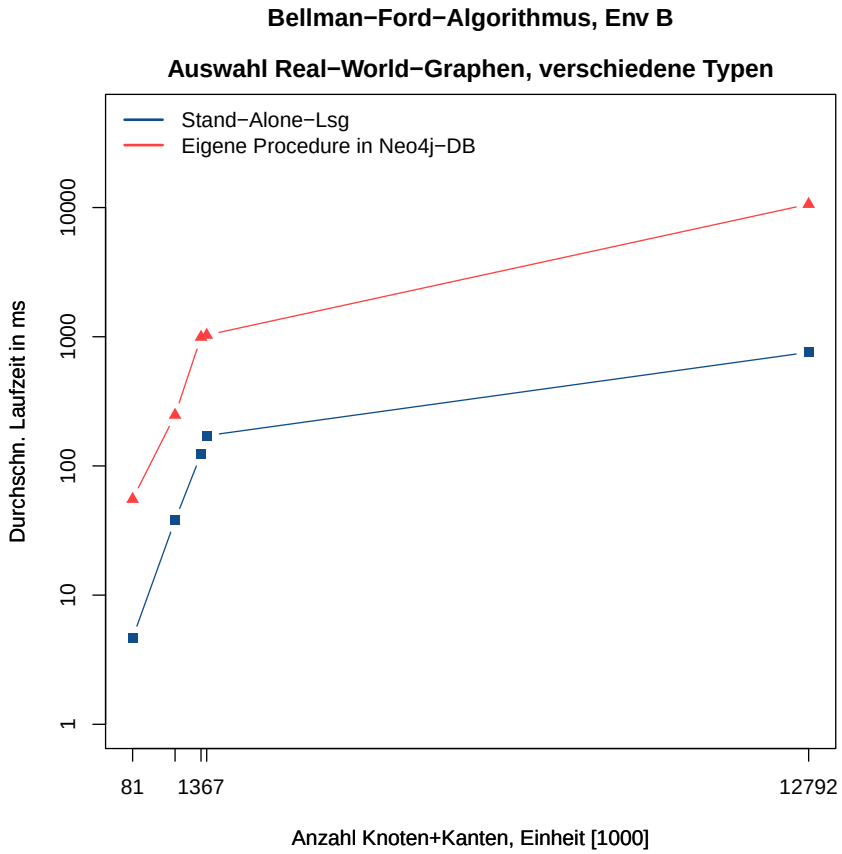


Abbildung 46: Bellman–Ford-Algorithmus, Betrachtung Real-World-Graphen mit unterschiedlicher Knoten- und Kantenzahl.

Nach der hier durchgeführten Analyse und den Erkenntnissen aus dem Experiment 2, indem ja auch eine eigene Neo4j-Procedure mit der Stand-Alone-Lösung verglichen wurde, kommen wir zu dem Schluss, dass die unterschiedlichen Laufzeiten zwischen der eigenen Neo4j-Procedure und der Stand-Alone-Lösung durch einen Datenbank-Overhead verursacht wird. Da die Laufzeiten der Lösungen sich in dem gemessenen Bereich nur um einen konstanten Faktor unterscheiden, verhalten sich beide Lösungen tatsächlich in etwa gleich. Daher können wir die in Kapitel 6.3 aufgestellte Hypothese 3.1 bestätigen. Diese gilt also als angenommen.

Auf der anderen Seite können wir Hypothese 3.2 trotz des in den Experimenten bestätigten Datenbank-Overheads nicht bestätigen. Der Grund ist, dass aus den Messergebnissen nicht ersichtlich ist, bis zu welcher Größe eines Graphen, ausgedrückt in Anzahl Knoten oder Kanten, die Stand-Alone-Lösung performanter als die eigene Neo4j-Procedure ist. Schließlich sind die gemessenen Laufzeiten der Stand-Alone-Lösung im gesamten gemessenen Bereich kleiner als die der eigenen Neo4j-Procedure.

8 Fazit und Ausblick

Im Folgenden fassen wir unsere Erkenntnisse zusammen und geben einen Ausblick mit Ideen für weitere Experimente.

Automatisierung der Messexperimente ist wichtig

Nach anfänglichen Messexperimenten mit wenigen Graphen und vielen Wiederholungsmessungen, stellten wir fest, dass die gemessenen Laufzeiten starke Schwankungen hatten, je nachdem, ob eine günstige oder ungünstige Konstellation für die Suche nach dem kürzesten Pfad in dem Graphen vorlag. Aus diesem Grund wurden die ersten Messergebnisse verworfen und stattdessen Messexperimente mit vielen Eingabegraphen, aber dafür mit weniger Wiederholungsmessungen durchgeführt. Dadurch bekamen wir Messwerte mit weniger Streuungen. In einigen Fällen erhalten wir allerdings einen höheren Durchschnittswert, wenn zum Beispiel die erste gemessene Laufzeit signifikant höher ausfällt, als bei den übrigen Messungen. Das war zum Beispiel bei der bidirektionalen Breitensuche der Fall.

Die Umstellung der Messexperimente brachte allerdings auch ein paar zusätzliche Herausforderungen mit sich: Die Handhabung von deutlich mehr Dateien in den Verzeichnissen, u.a. Messdateien, Skripte und die Dateien für die vielen Eingabegraphen. Speziell bei den Eingabegraphen stieg dadurch der Speicherplatzbedarf rasant an. Denn für die Vorbereitung für den Neo4j-Import werden für jeden Eingabegraphen vier Dateien erzeugt, die in einem Format gehalten sind, das in die Neo4j-Graph-DB importiert werden kann. Bei einem Messexperiment mit 100 Graphen pro Knotenanzahl im Bereich von 50 T bis 250 T Knoten mit einer Schrittweite von 10 T Knoten ergibt das 2100 Dateien mit Eingabegraphen. Zusammen mit den je 4 Import-Dateien sind das 10500 Dateien. Hinzu kommen zum Beispiel 6300 Konfigurationsdateien für die Steuerung der Experimente, weitere Dateien mit Skripten, die für die Neo4j-Queries mit Cypher-Shell ausgeführt werden etc. Da können durchaus inklusive der Dateien mit den Messdaten mehrere GB an Speicherplatzverbrauch für ein Messexperiment zusammenkommen.

Insgesamt haben wir 14 Messexperimente durchgeführt. Eine weitere Herausforderung dabei war, dass bei der Durchführung der Experimente sehr viele Datenbankimporte ausgeführt werden. Um gleiche Bedingungen für die Messungen zu schaffen, wird der Import immer in eine neue Datenbank durchgeführt. Dazu muss zunächst der Neo4j-DB-Service gestoppt, die bisher verwendete Datenbank gelöscht und der DB-Service wieder gestartet werden, bevor eine Anfrage für die Messungen an die Neo4j-DB gestellt werden kann. Der Start des DB-Service dauert ein paar Sekunden, was bei 2100 Importen die Dauer eines Messlaufs um einige Stunden verlängern kann.

Auch das Generieren der vielen Graphen hat einige Zeit in Anspruch genommen, da die Generierung einer Serie von Graphen mit mehr als 600 T Knoten auch bereits mehrere Stunden in Anspruch nimmt. So waren einige Laptops und PC's nur mit der Generierung von Graphen betraut, während parallel auf den in Kapitel 7.2 aufgeführten Environments Messexperimente durchgeführt wurden.

