Endbericht

PG573 - Multi-Objective NETwork design
Implementierung und experimenteller Vergleich von Algorithmen zur mehrkriteriellen Optimierung von Netzwerk-Design-Problemen

17. April 2014

Jakob Bossek, Michael Capelle, Hendrik Fichtenberger, Max Günther, Johannes Kowald, Marco Kuhnke, David Mezlaf, Christopher Morris, Andreas Pauly, Sven Selmke, Sebastian Witte

Betreuer:
Prof. Dr. Petra Mutzel
Dipl.-Inf. Fritz Bökler
Dipl.-Inf. Denis Kurz
Dipl.-Inf. Bernd Zey

Fakultät für Informatik
Algorithm Engineering (LS11)
Technische Universität Dortmund

Inhaltsverzeichnis

1

Einleitung

Der vorliegende Bericht ist der Endbericht der Projektgruppe 573, *Multi-Objective NETwork design: Implementierung und experimenteller Vergleich von Algorithmen zur mehrkriteriellen Optimierung von Netzwerk-Design-Problemen*, kurz MONET, die im Sommersemester 2013 und Wintersemester 2013/2014 veranstaltet wurde.

In Kapitel 2 wird zuerst eine Übersicht über die Thematik, die Minimalziele, die Organisatoren und Teilnehmer sowie die formalen Anforderungen gegeben. Anschließend werden in Kapitel 3 die zu Beginn der Projektgruppe (PG) aufbereiteten Seminarthemen und Vorträge zusammengefasst, bevor in Kapitel 4 auf die weitere Organisation der Durchführung der Projektgruppe, und dabei insbesondere auf *Scrum*, eingegangen wird. Kapitel 5 beschreibt die grundlegende Struktur der im Rahmen der Projektgruppe entwickelten Software. Die Hauptkomponenten der Plattform umfassen die *Worker*, den *Controlserver* und entsprechende *Kommunikation*. Weitere technische Aspekte werden in Kapitel 6 vorgestellt und umfassen *Java*, *MongoDB*, *OSGi*, *Apache Felix*, *Maven*, *GML*, *Git* sowie diverse Webtechnologien. Da es sich bei den in der Projektgruppe behandelten Problemen um Problemstellungen mit Graphen handelt, wird in Kapitel 7 das von den implementierten Algorithmen verwendete Graph-Framework vorgestellt. Neben einer Graph-Klasse selbst wurden ein Graph-Generator und mehrere Parser für Graphen implementiert. Die für das Projekt implementierten *Kürzeste-Wege-* und *Minimaler-Spannbaum-Algorithmen* für zwei oder mehr Kriterien werden in Kapitel 8 ausführlich beschrieben. Schließlich

folgt in Kapitel 9 das Handbuch mit Hinweisen zur Bedienung und Weiterentwicklung der Software. Zur Auswertung der Algorithmen durchgeführte Experimente werden in Kapitel 10 diskutiert. Der Bericht endet mit einem kurzen Fazit in Kapitel 11.

2

Übersicht

In diesem Kapitel wird ein kurzer Überblick über die Thematik mehrkriterieller kombinatorischer Optimierung auf Graphen gegeben. Grundlegende Begriffe werden hier genannt und es wird grob beschrieben, welche Ziele der Projektgruppe gegeben sind.

2.1. Einleitung

Thema unserer Projektgruppe sind kombinatorische Optimierungsprobleme auf gewichteten Graphen mit mindestens zwei Kriterien. Als Beispiele für mehrkriterielle Optimierungsprobleme dienen uns dabei das Finden kürzester Wege oder minimaler Spannbäume. Dieser Aspekt der Optimierung findet zum Beispiel Anwendung, wenn im Falle der kürzesten Wege nicht genau ein Kriterium, beispielsweise die benötigte Zeit oder die Kosten (in Form von erwarteten Treibstoffkosten oder Ähnlichem), sondern eben mehrere Kriterien optimiert werden sollen. Hierbei tut sich häufig das Problem auf, dass sich eine optimale Lösung bei mehreren Kriterien nicht oder nicht sinnvoll definieren lässt. Im Vergleich zweier Wege kann jeweils genau einer von ihnen zeit- oder kosteneffizienter sein. Dies ist ein Fehlen von *Dominanz* zwischen Lösungen. Der Begriff der Dominanz wird in Kürze erklärt. Ein sinnvoller Umgang mit diesem Verhalten der Dominanz, oder eben fehlender Dominanz, zwischen Lösungen ist stark

von den Anforderungen des Anwenders abhängig, lässt sich jedoch mit gewissen plausiblen Annahmen handhaben. So kann anstelle einer Lösung eine Lösungsmenge zurückgegeben werden, wobei jede dieser Lösungen innerhalb der Lösungsmenge von keiner anderen gültigen Lösung auf der gegebenen Probleminstanz dominiert wird. Diese Lösungsmenge ist dann die sogenannte *Pareto-Menge* oder eine Teilmenge dieser. Unter der Annahme, dass der Anwender von zwei Lösungen immer diejenige Lösung bevorzugt, die jedes gegebene Kriterium mindestens genauso gut erfüllt wie die andere, findet sich die bevorzugte Lösung des Anwenders immer in dieser Pareto-Menge.

2.2. Thematik

Im Folgenden werden grundlegende Definitionen und kapitelübergreifend verwendete Begriffe und Methoden eingeführt.

2.2.1. Definitionen

Wir haben nun eine Veranschaulichung des Problems fehlender Dominanz gesehen. Sie bildet für uns eine solide Basis für den Umgang mit der Lösungsmenge mehrkriterieller Optimierungsprobleme. Für zwei Vektoren $x := (x_1, \ldots, x_n)^T, y := (y_1, \ldots, y_n)^T \in \mathbb{R}^n$ *dominiert* der Vektor x den Vektor y genau dann, wenn für alle $i \in \{1, \ldots, n\}$ und für mindestens ein $j \in \{1, \ldots, n\}$ gilt: $x_i \leq y_i$ und $x_j \lneq y_j$.

Ein mehrkriterielles Optimierungsproblem unterscheidet sich nur in der Bewertungsfunktion sowie im Umgang mit dieser von den bekannteren einkriteriellen Optimierungsproblemen. Anders als im einkriteriellen Fall liefert die Kostenfunktion $c\colon E \to \mathbb{R}^n$ einen Vektor und für eine Lösung $L \subset E$ eines (kombinatorischen) Optimierungsproblems ist der Lösungsvektor $c(L) := (c_1(L), \ldots, c_n(L))$ für alle $i \in \{1, \ldots n\}$ gegeben durch die komponentenweise Addition $c_i(L) = \Sigma_{e \in L} c_i(e)$ der Kostenvektoren der einzelnen Elemente innerhalb der Lösung. Ziel ist nach wie vor das Finden optimaler Lösungen. Wir verwenden dabei die *Pareto-Optimalität*. Hierbei

gilt ein Vektor x in einer Menge A von Vektoren als *Pareto-optimal* oder *effizient* bezüglich A, wenn es keinen Vektor y in dieser Menge A gibt, der x dominiert. Die Menge aller effizienten Lösungen in einer Menge von möglichen Lösungen, wie sie zum Beispiel als Menge aller Spannbäume eines Graphen gegeben sein könnte, bildet dann die Pareto-Menge, die wir algorithmisch zu finden versuchen.

Es gibt verschiedene algorithmische Ansätze, um die Pareto-Optima einer mehrkriteriellen Zielfunktion zu finden. Ein Beispiel ist die Skalarisierung der Zielfunktion c durch eine gewichtete Summe, sodass eine einkriterielle Zielfunktion $\hat{c}$ entsteht (vgl. Abschnitt 3.2). Optimiert man diese mit einem für das entsprechende Optimierungsproblem bekannten einkriteriellen Lösungsalgorithmus, so erhält man eine Lösung L, deren Bewertungsvektor $c(L)$ auf der konvexen Hülle der Menge aller zulässigen Lösungen unter $c(\cdot)$ liegt. Die Menge aller Lösungen, welche auf der konvexen Hülle der Pareto-Front liegen, bezeichnen wir als *extrem effiziente* Lösungen. Die Menge aller extrem effizienten Lösungen wird häufig in der ersten Phase sogenannter Two-Phase-Algorithmen bestimmt, wo sie anschließend als Basis für die Suche nach weiteren Pareto-Optima innerhalb der zweiten Phase dient. Auf diese Weise kann man die gesamte Pareto-Menge oder eine Teilmenge finden. Ein weiterer Ansatz, mit dem wir uns beschäftigen, sind evolutionäre Algorithmen. Ein Beispiel für einen evolutionären Algorithmus wird in Abschnitt 8.3.1 erläutert.

Beispiel: Bikriterielles Problem

Abbildung 2.1 veranschaulicht die Kostenpaare einer Instanz eines bikriteriellen Problems. Die konvexe Hülle aller Kostenpaare (c_1, c_2) zulässiger Lösungen bezeichnen wir als gültigen Lösungswerteraum.

Die Lösungen s_1 und s_2 bezeichnen wir hierbei als *Extrempunkte*. Sie repräsentieren jeweils das lexikographische Minimum für (c_1, c_2) respektive (c_2, c_1).

Die effizienten Lösungen können zwei disjunkten Mengen zugeordnet werden: Die Menge der extrem effizienten Lösungen (z. B. p_1 oder p_4) und die verbleibenden Lösungen, die nicht extrem effizient sind (z. B. p_5). Diese befinden sich in dem von zwei benachbarten extrem effizienten Lösungen aufgespannten „Dreieck“. Alle

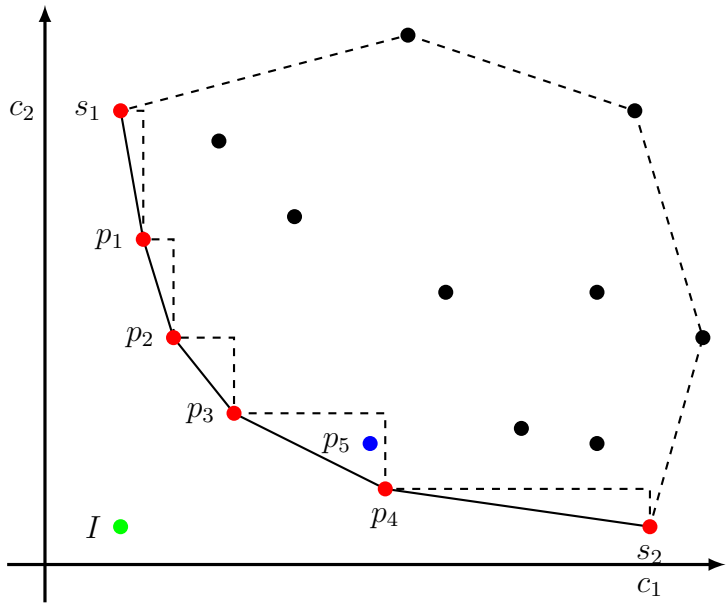

Abbildung 2.1.: Veranschaulichung der gültigen Lösungswerte

anderen Lösungen werden durch die Lösungen aus den oben genannten Mengen dominiert.

