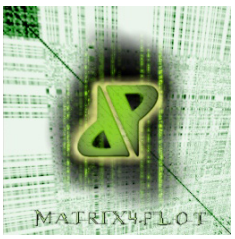

UNIVERSITY OF APPLIED SCIENCES FACHHOCHSCHULE GIESSEN FRIEDBERG

Diplomarbeit

Visualisierung großer Datenmengen

Anwendung von Grid-Computing und des Information-Mural-Algorithmus am Beispiel des Eclipse-Plugins „Dotplot“

---

zur Erlangung des akademischen Grades

Diplom-Informatiker (FH)

vorgelegt dem Fachbereich Mathematik, Naturwissenschaften und Informatik
der Fachhochschule Gießen-Friedberg

Tobias Gesellchen

31. Mai 2005

Referent: Prof. Dr. Klaus Quibeldey-Cirkel
Korreferent: Prof. Dr. Oskar Hoffmann

Inhaltsverzeichnis

Vorwort

Diese Arbeit ist die Krönung eines wichtigen Abschnitts in meiner Ausbildung zum Informatiker. Ohne die Unterstützung einiger Personen wären das Studium und speziell diese Arbeit nicht möglich gewesen. Ich möchte hier die Gelegenheit nutzen, diesen Personen zu danken.

In erster Linie geht mein Dank an meinen Betreuer Prof. Dr. Klaus Quibeldey-Cirkel, der mir viel Vertrauen, aber auch sehr viele Ideen und Vorschläge entgegenbrachte.

Ich möchte mich bei Dipl.-Ing. (FH) Jens Foerster für die Vermittlung der kleinen Rechnerfarm bedanken, ohne die die Performancetests nicht realisierbar gewesen wären, und auch für die Ratschläge zur Verbesserung der Arbeit. Für die vielen Hinweise und Korrekturvorschläge möchte ich mich bei allen Korrekturleserinnen und -lesern bedanken.

Melanie Wagner gilt mein besonderer Dank. Sie hat mir die letzten Semester in Gießen verschönert und durch einmalige Erfahrungen bereichert.

Nicht zuletzt möchte ich meinen Eltern für die vielen Diskussionen und die Geduld danken. Es war nicht einfach, aber irgendwie haben wir uns immer wieder arrangiert.

Tobias Gesellchen

Gießen, Mai 2005

Kapitel 1

Einleitung

Die Idee zu dieser Arbeit entstand während des Schwerpunktpraktikums „Java für Fortgeschrittene“ an der Fachhochschule Gießen-Friedberg im Sommersemester 2004. Dort wurde ein *Eclipse-Plugin*[1] zur Erzeugung von Dotplots entwickelt, die – grafisch aufbereitet – Gemeinsamkeiten innerhalb einer analysierten Menge von *Tokens*[2] aufzeigen sollen.

Besonderes Augenmerk lag auf der Analyse von Quelltexten aus der Rechnerprogrammierung. Deshalb verfügt das Plugin über spezielle Einstellmöglichkeiten, die nur auf Programmquelltexte anwendbar sind, wie beispielsweise das Ausblenden von Schlüsselwörtern. Trotzdem kann das Plugin auch für jegliche anderen Textarten verwendet werden.

Die interne Datenstruktur zur Speicherung der Matches jedes Vergleichs ist die sogenannte *F-Matrix*[3]. Der Begriff *Match* bezeichnet die Treffer, er wurde aus [1] übernommen. Den Matches werden Gewichtungen zugeordnet, die im Dotplot einer Graustufe entsprechen. Dem Benutzer wird die Möglichkeit gegeben, eine *Lookup Table* (LUT), also eine Farbskala, auszuwählen, die die Graustufen im Quellbild auf einen Farbwert im Zielbild abbildet. So können interessante Stellen noch stärker hervorgehoben werden.

Durch die Art und Weise, wie die Gemeinsamkeiten im Ergebnisbild erzeugt und dargestellt werden, treten Performanceprobleme im Bereich der Bildverarbeitung auf. Speicherverbrauch und Prozessorleistung sind die begrenzenden Faktoren. Durch Swap-Mechanismen des Betriebssystems, das Auslagern von Arbeitsspeicher auf die Festplatte, trägt auch die geringe Festplattengeschwindigkeit zum

[1] Eclipse ist eine erweiterbare Plattform, hauptsächlich verwendet als integrierte Entwicklungsumgebung. Die Erweiterungen werden *Plugins* genannt.

[2] Die kleinsten Einheiten, mit denen gearbeitet wird. In der Regel sind dies Wörter, für manche Anwendungen sind grammatische Sätze oder Programmanweisungen als Tokens zu behandeln.

[3] siehe auch [11, Seite 9]

Performanceverlust bei. Solche Probleme kann man durch unterschiedliche Ansätze lösen. Diese Arbeit zeigt zwei Möglichkeiten auf.

1.1 Aufgabenstellung

Die Diplomarbeit beschäftigt sich mit der Implementierung des Information-Mural-Algorithmus und der Verwendung eines Rechnernetzes, dem sogenannten Grid-Computing, zur Lastverteilung bei der Berechnung und Erzeugung großer Dotplots. Es wird gezeigt, wie viel Performance durch beide Methoden gewonnen werden kann. Ziel ist es, den vorhandenen Arbeitsspeicher und die Prozessorleistung effizient zu nutzen.

Da das Information Mural bereits in die offizielle Version des Plugins eingeflossen ist, kann auf positive Erfahrungswerte zurückgegriffen werden. Kapitel 5 zeigt, dass der Performancegewinn groß ist und eine Verwendung Sinn macht. Leider gilt dies bisher nur für die Arbeit am Bildschirm, denn beim Export in Dateien werden noch Standardtechniken verwendet.

Die Verwendung eines Grids soll den Speicherbedarf pro beteiligtem Rechner verringern. Einziger Nachteil ist das anschließende Zusammenfügen der einzelnen Teilbilder: Der Server muss von allen angeschlossenen Clients die Ergebnisse sammeln und zu einem Gesamtbild vereinen, wozu noch viel Rechenleistung und Speicher auf dem Server notwendig sind. Dem Anwender muss es darüber hinaus auf einfachste Weise möglich sein, das Grid aufzubauen, um mit dessen Hilfe Dotplots zu erstellen.

Es wird hier auf die Plugin-spezifischen Probleme eingegangen, die erwähnten Techniken lassen sich auch leicht auf andere Umgebungen umsetzen. Die Grundlage für die Implementierung des Grids bildet der in [5] vorgestellte Rahmen zum Nachrichtenaustausch zwischen den beteiligten Rechnern im Netz. Er wurde an einigen Stellen angepasst und das Softwaredesign flexibler gestaltet.

1.2 Kapitelübersicht

Als Orientierungshilfe für den Leser folgt eine kurze Übersicht über die in diesem Dokument enthaltenen Kapitel.

Kapitel 1 führt den Leser in die behandelten Themen ein, gibt einen Überblick über die angestrebten Ziele und definiert die Schriftkonventionen.

Kapitel 2 dient der Beschreibung von Dotplots. Ihr Zweck und spezifische Muster werden erläutert. Anhand von diversen Beispielen wird die Anwendung dieser Technik verdeutlicht.

Kapitel 3 behandelt das erste Hauptthema der Arbeit, das Grid-Computing. Verschiedene Möglichkeiten zur Implementierung werden kurz vorgestellt und der in dieser Arbeit eingeschlagene Weg aufgezeigt.

Kapitel 4 zeigt die Probleme bei der Skalierung von Bildern, stellt den Algorithmus „Information Mural“ vor und gibt Details zur Implementierung. Danach wird auf die Arbeit mit dem Information Mural und auf Anwendungen außerhalb des Dotplot-Plugins eingegangen.

Kapitel 5 diskutiert die Ergebnisse aus den Kapiteln 3 und 4. Ein Schwerpunkt wurde auf den Performancevergleich der vorgestellten Techniken gesetzt, da dies die Hauptmotivation der Arbeit darstellt.

Kapitel 6 fasst die Ergebnisse der Arbeit zusammen und macht Vorschläge für zukünftige Erweiterungen des Plugins.

Im Anhang befinden sich eine kurze Einführung in die Arbeit mit dem Plugin und eine Übersicht zur Begleit-CD.

1.3 Schriftkonventionen

Die verwendeten Schrifthervorhebungen werden hier kurz erläutert.

Kursive Schrift wird als Hervorhebung wichtiger Begriffe, Namen und Definitionen verwendet.

Beispiele: Programmiersprache *C*; Die Erweiterungen werden *Plugins* genannt.

`Nichtproportionalschrift` wird für Programmcode, Variablen, Klassen- und Methodennamen verwendet.

Beispiel: Er leitet die Teilbilder mit dem Signal `onImageReceived` an den `GridPlotter` weiter.

Kapitel 2

Dotplots

Dotplots wurden ursprünglich für die Genforschung entwickelt. Dort interessiert man sich für Gemeinsamkeiten im Genpool verschiedener Arten von Lebewesen. Die dabei gewonnenen Erkennisse lassen sich auf verschiedene Weisen anwenden. Man erhält Lösungsansätze bei der Entwicklung von Medikamenten, kann Erbkrankheiten besser verstehen und allgemein auf den Grad der Verwandtschaft der untersuchten Arten schließen [1, Seite 2].

In der Genforschung wird bei der Erzeugung von Dotplots die Darstellung der Gene durch die vier grundlegenden Aminosäuren Guanin, Cytosin, Adenin und Thymin miteinander verglichen. Eine ähnliche Anordnung der Aminosäuren bei zwei verschiedenen Arten kann auf gleiche Funktionen oder Wirkungen für das entsprechende Lebewesen hinweisen. So will man auf einen Code schließen, der es erlaubt, bei gleichartigen Anordnungen an anderen Stellen die Auswirkungen besser zu verstehen.

Verallgemeinert man diese Technik, kann man diese vier Aminosäuren als Wörter in der Sprache der Gene auffassen. Man kann also theoretisch jede Sprache mit Hilfe von Dotplots analysieren. Eine Sprache lässt sich in ihre einzelnen Wörter, sogenannte Tokens, zerlegen. Jedes Token muss dann als Datenstruktur in einem Rechner darstellbar sein, um mit einem entsprechenden Programm ein Dotplot erzeugen zu können.

Bei der Generierung eines Dotplots wird eine Matrix aufgespannt, die, auf jeder Achse verteilt, alle Tokens aufreiht. Jedes Token einer Achse wird dann mit jedem Token der anderen Achse verglichen. Sind zwei Tokens T_i und T_j gleich, wird an der Koordinate (i, j) in der Matrix eine Markierung gesetzt. Später wird, für die Darstellung am Bildschirm oder für die Ausgabe in eine Datei, eine Grafik erzeugt, die anstelle jeder Markierung einen Punkt zeigt. Man erhält so eine komprimierte Darstellung der Quelldaten.

Die Betrachtung eines solchen Dotplots lässt schnell erkennen, welche Bereiche Gemeinsamkeiten aufweisen. Dichter mit Bildpunkten besetzte Stellen weisen eine höhere Trefferquote auf. Bei einer entsprechenden Implementierung kann man die interessanten Bereiche direkt in den Quelldaten anzeigen lassen.

Der Algorithmus kann, um eine Überladung mit Informationen zu vermeiden, so erweitert werden, dass Treffer mit häufig vorkommenden Wörtern einen geringeren Wert oder *Gewichtung* erhalten [1, Seite 8]. Weniger Gewichtung bedeutet bei der Erzeugung des Dotplots, dass mit verschiedenen Helligkeitsstufen oder auch Farben gearbeitet wird. Höherwertige (seltene) Treffer erhalten dann eine auffälligere Farbe als ein Treffer von häufigen Wörtern.

Abbildung 2.1 verdeutlicht die Konstruktion eines Dotplots. Links zur Verdeutlichung der Technik nur ein Satz, rechts der Vergleich aller Werke Shakespeares (siehe auch die vergrößerte Abbildung 2.9 auf Seite 10). Die notwendigen Kenntnisse zur Analyse eines Plots, wie er auf der rechten Seite gezeigt wird, vermittelt der folgende Abschnitt.

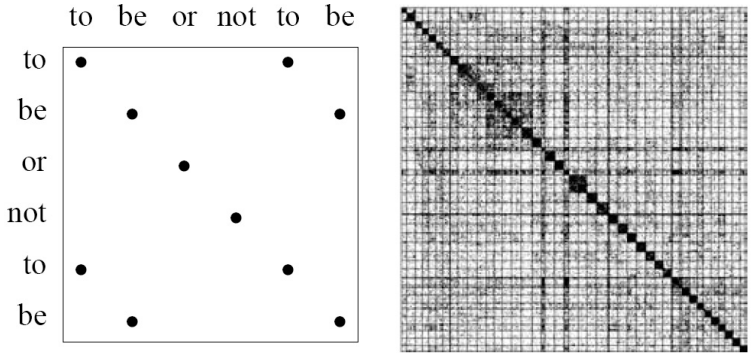

Abbildung 2.1: Zwei Texte von Shakespeare, dargestellt durch Dotplots [9, Seite 1].

2.1 Dotplot-Muster

In der aktuellen Implementierung des Plugins wird jede selektierte Datei mit *jeder* anderen verglichen. Das impliziert, dass auch Selbstvergleiche durchgeführt werden. Resultat ist die markante Hauptdiagonale in jedem Plot, von oben links nach unten rechts. Die Hauptdiagonale ist jedoch nicht das einzige Muster, das beim Betrachten eines Dotplots auffällt. Es entstehen häufig weitere Strukturen, die auf spezielle Arten von Gemeinsamkeiten oder Unterschieden hinweisen. Kennt man diese Muster, kann man ein Dotplot gezielt auswerten.

Im Folgenden werden häufig auftretende Muster näher erläutert. Einige der durch die synthetischen Grafiken gezeigten Muster werden durch Original-Plots veran-

schaulicht. Die Beispiele im Abschnitt 2.2 zeigen eine Kombination solcher Muster, es entstehen so genannte *Texturen*.

Die Grundlagen für alle Muster werden in Abbildung 2.2 gezeigt. Ein einzelner Block indiziert eine hohe Dichte von Treffern, wobei die Wörter nicht zwingend sortiert wurden. Ein solches Muster kann bei der Analyse von größeren Texten entstehen, die aus einem gemeinsamen Vokabular stammen. Ein Wechsel des Vokabulars resultiert in einem weiteren Block, wie die Abbildung zeigt [8]. Diagonalen entstehen erst bei einer beliebigen Anordnung von Wörtern. Die Hauptdiagonale stellt eine entsprechende *Anordnung* dar[1]. Wenn sich Teile einer sortierten Reihe von Wörtern wiederholen, entstehen die in Abbildung 2.2 gezeigten Nebendiagonalen [1, Seite 1].

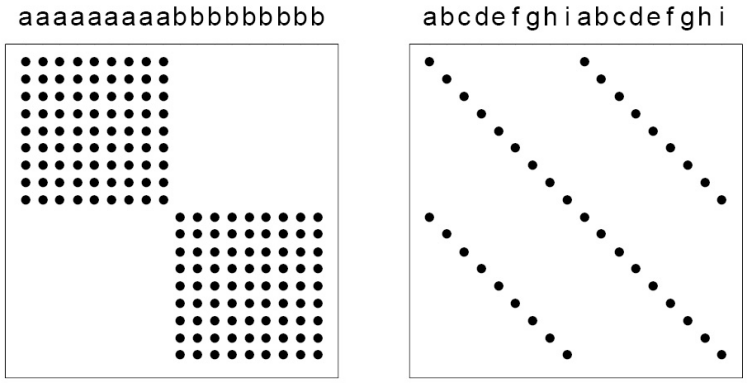

Abbildung 2.2: Die grundlegenden Muster sind Blöcke (links) und Diagonalen (rechts) [7, Seite 3].

Werden Wörter zusätzlich zu einer Anordnung in der umgekehrten Anordnung wiederholt[2], entstehen Diagonalen senkrecht zur Hauptdiagonalen, Abbildung 2.3 zeigt ein Beispiel. Bei der Analyse der Boot- und Shutdown-Scripte eines Unix-Systems entstehen solche Muster. Der während des Bootvorgangs zuerst gestartete Dienst wird während des Shutdowns als letzter beendet. Dieses Schema entspricht dem LIFO-Konzept[3], Dotplots können also auch zur Wiedergabe von Arbeitsbläufen oder einem Softwaredesign verwendet werden [7, Seite 10].

Diagonalen können auch bei kleineren Texten oft auftreten, Blöcke bedürfen aber einer größeren Wörterzahl. Der Betrachter benötigt zur entsprechenden Gruppierung bzw. Blockbildung die höhere Dichte[4].

[1] Es muss nicht lexikografisch sortiert werden, sondern es ist entscheidend, dass die Anordnung auf den beiden Achsen gleich ist.

[2] „Ein *Palindrom* (v. griech.: palíndromos = rückwärts laufend) ist ein Wort oder ein Satz, der von vorne und hinten gelesen gleich bleibt oder einen Sinn ergibt wie zum Beispiel beim Wort *Rentner*.“ [19] Die Bezeichnung des Musters wurde aus [7] übernommen.

[3] *Last In, First Out*: Das zuletzt hinzugefügte Element einer Liste wird beim Auslesen als erstes behandelt

[4] Bei der Verarbeitung von Bildern versucht das Gehirn Formen und Strukturen zu erkennen.

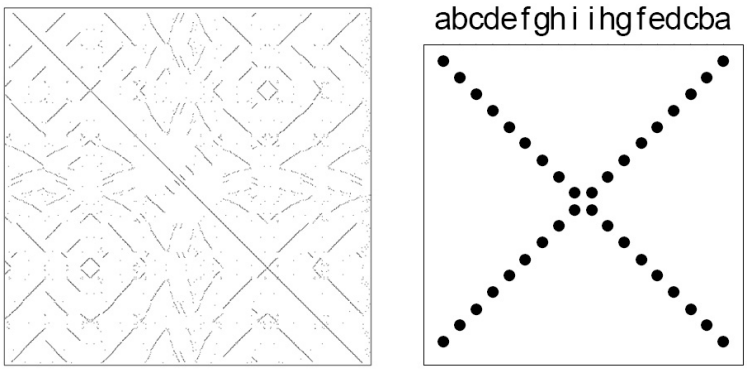

Abbildung 2.3: Dotplot-Muster: Palindrome lassen Diagonalen senkrecht zur Hauptdiagonale entstehen [7, Seite 3].

Die linke Grafik in Abbildung 2.4 demonstriert oben links den seltenen Fall, dass ein einziges Wort in direkter Nachbarschaft häufiger wiederholt wird. Im Allgemeinen erscheinen zwischen den Wiederholungen andere Wörter, so wie der Bereich unten rechts verdeutlicht.

Die rechte Hälfte in Abbildung 2.4 zeigt, im Gegensatz zum *hellen* Kreuz weiter unten, ein *dunkles* Kreuz. Es weist auf viele Treffer hin, besitzt aber gegenüber einem Block eine geringere Dichte. Die an dem dunklen Kreuz beteiligten Wörter besitzen in allen Bereichen des Textes eine hohe Häufigkeit, während ein kleinerer Block mit höherer Dichte erst für das Erscheinen dieser Struktur sorgt. Er wirkt horizontal und vertikal auf die ähnlichen Wörter.

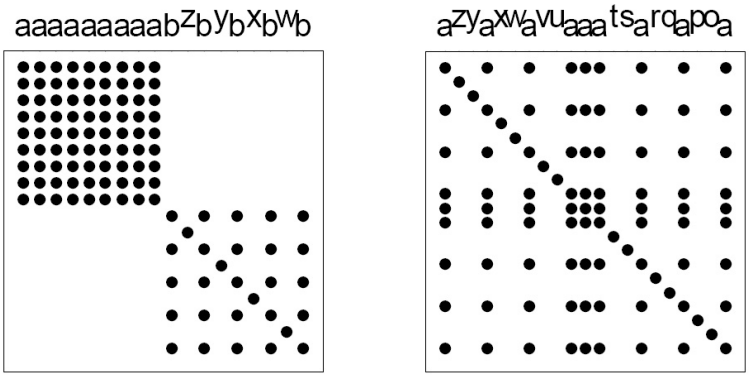

Abbildung 2.4: Links: Variation der Dichte von Blöcken. Rechts: das *dunkle* Kreuz [7, Seite 3].

Das helle Kreuz in Abbildung 2.5 zeigt den Effekt, der beim Einfügen von Text in einen selbstähnlichen Block auftritt. *Selbstähnlich* bedeutet hier, dass das Dotplot eine allgemein hohe Trefferquote zeigt. Die eingefügten Wörter treten an keiner

Für die Erkennung eines Bereiches als *Block*, muss sich dieser ausreichend von seiner Umgebung absetzen. [18], [23]

anderen Stelle im untersuchten Text auf, Resultat sind die Lücken zwischen den großen Blöcken, verbunden mit dem kleinen Block an der Kreuzung.