Als Fazit können wir hierzu geben, dass es wirklich hilfreich ist, sich zu überlegen wie man die Schritte von solchen Experimenten automatisieren kann. Hätten wir zum Beispiel keine entsprechenden Programme und Skripte für die automatisierte Durchführung erstellt, so wäre die oben beschriebene Umstellung der Messexperimente in dem vorgegebenen Zeitrahmen nicht mehr möglich gewesen. Die Masse an Daten, die dabei erzeugt wird, kann man nicht mehr manuell bewältigen und auch nicht mehr manuell auswerten. Durch den modularen Aufbau, siehe Beschreibung zum Ablauf in Kapitel 7.1, ist es auch möglich, gezielt Nachmessungen zu Experimenten durchzuführen, bei denen es zu Messfehlern kam.

Zusammenfassung der Ergebnisse der Experimente

Insgesamt waren wir überrascht, dass bei der eigenen implementierte Stand-Alone-Lösung im Vergleich mit dem Neo4j-GDBMS durchweg kürzere Laufzeiten gemessen wurden. Es konnte zwar ein Datenbank-Overhead mit Neo4j bestätigt werden, doch aufgrund der Messungen in den meisten Fällen kein Schwellwert angegeben werden, ab wann denn die Ausführung mit Neo4j performanter ist. Dies war für den gemessenen Bereich nicht möglich. Lediglich bei der bidirektionalen Breitensuche konnte festgestellt werden, dass bei einem konsequenten Warm-Up des Caches vor der ersten Anfrage ein Schwellwert von 160 T Knoten angegeben werden kann. In dem Fall hätte dann bei Graphen ab einer Größe von über 160 T Knoten die bidirektionale Breitensuche eine kürzere Laufzeit als die Stand-Alone-Lösung. Damit konnten allerdings beide Hypothesen für die bidirektionale Breitensuche aufgestellt worden waren, nicht bestätigt werden.
Beim Dijkstra-Algorithmus konnte die Hypothese 2.1 bestätigt werden, da eine Abhängigkeit von der Wahl der Min-Prioritätswarteschlange nachgewiesen werden konnte. Hypothese 2.2 konnte nicht bestätigt werden, da es zwar einen Datenbank-Overhead gibt, wir aber keinen Schwellwert in dem gemessenen Bereich mit den in Kapitel 7.3 festgelegten Eingabegraphen angeben können, ab dem dann Neo4j performanter als die Stand-Alone-Lösung ist.
Die Hypothese 3.1 gilt als angenommen, da sich bei der Analyse der Laufzeiten des Bellman–Ford-Algorithmus herausgestellt hat, dass sich die Laufzeiten der beiden Lösungen in dem gemessenen Bereich nur um einen konstanten Faktor unterscheiden, also damit beide asymptotisch gleich performant sind (gemäß O-Notation). Auch beim Bellman–Ford-Algorithmus konnte ein Datenbank-Overhead bestätigt werden, allerdings auch ohne Angabe eines Schwellwertes, weswegen auch hier die Hypothese 3.2 nicht bestätigt werden kann.

Zum Datenbanksystem Neo4j

Viele Probleme der heutigen Zeit lassen sich als Graph modellieren und in einer Graph-DB abbilden. Durch den Einsatz eines Graph-DBS können insbesondere vernetzte Daten mit Graph-Algorithmen effizienter ausgewertet werden, als dies mit in RDBS üblichen Join-Operationen möglich wäre [37].

Mit Neo4j sind die Eigenschaften und Labels eines Graphen flexibler wartbar als mit eigenen Programmen, die sonst ständig angepasst werden müssten, siehe dazu auch die in Kapitel 2.1 aufgeführten Aspekte.

Ein gewisser Datenbank-Overhead ist auch bei Graph-DBS vorhanden, da Anforderungen wie Transaktionsmanagement, Nebenläufigkeit, Recovery und weitere Anforderungen, siehe Kapitel 2.3, erfüllt werden müssen.

Die Untersuchung der Algorithmen in Neo4j hat gezeigt, dass diese teilweise noch verbessert werden können. Wir empfehlen daher, diese vor dem produktiven Einsatz zu testen und bei unerwarteten Ergebnissen genauer zu untersuchen (Code-Review).

In Neo4j werden nur Graphen mit gerichteten Kanten angelegt. Dabei sollen die Kanten wegen der besseren Performanz nur in einer Richtung angelegt werden. Da in Neo4j Adjazenzlisten für Vorgänger und Nachfolger geführt werden, ist das Navigieren im Graphen auch entgegen der definierten Richtung der Kanten möglich, ja sogar gewollt. Dies wird allerdings noch nicht von allen Algorithmen in Neo4j unterstützt. Das hat zur Folge, dass wir diesen Sachverhalt beim Datenbankentwurf berücksichtigen müssen. Es handelt sich hierbei um eine physische Gegebenheit, die auf konzeptueller und externer Ebene eigentlich verborgen bleiben und keinen Einfluss haben sollte, siehe Abb. 1. Auch aus dem Blickwinkel der Graphentheorie ist es irritierend, dass ein Graph mit einer gerichteten Kante definiert wird und dann trotzdem in allen Richtungen durchlaufen werden kann.

Neo4j erfüllt viele Anforderungen, die an ein DBS gestellt werden. Insbesondere hervorzuheben ist hier das Transaktionsmanagement nach dem ACID-Prinzip, da dies bei vielen GDBS nur in einer abgeschwächten Form unterstützt wird. Der Aufwand für den Datenbankentwurf ist gering, da ein Graph einfach abgebildet werden kann. Allerdings birgt das auch Gefahren, da man dennoch Fehler beim Datenbankentwurf machen kann. Daher empfehlen wir, beim Entwurf gedanklich die Auswertungen durchzuspielen, die benötigt werden. Dadurch können wir prüfen, ob alle relevanten Daten beim Datenbankentwurf erfasst wurden.

Ausblick und Ideen für weitere Experimente

Aufgrund der Ergebnisse, bei denen einige der Hypothesen nicht bestätigt werden konnten, wäre es interessant, in einem weiteren Experiment mit noch größeren Graphen, oder auch Graphen mit vielen Eigenschaften (properties), festzustellen, ob fehlende Schwellwerte gefunden werden und damit die Aussagen zu den Hypothesen in Kapitel 6 weiter präzisiert werden können.

Zudem wäre es interessant zu testen, wie sich die bisherigen Ergebnisse verändern, wenn grundsätzlich ein Warm-Up des Caches, wie von Neo4j Inc. empfohlen [38], für den Graphen in der gestarteten Neo4j-Graph-DB vor der Ausführung der ersten Laufzeitmessung durchgeführt wird. Es könnte ja sein, dass die Stand-Alone-Lösung deshalb besser in den Messexperimenten performt, weil beim Einlesen des Graphen in die Datenstrukturen auch bereits Daten in den Cache geladen werden, während das bei Neo4j bisher nur beim Start der ersten Anfrage erfolgt. Auch hier könnten die Aussagen zu den aufgestellten Hypothesen in Kapitel 6 noch verfeinert werden.

Weitere neue Experimente könnten die Betrachtung der Auswertung von Graphen in verteilten Systemen sein, entweder mit geschickter Partitionierung der Graphen oder mit der Nutzung von Cache-Sharding. Mit Cache-Sharding ist hier gemeint, dass der Graph zwar auf jedem System als Ganzes vorhanden ist, aber nur Teile, auf die schnell zugegriffen werden soll, in den Cache geladen werden. So kann man in jedem beteiligten System einen anderen Teil des Graphen in den Cache laden und damit auf dem jeweiligen System Teilauswertungen sehr schnell durchführen und, falls gewünscht, die erhaltenen Teilergebnisse zusammentragen. Insgesamt könnte man so auch ein Environment aufbauen, das skalierbar ist [29].