Ferner stellt der Punkt I in Abbildung 2.1 den *idealen Punkt* dar; dieser repräsentiert das (potentielle) gemeinsame Minimum für beide Kriterien. Liegt der ideale Punkt im Lösungsraum, so ist I die einzige effiziente Lösung und die Pareto-Menge ist einelementig. In der Regel tritt dieser Fall jedoch nicht ein, d. h. bei konfliktären Zielen ist der ideale Punkt nicht erreichbar.

2.2.2. Bikriterieller Two-Phase-Ansatz

Bei Verwendung des Two-Phase-Ansatzes wird die Suche nach Pareto-optimalen Lösungen zur Vereinfachung in zwei Schritte aufgeteilt. Im Folgenden beschränken wir uns auf den bikriteriellen Fall.

Erste Phase In einem ersten Schritt werden die Extrempunkte $s_1 = (x_1, y_1)$ und $s_2 = (x_2, y_2)$ berechnet. Hierzu wird das einkriterielle lexikographische Minimum für

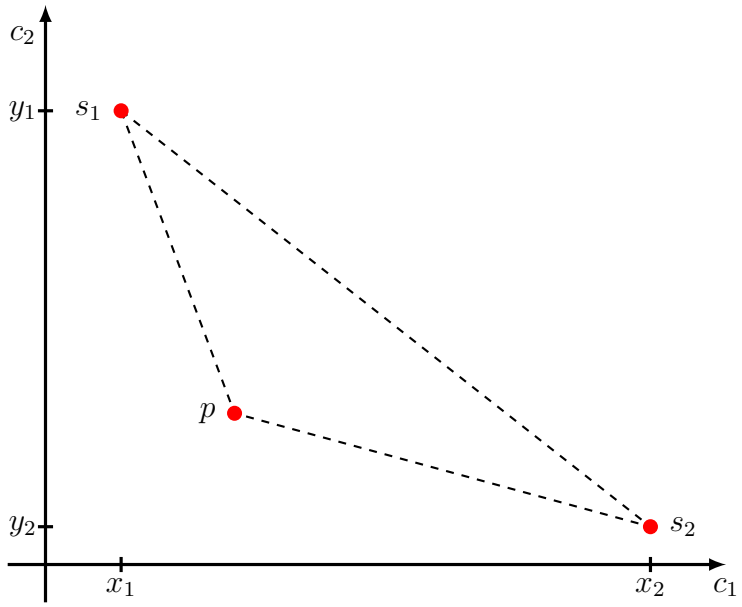

Abbildung 2.2.: Veranschaulichung der ersten Phase

Kriterium c_1 respektive c_2 berechnet. Falls s_1 und s_2 identisch sind, ist der ideale Punkt die einzige effiziente Lösung und dominiert alle anderen.

Andernfalls wird die bikrtierielle Zielfunktion $c = (c_1, c_2)$ durch Skalarisierung in eine monokriterielle Funktion $\hat{c}(\cdot)$ überführt und eine optimale Lösung bezüglich $\hat{c}$ mit dem einkriteriellen Algorithmus berechnet:

$$\hat{c}(e) = c_1(e)(y_1 - y_2) - c_2(e)(x_2 - x_1) \ . \tag{2.1}$$

Anschaulich gesprochen wird die effiziente Lösung p berechnet, die am weitesten von der Verbindungslinie zwischen s_1 und s_2 entfernt ist (siehe Abbildung 2.2). Falls die so berechnete Lösung nicht gleich s_1 oder s_2 ist, wird für s_1 und p respektive s_2 und p rekursiv fortgefahren. Auf diese Weise werden mindestens alle Ecklösungen in der ersten Phase gefunden.

Zweite Phase Im zweiten Schritt werden alle oder ein Teil der effizienten Lösungen gefunden, die nicht auf dem Rand der konvexen Hülle des gültigen Lösungsraums

liegen. Die Realisierung der zweiten Phase ist dabei abhängig vom verwendeten Algorithmus. Allgemein werden die in der ersten Phase gefundenen extrem effizienten Lösungen genutzt, um den Suchraum für die verbleibenden Lösungen einzuschränken.

2.3. Ziele und Anforderungen

Unser Ziel ist es, verschiedene Algorithmen für mehrkriterielle, kombinatorische Optimierungsprobleme zu untersuchen, sie zu analysieren und miteinander zu vergleichen. Dazu soll eine Plattform entwickelt werden, die das Ausführen, Analysieren und Vergleichen verschiedener Algorithmen vereinfacht. Sie soll möglichst modular gestaltet werden und so das Erweitern um weitere Optimierungsprobleme und Algorithmen ermöglichen. Zudem sollen für das Kürzeste-Wege- und das Minimale-Spannbaum-Problem jeweils mindestens drei Algorithmen entwickelt, implementiert und analysiert werden.

Wir setzen uns zudem das Ziel, eine Umgebung für konfigurierbare Algorithmen zu schaffen, um so zum Beispiel den Einfluss verschiedener Parameter und sogar Datenstrukturen auf das (Laufzeit-)Verhalten eines Algorithmus untersuchen zu können. Daten jeglicher Art sollen zudem durch Algorithmen in einer Datenbank abgelegt werden können, sodass sie zu einem späteren Zeitpunkt durch den Anwender der Plattform abrufbar sind. Auch eine Präsentation verschiedener Daten soll möglich sein. Dabei gelten Messwerte wie Laufzeit oder Speicherverbrauch sowie die Lösungsmengen selbst als Daten.

2.4. Organisatoren und Teilnehmer

Die Projektgruppe MONET fand im Sommersemester des Jahres 2013 sowie im darauf folgenden Wintersemester 2013/14 statt. Organisiert wurde die Projektgruppe vom Lehrstuhl 11 der Informatik der TU Dortmund. Als Veranstalter sind Prof. Dr. Petra Mutzel, Dipl.-Inf. Fritz Bökler, Dipl.-Inf. Denis Kurz und Dipl.-Inf. Bernd Zey zu

nennen. Die Teilnehmer sind Jakob Bossek, Michael Capelle, Hendrik Fichtenberger, Max Günther, Johannes Kowald, Marco Kuhnke, David Mezlaf, Christopher Morris, Andreas Pauly, Sven Selmke und Sebastian Witte.

3

Seminar

Zu Beginn der Projektgruppe wurde eine Seminarphase durchgeführt. Jeder Teilnehmer untersuchte ein spezifisches Thema des Algorithm Engineering insbesondere im Bereich der mehrkriteriellen kombinatorischen Optimierung. In diesem Kapitel werden die Seminarthemen und darin behandelte Probleme kurz vorgestellt.

3.1. Algorithm Engineering Cycle

Der *Algorithm Engineering Cycle* (AEC) ist ein Modell zur Entwicklung von Algorithmen, das 2005 von Sanders et al. [20] entwickelt wurde. Bei diesem Modell wird über einen Zyklus iteriert, anstatt sequentiell zu arbeiten, wie es z. B. beim Wasserfallmodell der Fall ist. Dabei zielt der AEC primär auf den Entwurf von Algorithmen ab. Der Ablauf des AEC ist in Abbildung 3.1 dargestellt, die einzelnen Schritte werden im Folgenden erläutert. Zu Anfang betrachtet man eine spezifische Anwendung und stellt dafür ein Modell auf. Das Modell sollte möglichst genau die zu lösende Anwendung beschreiben. Der Hauptzyklus des AEC startet mit einem initialen Algorithmus zur Lösung des aufgestellten Modells. Im nächsten Schritt wird der Algorithmus theoretisch analysiert, wodurch sich z. B. asymptotische Laufzeiten und/oder Approximationsgüten ergeben. Einer der wichtigsten Schritte ist die Implementierung. Dabei können Fehler und vergessene Anforderungen des Designs

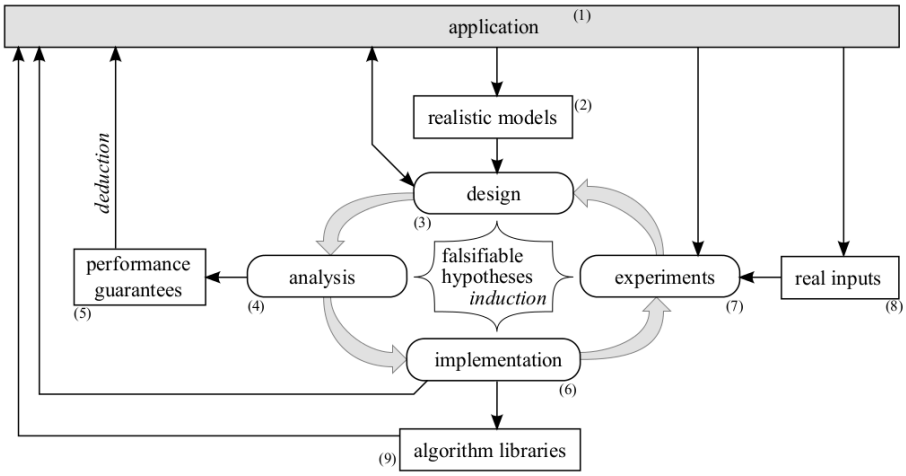

Abbildung 3.1.: Darstellung des Algorithm Engineering Cycle.

aufgedeckt werden und es wird ersichtlich, ob der Algorithmus implementierbar und somit praktisch nutzbar ist. Den Abschluss einer Iteration des Zyklus bilden die Experimente. Zum Experimentieren verwendet man praxisnahe Daten, die so auch in der Anwendung vorkommen. Nachdem ein Algorithmus gefunden wurde, welcher das Modell gut löst und ggf. gute Leistungskenndaten besitzt, sollte der Algorithmus in eine Bibliothek eingefügt und dokumentiert werden. Dadurch wird die Wiederverwendbarkeit erleichtert und der implementierte Algorithmus kann einfach veröffentlicht werden [5].

3.2. Skalarisierungstechniken für ILPs

Ehrgott [6] diskutiert in seiner Arbeit „*A discussion of scalarization techniques for multiple objective integer programming*“ diverse Ansätze zur Skalarisierung mehrkriterieller ganzzahliger linearer Programme zwecks Approximation der Menge effizienter Punkte.

Bevor auf einige der Verfahren eingegangen wird, soll an dieser Stelle eine kurze Einführung in die lineare Programmierung gegeben werden. Kombinatorische Optimierungsprobleme wie etwa das Problem des Handlungsreisenden (TSP) oder auch das Vertex-Cover-Problem (VC) sind als $\mathcal{NP}$-schwer eingestuft. Bisher sind

Mathematiker und Informatiker daran gescheitert – im Sinne der Komplexitätstheorie – effiziente, deterministische Algorithmen für diese und viele andere Probleme zu entwickeln. Die Vermutung, dass solche Algorithmen nicht existieren, ist in der Wissenschaft weit verbreitet. Nichtsdestotrotz treten jene Probleme in der Praxis auf und es gilt diese zumindest näherungsweise zu lösen. Eine Überführung der Problemstellung in ein lineares Programm (LP) ist eine dieser Methoden. Ein lineares Programm ist dabei eine *lineare Zielfunktion* in n Variablen, welche es unter mehreren *linearen* Nebenbedingungen zu optimieren gilt. Jede Lösung entspricht dabei einem Punkt im n-dimensionalen Hyperraum. Jede Nebenbedingung spaltet den Raum in zwei Teilräume; der Schnitt sämtlicher Teilräume ergibt die Menge der *zulässigen Lösungen*. Ein LP hat damit allgemein die Form

$$\begin{aligned} &\min c^T x \\ \text{u. d. N. } &Ax \leq b \end{aligned} \tag{LP}$$

mit $c \in \mathbb{R}^n, b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}$. Bei ganzzahligen linearen Programmen (ILP) wird zusätzlich die Forderung der Ganzzahligkeit an die Variablen gestellt.