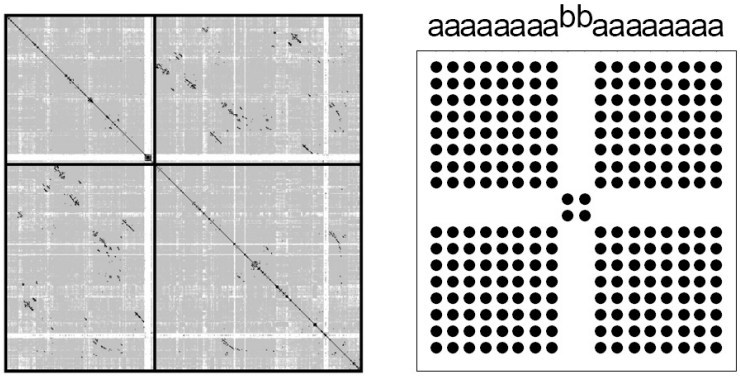

Abbildung 2.5: Dotplot-Muster: Das *helle* Kreuz indiziert ein nachträgliches Einfügen seltener Wörter [7, Seiten 3 und 6].

Neben Blöcken können auch die Diagonalen unterbrochen werden. Abbildung 2.6 zeigt wieder das Einfügen von Text in einen selbstähnlichen Bereich. Die Nebendiagonalen werden durch die fremden Wörter unterbrochen und voneinander weggeschoben. In größeren Dotplots erscheinen die unterbrochenen Diagonalen als gebogene Linien.

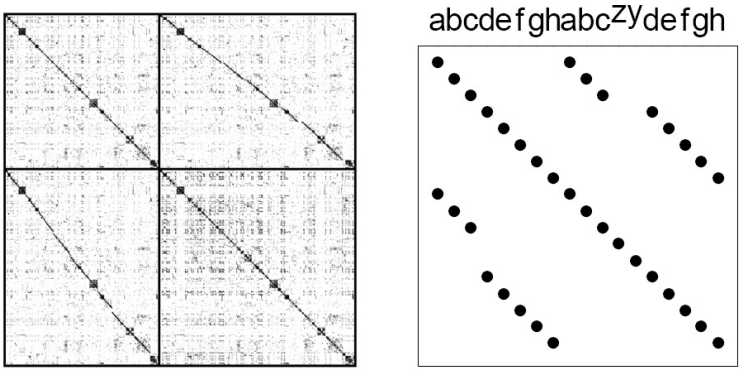

Abbildung 2.6: Dotplot-Muster: unterbrochene Diagonalen zeigen, wie das helle Kreuz, das Einfügen neuer Wörter [7, Seiten 3 und 5].

Bei der Umstrukturierung eines Textes können die Muster komplett neu angeordnet werden. Die Abbildungen 2.7 und 2.8 demonstrieren den Effekt, einerseits für Diagonalen und andererseits für Blöcke. Im Gegensatz zu den Diagonalen lassen die Blöcke keinen sicheren Rückschluss mehr auf die ursprüngliche Anordnung zu.

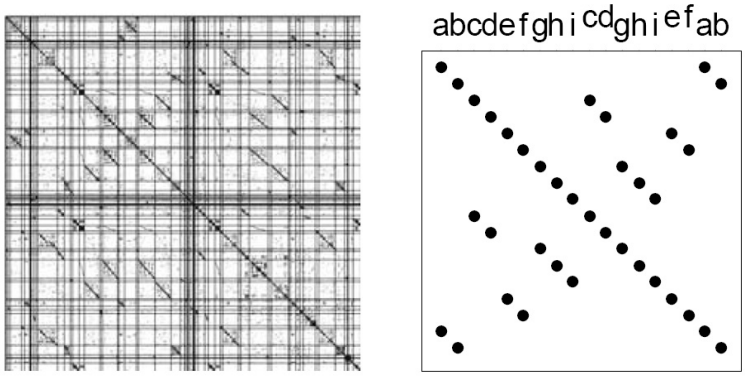

Abbildung 2.7: Unterbrochene und neu angeordnete Diagonalen [9, Seite 2] und [7, Seite 3].

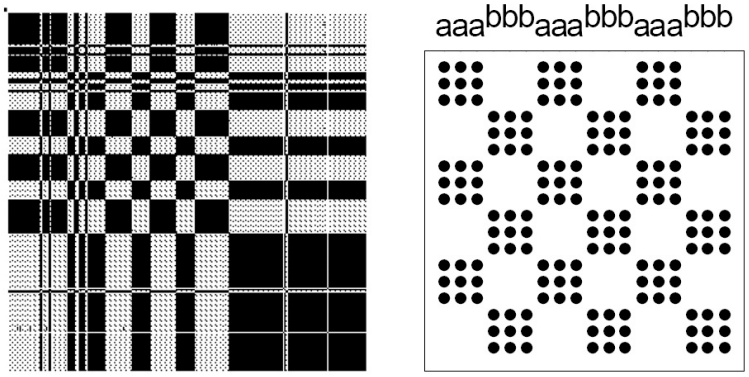

Abbildung 2.8: Kleinere Blöcke als Folge einer Umsortierung von Textblöcken [7, Seiten 3 und 6].

2.2 Beispiele

Alternative Vorgehensweisen zur Erkennung der Ähnlichkeit von Texten verwenden Ansätze aus der Bildverarbeitung und der Mathematik, die in Programme umgesetzt werden können. Bis heute ist aber die Leistungsfähigkeit des Menschen bei der visuellen Mustererkennung unerreicht. Dotplots geben dem Menschen die Möglichkeit, große Datenmengen auf einen Blick zu erfassen und unter Betrachtung der Ähnlichkeit zu bewerten. Mit Hilfe von normalen Editoren wäre ein solches Vorgehen unmöglich, da in der Regel die Übersicht fehlt. Bei längeren Texten muss man immer die Ansicht verschieben. Ein Vergleich von weit entfernten Textstellen ist so nur mit Aufwand möglich und nicht praktikabel.

Um einen Eindruck von der Leistungsfähigkeit der Dotplots zu geben, werden in diesem Abschnitt einige Beispiel-Plots gezeigt. Die darin auftretenden Muster und deren Entstehung wurden bereits im Abschnitt 2.1 beschrieben.

Abbildung 2.9 zeigt ein Dotplot der kompletten Werke von William Shakespeare, insgesamt ca. 1 Million Wörter. Um einen besseren Effekt zu erreichen, wurden mit Hilfe eines optimierenden Algorithmus die einzelnen Werke so angeordnet, dass höhere Ähnlichkeiten näher an der Hauptdiagonale liegen [8].

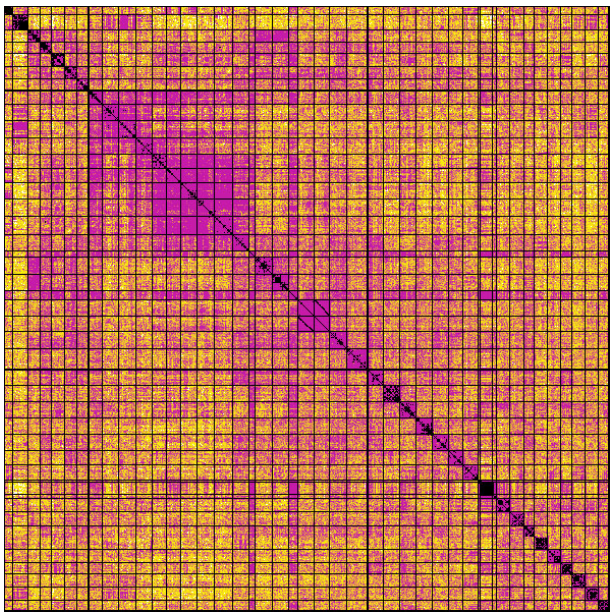

Abbildung 2.9: Dotplot aller Werke von Shakespeare [8].

Dem Dotplot in Abbildung 2.10 liegen die Dateinamen von 290.000 Dateien eines Unix-Dateiservers zugrunde. Das dominante helle Kreuz stammt aus einer Reihe von Dateinamen, sortiert nach der *inode*[5], die temporär durch das Unix-Tool *split* erzeugt wurden. Durch den Algorithmus zur eindeutigen Benennung dieser temporären Dateien sind Treffer mit anderen Dateinamen unwahrscheinlich – dies äußert sich in einem Dotplot durch die Abwesenheit von Treffern. Die zweite auffällige Struktur in diesem Dotplot sind die dunklen Blöcke. Sie weisen auf *Häufungen* oder Gruppen gleicher Dateinamen hin. In diesem speziellen Plot entstanden sie durch mehrere Kopien bzw. Versionen von Dateien, die auf der Festplatte verteilt gespeichert wurden [7, Seite 4].

Das Dotplot gibt also in diesem Fall die Tätigkeiten von Anwendern zum Zeitpunkt der Auswertung wieder. Ein Anwender bedient sich des Tools *split*, um aus einer großen Datei kleine handliche Pakete zu erzeugen. Andere Anwender arbeiten an einer Datei, sichern aber alte Versionen oder Kopien.

Interessant für die Informatik ist die Analyse des Quellcodes von Programmen oder einzelnen Softwaremodulen. Abbildung 2.11 zeigt das Dotplot von zwei Mil-

[5]Die *inode* ist im Unix-Dateisystem eine Referenz auf den Speicherplatz der entsprechenden Datei. Die Sortierung nach der *inode* entspricht in der Regel einer Sortierung nach dem Erstellungsdatum der Dateien.

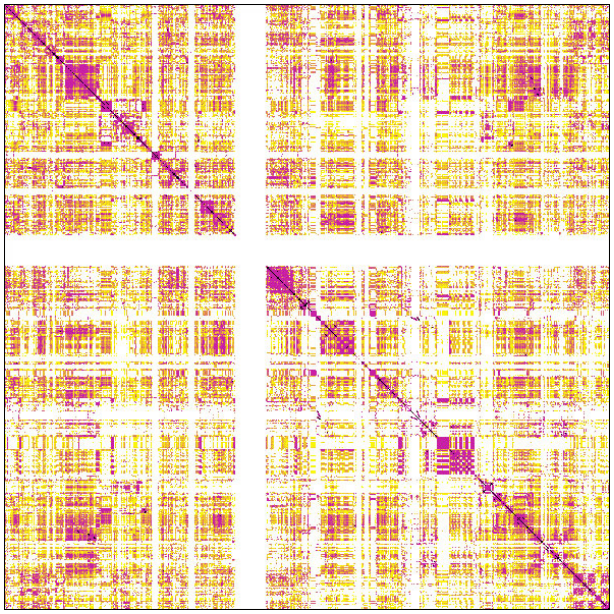

Abbildung 2.10: Dotplot von 290.000 Dateinamen [8].

lionen Zeilen Code eines Telekommunikations-Switches in der Sprache *C*. Die insgesamt recht dunkle Färbung lässt auf große Redundanz schließen. Redundanzen entstehen bei Quelltexten von Sprachen wie *C* durch Ähnlichkeiten bei der Initialisierung von Variablen und auch durch die vorgeschriebene Syntax[6]. In diesem Fall sind darüber hinaus große Tabellen für die Arbeit der Software notwendig, eine weitere Ursache für hohe Redundanz [8].

Ein weiteres Beispiel für Programm-Quelltexte zeigt Abbildung 2.12. Das Dotplot wurde aus den PHP-Sourcen des E-Study-Portals[7] erzeugt und mit Hilfe des Information-Mural-Algorithmus aus Kapitel 4 skaliert. Das E-Study-Portal ist ein auf der Scriptsprache *PHP* basierendes Kollaborationswerkzeug, entwickelt an der Fachhochschule Gießen-Friedberg, Fachbereich MNI. Der zentral liegende Bereich entstand aus HTML-Code, der Dokumentation zu den PHP-Sourcen. HTML-Code erscheint heller, da HTML als *Tag*-basierte Auszeichnungssprache[8] viele Redundanzen produziert. Darüber hinaus wurde die Dokumentation durch ein externes Programm automatisch generiert. Solche Prozesse erzeugen mehr Redundanzen, da sie auf Vorlagen basieren und den vom User geschriebenen Text nur darin einbetten.

[6]Die Syntax, oder *Grammatik*, schreibt dem Programmierer das Setzen von Klammern, Verwendung von Schlüsselwörtern und Ähnlichem vor [21]. In natürlichen Sprachen ist die Grammatik wesentlich flexibler. Außerdem werden dort (abhängig vom Kontext) die Wörter oft abgewandelt. Eine Analyse von natürlichen Texten würde bei der automatischen Reduktion auf Wortstämme mehr Treffer liefern.

[7]siehe auch `http://estudy.mni.fh-giessen.de/`

[8]Tags sind Meta-Informationen, die den ihnen zugeordneten Inhalt beschreiben und so die Interpretation und Darstellung beeinflussen können. [22]

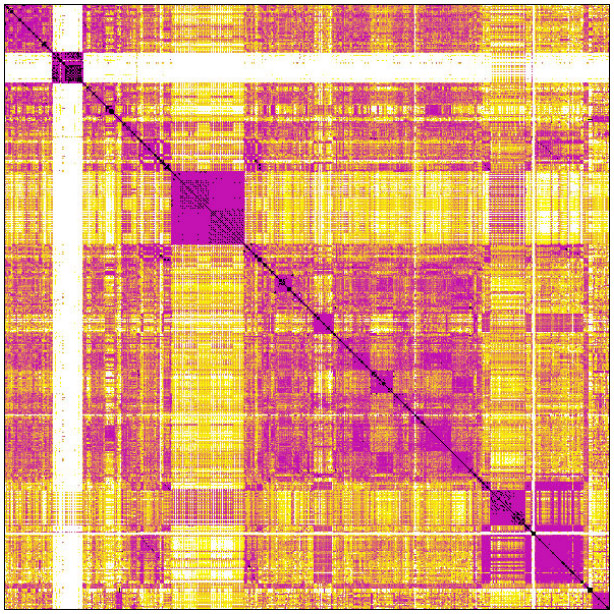

Abbildung 2.11: Dotplot von 2 Millionen Zeilen Code eines Telekommunikations-Switches in der Sprache *C* [8].

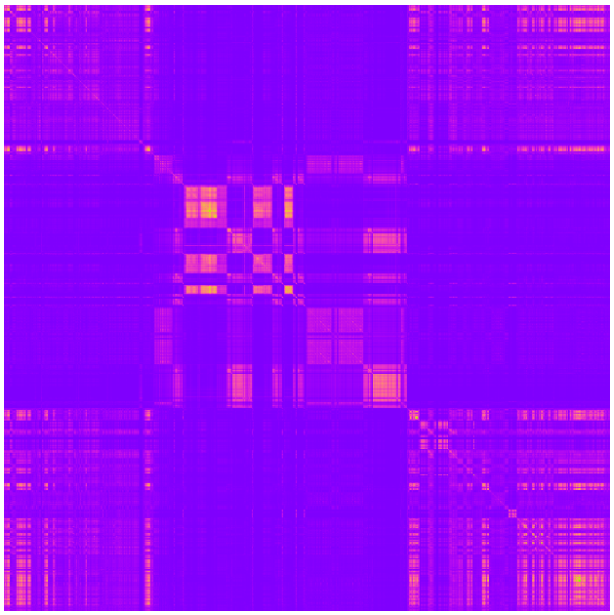

Abbildung 2.12: Dotplot der Sourcen des E-Study-Portals.

2.3 Analyse von Sprachen

Abbildung 2.13 zeigt das Dotplot eines Handbuchkapitels in den sechs Sprachen Dänisch, Französisch, Deutsch, Italienisch, Spanisch und Schwedisch von links nach rechts. Es wurden sogenannte *4-Gramme* gebildet, das sind Gruppen von jeweils vier Wörtern. Man wendet diese Gruppierung auch bei der Plagiaterkennung an, denn das Auftauchen gleicher Gruppen lässt auf eine hohe Ähnlichkeit schließen [2]. Die Kombination nutzt den Vorteil der sprachwissenschaftlichen Gruppenbildung zur effizienten Erkennung von Kopien mit einer schnell analysierbaren Visualisierung. Plagiaterkennung wird also mit Hilfe von Dotplots vereinfacht.

Die Blockbildung mit variierender Dichte zeigt, welche Sprachen untereinander näher oder ferner verwandt sind. Ein dunklerer Block weist auf höhere Ähnlichkeit hin, die Sprachen besitzen viele gemeinsame Wörter. Die in allen Blöcken enthaltenen Diagonalen lassen auf Zahlen oder Namen, bzw. nicht übersetzte Wörter schließen.

Man erkennt, dass ein Vergleich von Französisch, Italienisch und Spanisch eine höhere Trefferquote liefert, als beispielsweise der Vergleich von Deutsch mit Spanisch. Dotplots bieten also eine Grundlage für eine sprachübergreifende Analyse von Texten.

Auf der rechten Seite der Abbildung 2.13 wird die Entstehung des Musters etwas verdeutlicht. Es liegt das Blockmuster aus Abbildung 2.2 zugrunde, wobei einige gemeinsame Wörter in beiden Bereichen eingestreut wurden.

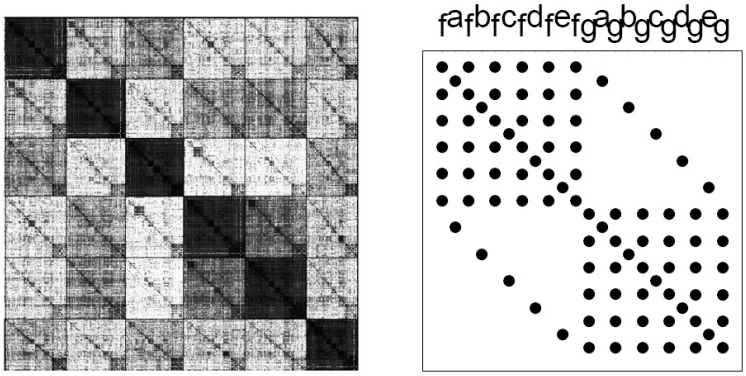

Abbildung 2.13: Links: Vergleich eines Handbuch-Kapitels in sechs Sprachen. Rechts: Schema für die Bildung dieses Musters [7, Seiten 3 und 4].

Bei der Verwendung von Wortstämmen könnte man sicher bessere Rückschlüsse auf die Verwandtschaft von Sprachen treffen. Dotplots werden bereits bei Firmen wie *AT&T* zur Pflege von Übersetzungen eingesetzt [1, Seite 4].

2.4 Anwendungen in der Informatik

Die Bildung von kleinen Wortgruppen wird durch das Plugin in grundlegender Weise unterstützt: Der Anwender kann einen Filter aktivieren, der die Eingabedaten zeilen- oder satzweise einliest. Bei der Erzeugung der F-Matrix werden so nicht die einzelnen Wörter verglichen, sondern komplette Gruppen. Die Abbildungen 2.14 und 2.15 zeigen die Auswirkungen auf einen Plot von Java-Quelldateien[9].

Die erste Abbildung entspricht einem normalen Plot mit wortweisem Vergleich, die senkrechten und waagrechten Linien zeigen die Dateigrenzen. Wie dem Bild zu entnehmen ist, wurden hier sechs verschiedene Java-Dateien (a-f) paarweise mit sich selbst verglichen: a mit a, a mit b, . . . , a mit f.

Bei der zweiten Abbildung wurden die Dateien zeilenweise eingelesen und verglichen. Dadurch sinkt die Trefferquote ab, was sich in einem geringeren Vorkommen von blauen und grünen Pixeln äußert. *Pixel* sind die Bildpunkte in einer Grafik. Besonders die dichten Blöcke am Anfang jeder Datei fallen komplett weg, denn sie entstanden durch wiederholte `import`-Anweisungen.

Eine weitere Auffälligkeit ist die Verschiebung der Dateigrenzen. Dieser Effekt resultiert aus der Zusammenfassung verschieden langer Zeilen, denn die Länge einer Zeile ist beim zweiten Plot irrelevant.

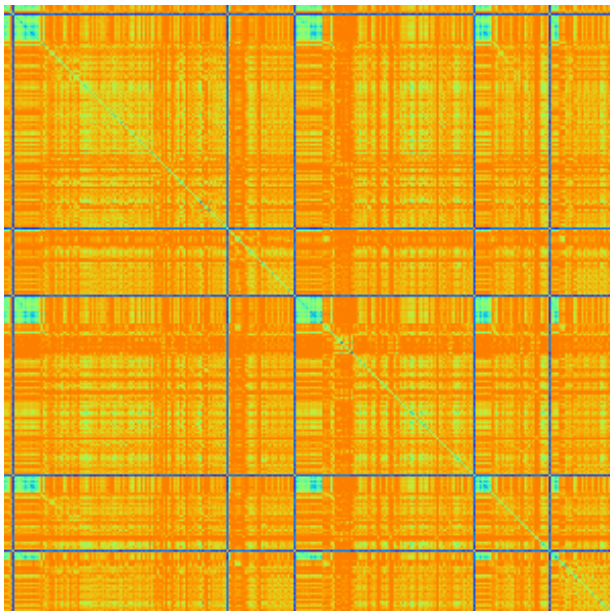

Abbildung 2.14: Dotplot von Java-Quelldateien, es wurde wortweise verglichen.

Jonathan Helfman beschreibt diverse Anwendungen für Dotplots im Zusammenhang mit Softwareentwicklung und -design [7, Seiten 6-11]. Eine Möglichkeit wird

[9] Es wurden dieselben Dateien verwendet wie für den Performance-Test in Abschnitt 5.2. Die Abbildungen wurden mit Hilfe des Information-Mural-Algorithmus skaliert.