Darüber hinaus wären Experimente bzw. Untersuchungen von Algorithmen im Bereich der Visualisierung von Auswertungsergebnissen bei großen Graphen interessant. Hier sollte ein ausgewertetes Muster gut erkennbar sein. Dabei wird zunächst eine grobe Darstellung der interessierenden Teile des Graphen gezeigt und erst bei einem Drilldown (Doppelklick) erscheinen dann weitere Details.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] Hiroyuki Akama u. a. “Random Graph Model Simulations of Semantic Networks for Associative Concept Dictionaries”. In: *Proceedings of the 3rd Textgraphs Workshop on Graph-Based Algorithms for Natural Language Processing*. TextGraphs-3. Manchester, United Kingdom: Association for Computational Linguistics, 2008, S. 57–60. ISBN: 978-1-905593-57-6. URL: http://dl.acm.org/citation.cfm?id=1627328.1627337.

[2] *arXiv cond-mat network dataset – KONECT*. Im Internet verfügbar am 03.09.2018, um 20:16 Uhr. Apr. 2017. URL: http://konect.uni-koblenz.de/networks/opsahl-collaboration.

[3] Michael Bachman. *Modelling in Neo4j: Bidirectional Relationships*. Im Internet verfügbar am 06.10.2018, um 18:21 Uhr. DZone Community. URL: https://dzone.com/articles/modelling-data-neo4j.

[4] Albert-László Barabási und Réka Albert. “Emergence of Scaling in Random Networks”. In: *Science* 286.5439 (1999), S. 509–512. ISSN: 0036-8075. DOI: 10.1126/science.286.5439.509. eprint: http://science.sciencemag.org/content/286/5439/509.full.pdf. URL: http://science.sciencemag.org/content/286/5439/509.

[5] E. F. Codd. “Relational Database: A Practical Foundation for Productivity”. In: *Commun. ACM* 25.2 (1982), S. 109–117. ISSN: 0001-0782. DOI: 10.1145/358396.358400. URL: http://doi.acm.org/10.1145/358396.358400.

[6] Thomas H. Cormen u. a. *Algorithmen - Eine Einführung, 4. Auflage*. Oldenbourg Verlag München, 2013, S. 153–160, 599–611, 655–675. ISBN: 978-3-486-74861-1.

[7] *DBLP co-authorship network dataset – KONECT*. Im Internet verfügbar am 03.09.2018, um 20:16 Uhr. Apr. 2017. URL: http://konect.uni-koblenz.de/networks/com-dblp.

[8] *DBMS Popularität pro Datenbankmodell, Vollständiger Trend, beginnend mit Jänner 2013*. Im Internet verfügbar am 29.04.2018, um 13:16 Uhr. solidIT consulting & software development gmbh. URL: http://www.db-engines.com/de/ranking_categories.

[9] Martin Dietzfelbinger, Kurt Mehlhorn und Peter Sanders. *Algorithmen und Datenstrukturen, die Grundwerkzeuge*. Berlin, Heidelberg: Springer-Verlag, 2014, S. 161–180, 211–274, 299–311. ISBN: 978-3-642-05471-6. DOI: 10.1007/978-3-642-05472-3.

[10] E. W. Dijkstra. “A Note on Two Problems in Connexion with Graphs”. In: *Numer. Math.* 1.1 (Dez. 1959), S. 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: http://dx.doi.org/10.1007/BF01386390.

[11] *Download Neo4j, Free. Includes Neo4j Desktop and Neo4j Enterprise Edition for Developers*. Im Internet verfügbar am 22.09.2018, um 21:25 Uhr. Neo4j, Inc. URL: https://neo4j.com/download/?ref=whats-new.

[12] *Gnutella network dataset – KONECT*. Im Internet verfügbar am 03.09.2018, um 20:21 Uhr. Apr. 2017. URL: http://konect.uni-koblenz.de/networks/p2p-Gnutella31.

[13] Ali Hamlili. “Intelligibility of Erdös-Rényi Random Graphs and Time Varying Social Network Modeling”. In: *Proceedings of the 2017 International Conference on Smart Digital Environment*. ICSDE ’17. Rabat, Morocco: ACM, 2017, S. 201–206. ISBN: 978-1-4503-5281-9. DOI: 10.1145/3128128.3128159. URL: http://doi.acm.org/10.1145/3128128.3128159.

[14] Winfried Hochstättler und Alexander Schliep. *CATBox An Interactive Course in Combinatorial Optimization*. Springer-Verlag Berlin, Heidelberg, New York, 2010, S. 53–64. ISBN: 978-3-540-14887-6.

[15] *Java SE Development Kit 8 Downloads*. Im Internet verfügbar am 04.10.2018, um 21:30 Uhr. Oracle Corporation. URL: https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html.

[16] Dieter Jungnickel. *Graphs, Networks and Algorithms, 4. Auflage*. Berlin, Heidelberg, New York: Springer-Verlag, 2013, S. 40, 47, 65–70, 86–88. ISBN: 978-3-642-32277-8. DOI: 10.1007/978-3-642-32278-5.

[17] Steven M. LaValle. *Planning Algorithms*. New York, NY, USA: Cambridge University Press, 2006, S. 32–43. ISBN: 0521862051.

[18] Jure Leskovec, Jon Kleinberg und Christos Faloutsos. “Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations”. In: *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. KDD ’05. Chicago, Illinois, USA: ACM, 2005, S. 177–187. ISBN: 1-59593-135-X. DOI: 10.1145/1081870.1081893. URL: http://doi.acm.org/10.1145/1081870.1081893.

[19] Jure Leskovec und Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. Im Internet verfügbar am 03.09.2018, um 18:16 Uhr. Juni 2014.

[20] Jure Leskovec u. a. “Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters”. In: *Internet Mathematics* 6.1 (2009), S. 29–123.

[21] Silviu Maniu, Talel Abdessalem und Bogdan Cautis. “Casting a Web of Trust over Wikipedia: An Interaction-based Approach”. In: *Proc. Int. Conference on World Wide Web Posters*. 2011, S. 87–88.

[22] Andreas Meier und Michael Kaufmann. *SQL- & NoSQL-Datenbanken*. 8. überarbeitete und erweiterte Auflage. Serie eXamen.press, Springer Vieweg, Berlin, Heidelberg, 2016, S. 136–154, 187–237. ISBN: 978-3-662-47663-5, 978-3-662-47664-2 (eBook).

[23] *Neo4j Documentation*. Im Internet verfügbar am 23.09.2018, um 13:00 Uhr. Neo4j, Inc. URL: `https://neo4j.com/docs`.

[24] *neo4j-graph-algorithms 3.4.0.0*. Im Internet verfügbar am 03.06.2018, um 13:22 Uhr. Neo4j, Inc. URL: `https://github.com/neo4j-contrib/neo4j-graph-algorithms/releases/tag/3.4.0.0`.

[25] *NetBeans IDE Download*. Im Internet verfügbar am 04.10.2018, um 21:30 Uhr. NetBeans open-source project. URL: `https://netbeans.org/`.

[26] Mark E. J. Newman. “The Structure of Scientific Collaboration Networks”. In: *Proceedings of the National Academy of Sciences* 98.2 (2001), S. 404–409.

[27] *Relationship direction in cypher is important*. Im Internet verfügbar am 06.10.2018, um 18:27 Uhr. Graphgrid, Inc. URL: `https://www.graphgrid.com/relationship-direction-in-cypher-is-important/`.