Am Beispiel der Optimierungsvariante des Vertex-Cover-Problems soll das Vorgehen der Modellierung als ILP veranschaulicht werden. Gegeben sei ein Graph $G = (V, E)$ mit o. B. d. A. $V = \{1, \ldots, n\}$. $VC \subseteq V$ heißt Vertex-Cover, falls $\forall\, (u, v) \in E : u \in VC \vee v \in VC$. Gesucht ist ein VC minimaler Kardinalität. Durch Einführen von n binären Variablen $x_i \in \{0, 1\}$ lässt sich das Problem durch das folgende ILP

$$\begin{aligned} &\min \sum_{i=1}^{n} x_i \\ \text{u. d. N. } &x_i + x_j \geq 1 \quad \forall\, (i, j) \in E \\ &x_i \in \{0, 1\} \quad \forall\, i = 1, \ldots, n \end{aligned}$$

modellieren. Die Nebenbedingungen stellen dabei sicher, dass eine zulässige Lösung jeden Knoten überdeckt.

Lineare Programme lassen sich mit diversen Methoden in polynomieller Zeit lösen. Dennoch wird in der Praxis i. d. R. das Simplex-Verfahren benutzt, welches exponentielle Laufzeit aufweisen kann. Das ILP-Optimierungsproblem hingegen ist

weiterhin $\mathcal{NP}$-schwer. Jedoch lässt sich durch Relaxierung der Ganzzahligkeitsbedingung heuristisch eine untere Schranke mit akzeptablem Berechnungsaufwand finden.

Oftmals ist man mit Optimierungsproblemen mit mehreren Zielfunktionen konfrontiert. Ein mehrkriterielles ganzzahliges lineares Programm (MOIP) ist von der Form

$$\begin{aligned} &\min Cx \\ &\text{u. d. N. } Ax \leq b, x \geq 0, x \in \mathbb{Z}^n \end{aligned} \tag{MOIP}$$

mit Zielfunktionsmatrix $C \in \mathbb{N}^{p \times n}$. Im Seminarvortrag wurden verschiedene Verfahren vorgestellt, welche ein gegebenes mehrkriterielles Optimierungsproblem in Form eines MOIP durch *Skalarisierung* in ein monokriterielles Ersatzproblem überführen [6]. Durch wiederholtes Lösen dieses Ersatzproblems wird die Pareto-Front bzw. die Pareto-Menge approximiert. Die einfachste vorgestellte Methode ist die *Methode der gewichteten Summe*, bei der eine Konvexkombination der p Zielfunktionen gebildet wird. Die Ersatzfunktion hat die Form

$$\min_{x \in X} \sum_{i=1}^{p} \lambda_i c_i x \tag{SIP}$$

mit $\lambda_i \in [0, 1]$ und $\sum_{i=1}^{p} \lambda_i = 1, X = \{x \in \mathbb{Z}^n \mid Ax \leq b, x \geq 0\}$. Mit dieser Methode lassen sich nicht alle effizienten Punkte bestimmen, jedoch ist der Berechnungsaufwand im tolerablen Bereich. Der *ε-Constraint-Methode* liegt die Idee zugrunde nur eine Zielfunktion zu optimieren und die übrigen als Nebenbedingungen, sogenannte ε-Constraints, in das lineare Programm einfließen zu lassen:

$$\begin{aligned} &\min_{x \in X} c_j x \\ &\text{u. d. N. } c_k x \leq \varepsilon_k \text{ für } k \neq j. \end{aligned} \tag{SIP_ε}$$

Diese Methode eignet sich zur Auffindung sämtlicher effizienter Punkte durch Variation der ε_k-Werte. Das Prinzip ist exemplarisch in Abb. 3.2 dargestellt. Man erkennt, dass durch Änderung der Constraints unterschiedliche effiziente Punkte auf der Front gefunden werden. Jede Methode hat Vor- und Nachteile. So muss schließlich ein

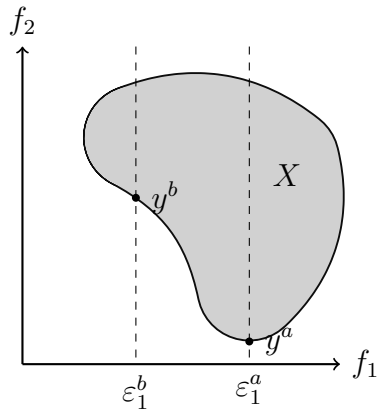

Abbildung 3.2.: ε-Constraints geometrisch für $p = 2$ dargestellt. Das Ersatzproblem ist dabei gegeben durch $\min_{x \in X} f_2(x)$ u. d. N. $f_1(x) \leq \varepsilon_1^k, k = a, b$.

Kompromiss zwischen Berechnungskomplexität und Güte der approximierten Pareto-Front/-Menge gefunden werden. Die ebenfalls vorgestellte *Methode der elastischen Constraints*, bei der die Verletzung von ε-Constraints durch Penalty-Funktionen bestraft wird, soll (bei geeigneter Parametrisierung) die Vorteile der übrigen Methoden vereinen. Eine solche Parametrisierung zu finden ist jedoch sehr schwierig.

Es gibt eine Vielzahl von Skalarisierungstechniken, mittels derer die Menge der effizienten Punkte mehrkriterieller ganzzahliger linearer Programme näherungsweise bestimmt werden kann. Es existiert keine Methode, die den alternativen Ansätzen in jeder Hinsicht überlegen ist. Daher gilt es je nach Anforderung die geeignete Methode auszuwählen und experimentell adäquate Parametrisierungen zu finden.

3.3. Mehrkriterielles Minimaler-Spannbaum-Problem

In diesem Abschnitt werden alle Arbeiten der Seminarphase zum mehrkriteriellen Minimaler-Spannbaum-Problem (MST-Problem) vorgestellt.

3.3.1. Repräsentative Pareto-Mengen

In [9] geben Hamacher und Ruhe unter anderem einen Algorithmus an, mit dem eine repräsentative Teilmenge der Pareto-optimalen Spannbäume eines bikriteriellen Graphen berechnet werden kann. Repräsentativ soll hierbei heißen, dass nicht zwangsläufig die gesamte Pareto-Front ermittelt wird, sondern versucht wird, eine Näherung anzugeben, sodass zwei im Bewertungsraum benachbarte Lösungen nicht allzuweit voneinander entfernt liegen.

Dabei verwenden die Autoren einen Two-Phase-Algorithmus, der zunächst mit Skalarisierung effiziente Lösungen berechnet und anschließend mit einer Nachbarschaftssuche zu große Lücken im Bewertungsraum zwischen den gefundenen Lösungen zu erschließen versucht. Beide Phasen werden nun genauer erklärt.

Erste Phase Die erste Phase verläuft grundsätzlich wie in Abschnitt 2.2.2 beschrieben. Das Ziel ist es jedoch, statt aller extrem effizienten Lösungen eine repräsentative Teilmenge zu finden, sodass der Abstand $\|c(T_i)-c(T_{i+1})\|_2$ zweier im Bewertungsraum benachbarter gefundener Lösungen T_i, T_{i+1} kleiner ist als ein gegebenes $\varepsilon > 0$, sofern solche Lösungen existieren. Die Korrektheit sowie eine effiziente Laufzeit werden dabei bewiesen.

Zweite Phase Die zweite Phase ist eine Nachbarschaftssuche. Dabei definieren die Autoren die Nachbarschaft $\Gamma(T)$ eines Spannbaums T in G als die Menge aller Spannbäume T', die aus T durch Ersetzen einer Kante aus T mit einer Kante hervorgeht, die nicht in T liegt. Mit Λ als Menge aller Spannbäume von G gilt

$$\Gamma(T) := \{T' \in \Lambda \mid T' = (T \cup \{e\}) \setminus \{f\}, e \in E \setminus T, f \in T\}.$$

In der zweiten Phase geht der Algorithmus nun alle Paare (T_i, T_{i+1}) im Bewertungsraum benachbarter und bisher gefundener Lösungen durch, für die der Abstand wie oben definiert größer als $\varepsilon > 0$ ist, und sucht in $\Gamma(T_i) \cup \Gamma(T_{i+1})$, also in der Vereinigung der Nachbarschaften der betrachteten Lösungen nach einer weiteren Lösung T^*, deren Wert $c(T^*)$ im Bewertungsraum im von $c(T_i)$ und $c(T_{i+1})$ aufge-

spannten Rechteck liegt. Dieses Vorgehen bleibt im Paper unbegründet, ist jedoch eine berechtigte Herangehensweise für die Erschließung zu großer Lücken im Bewertungsraum zwischen bereits gefundenen Lösungen, da man erwarten darf, dass unterschiedliche Spannbäume T, T' Werte $c(T), c(T')$ aufweisen, die sich nicht deutlich unterscheiden, wenn die Menge $T \cap T'$ der gemeinsamen Kanten nicht viel kleiner ist als T. Eine effiziente Laufzeit wird bewiesen; jedoch wird nicht bewiesen, dass der so vorgehende Algorithmus eine Annäherung der Pareto-Front nach beschriebenem Abstandskriterium liefert, sofern diese existiert.

3.3.2. Prüfer-EA

Prüfer-EA: Der evolutionäre Algorithmus (EA) von Zhou und Gen [28] ist eine randomisierte Suchheuristik. Der Algorithmus erzeugt initial eine sogenannte Population von potentiellen Lösungskandidaten, welche auch Individuen genannt werden. Diese Population besteht aus Kodierungen von Spannbäumen in Form von sogenannten Prüfer-Nummern. Die Kodierung eines Spannbaums wird als Genotyp bezeichnet, der dazu gehörige Spannbaum ist die phänotypische Ausprägung des Genotyps. Eine bijektive Genotyp-Phänotyp-Abbildung liefert zu jeder Prüfer-Nummer den entsprechenden Spannbaum. Der Algorithmus bewertet sämtliche Individuen der Population anhand einer sogenannten Fitnessfunktion, wählt Elternindividuen aus der Population aus und erzeugt daraus Nachkommen. Ein Selektionsmenachnismus wählt die besten Individuen unter der Elternpopulation und den Nachkommen aus. Diese Individuen bilden die Population der nächsten Generation. Dieses Prozedere wird fortgeführt, bis ein gewisses Abbruchkriterium erfüllt ist. Es kann nicht garantiert werden, dass der Algorithmus für eine gegebene Probleminstanz alle effizienten Punkte findet. Er wurde jedoch ausgewählt, da er sich vergleichsweise einfach implementieren lässt. Abbildung 3.3 zeigt den Ablauf eines allgemeinen EAs, der ebenfalls dem Ablauf des Prüfer-EAs entspricht.