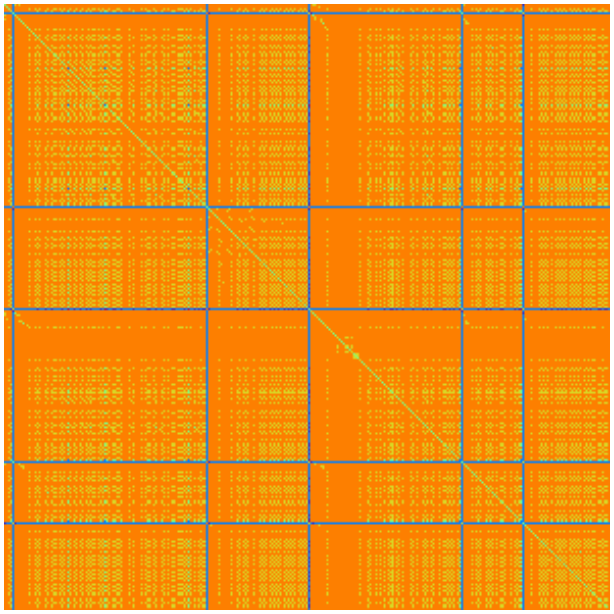

Abbildung 2.15: Dotplot von Java-Quelldateien, es wurde zeilenweise verglichen.

im Folgenden kurz vorgestellt: die Quelltextanalyse in Kombination mit Refactoring. Doppelt vorhandener Code kann durch die Einführung entsprechender Methoden zusammengefasst und eliminiert werden. *Refactoring* dient so der Vereinfachung und Optimierung des Software-Designs [20]. Die Software soll dadurch eine höhere Wartbarkeit und Flexibilität erreichen. Wenn ein Refactoring ausgeschlossen ist, können Dotplots immer noch helfen beim Bugfixing gleichartigen und möglicherweise ebenfalls fehlerhaften Code zu finden.

Refactoring bezieht sich in der Regel nur auf den vom Entwickler produzierten Code. Dotplots können jedoch sogar das Design von Programmiersprachen beeinflussen [7, Seite 6]. Eine in vielen aktuellen Sprachen vorhandene `switch`-Anweisung sieht so oder ähnlich aus:

Listing 2.1: Eine Switch-Anweisung in herkömmlicher Weise

```
switch(someInt) {
    case 0:         /* do something and break */
        break;
    case 1:         /* do something and break */
        break;
    case 2:         /* do something and continue with case 3 */
    case 3:         /* do something and break */
        break;
    case 4:         /* do something and break */
        break;
    default:        /* do something and break */
}
```

Die Anweisung vereint eine ganze Kette von `if`-Anweisungen in einer kompakten Form. Um dem Compiler mitzuteilen, dass er nach Abarbeitung eines Falles die Anweisung verlassen soll, muss vor dem nächsten Fall die Anweisung `break;` eingefügt werden. Nur selten sollen die Anweisungen für zwei verschiedene Fälle nacheinander abgearbeitet werden. In dem oben gezeigten Beispiel fällt der Programmfluss aus dem Fall `2` weiter in Fall `3` und verlässt danach erst die `switch`-Anweisung. Ein Dotplot zeigt aufgrund der vielen `break;`-Anweisungen eine hohe Trefferquote.

Das Einfügen des `break;` wird von unerfahrenen Programmierern oft vergessen, so dass Programmfehler entstehen. Eine Optimierung wäre also, das `break;` implizit auszuführen und nur bei Angabe von `no_break;` die `switch`-Anweisung nicht zu verlassen. Listing 2.2 zeigt einen Lösungsvorschlag.

Listing 2.2: Vorschlag für eine optimierte Switch-Anweisung

```
switch(someInt) {
    case 0:         /* do something and break */
    case 1:         /* do something and break */
    case 2:         /* do something */
            /* continue with case 3 */
        no_break;
    case 3:         /* do something and break */
    case 4:         /* do something and break */
    default:        /* do something and break */
}
```

Auch das oben schon angesprochene LIFO-Muster oder gleichartige Initialisierungen von Variablen lassen sich durch Dotplots hervorheben [7, Seiten 6-11]. Beim Softwaredesign und bei der -optimierung kann man also auf die Erkenntnisse aus den erzeugten Dotplots zurückgreifen.

2.5 Einschränkungen bei der Arbeit mit Dotplots

Dotplots besitzen bei einer Eingabe von n Tokens n^2 Punkte. Auch dort, wo keine Treffer gefunden wurden, muss in der Hintergrundfarbe ein Pixel gesetzt werden.

Man erkennt in den oben gezeigten Beispielen, dass die Dotplots an der Hauptdiagonale gespiegelt sind. Das liegt an den in der aktuellen Implementierung immer durchgeführten Selbstvergleichen. Bei der Erzeugung der F-Matrix wird diese Tatsache genutzt, um Speicher und Rechenaufwand zu sparen, indem intern nur eine Hälfte erzeugt wird. Jedoch muss für die Darstellung am Bildschirm oder beim

Export in eine Datei auch die zweite Hälfte angezeigt werden, die Datenmenge verdoppelt sich also.

Um ein Dotplot farbig anzeigen zu können, wird pro Pixel ein Integer, also 4 Bytes, verwendet. Bei nur 1.000 Tokens beansprucht ein Dotplot also schon 4 MB Speicher, 10.000 Tokens benötigen bereits 400 MB Speicher. Dieses hohe Datenaufkommen lässt einen Rechner schnell an seine Belastungsgrenzen stoßen. Kapitel 3 geht auf eine Möglichkeit ein, die Rechenlast auf einen Rechnerverbund, ein sogenanntes Grid, zu verteilen.

Neben dem Performanceproblem ist auch die *Skalierung* ein wichtiger Aspekt. Sie ist notwendig, um ein Dotplot am Bildschirm in einer brauchbaren Weise anzuzeigen. 10.000 Tokens würden bei einer Auflösung von 72 dpi ein Dotplot mit ca. $3,5$ m Seitenlänge produzieren. Da solche Ausmaße schlecht zu überblicken sind, werden Dotplots so weit verkleinert, dass sie komplett am Bildschirm angezeigt werden können. Eine solche Skalierung ist aber mit Qualitätseinbußen verbunden. Kapitel 4 geht näher auf die Problematik ein und stellt eine Lösung vor.

Kapitel 3

Grid-Computing

Das im vergangenen Kapitel vorgestellte Verfahren zur Auswertung von Texten ist bei der Arbeit mit großen Datenmengen ineffizient. Die Erzeugung der F-Matrix benötigt zwar nur wenig Arbeitsspeicher und Rechenleistung, da sie mehrere Optimierungsmöglichkeiten[1] nutzt und dabei auf ein Minimum an Datensätzen komprimiert wird. Bei ihrer Darstellung am Bildschirm oder beim Export in eine Datei müssen diese Datensätze aber wieder entpackt werden. Dieser Vorgang lässt einen Rechner schnell an die Grenzen seiner Belastbarkeit stoßen. Im Folgenden wird eine Technik beschrieben, die die Rechenlast auf mehrere durch ein Netz verbundene Rechner aufteilt.

3.1 Einführung und Anforderungen

„Der Begriff *Grid-Computing* existiert nun schon viele Jahre“ [16, Seite 109]. Im ursprünglichen Sinne verstand man darunter das, was man heute als *Verteiltes System* oder *Distributed Computing* bezeichnet. Dieses klassische Grid dient dem Zweck, ein komplexes Problem zu zerlegen und anschließend von verschiedenen Rechnern im Verbund berechnen zu lassen. Das Ergebnis der Berechnungen wird dann zu dem im Verbund agierenden Master zurückgesandt. Dieser Master koordiniert und steuert das Lösen des Problems [16, Seite 110]. In der Regel wird man für die Kommunikation mit den angeschlossenen Slaves ein Netzwerk verwenden. Dies ermöglicht die Verknüpfung mehrerer Teilnetze miteinander, auch das Internet wird nutzbar. Musterbeispiel ist das *SETI@home*-Projekt[2], das mit Millionen über das Internet angeschlossenen Rechnern versucht, extraterrestrische Signale zu durchsuchen und auszuwerten.

[1] Zum Beispiel die bereits angesprochene Ausnutzung der „Spiegelung“ an der Hauptdiagonale oder die Verwendung von Hashtables zur Zusammenfassung der Trefferliste gleicher Tokens.

[2] siehe `http://setiathome.ssl.berkeley.edu/`

Man interessiert sich nicht nur für das effiziente Bearbeiten von Problemen, sondern auch für das *Sharing*. Darunter versteht man das Teilen und gemeinsame Nutzen von Ressourcen *jeder* Art – nicht nur Rechenleistung und Speicherkapazität, sondern auch Peripherie. Der Datenaustausch steht stärker im Vordergrund. Man möchte zeitnah mit den neuesten Daten versorgt werden. Das Verständnis von einem modernen Grid ist so weit verallgemeinert, dass nicht nur Parallelisierung ermöglicht wird. Durch die Definition von Mechanismen, Konzepten und Protokollen können sowohl Firmen als auch Haushalte spontan beliebige Ressourcen anbieten und nutzen [16, Seite 111].

Durch die steigende Popularität entstehen immer mehr Projekte, die sich dem Aufbau und der Standardisierung von modernen Grid Technologien widmen. Neben den speziellen Implementierungen von Firmen wie z. B. *Oracle*[3] existieren auch offenere Projekte wie z. B. die der *Globus Alliance*[4]. Hier wurde kein solches Framework verwendet, da es mehr als nur den einfachen Versand von Nachrichten und grundlegenden Datenaustausch bieten würde, allerdings verbunden mit höherem Aufwand.

In dieser Arbeit wird das klassische Grid verwendet. Es soll als verteiltes System das effiziente Plotten großer Bilder ermöglichen. Anforderung ist, dass möglichst viele Systeme mit geringem Aufwand ihre Ressourcen bereitstellen können. Weitreichende Sicherheitsmechanismen wurden nicht gefordert. Wenn sich ein System im Netzwerk für das verteilte Plotten anbietet, so geschieht dies durch den jeweiligen Verantwortlichen, der entsprechende allgemeine Vorkehrungen zur Absicherung des Rechners treffen kann.

Es soll auch für unerfahrene Anwender mit vertretbarem Aufwand möglich sein, ihre Rechner in das Grid einzubinden. Der Anwender ist in der Regel nicht an Details über den internen Aufbau des Grids interessiert.

Die Verwendung von offenen Standards soll spätere Erweiterungen oder Korrekturen vereinfachen. Weiterhin ist auf einen fehlertoleranten Aufbau zu achten. Da das Grid auf einem Netzwerk basiert, können Knoten zu unvorhergesehenen Zeitpunkten ausfallen oder die Verbindung zum Server verloren gehen. Spontane An- und Abmeldungen eines Clients sind zu ermöglichen.

[3] siehe `http://www.oracle.de/`
[4] siehe `http://www.globus.org/`

3.2 Alternativen zur Implementierung des Grids

Für das mit Java[5] entwickelte Plugin bot sich auch die Umsetzung des Grids mit Hilfe dieser Programmiersprache an. Das Konzept des verteilten Rechnens ist schon relativ alt und es existieren bereits diverse Implementierungen. Die Kommunikation kann durch verschiedene Methoden erfolgen, zwei Möglichkeiten werden in den folgenden Abschnitten erläutert. Die zweite Variante wurde als Grundlage für diese Arbeit verwendet.

3.2.1 Kommunikation mit verteilten Objekten

Mit manchen Konzepten kann direkt auf Objekte eines entfernten Rechners zugegriffen werden. Populäre Beispiele dafür sind *RMI*[6] und *CORBA*[7]. Beide liefern die Grundlage zur Interaktion von Objekten über ein verteiltes Netz. Sie bringen Mechanismen zum Management der Objekte, Clients und Server mit.

In der Regel werden sogenannte *Interfaces*, *Skeletons* und *Stubs* benötigt, um die im System verwendeten Objekte verwenden zu können. Das *Skeleton* dient dem Server als Grundlage bei der Erzeugung von Objekten einer Klasse und zum Routen der Methodenaufrufe zur gewünschten Instanz. Die *Stubs* werden vom Client bei Transaktionen (z. B. Methodenaufrufe) zu und von den Objekten auf dem Server benötigt. Auf dem Server laufen darüber hinaus Dienste, wie *Registration Service*, *Naming Service* und ein *Object Manager*, die für die Registrierung, Verwaltung und Speicherung der Objekte zuständig sind [5, Seite 72].

RMI wird bei Java schon mitgeliefert, man muss nur noch die *RMI registry* starten und darin entsprechende Objekte registrieren. Das Erzeugen der clientseitigen Stubs und der serverseitigen Skeletons wird mit Hilfe des *RMI stub compilers* `rmic` ausgeführt.

Man kann mit Hilfe von RMI oder ähnlichen Systemen Objekte virtuell im Netz verteilen. Dadurch wird jedoch der Verwaltungsaufwand relativ hoch. Um die Kommunikation einfach zu halten, wurde sich gegen die Verwendung von RMI entschieden. Es sollen lediglich auf grundlegende Weise Nachrichten ausgetauscht werden können.

[5] *Java* ist eine von der Firma *Sun* entwickelte portable Programmiersprache und Laufzeitumgebung, siehe auch `http://www.sun.com/`.

[6] Remote Method Invocation

[7] Common Object Request Broker Adapter

Außerdem ermöglicht die Vermeidung von speziellen Protokollen die spätere Umsetzung des Plugins als *Applet*[8] und Probleme mit Firewalls sind leichter zu umgehen. Weiterhin war Effizienz ein wichtiges Kriterium bei der Auswahl des Frameworks.

3.2.2 Kommunikation durch Nachrichten

Um spätere Erweiterungen zu ermöglichen, sollte das Framework zum Nachrichtenaustausch von den anwendungsspezifischen Details gut gekapselt sein, andererseits muss eine Verknüpfung von Nachrichten mit den gewünschten Methodenaufrufen aufgebaut werden. Die hier vorgestellte Alternative verwendet nur rudimentäre Techniken zum Nachrichtenaustausch. Dadurch erscheint der Aufbau zwar weniger „elegant“, er kann aber leichter angepasst und für spezifische Anforderungen optimiert werden.

Das in [5, Kapitel 9] vorgestellte Konzept wurde hier als Grundlage übernommen. Es bietet einen einfachen Rahmen für ein *Collaborative System,* also ein System zur gemeinsamen Bearbeitung von Aufgaben. Das Konzept bietet eine einfache Infrastruktur (die Bezeichnungen wurden aus [5] übernommen):

- ein einzelner Server, der so genannte *Mediator*
- mehrere Clients, die *Collaborators*

Der Mediator wartet auf Anmeldungen der Collaborators und teilt ihnen Identitäten zu. Die Collaborators sind dann in der Lage, mit Hilfe des Mediators, Nachrichten an alle am Mediator registrierten Collaborators zu senden, dies wird dann als *Broadcast* bezeichnet. Sie können außerdem eine Nachricht an einen einzelnen Collaborator senden. Abbildung 3.1 zeigt beispielhaft die Struktur eines Grids.

Dies ist bereits ausreichend für die gestellten Anforderungen: es lassen sich effizient Plot-Aufträge an jeden Collaborator senden. Jeder einzelne kann nach der Bearbeitung sein Ergebnis an den Mediator zurücksenden, von dem der Auftrag abgeschickt wurde.

Der Nachrichtenaustausch findet nicht auf Objektebene statt, sondern es werden nur Zeichenketten, *Strings*, zwischen den beteiligten Clients und dem Server ausgetauscht. Im Gegensatz zu der im letzten Abschnitt vorgestellten Technik ist

[8] Ein Applet ist eine kleine Java-Anwendung, die auch aus Sicherheitsaspekten nur eingeschränkte Funktionalität bietet. Applets werden in der Regel auf Internetseiten eingesetzt.

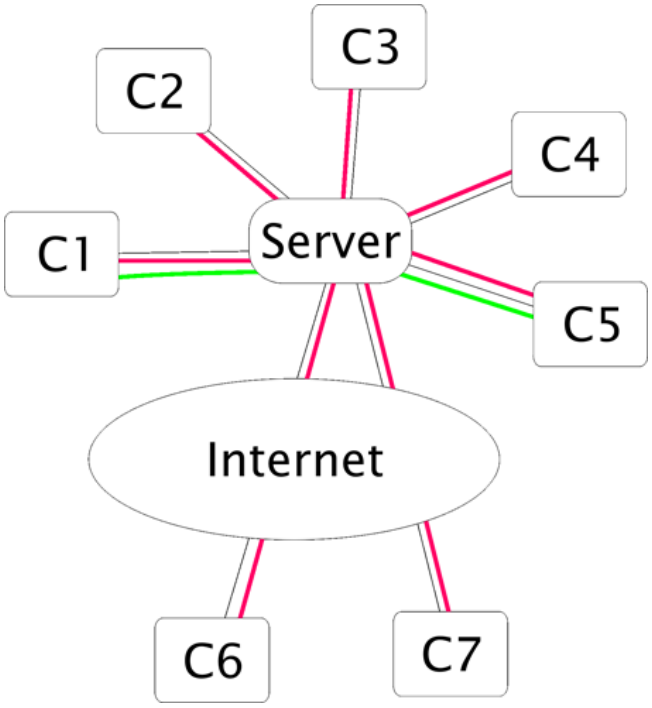

Abbildung 3.1: Die Grundstruktur des hier entwickelten Grids. Alle Nachrichten laufen über den Server. Rot: ein *Broadcast* an alle Clients. Grün: Nachricht von Client *C1* zu Client *C5*.

dazu weniger Verwaltungsaufwand notwendig. Statt spezielle Dienste und andere Mechanismen starten zu müssen, kann direkt Kommunikation betrieben werden.

Das Framework aus [5, Kapitel 9] war noch dahingehend zu erweitern, dass es eine Dateiübertragung auf einfache Weise erlaubt. Dies ist notwendig für das in Abschnitt 3.3.3 im Detail erläuterte Protokoll bei der Erstellung eines Dotplots.

Eine weitere Möglichkeit für den Aufbau eines Grids stellt das von Sun ins Leben gerufene Projekt *JXTA* dar, welches der Open Source Community übergeben wurde[9]. Es soll im Gegensatz zu Client-/Server- und webbasiertem Computing eine standardisierte P2P-Plattform anbieten. *P2P* steht für *Peer-to-Peer*, eine Technik zur Interaktion von mehreren Rechnern, die durch Tauschbörsen wie *Napster* und *Kazaa* bekannt wurde. JXTA stellt keine API[10] dar, da es sprachunabhängig ist. Es existieren zwar Referenz-Implementierungen, die aber nicht zwingend verwendet werden müssen. JXTA wird alleine durch eine Sammlung von Protokollen definiert. Diese Tatsache macht es gleichzeitig plattformunabhängig. Darüber hinaus ist es auch netzwerkunabhängig: Die definierten Protokolle lassen sich über *TCP/IP*, *HTTP*, *Bluetooth*, *Home-PNA* usw. übertragen [15, Seite 4].

[9] Project Juxtapose, siehe `http://www.jxta.org/`

[10] Application Programmer's Interface, die Anwendungsschnittstelle für den Programmierer. Eine API ermöglicht es dem Programmierer auf fertige Funktionen zuzugreifen.

Für zukünftige Erweiterungen des Dotplot-Plugins kann die Verwendung von JXTA berücksichtigt werden. Die im Folgenden erläuterte Implementierung ermöglicht diesen Schritt, indem die Schnittstelle zum Plugin einfach gehalten wurde.

3.3 Details zur Implementierung

Das Klassendiagramm in Abbildung 3.2 zeigt das grundlegende Design des Frameworks. Auf der rechten Seite (grün und blau) befinden sich die Klassen, die die einzelnen Knoten im Grid darstellen. Sie sind die aktiven *Agenten* innerhalb des Grids. Der Nachrichtenaustausch erfolgt mit Hilfe der links abgebildeten Klassen (gelb).

Die zentrale Klasse `MessageHandler` (weiß) übernimmt dabei die Koordination für einen Knoten während des Empfangs und Versands von Nachrichten. Er wird für jeden Knoten als eigener Thread gestartet, um Blockierungen der *Java-VM*[11] während Lese- und Schreibvorgängen zu minimieren.

Die Klasse `Identity` dient der eindeutigen Zuordnung der Knoten (`GridNode`) innerhalb des Grid. Es werden vom `GridServer` (siehe unten) fortlaufende Nummern vergeben, die bei einer Anmeldung in einer `Identity` verpackt an den jeweiligen Knoten verschickt werden. Der Knoten kann in dieser `Identity` noch einen beliebigen Namen setzen, so dass sich die im Hintergrund protokollierten Log-Ausgaben[12] bei Fehlern leichter zuordnen lassen.

Wie man in Abbildung 3.2 erkennen kann, wird die allgemeine Klasse `GridNode` in zwei Stufen abgeleitet. Die erste Stufe implementiert Methoden zum Versand von Nachrichten mit Hilfe des oben angesprochenen `MessageHandlers`. Bis zu dieser Stufe ist es noch nicht möglich, über das Netzwerk Verbindungen aufzubauen. Diese Möglichkeit wird erst in der zweiten Stufe implementiert. Die Trennung hat den Zweck, die interne Abwicklung der Nachrichten von der Art der Verknüpfung (dem Netzwerk) zu entkoppeln. Das Framework soll in dieser Hinsicht leicht an zukünftige alternative Techniken anzupassen sein. Die hier beschriebene Methode, das Grid über ein Netzwerk aufzubauen, wird also erst in der äußersten Schicht ermöglicht.

Die Einbindung in das Dotplot-Plugin erfolgt durch die in Abbildung 3.3 gezeigten Klassen `GridClient` und `GridServer` (blau). Sie sind Ableitungen der Klassen `MessageCollaborator` und `MessageMediator` des Grid-Frameworks.