[28] Matei Ripeanu, Adriana Iamnitchi und Ian Foster. “Mapping the Gnutella Network”. In: *IEEE Internet Computing* 6.1 (Jan. 2002), S. 50–57. ISSN: 1089-7801. DOI: `10.1109/4236.978369`. URL: `http://dx.doi.org/10.1109/4236.978369`.

[29] Ian Robinson, Jim Webber und Emil Eifrem. *Graph Databases: New Opportunities for Connected Data*. 2nd. O’Reilly Media, Inc., 2015, S. 1–64, 99–180. ISBN: 1491930896, 9781491930892.

[30] Gunter Saake, Kai-Uwe Sattler und Andreas Heuer. *Datenbanken Konzepte und Sprachen*. 5. Auflage. mitp Heidelberg, München, Landsberg, Frechen, Hamburg, 2013, S. 1–48, 67–103, 201–237. ISBN: 978-3-8266-9453-0.

[31] Gunter Saake, Kai-Uwe Sattler und Andreas Heuer. *Datenbanken Konzepte und Sprachen*. 6. Auflage. mitp Heidelberg, München, Landsberg, Frechen, Hamburg, 2018, S. 4–48, 67–103, 201–237, 717–733. ISBN: 978-3-95845-776-8.

[32] Markus Schneider. *Implementierungskonzepte für Datenbanksysteme*. Springer-Verlag Berlin, Heidelberg, New York, 2004, S. 1–15. ISBN: 978-3-540-41962-4.

[33] *Stanford network dataset – KONECT*. Im Internet verfügbar am 03.09.2018, um 20:23 Uhr. Apr. 2017. URL: http://konect.uni-koblenz.de/networks/web-Stanford.

[34] *The Internet-Scale Graph Platform*. Im Internet verfügbar am 01.05.2018, um 13:00 Uhr. Neo4j, Inc. URL: https://neo4j.com/product.

[35] *The Koblenz Network Collection*. Im Internet verfügbar am 23.09.2018, um 21:27 Uhr. Universität Koblenz Landau. URL: http://konect.uni-koblenz.de/.

[36] *The R Project for Statistical Computing, Download CRAN mirror*. Im Internet verfügbar am 04.10.2018, um 21:30 Uhr. The R Foundation, GNU Project. URL: https://www.r-project.org/.

[37] Aleksa Vukotic u. a. *Neo4j in Action*. Manning Publications Co, Shelter Island, NY, 2015, S. 4–11. ISBN: 9781617290763.

[38] *Warm the cache to improve performance from cold start*. Im Internet verfügbar am 06.10.2018, um 17:59 Uhr. Neo4j, Inc. URL: https://neo4j.com/developer/kb/warm-the-cache-to-improve-performance-from-cold-start/.

[39] *WikiSigned network dataset – KONECT*. Im Internet verfügbar am 03.09.2018, um 20:13 Uhr. Apr. 2017. URL: http://konect.uni-koblenz.de/networks/wikisigned-k2.

[40] Jaewon Yang und Jure Leskovec. "Defining and Evaluating Network Communities Based on Ground-Truth". In: *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*. ICDM '12. Washington, DC, USA: IEEE Computer Society, 2012, S. 745–754. ISBN: 978-0-7695-4905-7. DOI: 10.1109/ICDM.2012.138. URL: http://dx.doi.org/10.1109/ICDM.2012.138.

[41] Jaewon Yang und Jure Leskovec. "Defining and Evaluating Network Communities Based on Ground-truth". In: *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*. MDS '12. Beijing, China: ACM, 2012, 3:1–3:8. ISBN: 978-1-4503-1546-3. DOI: 10.1145/2350190.2350193. URL: http://doi.acm.org/10.1145/2350190.2350193.

Anhang

A Beschreibung der Schichten eines DBMS

Beschreibung der Schichten in Abb. 2 von unten nach oben [32]:

Schicht „Geräte- und Speicher-Manager“: In dieser Schicht erfolgt die Verwaltung der dem DBMS zur Verfügung stehenden Hardware-Betriebsmittel. Der Geräte- und Speicher-Manager führt alle physischen Zugriffe zwischen Hauptspeicher und Externspeicher, auf dem die Datenbank und der Systemkatalog (data dictionary) persistiert sind, aus.

Schicht „Systempuffer-Manager (buffer manager)“: Hier wird der verfügbare Hauptspeicherbereich entsprechend verwaltet, sodass die gelesenen Blöcke so gespeichert werden, dass schnell auf diese zugegriffen werden kann. Auf Leseanforderung werden Daten von einem externen Speichermedium in den Puffer gebracht, auf Schreibanforderung Seiten des Puffers auf einem persistenten Medium gesichert.
Nach oben hin stellt dieser Manager Segmente mit sichtbaren Seitengrenzen als lineare Adressräume im Systempuffer zur Verfügung und abstrahiert damit von der physischen Speicherung in Dateien oder Blöcken.

Schicht „Zugriffspfad- und Record-Manager“: Der Zugriffspfad-Manager verwaltet externe Datenstrukturen zur Abspeicherung von und zum Zugriff auf Kollektionen von Datensätzen (zum Beispiel Indexstruktur B*-Baum). Der Record-Manager ist für die Verwaltung der internen Darstellung eines logischen Datensatzes und den effizienten Zugriff auf dessen Komponenten verantwortlich. Nach unten hin werden interne Datensätze und physische Zugriffspfade auf Seiten von Segmenten abgebildet.

Schicht „Recovery-, Transaktions- und Sperr-Manager“: Der Transaktions-Manager ist für die Verwaltung und die zeitliche Verzahnung von Transaktionen zuständig. Hierfür stehen ihm spezielle Kontrollmechanismen zur Verfügung. Eine wichtige Komponente in diesem Zusammenhang ist der Sperr-Manager, der von Transaktionen angeforderte Datenbankobjekte gemäß einem festgelegten Sperrprotokoll sperrt und damit sicherstellt, dass Änderungen an den entsprechenden Objekten nur im erlaubten Kontext dieser Sperrung erfolgen. Dabei kann es zu Konflikten kommen, sodass eine Transaktion abgebrochen und wieder zurückgerollt und später erneut gestartet werden muss. In diesem Fall sorgt der Recovery-Manager dafür, dass die Datenbank in den Zustand vor Start der betreffenden Transaktion zurückversetzt wird. Um alle Änderungen der Transaktion in der Datenbank zurücknehmen zu können, bedient sich der Recovery-Manager eines Logs, in dem Änderungen protokolliert worden sind. Des Weiteren ist es Aufgabe des Recovery-Managers im Fall von Systemfehlern für die Wiederherstellung der Datenbank in einem konsistenten Zustand zu sorgen. Eventuell dabei verloren gegangene Transaktionen müssen nachgefahren werden.

Schicht „Autorisierungs- und Integritäts-Manager, Update-Prozessor, Anfrage-Optimierer, Anfrage- und Kommando-Ausführer“: Benutzeranfragen und Änderungskommandos liegen an der oberen Schnittstelle dieser Schicht in interner Form (Zwischencode) vor. Für Anfragen und Kommandos führt der Autorisierungs-Manager eine Zugriffskontrolle gemäß im Systemkatalog befindlichen Autorisierungstabellen durch. Bei Änderungen, die vom Update-Prozessor bearbeitet werden, werden mit Hilfe des Integritäts-Managers die hinterlegten Integritätsbedingungen überprüft und damit die semantische Korrektheit der Datenbank automatisch überwacht. Nach dieser Prüfung wird die Anfrage an den Anfrage-Optimierer übergeben, der die interne Zwischenform in eine effizientere Form überführt, ohne dass das Ergebnis der Anfrage dabei verändert wird. Als nächstes werden Zugriffspläne (execution plans) durch den Update-Prozessor oder durch den Anfrage-Ausführer erstellt. Hierzu benutzen beide Komponenten Implementierungen von Zugriffsstrukturen und Operatoren des DBMS.