Prüfer-Code: Nach dem *Satz von Cayley* existieren n^{n-2} verschiedene beschriftete Bäume mit n Knoten (und entsprechend n Knotenbeschriftungen) [28]. Für Spannbäume eines vollständigen Graphen mit n Knoten bietet sich daher eine Kodierung

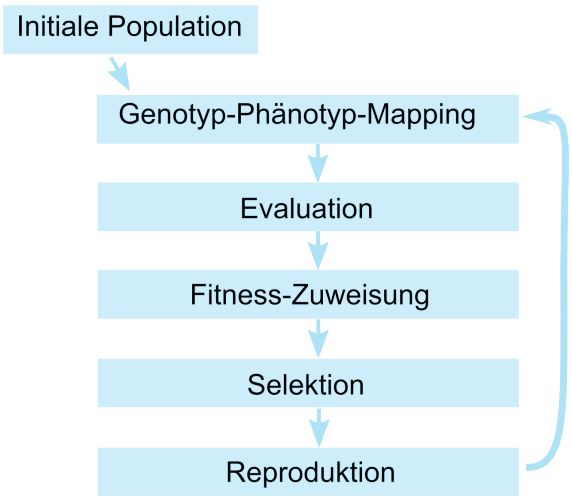

Abbildung 3.3.: Ablauf eines allgemeinen EAs.

der Länge $n-2$ an, da so eine Bijektion möglich ist.

Für die Kodierung eines Spannbaumes als Prüfer-Nummer werden folgende Schritte ausgeführt:

1. Entferne das Blatt mit der kleinsten Beschriftung aus dem Baum.

2. Füge die Beschriftung des Nachbarn an die Prüfer-Nummer an (ein Blatt hat immer genau einen Nachbarn; die Prüfer-Nummer ist zu Beginn leer).

3. Wiederhole dieses Verfahren, bis nur noch zwei Knoten übrig sind.

Umgekehrt berechnet sich die Dekodierung von einer Prüfer-Nummer zum Spannbaum wie folgt: Sei $\bar{P}$ die Menge der nicht in der Prüfer-Nummer P enthaltenen Knoten und sei j in jedem Schritt die kleinste Beschriftung aus $\bar{P}$ und k die erste (linke) Beschriftung in P.

1. Erzeuge die Kante $\{j,k\}$ und lösche j aus $\bar{P}$ und k aus P.

2. Falls k nicht im Rest von P enthalten ist, so füge k in $\bar{P}$ ein.

3. Wiederhole dieses Vorgehen, bis P leer ist und füge die Kante zwischen den letzten beiden Elementen aus $\bar{P}$ ein (es bleiben immer genau zwei Elemente in

$\bar{P}$ übrig).

Für die Betrachtung der eingesetzten Selektions- und Rekombinationsverfahren sei auf Abschnitt 8.3.1 verwiesen.

3.3.3. Two-Phase-Ansatz mit k-best-Algorithmus

Steiner und Radzik stellen in [24] einen Algorithmus zur Berechnung aller Pareto-optimalen Spannbäume unter bikriteriellen Zielfunktionen vor. Der Algorithmus folgt dem kanonischen Two-Phase-Ansatz: In der ersten Phase wird zunächst ein Teil[1] aller extrem effizienten Lösungen gefunden. Die verbleibenden effizienten Lösungen werden durch die zweite Phase konstruiert.

Die erste Phase verläuft wie in Abschnitt 2.2.2 beschrieben. Das entstandene monokriterielle Spannbaum-Problem wird mit einem beliebigen Algorithmus (z. B. Prim) gelöst.

In der zweiten Phase werden die verbleibenden effizienten Lösungen konstruiert (siehe Abbildung 3.4). Dazu wird für je zwei (bezüglich ihrer Lage auf der konvexen Hülle aller Pareto-optimalen Lösungen) benachbarte, extrem effiziente Lösungen die Zielfunktion erneut skalarisiert. Mit einem monokriteriellen k-best-Algorithmus (z. B. Gabow [7]) werden alle Lösungen aufgezählt, bis diese dominiert sein müssen. Dies ist spätestens dann der Fall, wenn die erste gefundene Lösung von beiden extrem effizienten Lösungen dominiert wird.

Alle gefundenen nicht-dominierten Lösungen aus der zweiten Phase ergeben zusammen mit den extrem effizienten Lösungen aus der ersten Phase die zurückgegebene Lösungsmenge. Der Algorithmus wurde in MONET implementiert; weitere Einzelheiten werden in Abschnitt 8.2.1 beschrieben.

[1] Es werden in jedem Fall alle extrem effizienten Lösungen gefunden, die auf Eckpunkten der konvexen Hülle der Lösungsmenge im Zielfunktionsraum liegen.

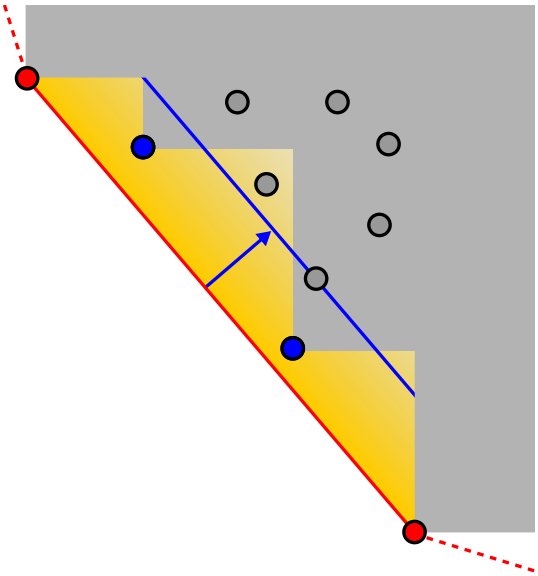

Abbildung 3.4.: 2. Phase: Berechnung der verbleibenden effizienten Lösungen zwischen je zwei extrem effizienten Lösungen durch einen monokriteriellen k-best-Algorithmus.

3.3.4. Two-Phase mit Branch-and-Bound

In [19] wird ein weiterer zweiphasiger Algorithmus für die Berechnung von Pareto-optimalen minimalen Spannbäumen vorgestellt.

Die erste Phase verläuft wie in Abschnitt 2.2.2 beschrieben. In der zweiten Phase werden nun die verbleibenden Lösungen mit Hilfe eines *Branch-and-Bound-Verfahrens* ermittelt. Dazu werden die Knoten mit 1 bis n durchnummeriert. Ferner bezeichne $A_k, k \in \{1, \ldots, n\}$ die Menge der zum Knoten k inzidenten Kanten.

Beginnend mit dem Knoten 1 wird nun ein Suchbaum T erzeugt, indem im i-ten ($i \in \{1, \ldots, (n-1)\}$) Branch-Schritt dem Knoten i der Graph mit Kantenmenge $T(E) \cup \{a\}$ für ein $a \in A_i$ und $a \notin T(E)$ als Kind zugewiesen wird, wenn dieser ein Baum ist und sein Kostenvektor addiert mit

$$\left(\sum_{j>i}^{n} \min\{c_1(e) \colon e \in A_j\}, \sum_{j>i}^{n} \min\{c_2(e) \colon e \in A_j\} \right) \tag{3.1}$$

nicht von bereits gefundenen Lösungen dominiert wird.

3.4. Mehrkriterielles Kürzeste-Wege-Problem

In diesem Abschnitt werden alle Arbeiten der Seminarphase zum mehrkriteriellen Kürzeste-Wege-Problem vorgestellt.

3.4.1. Warburtons Approximation

Der Algorithmus [26], der 1987 von A. Warburton vorgestellt wurde, ist eine Approximation der Pareto-optimalen Lösungsmenge eines mehrkriteriellen Kürzeste-Wege-Problems. Als Eingabe wird ein gerichteter azyklischer Graph erwartet, dessen Knoten nummeriert sind. Eine Kante darf nur zwischen Knoten existieren, bei denen der Zielknoten eine höhere Nummerierung hat als der Quellknoten. Außerdem müssen die Kantengewichte nicht-negativ sein. Der Knoten mit der niedrigsten Nummer ist die Quelle und der Knoten mit der höchsten das Ziel.

Als Grundlage für die Approximation stellt Warburton einen simplen, exakten, rekursiven Algorithmus vor, der mehrkriterielle Kürzeste-Wege-Probleme für azyklische Graphen lösen kann. Für das Verständnis seien folgende Definitionen gegeben: P_t ist die Menge aller $(1,t)$-Pfade für $1 \leq t \leq n$. Die aufaddierten Gewichtsvektoren aller Kanten pro Pfad aus P_t sind $C_t = c(P_t) = \bigcup_{p \in P_t}\{c(p)\}$. Da Pareto-optimale bzw. effiziente Pfade gesucht werden, ist C_t^* die Menge der effizienten Vektoren aus C_t. Der Algorithmus sucht also die Elemente der Menge C_n^*, was im Folgenden mit C^* abgekürzt wird. Der Algorithmus namens *ACYCLIC* definiert folgende Mengen:

1. $C_1^* = \{0\}$
2. $C_j^* = MIN_{i<j}\{C_i^* + c_{ij}\}$ für $2 \leq j \leq n$

wobei C_i^* alle Vektoren beinhaltet, die im Schritt i als effizient erkannt wurden und dementsprechend MIN die Menge auf die effiziente Teilmenge reduziert.

Damit die Approximation funktioniert, muss *ACYCLIC* um die Überprüfung von oberen Schranken für die aufaddierten Gewichte erweitert werden. Die Werte der

oberen Schranken sind im Vektor M_k enthalten. Die erweiterte Version *RESTRICT* ist so definiert:

1. $D_1^* = \{0\}$
2. $D_j^* = MIN_{i<j}\{Z \cap (D_i^* + d_{ij})\}$ für $2 \leq j \leq n$

wobei Z die Menge aller Vektoren $(x_1, \ldots, x_r)$ ist, die $0 \leq x_k \leq M_k$ für $1 \leq k \leq r-1$ erfüllen.

Als weitere Voraussetzung führt der Autor den Begriff der approximativen Effizienz ein. Ein Vektor x ε-dominiert einen Vektor y, wenn $x_k \leq (1+\varepsilon)y_k$ für $x, y \in \mathbb{R}_+^r, 1 \leq k \leq r$. Für $\varepsilon = 0$ entspricht dies der Definition des normalen Dominanzbegriffs. Ansonsten ist $\varepsilon > 0$ die Güte der Approximation.

Daraus lässt sich eine Definition für ε-effiziente Mengen ableiten: Sei X eine Menge von positiven r-dimensionalen Vektoren und $Y \subset X$. Dann ist $Y(\varepsilon) \subset X$ ε-effizient für Y, wenn jeder Vektor in Y von mindestens einem Vektor aus $Y(\varepsilon)$ ε-dominiert wird und kein Vektor in $Y(\varepsilon)$ einen anderen Vektor aus $Y(\varepsilon)$ dominiert, also $Y(\varepsilon) = (Y(\varepsilon))^*$.