[11] Die *Virtual Machine* ist die von der Java-Plattform zur Verfügung gestellte Schicht. Sie dient als Grundlage für Java-Programme und als Abstraktionsschicht über dem echten Betriebssystem. So ermöglicht die VM das Entwickeln von weitgehend plattformunabhängigem Code.

[12] die Log-Datei befindet sich unter `<user-dir>/dotplot.log`

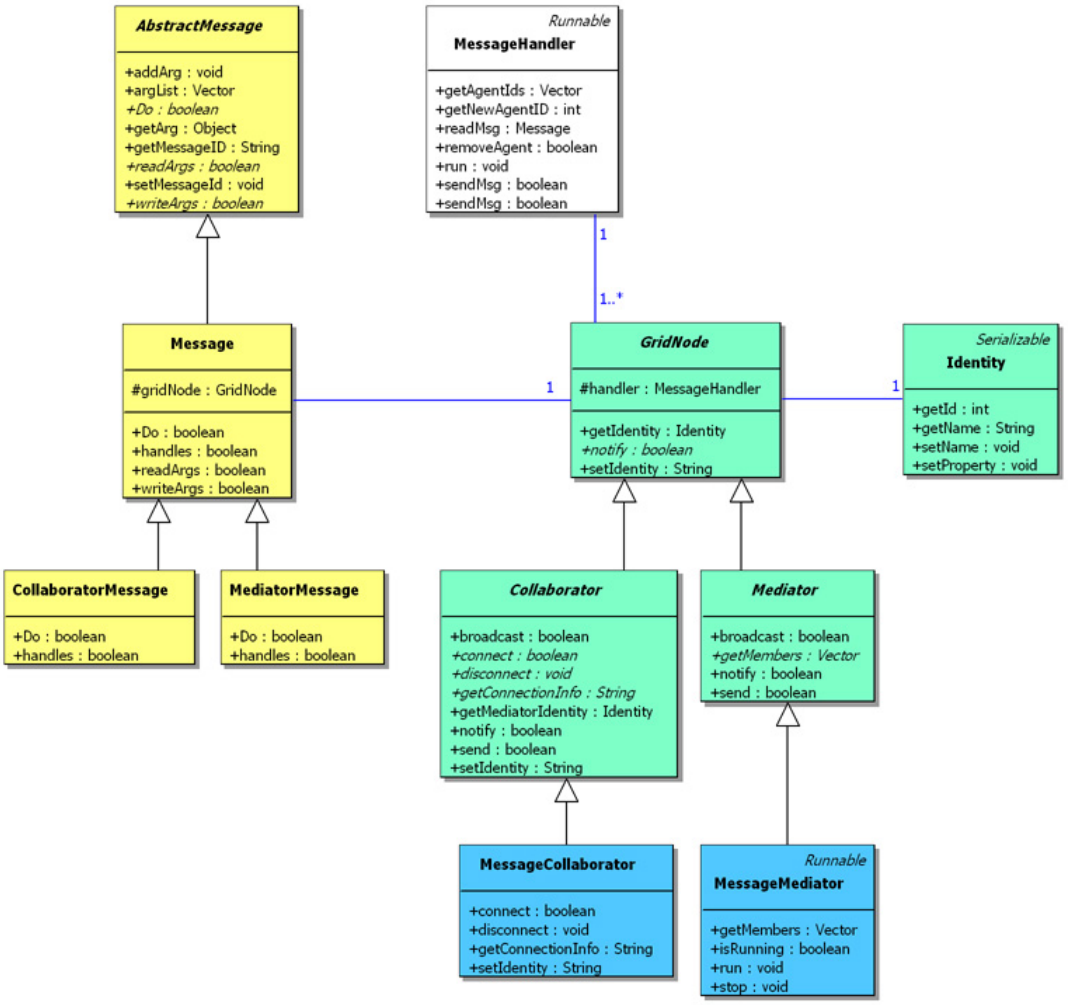

Abbildung 3.2: Das Klassendiagramm des Grid-Frameworks.

Der eigentliche Plot wird mit Hilfe der Klasse `GridPlotter` initiiert. Über das Interface `GridCallback` signalisiert er dem Plugin während eines Plots den kompletten Empfang der Ergebnisse von allen aktiven Knoten im Grid. Der `GridPlotter` wartet vorher auf die einzelnen Signale vom `GridServer`, der als zentraler Empfänger für alle Teilbilder fungiert. Er leitet die Teilbilder mit dem Signal `onImageReceived` an den `GridPlotter` weiter. Durch die Implementierung als Singleton wird vermieden, dass der `GridServer` auf einem Rechner mehrmals gestartet wird.

Die Klasse `PlotJob` repräsentiert den vom `GridPlotter` erzeugten Plot-Auftrag für die `GridClients`. Ein `PlotJob` enthält neben der notwendigen `TypeTable` (die interne Darstellung der F-Matrix) auch die aktuelle Konfiguration des Grids und eine Liste aller aktiven Clients.

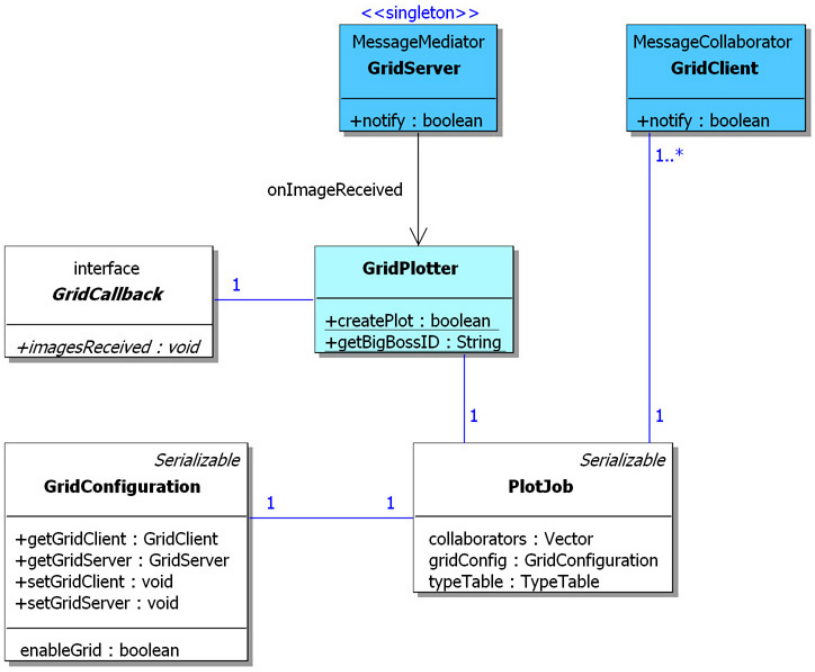

Abbildung 3.3: Das Klassendiagramm zur Einbindung des Grid-Frameworks in das Plugin.

3.3.1 Aufbau eines Grids

Um ein funktionierendes Grid aufzubauen, sind nur wenige Schritte notwendig. Grundlage stellt ein von allen Knoten erreichbarer Mediator dar. Er muss zuerst gestartet werden. Dazu öffnet er auf dem entsprechenden Rechner einen Port, über den sich Clients anmelden können. Der Port darf noch nicht an eine andere Anwendung vergeben sein und eine evtl. auf dem Rechner laufende Firewall muss das Öffnen des Ports zulassen.

Um Probleme mit anderen Anwendungen zu vermeiden, sollte man einen freien Port aus dem Bereich von 1024 bis 65535 wählen. Speziell unter Linux/Unix ist das Öffnen eines Ports < 1024 dem User *root* vorbehalten. Bei einem erfolgreichen Start des Mediators werden keine Meldungen ausgegeben.

Nun können sich die Clients am Mediator anmelden. Dazu wird jeweils ein *Socket*, also eine Netzwerkverbindung, geöffnet und auf eine spezielle Antwort des Mediators gewartet. Diese Antwort beinhaltet eine eindeutige `Identity` zur gridinternen Zuordnung der Clients. Der Client sendet als Bestätigung die `Identity` mit einem vom Anwender gewählten Namen zurück zum Mediator. So lassen sich Fehlermeldungen und auch im Hintergrund geführte Log-Ausgaben leichter dem entsprechenden Knoten zuordnen. Man sollte also einen sinnvollen Namen vergeben. Werden keine Meldungen ausgegeben, kann auf dem Server der Plot gestartet werden.

Das Grid ist ab diesem Zeitpunkt verwendbar. Alles Weitere geschieht allein durch gezielten Nachrichtenaustausch. Um als Client auf Nachrichten reagieren zu können, steht in der Klasse `GridNode` die Methode `notify()` zur Verfügung, siehe Listing 3.1. Sie kann in Unterklassen überschrieben werden, um die ankommenden Nachrichten abzufangen. Der erste Parameter `tag` benennt die aktuelle Nachricht, so dass der Knoten entscheiden kann, wie die Nachricht zu verarbeiten ist. Zum Beispiel wird während der Anmeldung eines Clients die `Identity` vom Mediator mit dem *Tag identity* versehen. Der zweite Parameter `data` enthält für die Abwicklung der Nachricht notwendige Daten, während `src` den Absender repräsentiert und somit eine Möglichkeit bietet, auf die Nachricht antworten zu können.

Listing 3.1: Ausschnitt der Klasse `GridNode`

```
// Incoming messages/data
public abstract boolean notify(String tag,
                               Object data,
                               Identity src)
```

3.3.2 Versand von Parametern

Innerhalb des Frameworks können Nachrichten an einen speziellen Empfänger oder auch per Broadcast an alle bekannten Knoten versandt werden. Nachrichten bestehen in der Regel aus speziellen Zeichenketten, die in jedem Knoten eine entsprechende Aktion auslösen. Benötigt eine Aktion zur korrekten Abwicklung eigene Parameter, so sind diese als Objekte in der Nachricht mitzusenden. Durch das zugrunde liegende Protokoll ist die Reihenfolge der Parameter definiert, ähnlich einem Funktionsaufruf.

Die als Parameter in Frage kommenden Objekte müssen, um über das Netz versandt werden zu können, entweder das Interface `java.io.Serializable` implementieren oder (z. B. bei zu komplexem internen Aufbau) selbst die (De-) Serialisierung durchführen. *Serialisierung* ist eine Technik, die im Speicher befindlichen Objekte in eine Folge von Bytes umzuwandeln, die sich problemlos über das Netzwerk verschicken oder auch als Datei speichern lassen. Beim Deserialisieren muss bekannt sein, welche Klasse den Bytes zugrunde lag, damit der Zustand wieder korrekt hergestellt werden kann. Die Basistypen wie `int`, `double`, `boolean` oder auch `String` stellen bei der Serialisierung kein Problem dar; Klassen, die ausschließlich Basistypen als Felder besitzen, lassen sich ebenfalls einfach serialisieren. In einem solchen Fall ist die Implementierung von `java.io.Serializable` ausreichend.

Es wird beim Versand von Nachrichten automatisch erkannt, ob Parameter als *String*, *Object* oder, im Falle von Dateien, als einzelne *Bytes* übertragen werden sollen. Dem Empfänger muss in den beiden letzten Fällen mitgeteilt werden, wie er den aktuellen Parameter aus dem Eingabestrom zu lesen hat. Dazu wird direkt vor dem eigentlichen Objekt ein kurzer String als Signal geschickt, der vom Empfänger ausgewertet wird und ihn die folgenden Daten auf korrekte Weise lesen lässt.

3.3.3 Protokoll zur Abwicklung eines Plots

Die grundlegende Einbindung des Grid-Frameworks in das Dotplot-Plugin geschieht durch die Klassen `GridServer` und `GridClient`. Sie sind Erweiterungen der Klassen `MessageMediator` und `MessageCollaborator`. Die Erweiterungen dienen der Verzahnung mit dem *UI*, also der Benutzerschnittstelle, und der korrekten Abwicklung von Nachrichten. Beide Klassen bieten Methoden zum Öffnen und Schließen der Netzwerkverbindungen, damit am UI alles per Knopfdruck gesteuert werden kann.

Sobald der Anwender das Grid aufgebaut und Dateien zum Plotten ausgewählt hat, kann er wie gewohnt auf den Button *Plot!* oder den Menüpunkt *DotPlot–Plot!* klicken[13]. Es wird dann zunächst die F-Matrix aus den Quelldateien gebildet. Aus der F-Matrix und der aktuellen Konfiguration wird ein `PlotJob` erzeugt, der dem `GridPlotter` zur Verarbeitung übergeben wird.

Dieser fügt dem `PlotJob` die aktuelle Liste der Collaborators hinzu und verschickt den kompletten Job per Broadcast mit dem Tag *plotjob* an jeden einzelnen Collaborator. Dazu erzeugt er intern einen `MessageCollaborator` – keinen `GridClient` – um den `GridServer` erreichen zu können. Es wird deshalb ein

[13]Die Kurzbeschreibung des Plugins in Abschnitt A.1 erläutert weitere Optionen.

`MessageCollaborator` verwendet, damit er sich nicht unbeabsichtigt selbst am Plot beteiligt.

Die `GridClients` empfangen nun per Netzwerk die Nachricht mit dem `PlotJob`. Um Missbrauch der Clients zu verhindern, kann über den VM-Parameter `grid.bigboss.id` eine beliebige Integer-ID gesetzt werden; per Default ist sie auf den Wert 0 gesetzt. Diese ID ist für jeden im Grid beteiligten Knoten auf den gleichen Wert zu setzen. Die Clients überprüfen beim Empfang der *plotjob*-Nachricht die ID und arbeiten nur weiter, wenn sie korrekt ist.

Nachdem dann eine temporäre Datei für das später erzeugte (Teil-) Bild angelegt wurde, ermittelt der Client die Größe und den ihm zugeteilten Teilabschnitt der gesamten F-Matrix. Die Teilbilder sind senkrechte Streifen des Gesamtbilds, das heißt, die jeweilige Breite ergibt sich durch das Verhältnis von *Gesamtbreite* zu *Anzahl der Clients*:

$$\text{Breite Teilbild} = \frac{\text{Gesamtbreite}}{\text{Anzahl Clients}}$$

Die Entscheidung, welcher Streifen vom jeweiligen Client erzeugt werden soll, geschieht durch ein einfaches Suchen in der Liste der Collaborators:

```
int index = plotjob.getCollaborators().indexOf(getIdentity());
```

Der `index` entspricht dabei genau dem Index aus der Menge aller Teilbilder.

Bei der Berechnung der Bildbreite können Rundungsfehler auftreten. Um diese zu kompensieren, prüft der Client, der das letzte Teilbild zu plotten hat, ob ein Rundungsfehler aufgetreten ist. In diesem Fall verbreitert er seinen Teilstreifen um den übrig gebliebenen Bereich – er benötigt also unter Umständen etwas länger bis zur Fertigstellung.

Nachdem alle Vorbereitungen abgeschlossen sind, wird das Teilbild erstellt und in der temporären Datei gespeichert. Diese Datei wird dann mit dem Tag *plotimg* an den Mediator, also an den zentralen `GridServer` verschickt. Dieser benachrichtigt bei jedem eingegangenen Teilbild den `GridPlotter`, der die Teilbilder zunächst nur sammelt. Erst wenn alle Teilbilder fertig sind wird über das `GridCallback` die gesamte Liste weitergegeben. Das Dotplot-Plugin braucht dann nur noch ein großes Gesamtbild aufzubauen und wie gewohnt am Bildschirm darzustellen.

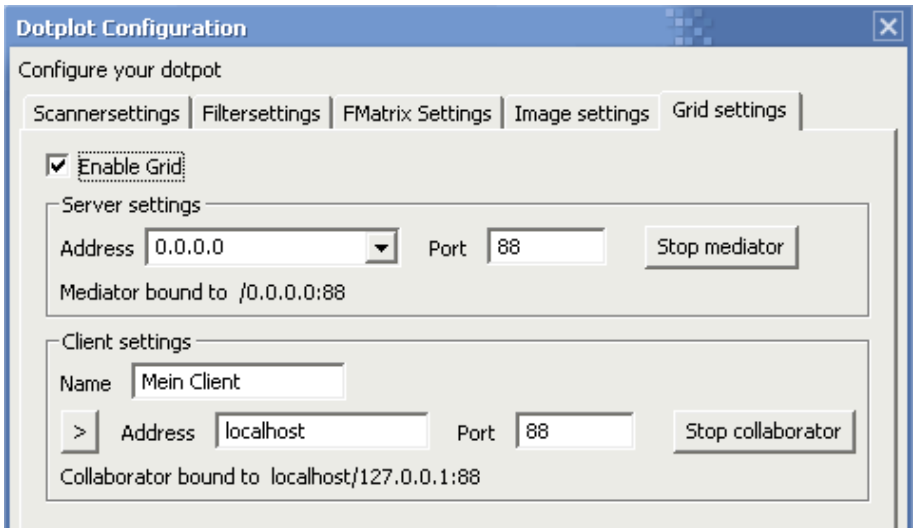

Abbildung 3.4: Modul zur Konfiguration des Grids. Die vier anderen Tabs werden im Anhang A.1 und im Plugin-Handbuch [11] erklärt.

3.4 Überblick über die Benutzerschnittstelle

3.4.1 Verwendung der graphischen Benutzerschnittstelle

Dem Anwender wird ein einfaches *GUI*[14] zur Verfügung gestellt, siehe Abbildung 3.4. Oben befinden sich die Einstellungen für den Mediator, darunter die Einstellungen für den Collaborator.

Um ein Grid aufzubauen, muss zuerst der Grid-Server gestartet werden. Er benötigt als Parameter die IP-Adresse und den Port, an denen auf Verbindungen von Clients „gelauscht“ wird. Der Port ist eine gundlegende Einstellung, während die IP-Adresse normalerweise auf den Wert `0.0.0.0` gesetzt wird. Dies hat zur Folge, dass der Server auf Anfragen an jeder lokalen Adresse antwortet. Durch das Setzen einer anderen gültigen IP-Adresse weist man den Server an, nur auf Anfragen an dieser speziellen Adresse (bzw. dem entsprechenden Interface) zu reagieren. Alle vorhandenen Interfaces bzw. Adressen stehen dem Anwender per Dropdown-Box zur Verfügung.

Die Einrichtung eines entsprechenden Grid-Clients ist ähnlich. Hier werden ebenfalls nur IP-Adresse und Port als Parameter benötigt. Sie müssen mit den Daten des im Grid aktiven Servers übereinstimmen. Der Grid-Client versucht bei einem Start sich mit dem Server zu verbinden und eine `Identity` von ihm zu erhalten. Dies ist notwendig, um mit den anderen Clients und dem Server kommunizieren zu können.

[14] *Graphical User Interface*, die graphische Benutzerschnittstelle. Mit ihrer Hilfe interagiert der Benutzer mit dem Programm.

Ein Plot im Grid kann nur von dem Rechner ausgelöst werden, auf dem ein Grid-Server läuft. Das alleinige Starten eines Grid-Clients dient lediglich der Weitergabe von Rechner-Ressourcen ins Grid. Will man also vom Grid profitieren, muss man einen Grid-Server starten und es müssen sich ausreichend viele Clients anmelden. Theoretisch können beliebig viele Clients an einem Server angemeldet werden, es ist aber zu berücksichtigen, dass der Verwaltungsaufwand für jeden aktiven Client steigt. Es macht keinen Sinn, eine kleine Datei mit Hilfe von 10 Clients im Grid berechnen zu lassen; auch langsame Rechner sollten mit Hilfe des in Kapitel 4 vorgestellten Information Murals in der Lage sein kleine Dateien effizient zu plotten.

Der Anwender erhält bisher noch keine direkte Rückmeldung über den Fortschritt des Plots, er kann jedoch in der Log-Datei die Abläufe kontrollieren und beobachten, was im Grid passiert. Fehlermeldungen werden dort ebenfalls ausführlich protokolliert.

3.4.2 Start eines Grid-Clients von der Konsole

Ein Grid-Client kann auch ohne die Eclipse-Umgebung gestartet werden. Die dafür notwendigen Dateien befinden sich im Eclipse-Programmverzeichnis unter `<eclipse>/plugins/org.dotplot.plugin_2.0.0/`, wobei `2.0.0` der aktuell installierten Plugin-Version entspricht. Ein Grid-Client wird dort durch den folgenden Befehl gestartet:

- Linux/Unix:
  ```
  java -cp $CP_DOTPLOT -Dheadless=true
  ↪ org.dotplot.grid.GridClient <name> <address> <port>
  ```
- Windows:
  ```
  java -cp %CP_DOTPLOT% -Dheadless=true
  ↪ org.dotplot.grid.GridClient <name> <address> <port>
  ```

Die Classpath-Variablen `$CP_DOTPLOT` bzw. `%CP_DOTPLOT%` müssen die folgenden mitgelieferten Bibliotheken enthalten:

- `plugin.jar`
- `lib/iText.jar`
- `lib/jai_codec.jar`
- `lib/jai_core.jar`

- `lib/jai_mlibwrapper.jar`
- `lib/log4j-1.2.8.jar`

Der Parameter `-Dheadless=true` weist das Plugin an, die internen Klassen für die Einbindung in Eclipse nicht zu starten. So werden Fehlermeldungen vermieden. Der Parameter ist für einen reibungslosen Ablauf jedoch nicht zwingend notwendig. Der Grid-Client benötigt, wie auch unter Eclipse, Server-Adresse und -Port. Zu diesen Parametern kommt noch ein frei wählbarer Name, wie schon in Abschnitt 3.3.1 angesprochen, hinzu. Unter Eclipse wird bei fehlender Eingabe automatisch ein Name gewählt. Auf der Konsole muss zwingend ein Name angegeben werden.