Schicht „Anfrage-Parser, DML-Präprozessor, DML- / DDL-Übersetzer, Ausgabe-Generierung“: Die oberste Schicht stellt nach oben hin eine mengenorientierte Schnittstelle für die verschiedenartigen Benutzerschnittstellen bereit. Wir unterscheiden hier Anfragen, die über eine Anwendungsschnittstelle gestellt werden und die bereits in Anwendungsprogrammobjectcode übersetzt worden sind und interaktiven Anfragen, die durch geübte Benutzer, Anwendungsprogrammierer oder Datenbankadministratoren gestellt werden. Adhoc-Anfragen erfolgen mittels einer deskriptiven, nicht-prozeduralen Anfragesprache, wie zum Beispiel SQL. Charakteristisch für solche Sprachen ist, dass sie das zu lösende Problem beschreiben, aber nicht den Lösungsweg beschreiben. Es ist Aufgabe des DBMS mittels implementierten Algorithmen einen Lösungsweg zu finden und Mengen- und Datenobjekte zu liefern, die die Anfrage erfüllen. Adhoc-Anfragen werden mit Hilfe des Anfrage-Parsers einer lexikalischen Analyse und einer Syntaxanalyse unterzogen und bei Korrektheit in eine äquivalente, interne Form (Syntaxbaum) überführt. Die Interaktion mit Anwendungsprogrammierern erfolgt mittels einer Datenmanipulationssprache (DML), die in die Programme eingebettet ist und mit dem DML-Präprozessor herausgefiltert und dann mittels DML-Übersetzer in Objectcode übersetzt wird. Der DDL-Übersetzer verarbeitet Schemadefinitionen, die in einer Datendefinitionssprache (DDL) spezifiziert sind, und speichert die Beschreibungen der Schemata im Systemkatalog.

Systemkatalog-Manager: Der Systemkatalog-Manager ist keiner speziellen Schicht zugeordnet. Er steht mit fast allen Komponenten des DBMS in Verbindung und verwaltet den Systemkatalog (data dictionary). Der Systemkatalog ist wie die Datenbank auf einem externen Speichermedium abgelegt und enthält Meta-Daten über die Struktur der Datenbank.

B Implementierte Lösungen - Dokumentation

B.1 Graphgenerator: Übersicht Pakete und Klassen

Für die Erstellung des Graphgenerators wurden keine zusätzlichen Bibliotheken eingebunden, sondern lediglich die Java SE 8 genutzt. Im folgenden geben wir eine Übersicht über die Aufteilung der Pakete und Klassen, siehe Abbildung 47.

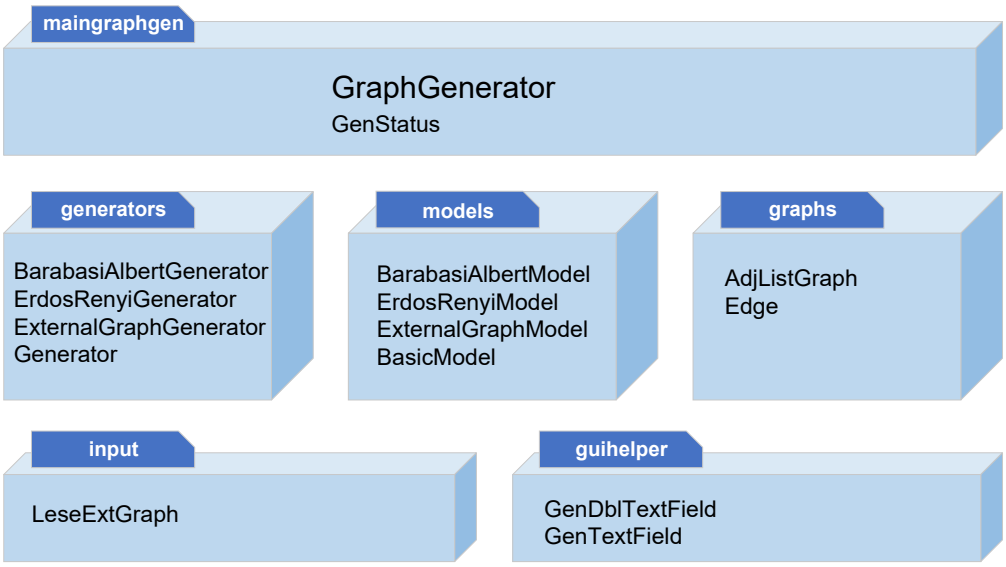


Abbildung 47: Graphgenerator: Übersicht der Pakete und Klassen.

In der Klasse `GraphGenerator` wird das Hauptfenster der Anwendung verwaltet, das Menüeinträge zum Starten der Generatoren, zum Sichern von Bildern in PNG-Format, sowie das Canvas für die graphische Darstellung und eine Status-Leiste bereitstellt. Der Status für die Statusleiste wurde in einer eigenen Klasse als `Enum` definiert. Die Dialogfunktionen für die Generatoren selbst sind in den Klassen mit der Endung Generator im Namen realisiert.
Im jeweiligen Generator wird ein neues Fenster zur Parametereingabe verwaltet, über das die Parameter des Generators erfasst werden können. Nach dem Betätigen des Buttons Generieren, wird die Eingabe geprüft, das zugehörige Modell (zum Beispiel BarabasiAlbertModel) instanziiert, die Startkonfiguration gesetzt und die Generierung der Graphen abhängig von den Eingabeparametern durchgeführt.

Im Gegensatz zu den anderen Generatoren wird beim `ExternalGraphGenerator` keine Generierung von Graphen durchgeführt, sondern nur einmalig die Methode `LeseGraph` aus dem Paket `input` aufgerufen. Der extern (aus dem Internet) geladene Graph, der in Form einer Textdatei vorliegt, wird mit dieser Methode gelesen und für die spätere Verarbeitung in den Experimenten aufbereitet. Dar-

über hinaus meldet der jeweilige Generator seinen Status für die Anzeige an das Hauptfenster.
Jeder Generator baut auf einem Basis-Generator auf, der den Namen `Generator` trägt. In diesem sind Methoden realisiert, die von allen Generatoren gemeinsam genutzt werden. Weiterhin werden hier auch gemeinsame Labels und Buttons definiert, sowie Methoden zum Setzen eines Hintergrundes oder zum Aktivieren und Deaktivieren von Buttons.

Im Folgenden wird eine Übersicht über die wichtigsten Methoden der Generatoren in den Tabellen 18 und 19 gegeben:

Tabelle 18: Wichtigste Methoden der abstrakten Klasse Generator.

Methode	Beschreibung
saveGraphToFile	Schreiben des übergebenen Graphen und seiner Konfiguration in Datei.
saveConfigData	Speichern der Konfigurationsdaten des Graphen in Datei. Wird von Methode saveGraphToFile aufgerufen.
saveGraphData	Speichern der Graphdaten in Datei. Wird von Methode saveGraphToFile aufgerufen.

Tabelle 19: Methoden, die von allen Generatoren implementiert werden.