Die eigentliche Approximation hat folgenden Ablauf:

1. **Einen Skalierungsvektor T_k bestimmen:**
   Zuerst muss B bestimmt werden, das mindestens so groß wie die größte Komponente der aufaddierten Kantengewichtsvektoren von allen Pfaden ist. Mittels B lässt sich die Menge I bestimmen, die das $(r-1)$-fache kartesische Produkt der Menge $\{0, \ldots, \lfloor \log B \rfloor\}$ ist. Für jeden Vektor $(i_1, \ldots, i_{r-1})$ aus I lässt sich nun der Vektor L durch $L = (\beta_{i_1}, \ldots, \beta_{i_{r-1}})$ belegen, wobei $\beta_j = 2^j$ ist. Für jede Möglichkeit, wie L belegt werden kann, wird die Approximation durchgeführt. Der Skalierungsvektor errechnet sich daraus wie folgt: $T_k = \varepsilon L_k/(n-1)$ für alle $1 \leq k \leq r-1$.

2. **Kantengewichte skalieren:**
   Für jede Kante e des Graphen werden nun die Gewichte skaliert.
   $d^r(e) = c^r(e)$
   $d^k(e) = \lfloor c^k(e)/T_k \rfloor$

$r^k(e) = (c^k(e)/T_k) - d^k(e)$
Der neue Graph mit den skalierten Kantengewichten sei G'.

3. **Berechnung der Effizienzmenge von G':**
   Da nun *RESTRICT* angewendet wird, muss der Vektor, der die oberen Schranken beinhaltet, gesetzt werden: $M_k = (n-1)(\theta_k/\varepsilon)$ wobei die Werte des Vektors θ frei wählbar sind. In der Praxis liefert $\theta = (2, \ldots, 2)$ gute Ergebnisse. Nun kann *RESTRICT* ausgeführt werden, wobei parallel in einer zweiten Menge von Vektoren R die Reste $r(e)$ zu den zur Lösung gehörenden Kantengewichte $d(e)$ aufaddiert werden. Das Ergebnis ist zum einen die Effizienzmenge D^* des skalierten Graphen und zum anderen die zugehörige Menge der Reste R.

4. **Berechnung der ε-Effizienzmenge für G:**
   Die ε-Effizienzmenge C'^* lässt sich aus D^* wie folgt berechnen: Für alle d^* aus D^* und dem zu d^* korrespondierenden r aus R sei $c'^r = d^{*r}$ und $c'^k = (d^{*k}+r^k)\cdot T_k$ für $1 \leq k \leq r-1$. Nachdem die Approximation für alle Skalierungen durchgeführt wurde, gilt $C(\varepsilon) = \bigcup C'^*$ für alle Vektoren L, die durch I indiziert wurden. Nun muss nur noch $(C(\varepsilon))^*$ bestimmen werden.

Die Laufzeit des exakten Algorithmus ist von der Größe der gesuchten Effizienzmenge abhängig. Da diese exponentiell groß werden kann, ist auch die Laufzeit im schlechtesten Fall exponentiell. Die Approximation zielt darauf ab, jeweils weitaus kleinere Effizienzmengen pro Skalierung zu errechnen und diese final zu einer angenäherten Lösungsmenge zu kombinieren. Bei großen Eingabeinstanzen kann damit die Laufzeit von exponentiell auf polynomiell in Abhängigkeit zur Güte $1/\varepsilon$ und zur Eingabemenge n reduziert werden.

3.4.2. Label-Correcting

Skriver und Andersen [22] stellen einen Algorithmus vor, der die effizienten Pfade bezüglich zweier Kriterien, also alle Pareto-optimalen Pfade von einem Start- zu einem Zielknoten berechnet. Er ist in weiten Teilen identisch mit einem Algorithmus von Brumbaugh-Smith und Shier [3], welcher hier zunächst beschrieben werden soll. Der Algorithmus nutzt die Tatsache, dass jeder Teilpfad eines effizienten Pfades

ebenfalls ein effizienter Pfad ist. Ausgehend vom Startknoten, werden für jeden Knoten die bisher berechneten kürzesten Pfade in Labels gespeichert. Für jede ausgehende Kante eines Knotens werden die Labels dieses Knotens um die Kosten der Kante erweitert und die so entstandene Labelmenge mit der des Zielknotens der Kante zusammengefügt. Hierbei müssen dominierte Labels gefunden und entfernt werden.

Sei *Labeled* eine Queue aller Knoten mit unbearbeiteten Labels, $Labels(i)$ die Menge der Labels von Knoten i, $Len(i, j)$ das Tupel $(c_1((i, j)), c_2((i, j)))$ der beiden Kostenfunktionen der Kante (i, j), $out(i)$ die Menge der von Knoten i ausgehenden Kanten und s und t ausgewiesene Start- beziehungsweise Endknoten.

Dann stellt sich der Algorithmus wie in Listing 3.1 dar.

Algorithmus 3.1 Label-Correcting Algorithmus.

```
1:  function labelCorrecting(Graph g, Node s, Node t)
2:      Labeled.enqueue(s)
3:      Labels(s) = {(0, 0)}
4:      while Labeled ≠ ∅ do
5:          i = Labeled.dequeue
6:          for all (i, j) ∈ out(i) do
7:              Labels(j) = Merge(Labels(j), Labels(i) + Len(i, j))
8:              if Labels(j) changed ∧ j ∉ Labeled then
9:                  Labeled.enqueue(j)
10:             end if
11:         end for
12:     end while
13:     return Labels(t)
14: end function
```

Hierbei ist Merge eine geeignete Methode, um zwei Labelmengen zusammenzufügen und dominierte Labels zu entfernen. Eine Möglichkeit, dies zu bewerkstelligen, besteht darin, alle Labels paarweise zu vergleichen. Eine effizientere Variante nutzt aus, dass eine Labelmenge ohne dominierte Labels, die aufsteigend bezüglich des ersten Kriteriums sortiert ist, bezüglich des zweiten Kriteriums absteigend sortiert ist. Die beiden Labelmengen lassen sich in Linearzeit zu einer sortierten zusammenfügen und Fehler in der Sortierung des zweiten Kriteriums, die in Linearzeit gefunden

werden können, zeigen dominierte Label an.

Skriver und Andersen nutzen die beschriebene Eigenschaft von sortierten Labelmengen, um frühzeitig zu erkennen, wenn ein Label der einen Labelmenge alle Labels der anderen dominiert. Eine anschließende experimentelle Evaluation zeigt, dass dieser Zusatz die Laufzeit des Algorithmus signifikant verkürzen kann. Der Laufzeitgewinn beträgt bei großen, dichten Graphen (500 Knoten, 7 bis 15 Kanten pro Knoten) nur ca 5%, wächst aber mit abnehmender Größe und Dichte und beträgt bei 200 Knoten mit je 1 bis 3 Kanten ca. 55%.

3.4.3. Near-Shortest-Path

Andrea Raith und Matthias Ehrgott [18] implementierten den von Carlyle und Wood [4] entworfenen zweikriteriellen Near-Shortest-Path-Algorithmus (NSP) und verglichen verschiedene Algorithmen auf unterschiedlichen Probleminstanzen. Hier wird kurz dargestellt, wie der NSP-Algorithmus funktioniert.

Der Near-Shortest-Path-Algorithmus geht alle möglichen Pfade in einem Graphen durch, beendet jedoch die Abarbeitung eines Pfades, sobald dieser länger wird als ein gewisses δ. Gelangt man an den Zielknoten und ist die Pfadlänge kürzer als δ, so wird der Pfad in der Lösungsmenge gespeichert. Dabei ist es möglich, dass dominierte Pfade aus der Lösungsmenge herausfallen. Wichtig für die Güte des NSP-Algorithmus ist ein gutes δ. Gewählt wird hier

$$\delta = \lambda_1(c_1(x_{lex(2,1)}) - 1)) + \lambda_2(c_2(x_{lex(1,2)}) - 1).$$

Dabei sind c_1 und c_2 die Gewichtfunktionen, während $x_{lex(1,2)}$ und $x_{lex(2,1)}$ die beiden lexikographisch optimalen Lösungen sind. Gewichtet wird dies durch die Faktoren λ_1 und λ_2, gewählt als $\lambda_1 = c_1(x_{lex(1,2)}) - c_2(x_{lex(2,1)})$ und $\lambda_2 = c_1(x_{lex(2,1)}) - c_1(x_{lex(1,2)})$. Zusammen beschreiben diese die Schranken, welche in Abb. 3.5. dargestellt sind. Außerhalb dieser Schranken sind die Wege zu lang und werden nicht weiter vom NSP-Algorithmus verfolgt.

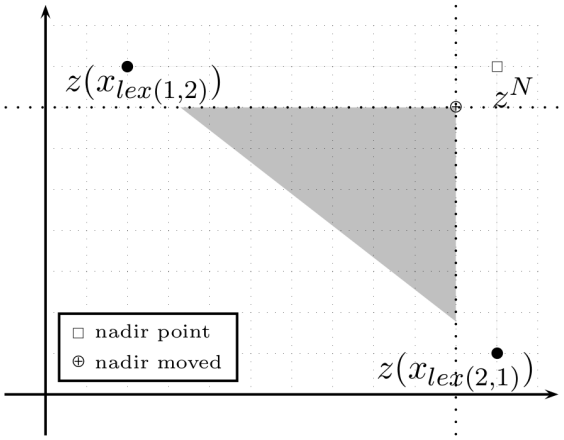

Abbildung 3.5.: Der Nadir-Punkt wird verwendet, um den Suchraum zu beschränken.

Als zusätzliche Verbesserung wurden noch lokale Nadir-Punkte verwendet. Diese setzen am Schwachpunkt des NSP-Algorithmus an, dem δ. Dabei wird ein lokaler Nadir-Punkt von zwei Punkten bestimmt: Sei $z^k = (z_1^k, z_2^k)$ und $z^l = (z_1^l, z_2^l)$ mit $z_1^l < z_1^k$ und $z_2^k > z_2^l$. Der lokale Nadir-Punkt ist dann $z^{LN} = (z_1^l, z_2^k)$. Damit lässt sich jetzt ein γ mit $\gamma = \max\{\lambda_1 c_1(x_c^1) + \lambda_2 c_2(x_{lex(1,2)}), \lambda_1 c_1(x_{lex(2,1)}) + \lambda_2 c_2(x_c^p)\}$ bestimmen. Dieses γ wird nun in der verbesserten δ-Funktion

$$\Delta = \max\{\gamma, \max\{\lambda_1 c_1(x_c^{j+1}) + \lambda_2 c_2(x_c^j j); j = 1, \ldots, p-1\}\}$$

verwendet. Dieses lokale Δ wird nun abhängig von der aktuellen Position im Graphen verwendet und schränkt den Suchbereich weiter ein.

3.4.4. Der Parametric approach

Der *parametric approach* von Mote et. al. [14] ist in der Lage bikriterielle kürzeste Wege zu berechnen. Dabei lässt sich der Algorithmus in zwei Phasen unterteilen, die im Folgenden kurz beschrieben werden.