Wenn keine Fehlermeldung ausgegeben wird, ist der Client ist für das Grid verfügbar. Durch das Starten von der Konsole kann Arbeitsspeicher gespart werden, die Performance wird sich erhöhen. Um dem Client gezielt mehr Arbeitsspeicher zuzuweisen, sind die Parameter `-Xms64M` und `-Xmx512M` möglich. Sie geben an, wieviel Speicher beim Start des Clients und wieviel maximal reserviert werden soll. Die Werte `64M` und `512M` sind hier nur Vorschläge und müssen auf den verwendeten Rechner abgestimmt werden.

3.5 Ergänzungen

Während eines Plots legt jeder Knoten in einem temporären Ordner seine Daten ab. Unter Windows ist dies der Ordner `<user-dir>/Lokale Einstellungen/Temp`, unter Linux befindet er sich unter `/tmp`. Beim Beenden der Clients und des Servers werden diese temporären Dateien wieder gelöscht.

Das Zusammenfügen der Teilbilder muss an einer zentralen Stelle erfolgen. Zur Zeit geschieht dies immer auf dem Rechner, auf dem der Grid-Server läuft. Leider benötigt das Zusammenfügen noch viel Rechenleistung und Arbeitsspeicher, es sollte also ein leistungsfähiger Rechner als Server eingerichtet werden. Wenn man ein professionelles Bildverarbeitungsprogramm besitzt, kann man die Teilbilder auch selbst zusammenfügen.

Dazu muss man in der Log-Datei beobachten, wann der Grid-Server den Empfang aller Teilbilder signalisiert. Dann liegen im temporären Ordner die einzelnen Bildstreifen im JPEG-Format und eine Textdatei mit Informationen über die korrekte Reihenfolge der Teilbilder.

Als Optimierung wurde der im folgenden Kapitel vorgestellte Algorithmus Information Mural eng mit dem Grid verknüpft. Den Clients kann per VM-Parameter

`-Dgrid.maxedge` eine maximale Kantenlänge (in Pixeln) gesetzt werden. Bei zu großen Plots wird dann auf diesen Wert skaliert. Man kann auf diesem Weg die Zielgröße des Dotplots exakt definieren, was neben dem Performancegewinn auch eine Hilfe in der Druckvorstufe bietet.

Kapitel 4

Information Mural

Das *Information Mural* wurde zur Visualisierung großer Datenmengen entwickelt [10]. Für den Anwender ist das Untersuchen solcher Datenmengen ein Problem, da ihm in der Regel der Überblick fehlt und jede Änderung des aktuellen Teilausschnitts mit Rechenaufwand und Zeitverlust verbunden ist.

Eine geschickte Visualisierungstechnik hilft Anwendern interessante Bereiche schnell einzugrenzen bzw. zu fokussieren und dabei Informationen effizient zu verarbeiten. Die Visualisierung des gesamten Datensatzes hilft, sich darin zu orientieren. Das Information Mural ist solch eine „[...] zwei-dimensionale, reduzierte Darstellung eines kompletten Informationsraumes [...]“ [10, Seite 1], die gleichzeitig in der Lage ist, die Informations*dichte* darzustellen. Die Darstellung der Informationsdichte ist eine Hauptaufgabe des Dotplots. Die Verwendung des Information Murals stellt also eine sinnvolle Möglichkeit dar, relevante Stellen im Dotplot hervorzuheben und gleichzeitig die Performance zu steigern.

Zur Erzeugung und Bearbeitung der Dotplots wird die von Sun entwickelte Bibliothek *JAI*[1] verwendet. Die Verwendung dieser Bibliothek ermöglicht Effizienzsteigerungen bei der Bildbearbeitung und den entsprechenden Dateioperationen und sichert gleichzeitig die Portabilität des Plugins. Eine Einschränkung entsteht jedoch durch die Einbindung des Plugins in die Eclipse-Umgebung: Eclipse verwendet das *SWT*[2] bei der Interaktion mit dem Benutzer, JAI basiert aber auf dem *AWT*[3]. Eine unter Umständen aufwendige Konvertierung zwischen deren internen Darstellungen ist notwendig.

[1] Java Advanced Imaging, eine optional installierbare Bibliothek

[2] Standard Widgets Toolkit. Durch die Wiederverwendung der nativen Elemente der zugrunde liegenden Plattform wird Performance gewonnen.

[3] Abstract Windows Toolkit. Die von Sun entwickelte Technik zur Anzeige und Bearbeitung von graphischen Elementen.

4.1 Skalierung von Bildern

Das Dotplot-Plugin gibt per Default pro Match einen Pixel aus. Bei einer Eingabe von nur 1.000 Tokens ergibt dies bereits 1.000.000 Pixel, während bei einem farbigen Bild pro Pixel mindestens 4 Bytes benötigt werden. Unkomprimiert ergibt das schon 4 MB – interaktiv kann hier auf einem durchschnittlichen Rechner[4] nicht mehr gearbeitet werden.

Die fehlende Interaktivität ist jedoch nicht das größte Problem. Das Bild soll auf dem Bildschirm dargestellt werden. Da die wenigsten durch das Plugin erzeugten Bilder komplett auf einen Bildschirm passen, muss man sie kleiner skalieren. Das Plugin ermöglicht es zwar, Dotplots in Originalgröße ausgeben zu lassen, der Speicherverbrauch steigt dann aber entsprechend an und man verliert gleichzeitig den Überblick.

4.1.1 Interpolation

Um das Bild zu verkleinern können verschiedene Methoden angewendet werden. Allgemein basieren die bekannteren Methoden aus der Bildverarbeitung auf dem gleichen Algorithmus, nur bei der Art der *Interpolation* (Glättung/Weichzeichnung) werden Unterschiede gemacht. Die Interpolation ist notwendig, weil man durch die Skalierung ohne Interpolation Muster und Artefakte im Bild erhält, die eine Analyse stören oder verfälschen können.

Interpolation ist eine Durchschnittsbildung der an einem Zielpixel beteiligten Farbwerte aus dem Quellbild. Man unterscheidet folgende qualitativ unterschiedlichen Typen[5] [17, Abschnitt 8.2]:

- Nearest-Neighbor
- Bilinear
- Bicubic

Nearest-Neighbor bezieht zur Ermittlung des Zielfarbwertes nur den direkten Nachbarn aus dem Quellbild mit ein. Das Ergebnis ist entsprechend grob und es erscheinen speziell an geraden Kanten des Quellbilds sogenannte *aliasing errors,* die bekannten Treppenmuster [17, Abschnitt 8.2.1]. Der große Vorteil dieser Technik ist, dass sie äußerst schnell arbeitet, da keine aufwendigen mathematischen Operationen verwendet werden.

[4] siehe auch Kapitel 5 für Details zu den getesteten Plattformen.

[5] dies ist nur eine Auswahl der bekanntesten und per JAI direkt verfügbaren Methoden.

Die *bilineare Interpolation* bezieht in die Berechnungen die horizontal und vertikal gelegenen Nachbarn ein. Dadurch werden schon wesentlich bessere Ergebnisse erzielt, man kann jedoch immer noch eine Stufenbildung erkennen [17, Abschnitt 8.2.2]. Diese Technik ist aber im Verhältnis von Rechenaufwand zu Qualität eine der Besten, daher wird sie z. B. auch in der Videobranche gern verwendet.

Bei der *bikubischen Interpolation* werden auch die diagonal gelegenen Bildpunkte aus der Umgebung des Pixels in die Berechnung mit einbezogen. Einerseits wird der Rechenaufwand recht hoch, denn es werden 16 Nachbarpixel in der Berechnung berücksichtigt [17, Seite 256]. Andererseits erhält man qualitativ hochwertige Bilder mit hoher Schärfe [17, Abschnitt 8.2.3].

Die Namensgebung der bilinearen und bikubischen Interpolation stammt von der Tatsache, dass man beide auch als Funktionen darstellt, also als lineare und kubische Funktionen (nearest-neighbor wird daher auch als Interpolation nullter Ordnung bezeichnet) [17, Abschnitt 8.2]. Abbildung 4.1 zeigt den Vergleich von nearest-neighbor und bilinearer Interpolation. In der vergrößerten Ansicht sieht man deutlich, dass die Kanten gleichmäßiger werden und die Stufen verschwinden.

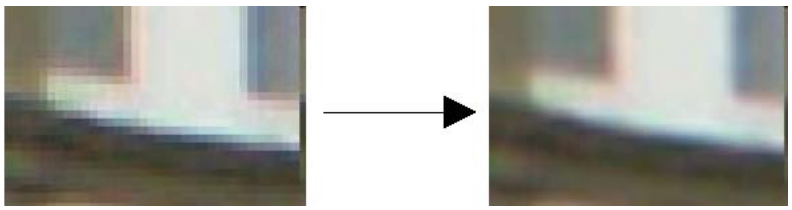

Abbildung 4.1: Nearest-neighbor (links) und bilineare (rechts) Interpolation [12].

JAI bietet darüber hinaus noch die Möglichkeit, mit Hilfe einer sogenannten *Table Interpolation* zu arbeiten. Neben dem höheren Rechenaufwand muss man aber genau wissen, wie die Quellbilder aufgebaut sind und welches Ergebnis man erwartet. Die oben genannten Typen lassen sich durch eine Table Interpolation nachbilden, nur der verwendete *Kernel*[6] ist entsprechend zu definieren [17, Abschnitt 8.2.5].

4.1.2 Pluginspezifische Probleme

Bei allen angesprochenen Typen fehlt der Bezug zwischen Bildpunkten und den im Bild enthaltenen Objekten. Nicht jedes Bild kann mit Metainformationen zu den darin befindlichen Objekten versehen werden. Es wird also ausschließlich

[6]Eine Matrix mit gezielt definierten Gleitkommawerten, die für jeden Pixel des Bildes einen neuen Wert aus den beteiligten Quellpixeln erzeugt. Die Gleitkommawerte und die Farb- bzw. Graustufenwerte des Bildes werden dazu mit einfachen Operationen verknüpft.

pixelbasiert gearbeitet. Es gibt Programme, die aus einem gegebenen Bild mit Hilfe von Kantendetektion versuchen eine entsprechende Vektorgrafik zu erzeugen. Vektorgrafiken lassen sich beliebig skalieren oder auch verzerren, ohne dass Interpolationsprobleme auftreten. Leider sind diese Programme nicht so effizient, dass deren Ergebnis direkt weiterverarbeitet werden könnte; es ist noch viel Handarbeit zu leisten.

Die Skalierung stellte für das Dotplot-Plugin deshalb ein großes Problem dar. Einerseits wird ein scharfes Bild benötigt. Andererseits existieren in der aktuellen Implementierung nach der Skalierung einmal das komplette Dotplot und einmal die skalierte Version für die Anzeige im Speicher. Eine weitere Kopie wird dann für eine an das Eclipse-Framework angepasste Version des Bildes benötigt, da zwischen den internen Darstellungen von AWT (`java.awt.Image`) und SWT (`org.eclipse.swt.graphics.ImageData`) konvertiert werden muss. Folge ist, dass der Speicher für die gestellte Anforderung schnell zu klein wird.

Der Code wurde im Zuge dieser Arbeit so verbessert, dass bei Kenntnis der gewünschten Darstellung (AWT/SWT) diese direkt erzeugt wird, um so weniger Kopien im Speicher halten zu müssen. Das Problem der Skalierung ist dadurch aber nicht gelöst: Da hochwertige Ergebnisse erwartet werden, reicht die im SWT implementierte Skalierung nicht aus. Sie arbeitet zwar sehr schnell, weist jedoch gegenüber der Skalierung mit JAI starke Einbußen in der Bildqualität auf.

Perfekte Skalierung und optimale Speichernutzung wird nur mit Vektorgrafik erreicht. Das Dotplot-Plugin kann aufgrund der Beschaffenheit der Quelldaten und deren interner Verarbeitung keine echte Vektorgrafik erzeugen. Dies erfordert eine andere Lösung.

4.2 Lösungsansatz Information Mural

Statt zuerst intern ein Ausgangsbild zu erzeugen und dieses kleiner zu skalieren, kann die skalierte Version direkt aus den Quelldaten gewonnen werden. Dadurch fällt das interne, nicht am Bildschirm verwendete Dotplot in voller Auflösung weg.

Eine Technik, die diese Idee verwendet, nennt sich Information Mural. Sie wurde entwickelt, um eine Ansicht über Datenmengen zu erzeugen, die so aussieht, als könne man alle Daten auf einmal anzeigen – praktisch ein Auszoomen, bis alles auf den Bildschirm passt.

Die Erzeugung eines Information Murals soll erreichen, dass möglichst wenige Informationen verloren gehen. Bei der bildpunktbasierenden Interpolation kann auf keine Informationen außer dem Bild selbst zurückgegriffen werden, daher gehen dort Einzelheiten oft verloren.

In Abbildung 4.2 wird am unteren Rand ein Information Mural als Navigationshilfe dargestellt. Das rote Rechteck entspricht dem im großen Bereich angezeigten Ausschnitt. Man kann durch Verschieben des Rechtecks den angezeigten Ausschnitt wählen und behält trotzdem immer die Übersicht über den gesamten Datensatz. Die Färbung wurde nicht durch das Information Mural erzeugt, sondern manuell durch eine LUT definiert, die links angezeigt ist. Information Murals sind zunächst nur Graustufenbilder.

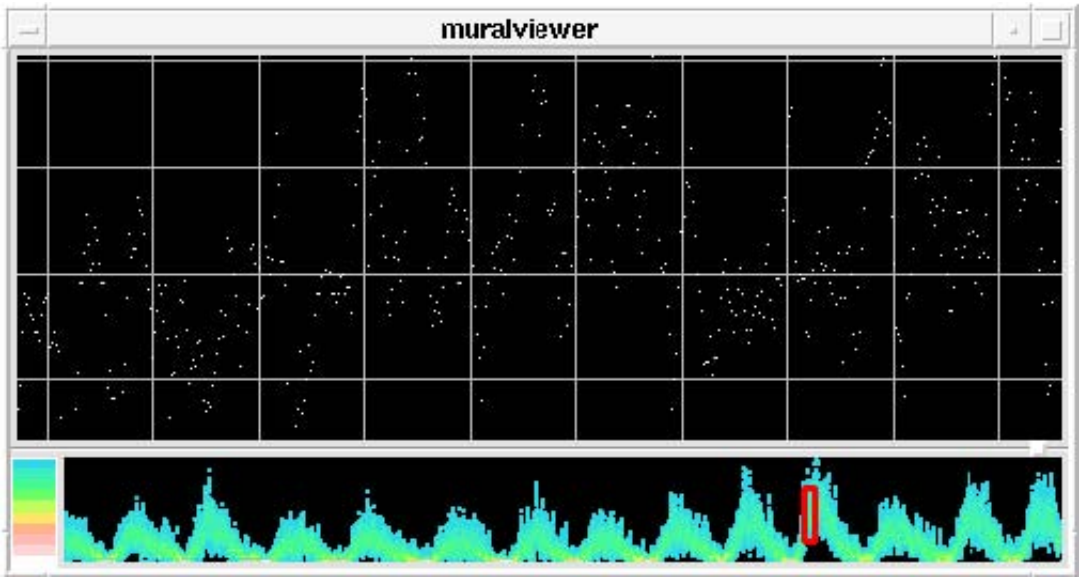

Abbildung 4.2: Das Information Mural (unten) als Navigationshilfe [10, Seite 12].

4.2.1 Algorithmus

Auf den ersten Blick betrachtet ähnelt der Algorithmus stark einer normalen Interpolation. Aus den Quelldaten wird ein Bereich ausgewählt und eine Funktion darauf angewendet. Es ergibt sich ein einzelner Wert, der als Bildpunkt im Zielbild erscheint. Der Unterschied besteht darin, dass eher auf einer logischen Ebene skaliert wird.

Die Quelldaten für einen Zielpixel werden aufeinander addiert statt direkt den Durchschnitt aus ihren Farbwerten zu berechnen. Wenn der gesamte Datensatz gelesen wurde, also für alle Zielpixel die korrespondierenden Quelldaten addiert wurden, hat man für jeden Bildpunkt des Information Murals eine Gewichtung, die der Summe aller Quellwerte entspricht. Man kann nun, über das gesamte Mural verteilt, die Gewichtungen von 0 bis 100 Prozent einer Graustufenskala zuordnen und erhält so das fertige skalierte Bild. Abbildung 4.3 verdeutlicht die interne Darstellung als Ansammlung von Säulen. Säulen gleicher Höhe entsprechen der gleichen Farbe im späteren Dotplot.

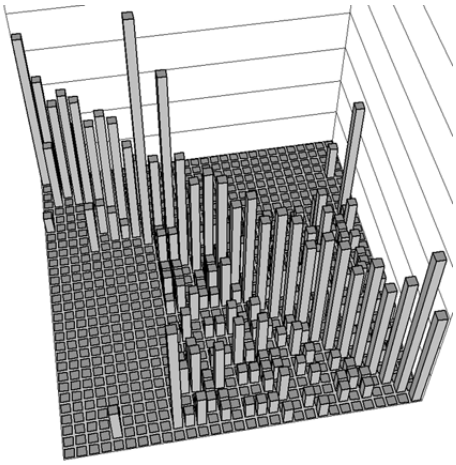

Abbildung 4.3: Die interne Darstellung des Information Murals.

Ein weiterer Unterschied zu den herkömmlichen Interpolationstechniken besteht darin, dass in diesem Algorithmus als Quelldaten nur Binärbilder brauchbar sind. Verwendet man Graustufenbilder, muss man in den Algorithmus eine Bewertung der einzelnen Stufen einbauen, damit er ein sinnvolles Ergebnis liefert. Dazu und zu einer anwendungsspezifischen Einfärbung des resultierenden Murals, siehe Abschnitt 4.2.2.

Der Algorithmus ermittelt zunächst die Zielkoordinate des Pixels. Diese Koordinate wird auf eine im Zielraster gültige Position gesetzt, da die Berechnung selten ganzzahlige Werte liefert. Die Nachkommastellen werden dazu einfach abgeschnitten. Nun wird für die ermittelte Position ein Zähler aktualisiert. Zusätzlich wird geprüft, ob der Wert des Zählers das Maximum im gesamten Mural darstellt. Gegebenenfalls wird dieses Maximum ebenfalls auf den neuen Wert gesetzt.

Nachdem diese Schritte für jeden Pixel durchgeführt wurden, ist das Information Mural eigentlich fertig. Da man es aber noch in eine Datei oder auf dem Bildschirm ausgeben möchte, *normalisiert* man die Zähler. Dieser Vorgang ist das Auslesen der absoluten Werte im Verhältnis zum Maximum. Man erhält dadurch Gleitkommawerte zwischen 0 und 1.

Hier der abgewandelte Algorithmus [10, Seite 18]:

- für jeden Pixel (m, n) im Original
  - berechne die Zielposition (x, y)
  - addiere 1.0 auf $mural\_array[floor(x)][floor(y)]$
  - aktualisiere max_mural_value
- normalisiere alle Werte in $mural\_array$ auf den Bereich von 0 bis 1

Der ursprüngliche Algorithmus [10, Seite 17] berechnete noch die Gewichtung des Zielpixels in Bezug auf die umgebenden Quellpixel. Ohne diese Gewichtung wird der Algorithmus jedoch schneller und, darüber hinaus, das Bild schärfer. Die Gewichtung würde einem *Anti-Aliasing*, also einer Art Interpolation, wie bei der konventionellen Skalierung entsprechen. Der hier gewünschte Nutzen des Information Murals wäre wieder verloren.

4.2.2 Anwendungsspezifische Anpassungen

Zu beachten ist, dass die Einschränkung auf Binärbilder aus der Addition mit 1.0 auf die Einträge im *mural_array* resultiert. Wandelt man diesen Schritt ab, so können auch Graustufenbilder verarbeitet werden. Es ist dann zu beachten, dass es in diesem Schritt zu keinem Datenverlust kommt. Durch das Aufsummieren verliert man die Information, ob zehnmal 0.1 addiert oder nur einmal 1.0 in das Array geschrieben wurde [10, Seite 20]. Diese Information wäre relevant für die spätere Entscheidung, welchen Farbwert (oder welche Graustufe) der Pixel erhalten soll.

Das Normalisieren auf den Bereich von 0 bis 1 ist die Ursache dafür, dass man als Resultat Graustufenbilder erhält. Hier muss man ansetzen, wenn der Algorithmus farbige Bilder generieren soll. Man kann dazu mit Hilfe von Lookup Tables die Graustufenskala auf das gewünschte Farbspektrum abbilden und so einen abgestuften Farbkeil, wie in Abbildung 4.2 auf Seite 37 verwendet, nutzen.

Will man die Farben aus einem farbigen Quellbild weiter verwenden, muss man sich bewusst sein, dass in dem Mural anstelle mehrerer verschiedenfarbiger Punkte ein einzelner tritt, dem natürlich nur eine einzelne Farbe zugeordnet werden kann. Es macht also Sinn, die Farbe in dem Mural zu verwenden, die dem höchsten Gewicht aus den Quelldaten entspricht. Dazu sollte aber entweder der ursprüngliche Algorithmus mit den Gewichtungen verwendet werden, oder man speichert parallel zu dem *mural_array* die Farbe (inkl. Gewicht) der aktuellen Zielfarbe in einem *color_array*. Wenn dann ein Bildpunkt mit höherem Gewicht einbezogen wird, muss dies in dem *color_array* dokumentiert werden. Später, bei der Normalisierung und der Zuordnung der Mural-Pixel zu Farben, kann dieses *color_array* ausgelesen werden. [10, Seite 20]

In der Regel ist aber die Erzeugung eines Graustufenbildes mit anschließender Abbildung durch eine LUT ein effizienterer Weg. Durch den gezielten Aufbau der LUT (z. B. mit hohem Kontrast) lassen sich viele Effekte erzielen. Es existieren noch weitere Möglichkeiten das Mural einzufärben [10, Seite 20], jedoch muss in den meisten Fällen die jeweilige Anforderung an das Ergebnis berücksichtigt werden.