Methode	Beschreibung
generate	Generiert Graphen basierend auf ausgewähltem Modell und der zuvor geprüften Eingabeparametern.
genDynFileName	Generiert dynamischen Dateinamen.
collectGenParams	Speichert spezifische Graph-Parameter als String.
checkInput	Überprüfung der Eingabeparameter.
initLabels	Initialisierung der Labels.
initTextFields	Initialisierung der Eingabefelder.
initButtons	Initialisierung der Buttons und Button-Events.
setGridLayout	Initialisierung des Grid-Layouts für die Anzeige.

Es folgt eine Beschreibung der wichtigsten Methoden des jeweiligen Modells, beginnend mit den Methoden des `BasicModel`, siehe Tabelle 20.

Die Methoden `displayCircleNodes` und `displayCircleGeneration` werden vom `ErdosRenyiGenerator` und dem `BarabasiAlbertGenerator` nach jeder Iteration aufgerufen, falls die eingegebene Knotenzahl kleiner oder gleich Tausend Knoten ist. Dadurch kann man kleinere Graphen praktisch wachsen sehen.

Tabelle 20: Wichtigste Methoden der abstrakten Klasse BasicModel.

Methode	Beschreibung
initGraph	Initialisierung des Graphen.
calculateCoordinates	Berechnung der Koordinaten für die Anzeige.
calculateArrowHead	Berechnung der Pfeilspitzen für die Anzeige.
calculateWeight	Berechnung des Gewichts einer neuen Kante.
displayCircleNodes	Anzeige der Knoten des Graphen.
displayCircleGeneration	Anzeige des Graphen nach einer Iteration.
calculateOffset	Berechnet Offsets für Anzeige Knotennummern.
genConfigSatz	Generieren des Satzes, der in die Konfigurationsdatei geschrieben werden soll.
calcConfPath	Bestimmt zufälligen Startknoten und durch weitere Navigation im Graphen einen Zielknoten, zu dem ein Pfad existiert.

Zusätzlich zu den Methoden der Klasse `BasicModel` gibt es noch zwei weitere Methoden, die in den Klassen `ErdosRenyiModel` und `BarabasiAlbertModel` implementiert sind: `setStartConfig` und `calculateNextGeneration`. Mit `setStartConfig` wird die Startkonfiguration des ausgwählten Modells gesetzt. Die Methode `calculateNextGeneration` berechnet die nächste Iteration und damit die nächste Generation des Graphen im Generierungsprozess. Die Klasse `ExternalGraphGenerator` implementiert ebenfalls eine Methode namens `setStartConfig`.

B.2 Stand-Alone-Lösung: Übersicht Pakete und Klassen

Im folgenden geben wir eine Übersicht über die Aufteilung der Pakete und Klassen, siehe Abbildung 48.

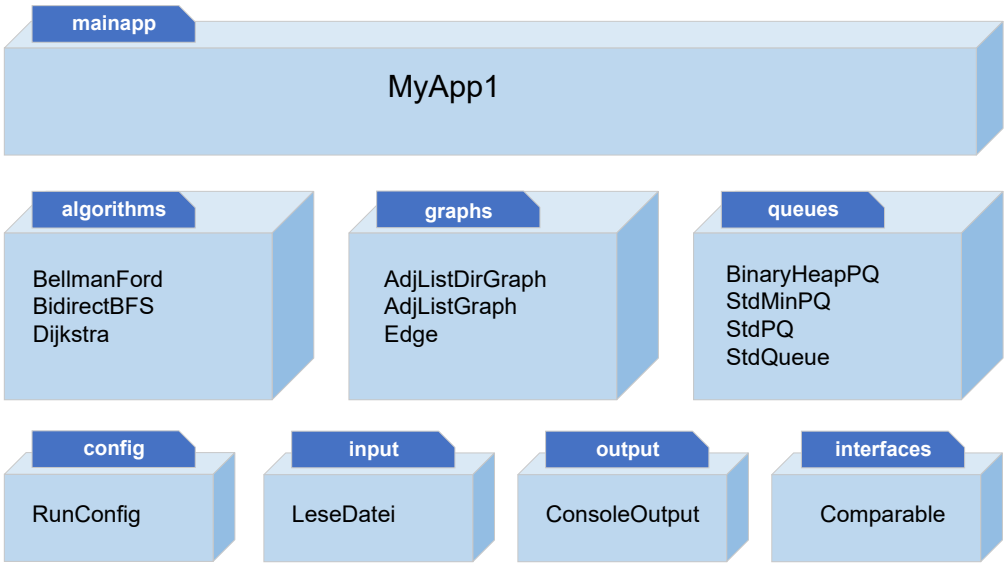


Abbildung 48: Stand-Alone-Lösung: Übersicht der Pakete und Klassen.

Paket algorithms, Methoden der Klasse BidirectBFS

In der Klasse `BidirectBFS` gibt es fünf Metheoden, die in Tabelle 21 kurz beschrieben werden. Die Methode `doBFS` implementiert die Standard-Breitensuche in einem Graphen von einem gegebenen Startpunkt aus. Diese wird allerdings nicht für die Experimente eingesetzt.

Der Einsatz der anderen in der Tabelle 21 aufgeführten Methoden ist abhängig davon, ob es sich um einen gerichteten oder ungerichteten Graphen handelt.

Beim gerichteten Graphen, der als `AdjListDirGraph` übergeben wird, wurde zusätzlich zu der üblichen Adjazenzliste mit den direkten Nachfolgern noch eine weitere Adjazenzliste mit den Vorgängerknoten angelegt. Diese wird bei der Suche vom Zielknoten ausgehend in Richtung Startknoten eingesetzt, um effizient direkte Vorgängerknoten zu finden, von denen man zum Zielknoten in der durch die Kante vorgegebenen Richtung navigieren kann.

Tabelle 21: Methoden der Klasse BidirectBFS.

Methode	Graph gerichtet	Beschreibung
doBFS	-	Breitensuche im Graphen mit vorgegebenen Startknoten, Rückmeldung Laufzeit.
doBidirectBFSUndirGraph	-	Bidirektionale Breitensuche im Graphen mit vorgegebenem Startknoten, Rückmeldung Laufzeit.
doBiBFSPartUndirGraph	-	Methode, die als Teilsuche von doBidirectBFSUndirGraph aufgerufen wird. Je nach Suchrichtung wird Start- oder Zielknoten übergeben.
doBidirectBFSDirGraph	X	Bidirektionale Breitensuche im gerichteten Graphen mit vorgegebenem Startknoten, Rückmeldung Laufzeit.
doBiBFSPartDirGraph	X	Methode, die als Teilsuche von doBidirectBFSUndirGraph aufgerufen wird. Je nach Suchrichtung wird Start- oder Zielknoten übergeben.

Die Methoden `doBidirectBFSUndirGraph` und `doBidirectBFSDirGraph` berechnen den kürzesten Pfad in einem ungewichteten Graphen zwischen dem übergebenen Startknoten und dem Zielknoten. Dafür führen sie eine Breitensuche in beiden Richtungen aus, eine ausgehend vom Startknoten und eine zweite ausgehend vom Zielknoten, wie in Kapitel 4.2 beschrieben. Dabei wird die für die Ausführung des Algorithmus gemessene Laufzeit als Ergebnis zurückgegeben. Als Parameter werden der Graph, der Startknoten, der Zielknoten und zwei weitere Parameter namens `outputOK` und `outputMD` zur Steuerung der Ausgabe übergeben.