Dem Algorithmus liegt ein lineares Programm zu Grunde, das mittels Parametrisierung fast wie ein monokriterielles lineares Programm berechnet werden kann. Diese Parametrisierung gibt diesem Algorithmus auch den Namen. Sei $G = (V, E)$ ein gerichteter Graph mit einer endlichen Knotenmenge V und einer Menge E gerichteter

Kanten. Bezeichne $s \in V$ die Quelle für das bikriterielle Kürzeste-Wege-Problem. Wenn nun x_{ij} die Indikatorvariable dafür ist, dass die Kante $(i,j) \in E$ in der Lösung des Kürzeste-Wege-Problems ist, dann lässt sich das Programm wie folgt definieren:

$$\begin{aligned} \min & \begin{cases} Z_1 := \sum_{(i,j)\in E} c_1((i,j))x_{ij} \\ Z_2 := \sum_{(i,j)\in E} c_2((i,j))x_{ji} \end{cases} \\ \text{u. d. N.} \sum_{(i,j)\in E} x_{ij} - \sum_{(j,i)\in E} x_{ji} &= \begin{cases} |N|-1 & \text{wenn } i = s, \\ -1 & \text{sonst} \end{cases} \\ x_{ij} &\in \{0,1\} \quad \forall\, (i,j) \in E \end{aligned}$$

Um dieses ganzzahlige lineare Programm zu lösen, wird die Zielfunktion auf ein Kriterium reduziert. Dafür werden die Zielfunktionen durch die folgende gewichtete Zielfunktion ersetzt:

$$Z_\lambda := \lambda Z_1 + (1-\lambda)Z_2 = \sum_{(i,j)\in E} x_{ij} \left(\lambda c_1((i,j)) + (1-\lambda)c_2((i,j))\right).$$

Die Gewichtung der Zielfunktionen mit dem Parameter λ ist hier aber nicht willkürlich, sondern dient als zusätzlicher Parameter für einen leicht modifizierten Simplexalgorithmus. Da es sich um ein bikriterielles Problem handelt, ist die Lösungsmenge eine Pareto-Front. Der modifizierte Simplexalgorithmus berechnet einen neuen Eckpunkt der konvexen Hülle der Pareto-Front in einer Ausführung des Simplexalgorithmus. Im Algorithmus, der in diesem Kapitel vorgestellt wird, entspricht das Finden eines neuen λ, beim Lösen des relaxierten Programms, dem Finden einer Kante im Kürzeste-Wege-Baum. Der bisherige Baum ist dann für die gewählte Parametrisierung Pareto-optimal und für die folgenden Parameter muss nur noch der Teilgraph betrachtet werden, der aus den verbleibenden Knoten besteht. Die dabei gefundenen Weglängen werden in Listen gespeichert. Dabei gibt es eine Liste, in der alle gefundenen Lösungen gespeichert werden und eine, in der nur Lösungen gespeichert werden, die nicht von den Eckpunkten der Pareto-Front der relaxierten Lösungen dominiert werden.

In der zweiten Phase wird dann jede nicht-dominierte Lösung betrachtet und es wird geprüft, ob es von dessen Vorgängerknoten einen Pfad zu einem anderen Knoten

gibt, der zu einer Pareto-optimalen ganzzahligen Lösung führt. Wenn dabei ein neuer nicht-dominierter Pfad erzeugt wird, wird dieser auch zu der Liste hinzugefügt. Wenn alle Lösungen abgearbeitet sind, werden alle nicht-dominierten Lösungen aus dieser Liste zurückgegeben, die in einem gewünschten Zielknoten enden.

4

Organisation

Dieser Abschnitt behandelt die Arbeitsteilung innerhalb der Projektgruppe in kleineren Subgruppen, die Kommunikation untereinander, sowie das umgesetzte Modell des Projektmanagements.

4.1. Aufteilung

Unsere Aufgabenstellung sah vor, Algorithmen für das Kürzeste-Wege- sowie das Minimale-Spannbaum-Problem zu entwickeln und diese dann auf einer eigens implementierten Experimentierplattform zu testen. So lag es zunächst nahe, drei Subgruppen zu bilden, die sich mit der Planung der Plattform bzw. mit je einem der beiden algorithmischen Probleme beschäftigten. Von den ursprünglich zwölf Teilnehmern der Veranstaltung bildeten anfangs fünf die Plattformgruppe, während sich die restlichen sieben auf das Kürzeste-Wege- und das Minimale-Spannbaum-Problem verteilten.

Nachdem wir die entsprechende Literatur innerhalb der Subgruppen gesichtet hatten und klar geworden war, dass die beiden Algorithmengruppen viele Gemeinsamkeiten aufwiesen und die Plattformgruppe darüber hinaus weitere Mitglieder benötigte, entschieden wir uns, die MST- und die Kürzeste-Wege-Gruppe zu einer großen Algorithmengruppe zusammenzulegen und aus dieser dann zwei Teilnehmer zugunsten der

Plattformgruppe abzuziehen. Im Laufe des ersten Semesters brach eine Kommilitonin die Teilnahme an der Veranstaltung ab, sodass sich die Anzahl der Mitglieder auf fünf in der Algorithmen- bzw. sechs in der Plattformgruppe reduzierte:

Algorithmen	**Plattform**
Jakob Bossek	Max Günther
Michael Capelle	Johannes Kowald
Hendrik Fichtenberger	Marco Kuhnke
Christopher Morris	David Mezlaf
Sven Selmke	Andreas Pauly
	Sebastian Witte

Diese Einteilung sollte ausdrücklich nicht zwangsweise bis zum Ende der Projektgruppe beibehalten werden müssen, sondern in sinnvollen Intervallen nach Bedarf oder je nach Wunsch der Teilnehmer angepasst werden können. So tauschten etwa Jakob Bossek und Andreas Pauly zu Beginn des zweiten Semesters die Gruppen.

4.2. Kommunikation

Unabhängig von der Gruppeneinteilung ist der ständige Austausch zwischen allen Entwicklern sowohl in der Planungs- als auch in der Implementierungsphase sehr wichtig. Zu diesem Zweck haben wir mit *Redmine* (siehe Sektion 4.4) ein webbasiertes Projektmanagementtool genutzt, das uns ein Repository, ein Wiki, ein Forum, sowie ein Ticketsystem zur Verfügung gestellt hat.

Das Wiki wurde unter anderem zu Beginn dazu genutzt, Tutorials zentral zu sammeln und das Fortschreiten des Projektes zu dokumentieren. Auch sind dort alle Protokolle der *Daily Scrums* (siehe Kapitel 4.3.1) und der weiteren Gruppentreffen hinterlegt. Die Tickets wurden darüber hinaus zur Verwaltung der einzelnen Teilaufgaben jedes Teilnehmers verwendet, etwa um einen Überblick über das momentane Aufgabenfeld einzelner Personen oder bisher nicht zugewiesene Arbeiten zu bekommen.

4.3. Scrum

Wir haben uns in der Anfangsphase mehrheitlich für ein agiles Projektmanagement mit *Scrum* entschieden. Bei diesem Modell werden Implementierungszyklen (sogenannte *Sprints*) durchlaufen, an deren Ende jeweils eine neue Version (Inkrement) des Produkts steht. Dadurch sollte die Unterteilung in ein reines Planungs- und ein Implementierungssemester vermieden werden, zumal sich die Vision des Endprodukts erst noch entwickeln musste. Durch frühe Umsetzungen ließ sich außerdem auch die zeitliche Realisierbarkeit besser abschätzen und etwaige Probleme frühzeitig erkennen. Da insgesamt wenig Erfahrung in der Entwicklung größerer Projekte vorhanden war, entschieden wir uns so für dieses agile Modell, dessen ausführlichere Beschreibung in den Veröffentlichungen von Ken Schwaber [21, 25] zu finden ist.

4.3.1. Regelmäßige Treffen

Elementar im Scrum-Ablauf sind die regelmäßigen Treffen: Vor jedem Sprint steht ein *Sprint Planning* an, während der eigentlichen Implementierungsphase steht das Element des Daily Scrum im Mittelpunkt und nach Beendigung dieser Phase bilden *Review* und *Retrospective* den Abschluss des Zyklus.

Vor der Planung des ersten Sprints ist zunächst die Erstellung einer priorisierten Liste von *User Stories* erforderlich, d. h. von in natürlicher Sprache verfassten Anwendungsfällen. Die Gesamtheit der User-Stories nennt sich *Product Backlog* und stellt die Basis für alle folgenden Sprints dar.

Sprint Planning

Aus dem relativ grob gehaltenen Product Backlog werden im Planungstreffen diejenigen Elemente ausgewählt, die im kommenden Sprint implementiert werden sollen. Diese werden detaillierter beschrieben und in eine neue Liste übernommen, dem sogenannten *Sprint Backlog*. Dort werden auch etwaige Hilfsarbeiten, wie die Ent-

wicklung ggf. nötig gewordener Tools beschrieben. Auf Grundlage des Sprint Backlog kann nun der eigentliche Sprint begonnen werden.

Daily Scrum

In jedem Zyklus hat die Kommunikation im Entwicklungsteam einen hohen Stellenwert. Um diese hoch zu halten, wird täglich ein Daily Scrum abgehalten. Bei diesem in einem möglichst kleinen Zeitrahmen stattfindenden Treffen beantwortet jedes Mitglied des Teams drei Fragen:

1. Was wurde seit dem letzten Daily Scrum erreicht?
2. Woran wird bis zum nächsten Treffen gearbeitet?
3. Gibt es Probleme oder Hindernisse?

Dabei sollen diese Informationen jedem Teilnehmer einen Überblick über die momentane Situation und den aktuellen Entwicklungsstand geben. Während dieser Runde wird noch nicht über einzelne Themen oder Probleme diskutiert; dies geschieht anschließend in entsprechenden Kleingruppen.

Sprint Review

Dieses Treffen bildet den Abschluss eines Sprints. Hier wird das „Was?“ zusammengefasst, d. h. inwiefern kann das nun zwar noch nicht vollständige, aber in jedem Fall funktionsfähige Produkt mehr leisten als nach dem letzten Sprint und was wurde aus dem Sprint Backlog (ggf. nicht) umgesetzt? Typischerweise sind dabei die Auftraggeber des Projekts anwesend, denen so das Zwischenergebnis präsentiert wird und die dazu entsprechende Kritik oder Änderungswünsche anbringen können.

Sprint Retrospective

Nach dem Abschluss eines Sprints folgt noch einmal ein Rückblick, der dem Team die Möglichkeit zur Reflektion über das „Wie?“ gibt. Hier werden die angewandten Methoden des letzten Sprints beleuchtet und herausgearbeitet, inwieweit sich diese optimieren lassen. Auf diese Weise soll die Produktivität in jedem Sprint weiter verbessert werden. Anschließend geht das Team in das nächste Sprint Planning.

4.3.2. Anpassungen an die Gegebenheiten der Projektgruppe

Das Scrum-Prinzip ist auf professionelle Softwareentwicklung mit einem Team aus Vollzeitbeschäftigten ausgelegt. Das hatte zur Folge, dass wir insbesondere die zeitlichen Vorgaben nicht unverändert übernehmen konnten. Beispielsweise war ein Daily Scrum im Wortsinn nicht umsetzbar; stattdessen einigten wir uns auf einen wöchentlichen Termin, an dem wir uns gegenseitig die neusten Entwicklungen präsentieren konnten.

Die Dauer eines Sprints wurde von uns auf vier Wochen festgesetzt. Die Findung einer idealen Zeitspanne pro Zyklus, die in Scrum nicht verändert werden soll, gestaltete sich nicht trivial, da die Projektgruppe an eine feste Gesamtzeit und darüber hinaus an die terminlichen Rahmenbedingungen von zwei Semestern gebunden ist.