Da beim Datenaustausch auch die Gewichte erhalten bleiben sollen, wird das Information Mural nicht beim Export eines Dotplots in Dateien verwendet.

4.3 Implementierung

Die Implementierung war recht einfach, da der Algorithmus keine besonderen Voraussetzungen benötigt und nur Basisoperationen beinhaltet. Um den Wiederverwendungsgrad möglichst hoch zu halten, wurde eine eigene Klasse `InformationMural` mit einem einzelnen Einstiegspunkt `getMural()` definiert.

Außerdem wurde, um den Algorithmus von den verschiedenen im Plugin verwendeten Bildformaten zu entkoppeln, ein Interface `ImageCallback` definiert, das vom `InformationMural` gefüllt wird. Der Aufrufer muss für eine Verwendung im Information Mural nur die Methode `setPixel()` implementieren.

Da das `ImageCallback` (siehe Listing 4.4) im Zuge der Umstellungen während dieser Arbeit auch an anderen Stellen im Plugin Verwendung fand, besitzt es mehr Methoden, als für das Information Mural nötig sind. Hier wird lediglich `setPixel()` verwendet.

Dieser Methode wird als Farbwert ein Integer übergeben, dessen Aufbau dem in Abbildung 4.4 entspricht. Für jeden Kanal stehen acht Bit zur Verfügung, wobei der Alpha-Kanal nicht benötigt wird. Dieser Aufbau wird auch bei der Darstellung am Bildschirm verwendet, es sind also keine weiteren Konvertierungen notwendig.

Abbildung 4.4: Interner Aufbau eines Farbwertes bei Verwendung eines Integers als Container.

Während eines Plots arbeitet `getMural()` in drei Stufen:

1. Erzeugen eines internen zweidimensionalen `int`-Arrays
2. Auslesen der F-Matrix und Füllen des `int`-Arrays
3. Daten-Transfer in das `ImageCallback`

Listing 4.1 zeigt einen kleinen Code-Ausschnitt. Das Interface `ITypeTableNavigator` ist die Verbindung zur F-Matrix, es liefert die einzelnen Matches. Es müssen noch die gewünschte Ausgabegröße und das `ImageCallback` übergeben werden. Bei späteren Erweiterungen kann man den

`ITypeTableNavigator` durch einen `Iterator` ersetzen, dadurch könnte man das Information Mural auch für andere Daten-Quellen verwenden.

Listing 4.1: Einstiegspunkt `getMural()` für das InformationMural

```
static void getMural(ITypeTableNavigator navData,
                     Dimension targetSize,
                     ImageCallback image)
```

Die erste Stufe dient nur der allgemeinen Vorbereitung der benutzten Datenstrukturen. Listing 4.2 zeigt die Implementierung der zweiten Stufe. Dort wird in den Zeilen 78 bis 81 das Verhältnis zwischen Original- und Zielgröße ermittelt. Dieses Verhältnis beeinflusst die Berechnung der Zielkoordinaten in den Zeilen 86/87. Der *Cast*[7] auf `int` dient dem „Einrasten“ auf gültige Bild-Koordinaten.

Listing 4.2: Auslesen der F-Matrix

```
66 private static float fillMuralArray(Dimension targetSize,
                                       Dimension originalSize,
                                       ITypeTableNavigator navData,
                                       int[][] mural_array)
   {
71  Match p;
    int mural_x, mural_y;
    float max_mural_array_value = 0;

    float xFactor
76       = (float) targetSize.width / (float) originalSize.width;
    float yFactor
         = (float) targetSize.height / (float) originalSize.height;

    while ((p = navData.getNextMatch()) != null)
81  {
      // calculate target indices
      mural_x = (int) ((float) p.getX() * xFactor);
      mural_y = (int) ((float) p.getY() * yFactor);

86    mural_array[mural_x][mural_y]++;

      // update max_value
      if (mural_array[mural_x][mural_y] > max_mural_array_value)
        max_mural_array_value = mural_array[mural_x][mural_y];
91  }
    return max_mural_array_value;
   }
```

[7]Programminterne Konvertierung zwischen Datentypen. Bei der Konvertierung eines Gleitkommawertes in einen Integer werden die Nachkommastellen einfach abgeschnitten, es erfolgt also eine Rundung auf den nächsten ganzzahligen Wert $\leq$ dem Quellwert.

Das `mural_array` wird im letzten Schritt durch das `ImageCallback` in das gewünschte Bild übertragen (Listing 4.3). Die `if`-Abfrage dient der Performance-Steigerung.

Die in Listing 4.3 verwendete Klasse `QImageConfiguration` wird zur Abfrage der LUT verwendet. Möchte man das `InformationMural` in einer Umgebung außerhalb des Dotplot-Plugins einsetzen, muss man die LUT als Parameter übergeben.

Der Algorithmus ist also leicht zu implementieren und ggf. an andere Anforderungen anzupassen. Für spätere Erweiterungen sollte man die Gewichtung der Matches in die Färbung einbeziehen. Dazu ist es, wie oben bereits erwähnt, notwendig eine weitere Datenstruktur mit dem *mural_array* zu verknüpfen und diese in geeigneter Weise auszulesen [10, Seite 20].

Listing 4.3: Übertragung ins `ImageCallback`

```
  private static void transferToImage(int[][] mural_array,
33                                    ImageCallback image,
                                      float max_mural_array_value,
                                      QImageConfiguration config)
  {
    int col;
38  float maxColVal = Util.COLOR_COUNT_PER_BAND - 1;

    // cached values to improve performance
    int colBackground = config.getLutBackground().getRGB();
    int colForeground = config.getLutForeground().getRGB();
43  int[][] lut = config.getLut();

    for (int x = 0; x < mural_array.length; x++)
    {
      for (int y = 0; y < mural_array[x].length; y++)
48    {
        if (mural_array[x][y] == 0) {
          image.setPixel(x, y, colBackground);
        }
        else if (mural_array[x][y] == max_mural_array_value) {
53        image.setPixel(x, y, colForeground);
        }
        else {
          col = (int) (mural_array[x][y]
                       / max_mural_array_value * maxColVal);
58        image.setPixel(x, y, new Color(lut[0][col],
                                         lut[1][col],
                                         lut[2][col]).getRGB());
        }
      }
63  }
  }
```

Listing 4.4: Das Interface `ImageCallback`

```
interface ImageCallback
{
   void setPixel(int x, int y, int rgb);
   int getPixel(int x, int y);
   void invertLine(int index);
   void updateProgress(int diff, int curStep, String msg);
   java.awt.Dimension getSize();
}
```

4.4 Arbeiten mit dem Information Mural

Nach der Implementierung des Information Murals war ein großer Leistungsgewinn erkennbar. Die erzeugten Dotplots entsprechen jedoch nicht den konventionell erzeugten Bildern, da das durch die F-Matrix erzeugte Gewicht aus oben genannten Gründen nicht berücksichtigt wird. Es sind in der Regel mehr Details erkennbar, da *alle* Matches aus der F-Matrix gleichwertig in das Information Mural eingebunden werden.

Das Information Mural filtert also keine irrelevanten Informationen aus; es ist zu beachten, dass man das Information Mural nicht als alleinige Entscheidungshilfe, sondern nur als Hilfe bei der Navigation im Dotplot verwenden sollte. Für zukünftige Versionen des Plugins könnte man einen kleinen Vorschaubereich einbinden, der immer ein Information Mural als Übersichtskarte zeigt, während im Hauptfenster konventionell erzeugte Ausschnitte des Gesamtplots angezeigt werden. Abbildung 4.2 auf Seite 37 zeigt eine Möglichkeit, wie dies umgesetzt werden könnte.

Um die Arbeit mit dem Plugin einfach zu gestalten, wird per Default mit Hilfe des Information Murals geplottet. Will man einen Plot auf herkömmliche Weise erzeugen, lässt sich dies durch eine Einstellung in der Konfiguration festlegen. Nur beim Speichern in eine Datei wird immer das JAI verwendet. Das liegt daran, dass beim Export jegliche Skalierung deaktiviert ist und JAI außerdem die besten Ergebnisse liefert.

Zum Qualitätsvergleich sind in den Abbildungen 4.5, 4.6 und 4.7 die Ergebnisse der im Plugin möglichen Techniken demonstriert. Es ist zu beachten, dass die Qualität stark von den Quelldaten abhängt.

In jeder Abbildung ist ein Bereich markiert, der für den Anwender interessant sein könnte. In diesem Bereich erkennt man nach der Skalierung durch das SWT keine

Auffälligkeiten mehr – allgemein sieht der Ausschnitt so aus, als seien kaum Matches gefunden worden. Das Information Mural ist wesentlich besser. Es werden zwar kaum Informationen ausgefiltert, man erkennt aber speziell die Trefferdichte sehr gut. Die Erzeugung des Dotplots braucht für beide Varianten (SWT/Information Mural) etwa die gleiche Zeit, siehe Kapitel 5. Man sollte also das Information Mural verwenden um einen groben Überblick zu erhalten und dann für kleinere Ausschnitte auf das JAI umschalten.

JAI produziert optisch ansprechende Bilder, was durch das hohe Niveau der Interpolation ermöglicht wird. Das Ziel, wichtige Matches hervorzuheben und irrelevante auszublenden, wird optimal erreicht. Rechenaufwand und Speicherbedarf sind jedoch entsprechend hoch.

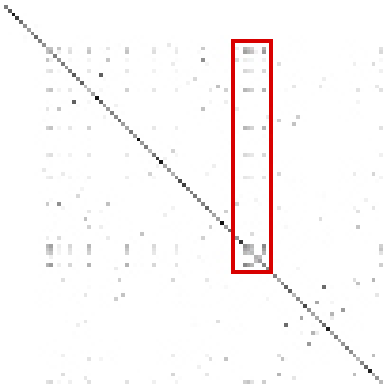

Abbildung 4.5: Qualitätsvergleich: Skalierung durch JAI.

Bei der Skalierung durch das *SWT* erkennt man starke Verluste in der Bildqualität. Auch höherwertige Matches können bei der Skalierung verloren gehen. Diese Technik ist also nur unter Vorbehalt verwendbar.

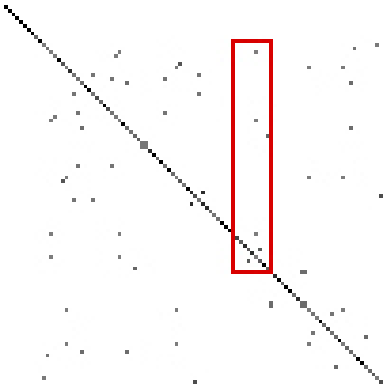

Abbildung 4.6: Qualitätsvergleich: Skalierung durch SWT.

Das *Information Mural* zeigt, wie oben schon angesprochen, *alle* Matches und berücksichtigt dabei nicht die Gewichtung. Details verschwinden stellenweise im allgemeinen Rauschen, Bereiche mit höherer Dichte fallen trotzdem noch auf.

Abbildung 4.7: Qualitätsvergleich: Skalierung durch das Information Mural.

4.5 Andere Anwendungen

Das Information Mural ist nicht nur für eine Skalierung sinnvoll, sondern kann allgemein zur Darstellung von beliebigen Daten herangezogen werden. Die folgenden Beispiele demonstrieren den praktischen Einsatz in verschiedenen Bereichen wie Wissenschaft/Forschung und sogar Demographie und verdeutlichen die Vorteile der Technik.

4.5.1 Analyse von Sonnenflecken

Abbildung 4.2 auf Seite 37 zeigt die bei der Beobachtung von Sonnenflecken gesammelten Daten [10, Seite 12]. Ohne das Information Mural am unteren Rand kann man sich in den recht stark gestreuten Punkten kaum zurecht finden.

Die zweite Eigenschaft des Information Murals, insbesondere die Treffer*dichte* optimal darzustellen, wird in den Abbildungen 4.8 und 4.9 deutlich: die erste zeigt auf konventionelle Weise die Sonnenaktivität aus dem Zeitraum von 1850-1993, wobei aus Platz- und Effizienzgründen lediglich die monatlichen Durchschnittswerte dargestellt werden.

Erst mit Hilfe des Information Murals in der zweiten Abbildung treten zwei Details hervor, die durch die Durchschnittsbildung verloren gehen: Die Lücke im unteren Bereich ist kein Druckfehler, sondern das Fehlen von Datensätzen. Durch die gezielte Einfärbung wird außerdem deutlich, dass viele Nullen im Datensatz enthalten sind (die helleren Bereiche am äußersten unteren Rand) [10, Seite 8]. Das Information Mural kann also nicht nur eine Durchschnittsbildung ersetzen, sondern auch die Verteilung der Daten darstellen. Die Menge der Daten ist nun unproblematisch, da immer passend skaliert werden kann ohne Datenverlust in Kauf nehmen zu müssen.

4.5.2 Darstellung der Bevölkerungsdichte

Die Verknüpfung von Geographie mit Bevölkerungsdaten ist eine Anwendung von *Geographischen Informations-Systemen* (GIS). Bei der Planung einer neuen Restaurantfiliale ist es beispielsweise notwendig, die potentielle Kaufkraft in dem gewünschten Gebiet zu ermitteln. Dazu bedient man sich demographischer Daten und Statistiken, die regelmäßig erhoben werden. Die Statistiken sind aufgrund ihrer Fülle schwierig auszuwerten, deshalb bedient man sich grafischer Darstellungen. [4]

Abbildung 4.10 zeigt eine solche Grafik, die pro Staat die Bevölkerungsdichte der USA zeigt. Die Darstellung kann auch für einen einzelnen Staat erfolgen.

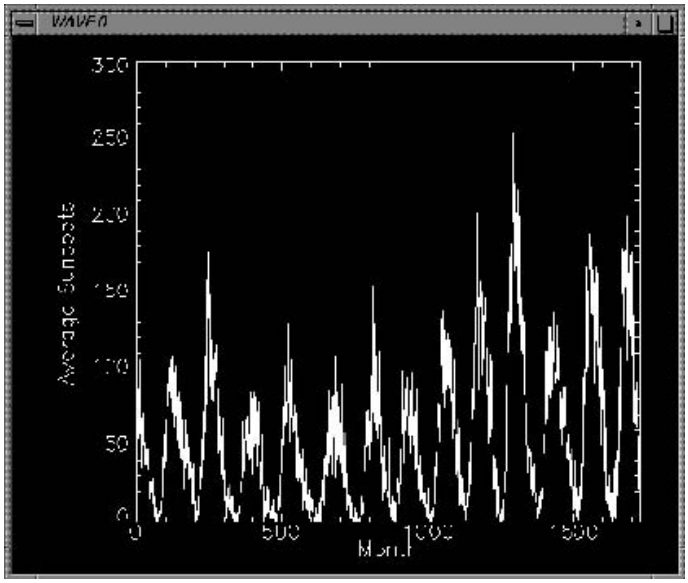

Abbildung 4.8: Monatliche Anzahl der Sonnenflecken, aus dem Zeitraum 1850-1993 [10, Seite 10].

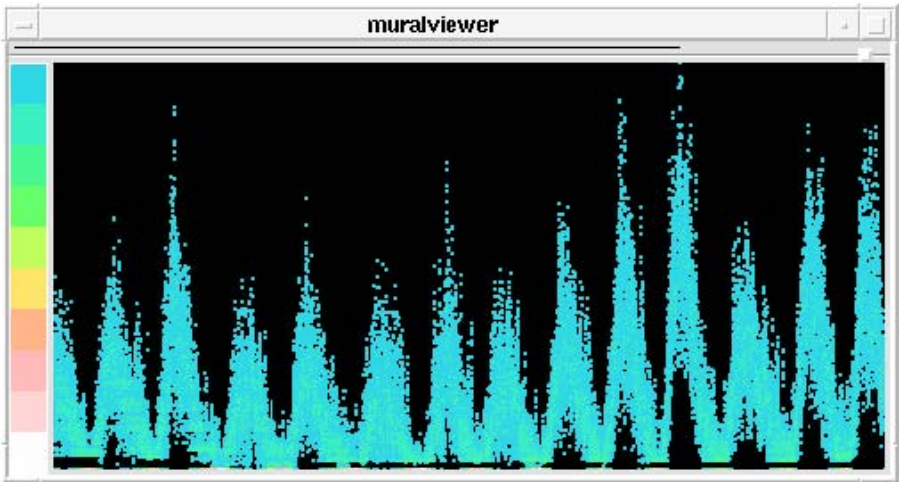

Abbildung 4.9: Anzahl der Sonnenflecken, aus dem Zeitraum 1850-1993, dargestellt mit dem Information Mural [10, Seite 10].

Dann müssten die Einwohnerzahlen der Städte mit einbezogen und die Grafik entsprechend angepasst werden. Ein alternativer Weg ist, die Bevölkerungsdichte mit Hilfe des Information Murals darzustellen, siehe Abbildung 4.11. Hier werden keine Durchschnittswerte (pro Staat) mehr gebildet, sondern alle verfügbaren Daten komplett angezeigt. Sinnvoll könnte das Einblenden der Staatsgrenzen sein, um eine bessere Orientierung im Bild zu erreichen. Doch schon in dieser groben Übersicht können Entscheidungen getroffen werden, die mit Hilfe der ersten Abbildung ausgeschlossen sind. Auch hier kann das Information Mural helfen, effizient zu arbeiten.

Interessant ist der Vergleich mit dem Satellitenfoto[8] in Abbildung 4.12, aufgenommen bei Nacht: Das Foto ist dem Information Mural sehr ähnlich. Man erkennt auf beiden Bildern, dass die Küstengebiete stärker bevölkert sind und beispielsweise die Rocky Mountains nur eine geringe Bevölkerungsdichte aufweisen. Vielleicht lassen sich aus Vergleichen solcher Bilder weitere Erkenntnisse oder auch neue Arbeitstechniken bei der Verwendung des Information Murals erzielen? Möglicherweise kann mit Hilfe genauerer Methoden aus den Farbwerten von Satellitenfotos auf die Bevölkerungsdichte zurückgerechnet werden.

[8]Das Foto ist eine Fotomontage, die Teilbilder wurden über einen längeren Zeitraum (Oktober 1994 bis März 1995) erstellt, um eine wolkenfreie Sicht vom Satelliten auf die Erde zu nutzen.

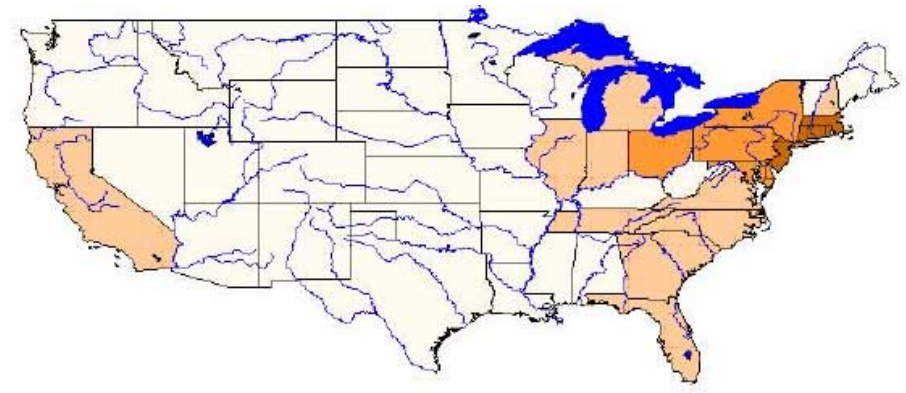

Abbildung 4.10: Bevölkerungsdichte pro US-Staat [3, Seite 9].

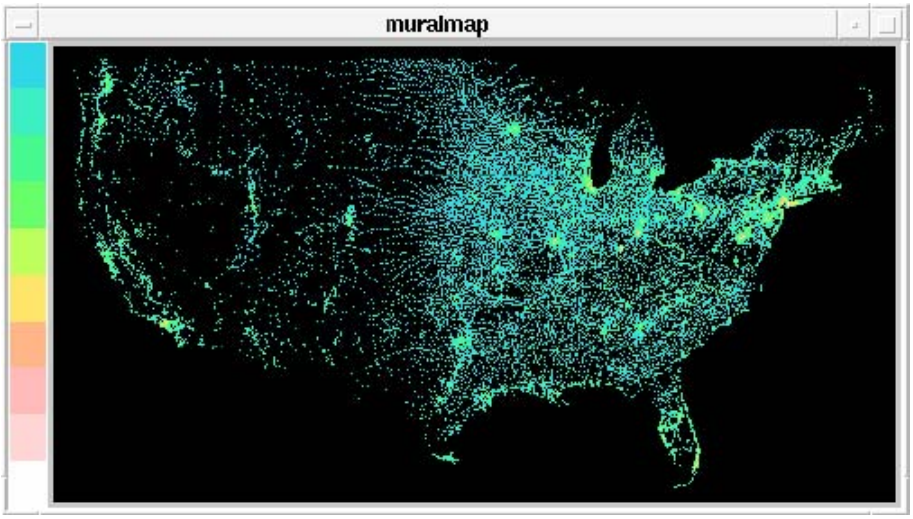

Abbildung 4.11: Information Mural zur Bevölkerungsdichte der USA [10, Seite 15].