Der Parameter `outputOK` ist ein Schalter, der, wenn er auf **true** gesetzt ist, die Ausgabe des gefundenen Pfades nach der Berechnung auf der Konsole veranlasst. Mit Hilfe dieses Schalters wurde, zum Beispiel bei kleineren Graphen, die Korrektheit des Ergebnisses für die in der Stand-Alone-Lösung implementierten Algorithmen getestet. Der Wert von `outputOK` wird in der Konfigurationsdatei zum Graphen gesetzt, die als Steuerung für Ausführung der Stand-Alone-Lösung und die Messung der Laufzeiten dient. Weiterhin gibt es den Parameter `outputMD`, mit dem gesteuert wird, ob die Ausgabe der gemessenen Laufzeit

in die Messdateien im Verzeichnis `EXPERIMENT_HOME/measure/runxxx` geschrieben wird. Messdaten werden also nur geschrieben, wenn `outputMD` in der Konfigurationsdatei auf **true** gesetzt ist.

Wird bei der bidirektionalen Breitensuche ein gewichteter Graph übergeben, so werden die Kantengewichte des Graphen bei der Ausführung der Methoden ignoriert.

Paket algorithms, Methoden der Klasse Dijkstra

Die Klasse Dijkstra implementiert zwei Methoden namens `doIntDijkstra` und `doDoubleDijkstra`. Mit diesen Methoden wird der kürzeste Pfad in einem Graphen von einem gegebenen Startpunkt aus zu allen anderen Knoten im Graphen mit dem Dijkstra-Algorithmus berechnet, wie im Kapitel 4.2 beschrieben. Beide Methoden erwarten daher einen gewichteten Graphen, bei dem die Kantengewichte nicht-negativ sind. Bei ungewichteten Graphen wird das Kantengewicht standardmäßig mit dem Wert 1 vorbelegt. Die Methode `doIntDijkstra` arbeitet mit ganzzahligen Kantengewichten, während die Methode `doDoubleDijkstra` mit Kantengewichten vom Typ Double rechnet. Da wir für die Experimente nur Graphen mit ganzzahligen Gewichten ausgewählt haben, kommt hier nur Methode `doIntDijkstra` zum Einsatz. Wie bei der bidirektionalen Breitensuche gibt es auch hier die Parameter `outputOK` und `outputMD` zur Steuerung der Ausgabe, deren Wert auch hier in der Konfigurationsdatei festgelegt wird.

Weiterhin gibt es den Parameter `typePQ`, mit dem die gewünschte Prioritätswarteschlange, die für die Berechnung verwendet werden soll, übergeben wird. Hat dieser Parameter den Wert 1, so wird ein binärer Heap verwendet, wie er in Kapitel 4.2.2 vorgestellt wurde. Wird hingegen `typePQ` mit Wert 0 übergeben, so wird eine Standard-Min-Prioritätswarteschlange verwendet. Für die Experimente zur Messung der Laufzeit wird allerdings nur `typePQ` mit Wert 1 genutzt, da die Berechnung des kürzesten Pfades damit performanter erfolgen kann, siehe Ausführungen im Kapitel 4.2.2, und die damit bessere Implementierung für den Vergleich der Performanz mit der Neo4j-Graph-DB herangezogen wird.

Paket algorithms, Methoden der Klasse BellmanFord

Die Klasse BellmanFord implementiert drei Methoden, die den kürzesten Pfad in einem vorgegebenen Graphen von einem gegebenen Startpunkt aus mit dem Bellman–Ford-Algorithmus, wie im Kapitel 4.2 beschrieben, berechnen. Als Gewichte werden hier positive und negative Kantengewichte zugelassen, wobei vorausgesetzt wird, dass es keinen negativen Zyklus im Eingabegraphen, gemäß Festlegungen zum betrachteten kürzesten-Pfad-Problem im Kapitel 4, gibt.
In den hier implementierten Methoden wird daher auch keine Prüfung auf einen negativen Zyklus im Graphen vorgenommen. Sollte trotzdem ein solcher Graph als Eingabegraph verwendet werden, wird dies zu einer Endlosschleife führen. In einer früheren Version der Stand-Alone-Lösung waren auch Varianten des

Bellman–Ford-Algorithmus implementiert, die einen negativen Zyklus feststellen. Diese wurden jedoch wieder entfernt, da sie für die eigentliche Durchführung der Experimente nicht benötigt werden.

Von den verbliebenen drei Methoden wird jedoch nur die Methode mit Namen `doNoNegCycleIntBellmanFord` für die Experimente eingesetzt. Das liegt daran, dass wir bei gewichteten Graphen nur Graphen mit ganzzahligen Gewichten für die Messung der Laufzeiten einsetzen, siehe Kapitel 7.3 zur Beschreibung der Auswahl der Eingabegraphen.
Die Methode `doNoNegCycleDoubleBellmanFord` arbeitet mit Kantengewichten mit Gleitkommazahlen und die Methode `doNoNegCycleBellmanFord` ist eine generische Methode, die sowohl mit ganzzahligen Gewichten, als auch mit Gleitkommagewichten ausgeführt werden kann. Trotzdem wird diese generische Methode in den Experimenten nicht eingesetzt, da sich bei Vorab-Tests, insbesondere bei kleineren Graphen bis 1000 Knoten, durch Vergleich der Laufzeiten herausgestellt hat, dass die Ausführung dieser Methode etwas langsamer ist.

Alle aufgeführten Methoden, die zur Berechnung des kürzesten Pfades in dem übergebenen Graphen von einem gegebenen Startknoten aus, in der Klasse `BellmanFord` eingesetzt werden, nutzen eine Standard-Prioritätswarteschlange, die nach dem FIFO-Prinzip (first in first out) arbeitet. Dies ist möglich, da bei dem implementierten Algorithmus keine Sortierung nach den bisher kürzesten Distanzen in der Warteschlange erforderlich ist. Das hat zur Folge, dass auch kein Aufwand für die Suche eines Eintrage oder die Sortierung der Einträge in der Warteschlange anfällt.

Paket input, Methoden der Klasse LeseDatei

Im Paket `input` sind Methoden für das Lesen von Verzeichnissen, von Konfigurationsdateien und von Eingabegraphen implementiert. Mit der Ausführung der Stand-Alone-Lösung werden zu Beginn mit der Methode `leseVerzeichnis` die Dateinamen der Konfigurationsdateien im Unterverzeichnis `config` des Verzeichnisses `EXPERIMENT_HOME` gelesen und als Dateiliste vom Typ `ArrayList` übergeben. Diese dient dann zur Steuerung des weiteren Ablaufs der Berechnungen und Messungen, die mit der Stand-Alone-Lösung ausgeführt werden.

Mit der Methode `leseKonfiguration` wird der Inhalt der aktuell bearbeiteten Konfigurationsdatei eingelesen und nach Prüfungen der Eingabe in einer globalen Instanzvariable `cfg` vom Typ `RunConfig`, wie in der Klasse `Runconfig` definiert, gespeichert.

Für das Einlesen einer Graphdatei sind drei Methoden implementiert: `leseGraph`, `leseDoubleGraph` und `leseDirGraph`. Durch die Typisierung, die durch das Lesen der Konfigurationsdatei zuvor erfolgt, wird, je nach Typ des Graphen, die entsprechende Methode zum Einlesen des Eingabegraphen in die Datenstruktur

AdjListGraph bzw. AdjListDirGraph ausgeführt. Dabei wird die Methode leseDirGraph nur zum Einlesen von ungewichteten, gerichteten Eingabegraphen verwendet, während die Methode leseDoubleGraph für das Einlesen von Graphen mit Kantengewichten vom Typ Double ausgeführt wird.
In den meisten Fällen kommt jedoch nur die Methode leseGraph zum Einsatz, da wir bei den Experimenten bei gewichteten Graphen nur Graphen mit ganzzahligen Kantengewichten zulassen.