4.3.3. Rollen-Einteilung

Diesen Prozess und die Einhaltung der Scrum-Regeln und -Abläufe zu überwachen und durchzusetzen ist Aufgabe des sogenannten *Scrum Masters*, dessen Position von Marco Kuhnke besetzt wurde. Eine weitere Sonderrolle kommt dem *Product Owner* zu. Dabei haben wir uns für eine Aufgabenteilung entschieden, sodass im ersten Semester mit Jakob Bossek und Andreas Pauly je ein Teilnehmer diese Rolle für die Algorithmen- bzw. die Plattformgruppe ausfüllte. Sie waren verantwortlich für die Pflege des jeweiligen Product Backlog und sind im ursprünglichen Scrum-Konzept die Ansprechpartner für den Auftraggeber. Im zweiten Semester wurden die Aufgaben der

Product Owner von Christopher Morris und Sebastian Witte übernommen, während der Scrum Master beibehalten wurde.

Diese Sonderrollen sind in unserer Projektgruppe anders als im eigentlichen Scrum ausdrücklich Teil des *Entwicklungsteams*, dem auch alle restlichen Teilnehmer angehören. Innerhalb des Teams gibt es dann ausschließlich gleichberechtigte Entwickler und keine klassischen Projektmanager wie das in anderen Modellen der Fall ist.

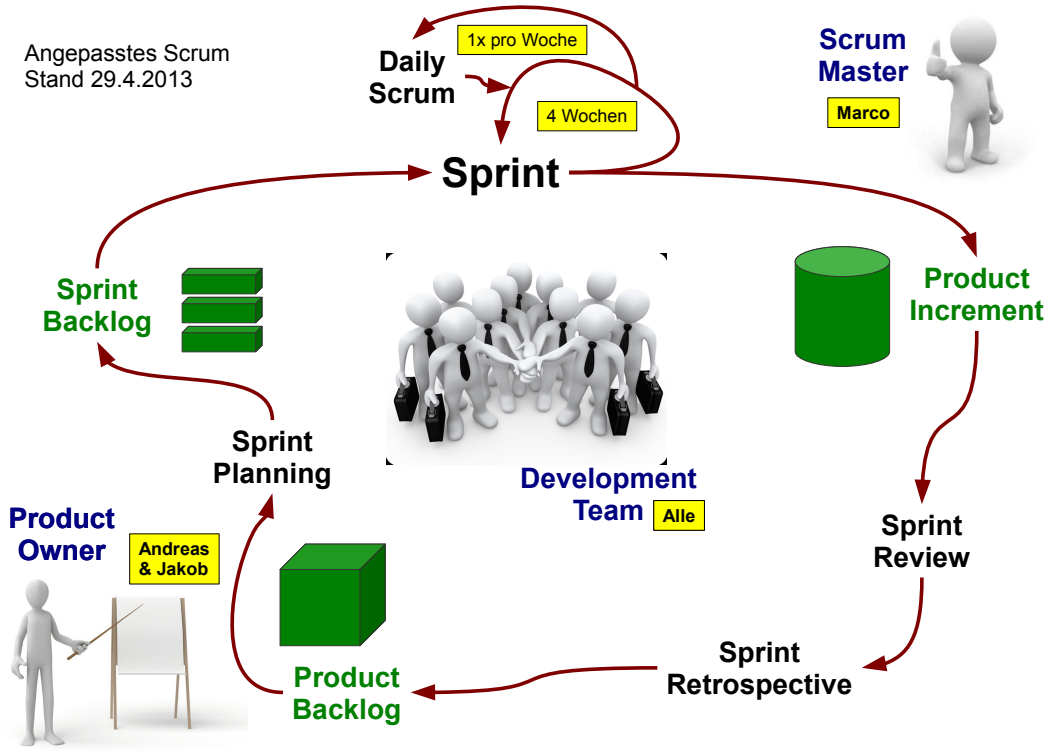

Abbildung 4.1.: Ablauf im Scrum-Prozess mit den vorgenommenen Anpassungen und den Rolleneinteilungen des ersten Semesters (gelbe Kästen). Eigene Darstellung.

4.3.4. Verlauf

Erstes Semester

Die Anfangszeit des Projekts haben wir darauf verwendet, uns mit der vorgegebenen Literatur zu beschäftigen, um allen Teilnehmern ein Grundwissen zu vermitteln.

Nach dieser Vorbereitungsphase, in der eine erste gemeinsame Vision des späteren Programms entstand, folgte vom 13. Mai bis zum 10. Juni 2013 der erste Sprint.

Es konnten in diesem ersten Zyklus einige Punkte ausgemacht werden, die zu kleineren Problemen geführt haben. Beispielsweise im Umgang mit dem Versionskontrollsystem Git (siehe Kapitel 6.7) oder bei der Nutzung des Ticketsystems wurden Fehler oder Uneinigkeiten angesprochen, die es zu beheben galt. Positiv hervorzuheben war jedoch die gute Zusammenarbeit der beiden Gruppen und die hauptsächlich durch die *Daily Scrums* forcierte gruppenübergreifende Kommunikation. Zusammenfassend mussten wir feststellen, dass wir uns quantitativ sehr hohe Ziele gesteckt hatten. Ein Grund dafür war sicherlich auch, dass zunächst das umfangreiche Fundament zu implementieren war, bevor die eigentlichen Anwendungsfälle aus den Backlogs abgearbeitet werden konnten.

Diese Erkenntnisse wurden im zweiten Sprint, der vom 17. Juni bis zum 8. Juli 2013 lief, zur Optimierung der Arbeitsweise genutzt. Am Ende des Sprints konnten wir festhalten, dass die vorgenommenen Verbesserungen hilfreich waren und wir insgesamt weit gekommen waren. Die restliche Zeit bis zum Ende des Semesters konnte dann noch für den Zwischenbericht genutzt werden.

Zweites Semester

Der dritte Sprint startete am 23. Oktober und endete vier Wochen später am 20. November 2013. Hier wurden weitere Algorithmen ausgewählt und implementiert, sowie der Grundstein für die spätere Benutzeroberfläche der Webanwendung gelegt. Der letzte Sprint im Kalenderjahr 2013 knüpfte nahtlos an den vorhergehenden an und lief bis zum 18. Dezember. Hier sollten die Ergebnisse der ausgeführten Algorithmen erstmals graphisch dargestellt werden. Im neuen Jahr 2014 wurde bis zum Semesterende hauptsächlich an der Behebung auftretender Fehler und der Fertigstellung des Endberichts gearbeitet.

4.3.5. Erfahrungen mit Scrum

Das Projektmanagement nach dem Scrum-Modell wurde in seinen Komponenten unterschiedlich wahrgenommen. Die Unterteilung in vierwöchige Intervalle hatte beispielsweise insbesondere in den ersten Sprints zur Folge, dass sich auch aufgrund der durchweg hohen Zielsetzung unsererseits bei der Vorgabe, ein lauffähiges Inkrement zu produzieren, die Arbeitszeit zum Ende eines jeden Zyklus deutlich erhöhte. Andererseits ließ sich daran regelmäßig der Fortschritt des Projekts sehr gut bestimmen und abschätzen, wie viel im nächsten Intervall in etwa zu schaffen sein würde.

Anhand der nach jedem Sprint durchgeführten Retrospectives und den darin angesprochenen Problemen der jeweils vorhergehenden Implementierungsphase ließ sich außerdem die Arbeitsweise gruppenübergreifend in mehreren Punkten stetig verbessern. So konnten etwa Unstimmigkeiten im Versionsmanagement mit Git frühzeitig erkannt werden und der Umgang mit dem Ticketsystem im Redmine vereinheitlicht werden, um so die Qualität der Arbeit schrittweise zu steigern und die Gefahr von großen Konflikten am Ende des zweiten Semesters zu minimieren.

Ebenfalls positiv empfunden wurden die in unserem Fall wöchentlich durchgeführten Daily Scrums, die allen Teilnehmern unabhängig von ihrer Zugehörigkeit zu den spezialisierten Subgruppen einen guten Überblick über den aktuellen Entwicklungsstand in jedem Bereich des Projekts ermöglichten. Das resultierte außerdem in einer guten Zusammenarbeit und in der Einbeziehung aller Teilnehmer in die Entscheidungsprozesse.

Trotz der nötigen Einarbeitung in Werkzeuge wie Wicket oder der Erstellung des Zwischen- und Endberichts, was den Arbeitsaufwand innerhalb der Sprints erhöhte und damit Entwicklungszeit kostete, konnten wir mit den einzelnen Inkrementen nach jedem Intervall sowie mit dem Endprodukt sehr zufrieden sein. Rückblickend lässt sich sagen, dass die Unterteilung in Sprints und die Einhaltung des Scrum-Konzepts anfänglich ein wenig problematisch war, uns jedoch insgesamt auf dem Weg zu einem erfolgreichen Projekt geholfen hat.

4.4. Redmine

Wir haben das Projektmanagementsystem *Redmine*[1] verwendet. Redmine erlaubt es sogenannte *Tickets* zu erstellen, die eine Arbeit oder ein Problem beschreiben. Konkret kann es sich dabei um ein geplantes Feature oder einen aufgetretenen Bug handeln. In einem Ticket wird der Prozess bis zur Lösung dokumentiert und koordiniert.

Tickets enthalten konkret ein Thema, eine Beschreibung, eine Menge von Kommentaren, eine Menge von zugeordneten Commits im Versionskontrollsystem, einen Zustand, einen Tracker, eine Kategorie, eine Priorität und eine Person, die dem Ticket zugeordnet ist. Wir haben diese Funktionen wie folgt interpretiert:

Zugeordnete Person Diese Person ist für das Ticket verantwortlich. Der Verantwortliche kann das Ticket auch jemand anderem zuweisen, der das Ticket vielleicht besser bearbeiten kann. Dies geschieht üblicherweise zusammen mit einem Kommentar.

Zustand *Zugewiesen* bedeutet, dass die Person das Ticket bearbeiten wird. *Neu* heißt, dass das Ticket gerade erstellt wurde und noch ein Verantwortlicher gesucht werden muss, der das Ticket bearbeitet. *Feedback* zeigt an, dass der Zugewiesene etwas zu dem Zustand des Tickets oder der Arbeit schreiben soll; üblicherweise steht im Kommentar, zu was genau Feedback gegeben werden soll. *Gelöst* bedeutet, dass der Bearbeiter der Meinung ist, dass das Problem gelöst ist. Typischerweise wird das Ticket gleichzeitig jemandem zugewiesen, der beurteilen soll, ob das Ticket wirklich gelöst worden ist. *Erledigt* heißt, dass das Ticket wirklich abgearbeitet ist. Ein Ticket kommt nie wieder aus diesem Zustand heraus. Sollte der Fehler nochmal auftreten, wird ein neues Ticket geöffnet. Dieser Zustand wird häufig von demjenigen gesetzt, der das Ticket ursprünglich erstellt hat.

Tracker *Feature* für neue Features, *Fehler* für Bugs und *Unterstützung* für Tickets, die ein bestehendes Feature erweitern.