Abbildung 4.12: Satellitenfoto der USA, aufgenommen bei Nacht [13].

Kapitel 5

Auswertung

Die beiden vergangenen Kapitel haben zwei verschiedene Ansätze zur Verringerung der dort und im Kapitel 2 genannten Performanceprobleme vorgestellt. Dieses Kapitel beschäftigt sich mit der Bewertung dieser beiden Techniken zur Effizienzsteigerung bei der Visualisierung großer Datenmengen.

5.1 Testsysteme

Die Performancetests wurden auf verschiedenen Plattformen durchgeführt, um eine bessere Aussage über die Performancesteigerung treffen zu können. Um auch eine mit durchschnittlichen Rechnern vergleichbare Bewertung erstellen zu können, wurde das Plugin auf einem Pentium III-PC mit 700 MHz und 512 MB RAM getestet. Ein weiterer Rechner der Pentium IV-Klasse mit 3 GHz und 512 MB RAM dient zum Vergleich mit aktuellen Rechnern. Auf beiden Systemen wurden Linux und Windows verwendet.

Darüber hinaus dienten der „Saturn“, ein Sun-Server *Sun Fire V880* unter dem Unix-Derivat *Solaris*, und Windows-Rechner aus dem PC-Labor des Fachbereichs MNI als Testplattformen. Der „Saturn“ verfügt über acht Prozessoren und 32 GB RAM. Es sind ca. 50 Clients an das System angeschlossen. Die Windows-PCs im MNI-Labor besitzen einen Pentium IV mit $1,5$ GHz und 256 MB RAM. Tabelle 5.1 zeigt den Überblick über die verwendeten Systeme.

Die Tests mit dem Information Mural sind gut mit anderen Systemen vergleichbar. Im Gegensatz dazu sind die Test-Ergebnisse zum Grid stark von der kompletten Infrastruktur (Server, einzelne Clients, Netzwerk) abhängig. Speziell bei zu wenig Arbeitsspeicher ist die jeweilige Festplattengeschwindigkeit durch betriebssysteminterne Swap-Mechanismen ein begrenzender Faktor. Die im folgenden aufgeführten Angaben zeigen also nur eine Tendenz.

Obwohl der „Saturn“ über acht Prozessoren verfügt, konnte das Plugin diese nicht ausnutzen: Die Prozessoren und der Arbeitsspeicher werden unter allen aktiven Clients aufgeteilt, so dass nicht die gesamte Infrastruktur zur Verfügung stand. Ein „normaler“ User erhält nicht die gesamte Rechenleistung, sondern er kann nur eingeschränkt (z. B. über das Tool *nice*, siehe auch dessen Manpage) die Priorität von Prozessen steuern. Darüber hinaus konnte die Java-VM unter Solaris nur maximal 3 GB Arbeitsspeicher ansprechen.

5.2 Ergebnisse der Performance-Tests

Den Auswertungen liegen die parallel zu einem Plot protokollierten Daten der Log-Datei zugrunde. Der aktuelle Status wird darin immer mit einem Zeitstempel markiert.

5.2.1 Information Mural

Das zugrunde liegende Betriebssystem hatte keinen signifikanten Einfluss auf die Performance. Nur bei der Verwendung von JAI in Kombination mit der von Sun bereitgestellten nativen Bibliothek *MediaLib* hat Windows minimale Vorteile. Das liegt an der unter Windows genutzten Möglichkeit zur Verwendung von *MMX*[1].

Als Referenz dienten die Plugin-Sourcen des Pakets `org.dotplot.grid` (siehe auch Abbildung 3.3 auf Seite 25) in der aktuellen Version `2.0.0`. In voller Auflösung ergeben diese Klassen ein Dotplot mit einer Seitenlänge von ca. 2800 Pixeln. Auf einem durchschnittlichen Rechner ist diese Größe bei Verwendung von JAI eine große Herausforderung. Die in dieser Arbeit vorgestellten Techniken sollten also einen spürbaren Performancegewinn zeigen.

Tabelle 5.2 zeigt die Ergebnisse der Zeitmessungen für einen Plot per Information Mural im Vergleich zum selben Plot per JAI und SWT. Als VM-Parameter wurden `-Xms768M` und `-Xmx768M` verwendet. Dadurch kann die Java-VM schon beim

[1] *M*ulti*m*edia E*x*tensions, ein im Prozessor integrierter Befehlssatz für multimediale Anwendungen.

Bezeichnung	CPU, Taktfrequenz	RAM
System 1	Pentium III, 700 MHz	512 MB
System 2	Pentium IV, 1.5 GHz	256 MB
System 3	Pentium IV, 3.0 GHz	512 MB
System 4	8 * UltraSparc III, 750 MHz	32 GB

Tabelle 5.1: Verwendete Systeme für die Performance-Tests.

Start des Plugins ausreichend Speicher alloziieren. Wenn nicht genügend physischer Arbeitsspeicher zur Verfügung steht, wird automatisch auf die Festplatte ausgelagert.

Um die Algorithmen möglichst isoliert bewerten zu können wurde die Ausgabe der Dateigrenzen deaktiviert. Die Einblendung der Dateigrenzen verlangsamt die Bilderzeugung abhängig von der Bildgröße und der Anzahl der Dateien. Skaliert wurde auf 290 Pixel Seitenlänge, also ca. 10% der Original-Größe. Die Zeiten sind in Sekunden angegeben. Die Angabe von Millisekunden war notwendig, um die Zeiten für das Information Mural noch erfassen zu können. Obwohl in diesem Bereich keine exakte Aussage getroffen werden kann, geben die Zahlen den Grad der Performancesteigerung wieder.

System	JAI	SWT	IM
1	118,718s	3,891s	1,015s
2	114,795s	2,023s	0,551s
3	39,469s	1,157s	0,354s
4	134,386s	12,209s	2,951s

Tabelle 5.2: Performancevergleich des Information Murals (*IM*) mit JAI und SWT für verschiedene Systeme.

Man kann gut erkennen, dass bei der Arbeit mit JAI längere Wartezeiten einkalkuliert werden müssen. Die Ursache dafür liegt aber in der schon angesprochenen Konvertierung des Bild-Formats von JAI/AWT nach SWT. Tabelle 5.3 zeigt die Zeitmessungen für JAI im Detail.

Schritt	System 1	System 2	System 3	System 4
Interne Erzeugung des Dotplots	12,359s	6,199s	5,047s	39,260s
Konvertierung nach SWT	106,359s	108,596s	34,422s	95,126s
Summe	118,718s	114,795s	39,469s	134,386s

Tabelle 5.3: Detaillierte Zeitmessung für JAI.

Die Zeitmessungen beim Plot per Information Mural lassen indirekt auch Aussagen über den Speicherverbrauch zu: Durch den effizienten Ablauf muss weniger ausgelagert werden und es wird häufiger direkt auf den Arbeitsspeicher zugegriffen. Die Vermeidung von Festplattenzugriffen schlägt sich natürlich auf die Verarbeitungsgeschwindigkeit nieder. Ein Plot per Information Mural benötigt so nur noch einen Bruchteil der Zeit gegenüber einem Plot per JAI. Tabelle 5.4 zeigt den Zeitgewinn, jeweils für einen kompletten Durchlauf per JAI und für die interne Erzeugung des Dotplots.

Um den unterschiedlichen Zeitbedarf zu verdeutlichen, zeigt Abbildung 5.1 die Daten aus Tabelle 5.4 grafisch aufbereitet. Es fällt auf, dass System Nr. 4 trotz

System	JAI komplett	JAI intern	IM	Zeitgewinn (Faktor)
1	118,718s	12,359s	1,015s	117x/12x
2	114,795s	6,199s	0,551s	208x/11x
3	39,469s	5,047s	0,354s	111x/14x
4	134,386s	39,260s	2,951s	46x/13x

Tabelle 5.4: Zeitgewinn bei Verwendung des Information Murals.

großem Arbeitspeicher lange für einen Plot benötigt. Das liegt an der Implementierung des nativen Codes für JAI, er kann die Fähigkeiten der Plattform nicht voll ausnutzen. Darüber hinaus wird ein Plot bisher nicht auf die vorhandenen Prozessoren verteilt. Während der Tests wurde beobachtet, dass dem Java-Prozess eine fortlaufend geringe Priorität zugewiesen wurde. Mit den entsprechenden Rechten (als User *root*) können solche Effekte vermieden werden. Hier sollten jedoch für den „normalen“ Anwender repräsentative Ergebnisse geliefert werden.

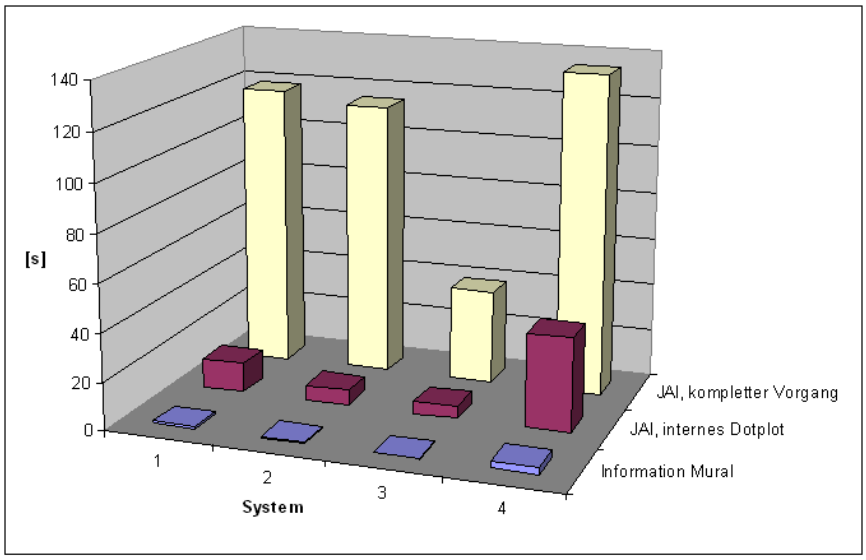

Abbildung 5.1: Grafik zum Zeitbedarf bei Verwendung des Information Murals. Die Zahlen 1 bis 4 entsprechen den vier Test-Systemen, siehe auch Tabelle 5.4.

Die Versuche zeigen, dass sowohl die Prozessorleistung als auch die Größe des Arbeitsspeichers Einfluss auf die Dauer eines Plots haben. Das Information Mural nutzt beide effizient aus und ermöglicht interaktives Arbeiten. Für eine auf die Bildschirmgröße skalierte Darstellung wird wenig Speicher benötigt und ein Plot in Sekundenbruchteilen angezeigt.

5.2.2 Grid

Für die Bewertung des Grids wurden sechs Systeme in einem Netzwerk verbunden. Alle sechs Rechner, im Folgenden als *C1* bis *C6* bezeichnet, besaßen die gleiche Ausstattung: Pentium `IV` mit 3, 2 GHz und 1 GB RAM. Die Rechner wurden über einen zentralen Switch zu einem sternförmigen 100 MBit-Netzwerk verbunden. C1 diente in jedem Durchlauf als Mediator.

Auf allen Rechnern wurde Windows XP mit Servicepack 2 installiert. Um einen Vergleich zwischen Windows und Linux durchzuführen, wurde auf drei der Rechner auch die Linux-Distribution SuSE 9.2 installiert. Die Performancevergleiche zeigten auch hier keine signifikanten Unterschiede zwischen den Betriebssystemen, daher sind unten nur die Ergebnisse für Windows aufgeführt.

Grund für die kaum unterschiedliche Performance ist die Tatsache, dass das Information Mural beim Aufbau des Dotplots verwendet wird und erst beim Export in eine Datei auf JAI zurückgegriffen wird. Sun hat speziell für Dateioperationen die eng mit JAI verzahnte Bibliothek *ImageIO* entwickelt. JAI wurde also in Bezug auf Dateiimport und -export bereits gut optimiert. Die oben genannten Schwächen beziehen sich auf andere Operationen wie z. B. die Skalierung oder das Zuweisen einer LUT.

Die Eclipse-Umgebung wurde nur auf C1 gestartet. Die Grid-Clients auf C2 bis C6 wurden über die Konsole, wie schon in Abschnitt 3.4.2 auf Seite 30 beschrieben, gestartet.

Als VM-Parameter wurden auf allen Systemen `-Xms1024M` und `-Xmx1024M` verwendet. Die ebenfalls über VM-Parameter konfigurierbare maximale Kantenlänge (`-Dgrid.maxedge=...`) wurde bei den Tests variiert, in der Tabelle sind die entsprechenden Werte angegeben.

Für den Performancevergleich wurden die Sourcen des E-Study-Portals und die kompletten Sourcen des Dotplot-Plugins jeweils mit sich selbst verglichen. Bei jedem Plot waren Textscanner und Linefilter aktiviert (mit der Option, leere Zeilen zu ignorieren). Tabelle 5.5 zeigt die Ergebnisse der Performancetests. Es wurde gemessen, wieviel Zeit der Versand des Plotjobs an die Clients (Spalte *Versand*) und das Bearbeiten jedes Teilplots (Spalten *C1* bis *C6*) benötigte.

Für die E-Study-Sourcen ergab sich ein Dotplot mit einer Kantenlänge von 113.894 Pixeln, der an jeden Client verschickte Plotjob hatte dadurch eine Größe von ca. 2 MB. Das Dotplot der Plugin-Sourcen besaß eine Kantenlänge von 26.426 Pixeln, der entsprechende Plotjob eine Größe von ca. 0, 5 MB. Diese Größenverhältnisse wirken sich beim Versand der Daten vom Mediator an die Clients aus.

Die Angaben in den Spalten *C1* bis *C6* geben die Dauer vom Start des Plots bis zur Empfangsbestätigung des `GridPlotters` wieder. In dieser Zeit sind also der Plot auf dem jeweiligen Client und der Versand des Teilbildes an den `GridPlotter` enthalten. In Spalte ∅ sind die Durchschnittszeiten eingetragen. Drei Versuche, markiert durch „†“, schlugen fehl, die Clients brachen wegen Speichermangels ab.

Source	Clients	Versand	C1	C2	C3	C4	C5	C6	∅
Dotplot	1	5s	†	–	–	–	–	–	–
(4800 Pixel)	2	5s	68s	74s	–	–	–	–	71s
	3	5s	52s	53s	54s	–	–	–	53s
	4	5s	30s	32s	33s	33s	–	–	32s
	5	5s	27s	28s	28s	31s	32s	–	29s
	6	5s	25s	26s	26s	27s	27s	32s	27s
E-Study	1	33s	†	–	–	–	–	–	–
(6000 Pixel)	2	31s	†	†	–	–	–	–	–
	3	34s	249s	249s	354s	–	–	–	284s
	4	32s	93s	97s	101s	110s	–	–	100s
	5	32s	78s	79s	79s	81s	83s	–	80s
	6	33s	66s	67s	67s	68s	70s	73s	69s

Tabelle 5.5: Zeitmessungen des Performancetests zum Grid.

In Abschnitt 3.5 wurde schon angesprochen, dass das Zusammenfügen der einzelnen Teilbilder noch viel Rechenleistung und Arbeitsspeicher benötigt. Die Dauer für den Aufbau des Gesamtplots hängt daher auch stark von der Festplattengeschwindigkeit ab. Da sich bei diesem Vorgang die Wartezeiten auf das Ergebnis verlängern, wurden in Tabelle 5.5 nur die Zeiten bis *vor* diesen Schritt angegeben. Verglichen mit professionellen Grafikprogrammen ist das Dotplot-Plugin unter diesem speziellen Aspekt bisher nicht empfehlenswert.

Abbildung 5.2 zeigt den Vergleich der Gesamtzeiten für jeden Plot. Die Durchschnittswerte aus Spalte ∅ wurden dazu auf die Werte der Spalte *Versand* addiert, siehe Tabelle 5.6. So werden die vom Anwender messbaren Zeiten wiedergegeben.

	Anzahl Clients					
	1	2	3	4	5	6
Dotplot	–	71s	53s	32s	29s	27s
E-Study	–	–	318s	132s	112s	102s

Tabelle 5.6: Gesamtdauer der Plots über das Grid.

Man erkennt, dass der Zeitgewinn bei jedem zusätzlichen Client immer geringer wird. Der Grund ist die höhere Belastung des Servers, der bei einer höheren Anzahl von Clients auf mehr Anfragen reagieren muss. Bei jedem zusätzlichen Client

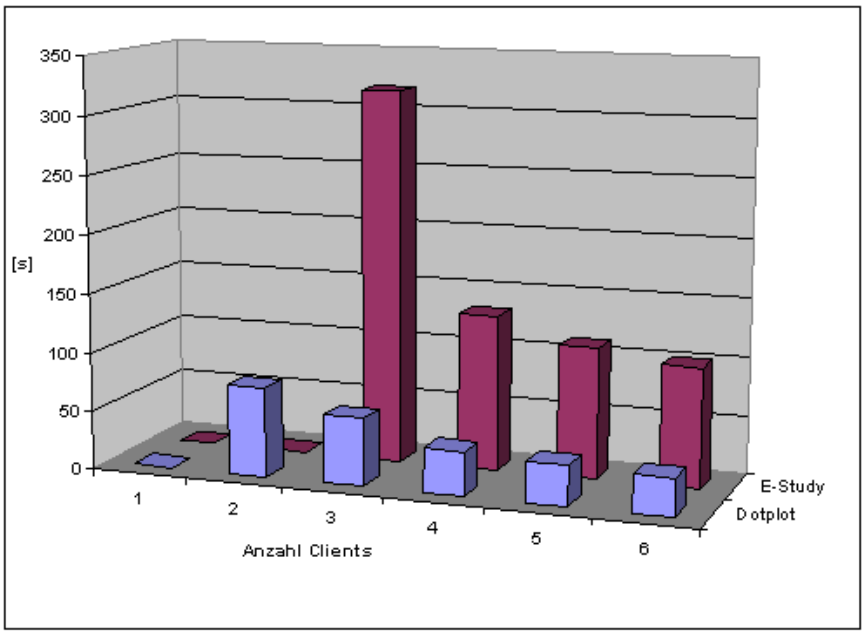

Abbildung 5.2: Grafik zum Zeitbedarf bei Verwendung des Grids, siehe Tabelle 5.6.

steigt die Wahrscheinlichkeit, dass zwei oder mehr Clients gleichzeitig ihren fertigen Teilplot schicken. Dies beansprucht speziell die Festplatte, da sie „gleichzeitig“ mehrere Dateien schreiben muss.

Unabhängig von der Anzahl der Clients wirkt sich auch die Größe der F-Matrix auf die Dauer eines Teilplots aus. Jeder Client muss den gesamten Datensatz lesen und entscheidet erst dann, welche Daten für ihn bestimmt sind. Die F-Matrix sollte daher hinsichtlich ihrer Aufteilung in kleine Blöcke optimiert werden. Weitere Anregungen zur Verbesserung und Erweiterung des Plugins stellt das folgende Kapitel vor.

Kapitel 6

Schlussbetrachtungen

Diese Arbeit zeigt neue Ansätze bei der Erzeugung und Verarbeitung von Dotplots. Die vorgestellten Techniken lassen sich leicht auf ähnliche Anwendungen übertragen. Dieses Kapitel weist auf bestehende Einschränkungen hin und macht Vorschläge für zukünftige Erweiterungen und neue Einsatzgebiete.

Exemplarisch für die Probleme bei der Visualisierung großer Datenmengen wurde die Technik zur Erzeugung von Dotplots vorgestellt. Obwohl sie auf einem einfachen Prinzip beruht, sind durchschnittliche Rechner schnell vom Datenaufkommen überlastet. Als Lösung hat diese Arbeit zwei Möglichkeiten vorgestellt.

Das Information Mural hat sich als stabil erwiesen und wurde im Plugin als Standard bei der Erstellung von Plots eingerichtet. Obwohl der Algorithmus einfach in ein Programm umzusetzen ist, scheint er noch keine große Verbreitung gefunden zu haben. Das wirft die Frage auf, ob das Information Mural noch zu unbekannt ist, oder ob es zwar schon in einigen Anwendungen implementiert wurde, aber aufgrund seiner Einfachheit nicht gesondert betont wird.

Bei der Trennung eines Clients vom Grid tritt unter bisher unbekannten Umständen auf dem Server ein Fehler auf, der die Arbeit mit den restlichen Clients stören kann. Wenn also die Struktur des Grid gestört wird und sich diese Störungen negativ auswirken, sollte das Grid neu aufgebaut werden.

Die Testergebnisse zu beiden Techniken machen deutlich, dass Problemen bei der Visualisierung großer Datenmengen auf verschiedenen Wegen begegnet werden kann. Durch die Kombination der Gridtechnologie mit dem Information Mural wurden die Optimierungsmöglichkeiten voll ausgereizt. Alleine der Vorgang zum Zusammenfügen der Teilbilder eines Plots über das Grid ist noch zu verbessern. Das Grid befindet sich aber noch in einem recht frühen Entwicklungsstadium, so dass bei den geplanten Erweiterungen auch dieses Problem gelöst werden sollte.