Paket output, Methoden der Klasse ConsoleOutput

Tabelle 22 gibt eine Übersicht der Methoden, die in der Klasse ConsoleOutput implementiert sind. Die Methode outputConfig wird bei jeder aktuell von der Stand-Alone-Lösung bearbeiteten Konfigurationsdatei als Information auf der Konsole ausgegeben. Dadurch weiß man bei einem Messlauf, welcher Eingabegraph gerade mit welchem Algorithmus bearbeitet wird.

Tabelle 22: Methoden der Klasse ConsoleOutput.

Methode	Beschreibung
outputConfig	Ausgabe des Inhalts der aktuell bearbeiteten Konfigurationsdatei.
outputPathlist	Ausgabe des in einer ArrayList übergebenen Pfades.
outputPath	Überladene Methode, die den berechneten kürzesten Pfad mit oder ohne Distanz ausgibt.
outputRunTime	Ausgabe der Laufzeit gemäß übergebener Start- und Endezeit in Einheiten ns, ms und s.
doOutAdjList	Ausgabe der Adjazenzliste für Testzwecke.

Die Methode outputRunTime und doOutAdjList wurden nur während der Tests der Anwendung als Hilfen benutzt. Die Methode outputPathList wird speziell bei der bidirektionalen Breitensuche eingesetzt, da hier der Pfad vor der Ausgabe in einer ArrayList aus den Einzelberechnungen zusammengesetzt wird. Im Allgemeinen wird die Ausgabe des kürzesten Pfades mit der Methode outputPath ausgeführt, vorausgesetzt der Parameter outputOK ist auf true gesetzt. Diese Methode ist überladen, sodass entweder nur die gefunden kürzesten Pfade ausgegeben werden können oder zusätzlich auch die ermittelte Distanz vom Startknoten zum jeweiligen Zielknoten.

B.3 PrepNoe4jGraphs: Übersicht Pakete und Klassen

Im folgenden geben wir eine Übersicht über die Aufteilung der Pakete und Klassen von `PrepNeo4jGraphs`, siehe Abbildung 49.

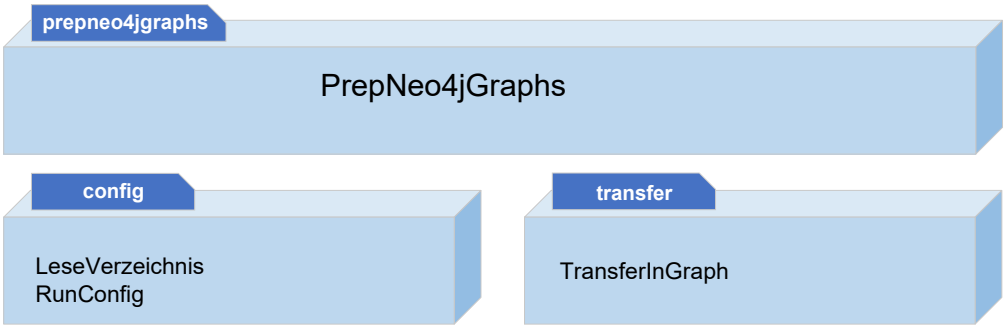


Abbildung 49: PrepNeo4jGraphs: Übersicht der Pakete und Klassen.

In der Klasse PrepNeo4jGraphs werden mit Hilfe von Methoden der Klasse `LeseVerzeichnis` die Verzeichnisse mit den Konfigruationsdateien gelesen und in einer Schleife abgearbeitet. Dabei wird, falls beim Programmstart nicht der Parameter `-S` angegeben wurde, mit der Methode `transferInGraph` die Graphdatei gelesen und im Noe4j-Importformat in das Verzeichnis `input` geschrieben, wie im vorherigen Kapitel erläutert.

Danach werden die Skript-Dateien erzeugt, mit denen der Messlauf mit Neo4j gestartet und die Laufzeit gemessen wird. Die Klasse `TrasferInGraph` implementiert fünf Methoden, die in Tabelle 23 kurz beschrieben werden.

Tabelle 23: Übersicht Methoden der Klasse TransferInGraph.

Methode	Beschreibung
leseKonfiguration	Lesen der aktuellen Konfigurationsdatei
outputConfig	Anzeige der Inhalte der aktuell bearbeiteten Konfigurationsdatei
transferGraph	Erstellen Import-Dateien für Neo4j-Import-Dateien
buildRunScripts	Erstellung der Einzelskripte für Import und Messlauf pro Graphdatei für aktuelles Environment
mergedRunScripts	Zusammenfassung aller Skripte für Mess-Experiment in einem Sammelskript für aktuelles Environment

B.4 PrepResults: Übersicht Pakete und Klassen

Mit der Ausführung von `PrepResults` werden die gesammelten Laufzeiten der Stand-Alone-Lösung mit den mit Neo4j gesammelten Messdaten zusammengeführt.
Die Klasse ist in zwei Paketen organisiert. Im Paket `prepresults` befindet sich das Hauptprogramm `PrepResults`, das zwei Methoden der Klasse `ConcatMeasure` aufruft, die sich im Paket `input` befindet, siehe Abbildung 50.

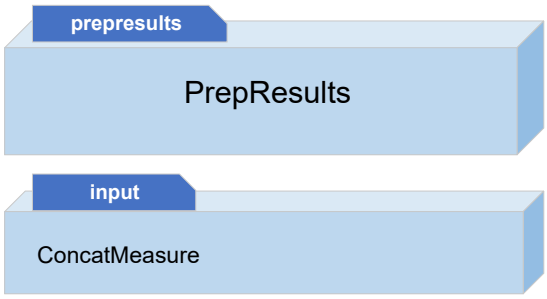


Abbildung 50: PrepResults: Übersicht der Pakete und Klassen.

Zunächst werden mit der Methode `leseDateinamen` die Namen der Messdateien im beim Programmstart übergebenen Messverzeichnis gelesen und anschließend mit der Methode `concatFileContents` dann die Zusammenführung der Messdaten je Eingabegraph und Algorithmus durchgeführt. Das erwartete Eingabeformat der Messdatei der Stand-Alone-Lösung ist in Abbildung 16 dargestellt. Die Messdaten mit der Neo4j-Graph-DB liegen in einem in Abbildung 51 dargestelltem Format vor, sodass während des Zusammenführens zunächst die interessierenden Informationen aus der entsprechenden Datei herausgefiltert werden müssen.

```
+--------------+
| evalDuration |
+--------------+
| 2412         |
+--------------+

1 row available after 3030 ms, consumed after another 2 ms
+--------------+
| evalDuration |
+--------------+
| 2276         |
+--------------+

1 row available after 2520 ms, consumed after another 0 ms
+--------------+
| evalDuration |
+--------------+
| 2265         |
+--------------+

1 row available after 2391 ms, consumed after another 1 ms
```

Abbildung 51: PrepResults: Beispiel Neo4j-Messdatei.

Erklärung

Name:	Gundula Swidersky
Matrikel-Nr.:	8913447
Fach:	Informatik
Modul:	Bachelorarbeit

Ich erkläre, dass ich die vorliegende Abschlussarbeit mit dem Thema

Vergleich der Performanz bei kürzesten-Pfad-Berechnungen zwischen Stand-Alone-Lösungen und Graphdatenbanken

selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich, inhaltlich oder sinngemäß entnommenen Stellen als solche den wissenschaftlichen Anforderungen entsprechend kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate überprüft und ausschließlich für Prüfungszwecke gespeichert wird.

Datum: _____ **Unterschrift:** _____