Mit Hilfe der beiden Erweiterungen kann man effizient arbeiten, was vorher nur eingeschränkt möglich war. Der Zeitgewinn bei der Arbeit mit dem Plugin ist der wichtigste Aspekt, aber die bessere Nutzung der Rechnerressourcen ermöglicht erst die Verwendung auf einer größeren Auswahl an Plattformen.

6.1 Ausblick

Das Information Mural kann in Zukunft als Grundlage für eine automatisierte Analyse von Dotplots verwendet werden: Die interne Darstellung in einem Integer-Array kann gut an andere Module weitergereicht werden, um beispielsweise eine automatisierte Mustererkennung umzusetzen. Die in Kapitel 2 vorgestellten Muster können durch ein Programmmodul extrahiert und für den Anwender hervorgehoben werden. Diese Muster können auch als Basis für eine Bewertung des Dotplots Verwendung finden.

Wie schon in Kapitel 4 angesprochen, ist eine Erweiterung zur Berücksichtigung der Gewichte aus der F-Matrix sinnvoll. Dies sollte unter Erhalt der aktuellen Performance geschehen, um den Nutzen des Information Murals nicht zu beeinträchtigen.

Das Grid bietet noch mehr Potential für Erweiterungen. Es wurde schon erwähnt, dass das Grid bisher nicht durch aufwendige Techniken vor Missbrauch geschützt wird. Man muss sich vor der Implementierung von Modulen zur Erhöhung der Sicherheit allerdings die Frage stellen, ob eine allgemeine Absicherung des Rechners nicht sinnvoller ist, da für das Grid lediglich das Öffnen eines einzelnen Ports notwendig ist.

Wichtiger ist wohl die Umsetzung einer besseren Fehlerbehandlung, die den Ausfall einzelner Knoten toleriert. Der Mediator sollte den Ausfall eines Knotens kompensieren, indem er den entsprechenden Teilplot an einen freien Knoten schickt. In diesem Zusammenhang könnte auch eine Bewertung der angeschlossenen Knoten implementiert werden. Dazu könnte vor dem eigentlichen Plot ein kurzer, aber aussagekräftiger Performancetest an jeden Client geschickt und ausgewertet werden. Abhängig von der einzelnen Performance kann der Mediator dann gezielt die Bereiche für jeden Client zuordnen.

Der in dieser Arbeit vorgeführte Vergleich zwischen wortweise und zeilenweise erstellten Plots hat verdeutlicht, dass es bei einer Auswertung sinnvoll ist, logische Gruppen von Wörtern zu vergleichen. Übertragen auf Programmcode bedeutet dies, dass ein noch fehlender Filter zur Auswertung von kompletten Programmanweisungen mehr Informationen liefert als der Vergleich von einzelnen Wörtern.

Für natürliche Sprachen ist ein bereits angesprochener Filter für die Reduktion auf Wortstämme nützlich. Dazu könnten sogar bei einer engen Verwandtschaft von Sprachen manche Regeln wiederverwendet werden.

Es ist geplant, das Plugin als *Rich-Client-Platform* aufzubauen, mehr Informationen dazu unter `http://www.eclipse.org/rcp/`. Im Zuge der notwendigen Anpassungen könnte man das Plugin in ein erweiterbares Framework für andere Module umwandeln.

Da man durch geschickte Darstellung beliebiger Daten auch andere Quellen als nur Texte im Plugin verarbeiten kann, stellt die Entwicklung von Eingabefiltern eine Möglichkeit dar, das Plugin zu erweitern. Beispielsweise könnte man Melodien, bzw. die entsprechenden Frequenzen, mit Hilfe von Dotplots vergleichen – die Plagiaterkennung wird so von Texten in den multimedialen Bereich übertragen.

Das Zusammensetzen der Client-Teilbilder könnte an externe Grafikprogramme ausgelagert werden. Professionelle Programme bieten dafür geeignete Schnittstellen an, ein anderer Weg wäre die Verwendung einer spezialisierten Bibliothek. Welcher Weg der bessere ist, müssen weitere Tests zeigen.

6.2 Persönliches Schlusswort

Das Dotplot-Plugin ist nur eine Anwendung der hier vorgestellten Techniken. Mit ihrer Hilfe macht die Arbeit mit dem Plugin aber erst Spaß und man kann „schnell mal“ ein paar Texte plotten. Neben den oben angeführten Ideen zur Verbesserung und Erweiterung des Plugins sind kleinere Erweiterungen geplant, die den Spaßfaktor noch erhöhen könnten. Zum Beispiel ist ein Tooltip nützlich, der zu dem unter dem Mauszeiger liegenden Pixel einen Ausschnitt der Quelldateien anzeigt. Das interaktive Anwenden von graphischen Effekten (Kontrast, Kantendetektion, ...) verleitet zum Spielen, kann aber auch bei einer Auswertung nützlich sein. Nach der geplanten Veröffentlichung des Plugins auf `http://www.sourceforge.net/` werden sicher weitere Anwendungsmöglichkeiten durch andere Entwickler entdeckt.

In den Gesprächen mit anderen Studenten, Betreuern und Bekannten wurden viele Fragen aufgeworfen, die mir gezeigt haben, dass noch viel Potential in dem Plugin steckt. Ich freue mich auf die weitere Arbeit an diesem Plugin und hoffe, dass es eine Verbreitung über die Grenzen der Fachhochschule Gießen-Friedberg hinaus findet.

Anhang A

Ergänzungen

Der Anhang enthält neben den Verzeichnissen und dem Index auch eine kurze Einführung zum Plugin und eine Übersicht zur Begleit-CD. Die Quelldateien des Dotplot-Plugins wurden hier aus Platzgründen nicht abgedruckt, sie befinden sich auf der CD.

Auszüge der Begleit-CD werden auf meiner privaten Homepage unter `http://www.gesellix.de/` veröffentlicht. Fragen und Anregungen zur Arbeit und zum Dotplot-Plugin können dort oder auf der Projekt-Homepage `http://www.dotplot.org/` gestellt werden.

A.1 Kurzbeschreibung des Plugins

Hier werden die Durchführung eines Plots, die im Plugin verfügbaren Optionen und technische Daten kurz erläutert. Eine detaillierte Beschreibung der einzelnen Optionen befindet sich im Handbuch [11]. Am Beispiel der E-Study-Sourcen werden nun die einzelnen Schritte zur Erstellung eines Dotplots aufgeführt. Es wird vorausgesetzt, dass die Perspektive „DotPlot“ bereits über *Window–Open Perspective–DotPlot* geöffnet wurde.

A.1.1 Erstellen eines Dotplots

Zunächst werden die Eingabedateien ausgewählt, dies geschieht durch einfaches Aktivieren der Checkbox neben dem gewünschten Ordner oder einer einzelnen Datei, siehe Abbildung A.1. Vor dem Plot können Einstellungen vorgenommen werden. Der entsprechende Dialog kann, wie in Abbildung A.2 gezeigt, geöffnet werden.

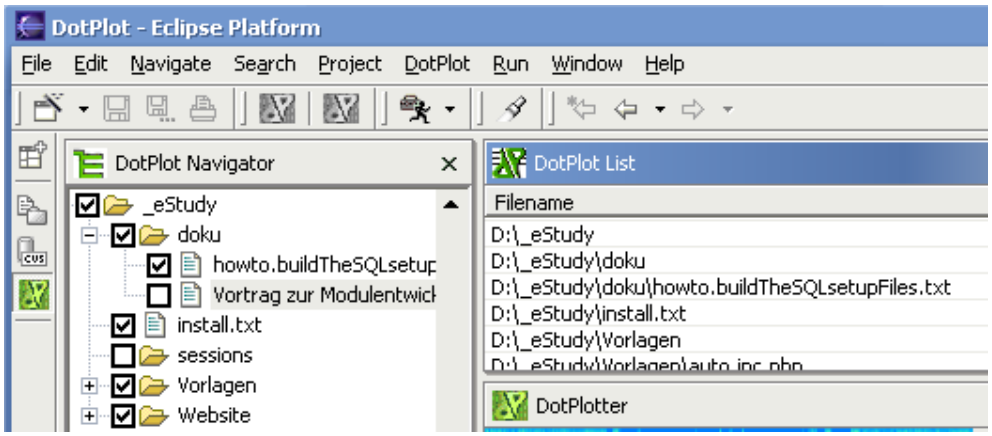

Abbildung A.1: Auswahl der Eingabedateien. Im „DotPlot Navigator“ wurden die E-Study-Sourcen ausgewählt, die einzelnen Dateien werden in der „DotPlot List“ aufgeführt.

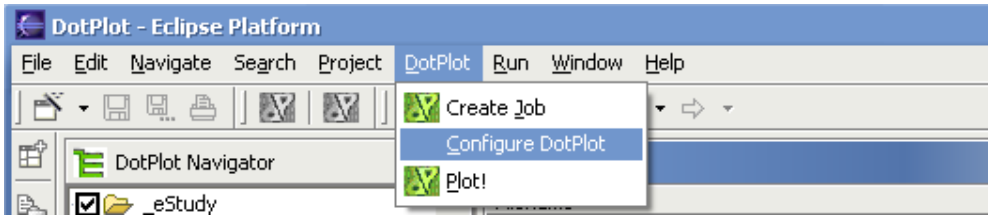

Abbildung A.2: Öffnen des Konfigurationsdialogs über *DotPlot–Configure DotPlot*.

Die Abbildungen A.4 bis A.6 auf Seite 62 zeigen eine mögliche Kombination von Einstellungen. Die dort vorgenommenen Einstellungen entsprechen den Parametern, die auch für die Performancetests in Abschnitt 5.2.2 verwendet wurden. Die Aktivierung des TextScanners und LineFilters wird in den meisten Fällen sinnvoll sein, während die Auswahl der LUT abhängig von den Zielen des Anwenders und den Eingabedateien ist. Beim Plot vieler Dateien und gleichzeitiger Skalierung auf eine kleine Bildgröße ist es empfehlenswert, die Ausgabe der Dateigrenzen zu deaktivieren.

Sind alle Einstellungen vorgenommen, kann über den Button „Plot“ des Konfigurationsdialogs oder über den Button in der Eclipse-Toolbar, siehe Abbildung A.3, der Plot gestartet werden. Das Dotplot wird dann im View „DotPlotter“, wie in Abbildung A.7 gezeigt, ausgegeben.

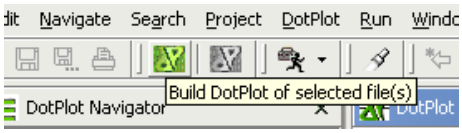

Abbildung A.3: Start des Plots über die Toolbar.

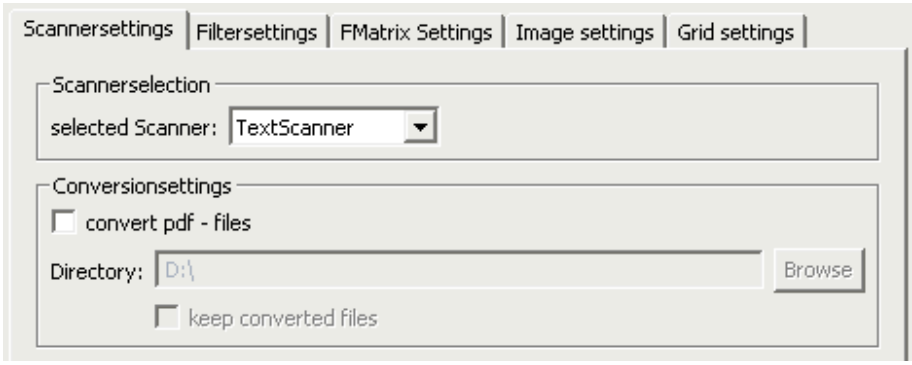

Abbildung A.4: Aktivierung des TextScanners.

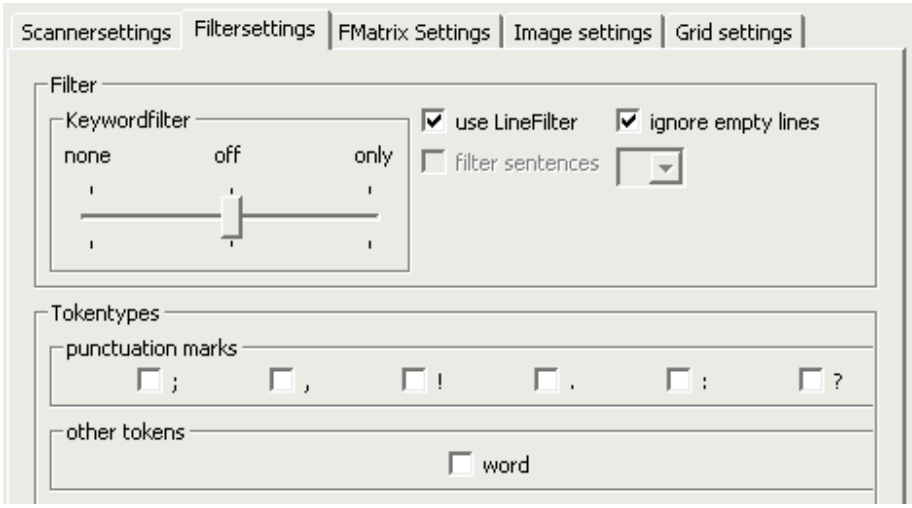

Abbildung A.5: Aktivieren des LineFilters mit der Option, Leerzeilen zu ignorieren.

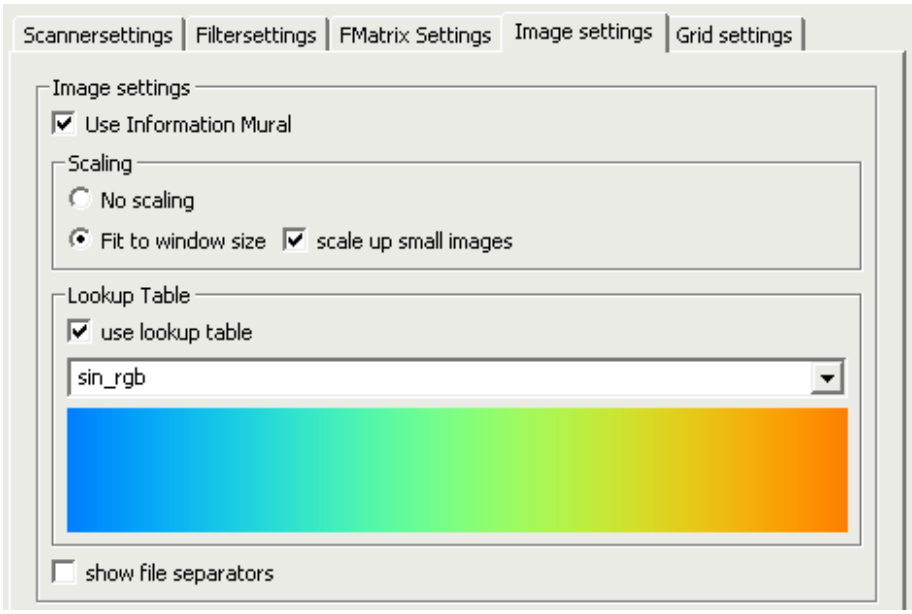

Abbildung A.6: Auswahl der LUT und Deaktivierung der Dateigrenzen.

Abbildung A.7: Ausgabe des Dotplots im View „DotPlotter“.

A.1.2 Technische Details zum Plugin

Die folgende Tabelle A.1 führt die wichtigsten Details zum Plugin auf. Die Liste der unterstützten Dateiformate ist noch fest im Code verankert, eine für den Anwender flexiblere Lösung ist jedoch geplant.

unterstützte Dateiformate	.txt, .xml, .htm, .html, .css, .js, .java, .cpp, .c, .cc, .h, .hpp, .php, .py, .ini, .cfg, .conf, .properties, .csv, .log, .doc, .rtf, .pdf, .tex
verfügbare Textscanner	Text, Java, C, C++, PHP
Anzahl Sourcedateien	106
Interfaces	16
Klassen	174
- abstrakt	8
- konkret	126
- Testklassen	40
Zeilen Code (inkl. Kommentaren)	30.674
Kompiliertes Plugin	10,60 MB
- plugin.jar	0,37 MB
- Integration in Eclipse	1,42 MB
- Bibliotheken	8,81 MB

Tabelle A.1: Technische Details des Dotplot-Plugins.

A.2 Begleit-CD

Einige Referenzen im Literaturverzeichnis verweisen auf Webseiten oder Downloads im Internet. Um die Flüchtigkeit des Internets zu meiden, wurden Kopien der Internetseiten und Dokumente auf der CD abgelegt. Ich weise darauf hin, dass diese Inhalte durch das Urheberrecht oder internationales Copyright geschützt sind und daher ausschließlich für private Zwecke bzw. nicht kommerziell genutzt werden dürfen.

Neben diesem Dokument, den kompletten Quellen des Dotplot-Plugins und hochauflösenden Dotplots befindet sich auch die Eclipse-Plattform für verschiedene Systeme auf der CD. So können die in dieser Arbeit genannten Beispiele nachvollzogen und mit eigenen Texten experimentiert werden.

Einige Dotplots auf der CD entsprechen den in der Arbeit erwähnten Plots, sie wurden mit einer ausführlichen Legende und der entsprechenden Lookup Table versehen. Die Muster können dadurch besser ausgewertet werden als es mit den verkleinerten Abbildungen im Text möglich ist.

Anhang B

Verzeichnisse

B.1 Abbildungsverzeichnis

B.2 Listings

B.3 Tabellenverzeichnis

B.4 Literaturverzeichnis

[1] Church, Kenneth Ward und Jonathan Helfman:
Dotplot: a Program for Exploring Self-Similarity in Millions of Lines of Text and Code.
Journal of Computational and Graphical Statistics, Vol. 2, Nr. 2, Seiten 153-174, Juni 1993.

[2] Clough, Paul:
Plagiarism in natural and programming languages: an overview of current tools and technologies.
Technischer Bericht, Department of Computer Science, University of Sheffield, 2000.

[3] Environmental Systems Research Institute, ESRI:
Geographic Information Systems „GIS“: Geography Matters.
`http://www.gis.com/whatisgis/whatisgis.pdf` (2005-05-14).

[4] Environmental Systems Research Institute, ESRI:
Geography Matters: An ESRI White Paper.
`http://www.gis.com/whatisgis/geographymatters.pdf` (2005-05-21).

[5] Farley, Jim:
Distributed Computing for Java Applications.
O'Reilly, 1998.

[6] Harold, Elliotte Rusty:
Java Network Programming.
O'Reilly, 3. Auflage, 1998.

[7] Helfman, Jonathan:
Dotplot Patterns: A Literal Look at Pattern Languages.
Theory and Practice of Object Systems (TAPOS), special issue on Patterns, Vol. 2, Nr. 1, Seiten 31-41, 1996.

[8] Helfman, Jonathan:
Dotplot Similarity Patterns.
`http://imagebeat.com/dotplot/` (2005-05-14).

[9] Helfman, Jonathan:
Similarity Patterns in Language.
Proceedings of the 1994 IEEE Symposium on Visual Languages, St. Louis, MO, Seiten 173-175, Oktober 1994.

[10] Jerding, Dean F. und John T. Stasko:
The Information Mural: A Technique for Displaying and Navigating Large Information Spaces.
Technischer Bericht, Georgia Institute of Technology, Atlanta, 1997.

[11] Matrix4.plot-Team:
Dotplot, das Eclipse-Plugin, 2004.
http://www.dotplot.org/downloads/Dotplot_User_Manual.pdf (2005-05-14).

[12] Matuschek, Daniel:
Vergleich verschiedener Interpolationsverfahren.
http://www.matuschek.net/photo/interpol/ (2005-05-14).

[13] National Oceanic & Atmospheric Administration, NOAA:
NOAA posts images online of northeast blackout.
http://www.magazine.noaa.gov/stories/s2015.htm (2005-05-14).

[14] Rodrigues, Lawrence H.:
Building Imaging Applications with Java Technology.
Addison-Wesley, 2001.

[15] Scott Oaks, Bernard Traversat und Li Gong:
JXTA in a Nutshell.
O'Reilly, 2002.

[16] Stoidner, Christoph:
Entwurf und Implementation eines Grids zur automatischen Programmgenerierung unter Verwendung moderner Grid Computing Technologien.
Diplomarbeit, Fachhochschule Gießen-Friedberg, 2004.

[17] Sun microsystems:
Programming in Java Advanced Imaging, 1999.
http://java.sun.com/products/java-media/jai/forDevelopers/jai1_0_1guide-unc/ (2005-05-14).

[18] Wikipedia:
Gestaltpsychologie.
http://de.wikipedia.org/wiki/Gestaltpsychologie (2005-05-21).

[19] Wikipedia:
Palindrom.
http://de.wikipedia.org/wiki/Palindrom (2005-05-19).

[20] Wikipedia:
Refactoring.
http://de.wikipedia.org/wiki/Refactoring (2005-05-21).

[21] Wikipedia:
Syntax.
http://de.wikipedia.org/wiki/Syntax (2005-05-21).

[22] Wikipedia:
Tag (Informatik).
http://de.wikipedia.org/wiki/Tag_%28Informatik%29 (2005-05-21).

[23] Wikipedia:
Wahrnehmung.
http://de.wikipedia.org/wiki/Wahrnehmung (2005-05-21).

B.5 Index

Eidesstattliche Erklärung

Hiermit erkläre ich, die vorliegende Diplomarbeit selbständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und habe wörtliche oder sinngemäße Zitate als solche gekennzeichnet.

Gießen, den 25. Mai 2005

_____

Tobias Gesellchen