[1] `http://www.redmine.org/`

Kategorie Entweder *Plattform Planung*, *Plattform Implementierung*, *Algorithmen Planung* oder *Algorithmen Implementierung*. Am Ende wurde zusätzlich die Kategorie *Endbericht* verwendet.

Thema Eine sehr kurze Beschreibung.

Beschreibung Eine ausführliche Beschreibung.

Einige weitere Features wie *geschätzter Aufwand* und *Abgabedatum* haben wir nicht verwendet.

Insbesondere im zweiten Semester wurde das Redmine intensiv benutzt. Insgesamt wurden über 250 Tickets geschrieben, davon über 90 im Tracker „Unterstützung“ und je über 60 im Tracker „Fehler“ und „Feature“. Das Redmine Ticketsystem und Wiki wurde insgesamt als sehr hilfreich empfunden und intensiv genutzt.

5

Architektur

Die Plattform ist grundlegend in zwei funktionale Einheiten aufgeteilt. Der Control Server bildet eine Einheit und verwaltet beliebig viele Worker, welche die zweite Einheit bilden. Der Worker ist in der Lage Aufträge auszuführen und die Ergebnisse dieser Aufträge in einer Datenbank zu speichern. Für die restliche Funktionalität sorgt der Control Server, der die gewünschten Aufträge an die ihm zur Verfügung stehenden Worker verteilt. Zudem bietet er die Benutzeroberfläche an und ist dementsprechend für die komplette Benutzerinteraktionslogik verantwortlich. In den folgenden Kapiteln wird detaillierter auf die beiden Einheiten eingegangen.

5.1. Worker

Die Klassenstruktur des Workers wird in einer abstrahierten Form in Abbildung 5.1 auf Seite 54 dargestellt. Die Interaktion der Klassen, die im Folgenden als thematischer Faden durch die Struktur dienen soll, wird durch das Sequenzdiagramm in Abbildung 5.2 auf Seite 55 modelliert.

Die zentrale Klasse ist der `Experimentor`. Die Hauptfunktionalität, das experimentelle Ausführen eines Algorithmus, wird durch diese Komponente initialisiert und überwacht. Wenn vom Control Server ein Auftrag zum Worker versendet wird, erhält

der `Communicator` die zugehörigen Informationen. Diese werden in einem neuen Objekt der Klasse `Job` gekapselt. Abgesehen vom Status sollen die Informationen nach der Erstellung des Objektes nicht mehr verändert werden. Deshalb bietet die Klasse zwar eine Reihe von Abfragemethoden, aber keine zum Setzen von Werten. Der Status des `Jobs` ist eine für den `Communicator` relevante Information, welche an den *Control Server* weitergeleitet werden soll. Damit nicht ständig der Status abgerufen werden muss, um auf dem aktuellen Stand zu bleiben, folgen die beiden Klassen dem *Observer Pattern*. Dafür implementiert die Klasse `Job` das Interface `Observable` und der `Communicator` das Interface `Observer`. Wenn der Status des `Jobs` neu gesetzt wird, so wird der `Observer` sofort darüber benachrichtigt. Das neu erstellte `Job`-Objekt wird nun an den `Experimentor` übergeben, der die Methode `startJob` aufruft. Sollte der `Experimentor` bereits einen Job ausführen, wird der neue zurückgewiesen und eine entsprechende Nachricht an den Server gesendet. Damit ein *Worker* möglichst selten Aufträge zurückweisen muss, wird der Status des `Experimentors` ebenfalls durch den `Communicator` beobachtet und an den Server weitergeleitet. Wurde der `Job` nicht zurückgewiesen, erstellt der `Experimentor` für das `Job`-Objekt einen neuen `JobThread`, übergibt alle wichtigen Referenzen und startet diesen mittels der `run`-Methode. Der `JobThread` holt sich nun über den `Experimentor` die Referenz der Klasse `ServiceDirectory`. Diese verwaltet die für die Ausführung von Algorithmen wichtigen *Bundles* und realisiert damit die Anbindung an das *OSGi-Framework*. Relevante Bundles sind solche, die entweder das Interface `Algorithm` oder das Interface `GraphParser` implementieren. Sie können aber auch unabhängig von diesen Schnittstellen Funktionalität zur Verfügung stellen, die vom Algorithmus oder Parser zur Laufzeit benötigt werden. Mit Hilfe der `ServiceDirectory` aktiviert der `JobThread` nun die Bundles des gewünschten `GraphParser` und `Algorithmus` mittels der zugehörigen Zugriffsmethoden `getGraphParser` und `getAlgorithm` und führt diese nacheinander aus. Dabei wird der Status des aktuellen `Jobs` je nach Stadium der Ausführung aktualisiert.

Um Messungen innerhalb der Algorithmen zu ermöglichen, wird über die Schnittstelle eine Referenz eines konkreten `Meters` übergeben, die vorher im `Experimentor` durch die `setMeter` Methode gesetzt wurde. Mittels der Methode `startExperiment` kann der Algorithmus eine neue Testreihe initialisieren und während der Laufzeit die zahlreichen `measure`-Methoden dazu nutzen, relevante Werte unter bestimmten

Pfaden abzuspeichern. Die Pfade sind Zeichenketten, die den Ort der Speicherung im Datenbankobjekt angeben. Durch das Interface können so verschiedene Datenbanken mit Inhalt befüllt werden, wobei die Standardimplementierung *MongoDB* anbindet. Ein zweites Interface `Aggregator` ermöglicht die Implementierung verschiedener Aggregationsvarianten wie zum Beispiel Mittelwertbildung oder einfache Summation, die durch die Messbibliothek genutzt werden können. Die Messbibliothek wurde im Sequenzdiagramm aus Platzgründen eingespart.

Wenn während der Initialisierung oder Laufzeit des Algorithmus ein Fehler entsteht, wird der `Job` abgebrochen und eine `JobFailedException` geworfen. Werden Algorithmus und Parser hingegen fehlerfrei ausgeführt, wird dem `Meter` mittels `endExperiment` das Ende der Messreihe mitgeteilt. Der `Experimentor` kann durch die Methode `experimentFinished` nun den Status des `Jobs` auf *„finished“* setzen und die Messergebnisse mittels des `Communicators` an den Control Server übertragen.

Die `HostActivator`-Klasse implementiert das Interface `BundleActivator`. Wenn das gesamte Worker-Bundle durch das OSGi-Framework (siehe Kapitel 6.3) aktiviert wird, wird die `start`-Methode ausgeführt, die den `Experimentor`, das `ServiceDirectory` und den eigenen *Thread* des `Communicators` initialisiert. Der Worker kann danach, wie oben beschrieben, Aufträge annehmen und ausführen sowie danach die Ergebnisse zurückschicken. Durch die Auslagerung des `Communicators` und des `Jobs` in eigene Threads wird gewährleistet, dass alle Teile des Workers auch während der Ausführung von Aufträgen interaktionsfähig bleiben.

5.2. Control Server

Der Control Server dient zur zentralen Verwaltung von Experimenten, Workern und Bundles. Dabei stellen die Methoden des Control Servers den Zugriffspunkt des Benutzers auf die Plattform. Der Zugriff geschieht über eine Weboberfläche.

Die wichtigsten Klassen des Control Servers werden kurz vorgestellt und in den folgenden Punkten genauer erläutert. Der `CSJob` enthält einen auszuführenden Algorithmus und Parameter. Im `WorkerDescriptor` werden die Hardwareinformationen zu einem Worker gespeichert. Ein `Experiment` speichert eine Liste von auszuführenden `CSJobs` und ggf. eine weitere Liste von `WorkerDescriptors`. Um zu bestimmen, welcher `CSJob` als nächstes ausgeführt wird, wird der `Scheduler` benötigt. Die Klasse `BundleManager` wird zur Verwaltung der OSGi-Bundles genutzt. Zur Kommunikation zwischen dem Control Server und Worker dient die Klasse `ControlServer`.

Der Ablauf zur Ausführung eines Algorithmus ist in Abbildung 5.3 skizziert. Zuerst wird jeweils ein Objekt der Klassen `Experiment` und `CSJob` instanziiert. Der Algorithmus im instanziierten Objekt der Klasse `CSJob` ist zu parametrisieren; falls gewünscht, kann ein Worker zur Bearbeitung im `Experiment`-Objekt eingetragen werden. Zur Ausführung wird das `Experiment` im `Scheduler` eingetragen. Der `Scheduler` bestimmt das auszuführende `Experiment` und die Reihenfolge, in welcher die eingetragenen `CSJobs` vom Worker abgearbeitet werden sollen.

CSJob

`CSJob` steht für Control Server Job. Dieser erbt von der Klasse `Job` des Worker und wird erweitert um die benötigten Teile für die Kommunikation zwischen Control Server und dem ausführenden Worker.

Die Erweiterung besteht aus den zwei Attributen `parentExperiment` und `worker`. Das Attribut `parentExperiment` enthält das `Experiment`, zu dem der `CSJob` gehört. Dies wurde getan, damit ein `CSJob` auf die Parameter des Experiments zugreifen kann. Im `Experiment` ist angegeben, auf welchem Worker der `CSJob` ausgeführt werden

soll. Für die Kommunikation zwischen dem Control Server und dem ausführenden Worker wird die IP-Adresse der Workers im CSJob eingetragen.

Für die Parametrisierung des dem CSJob zugeordneten Algorithmus wird die Web-GUI genutzt. Dafür wird die beim Algorithmus vorliegende XML-Datei geparst und die Web-GUI bietet die Möglichkeit, die Parameter zu setzen. In der XML-Datei sind die Namen, Wertebereiche und Standardwerte eingetragen.

WorkerDescriptor

Die Attribute des `WorkerDescriptors` enthalten die Hardwareinformationen eines Workers. Diese sind CPU, RAM, Computername, IP-Adresse und Port.

Die IP-Adresse und der Port werden für die Netzwerkkommunikation gebraucht. Die restlichen Attribute dienen der Beschreibung des Workers. Diese Informationen können unter Linux und Windows ausgelesen werden mit den Methoden der Klasse `SysInformation`.

In `SysInformation` gibt es für die einzelnen Attribute jeweils eine statische Methode, die die Hardwareinformation ausliest.

Für die CPU steht die Methode `getProcInfo()` zur Verfügung. Diese unterscheidet beim Aufruf zwischen Linux und Windows. Bei einem Linux-basierten System wird die Datei `/proc/cpuinfo` ausgelesen und der Modellname extrahiert. Unter Windows wird der CPU-Typ durch den Aufruf `System.getenv()` bestimmt.

Der RAM wird durch die Methode `getMemInfo()` ausgelesen. Unter Linux wird hierfür die Datei `/proc/meminfo` ausgelesen. Um unter Windows den RAM auszulesen, wird die Methode `Runtime.getRuntime().maxMemory()` genutzt; dies gibt die Größe des der JVM zugewiesenen RAM an.

Um den Computernamen zu bekommen, wird die Methode `getHostName()` genutzt. Diese Methode arbeitet unter Linux und Windows identisch.

Als letzte Methode steht `getIpAddress()`, womit die IP-Adresse ausgelesen wird. Hierfür werden unter Linux alle Netzwerk-Interfaces betrachtet. Dabei wird geprüft, ob es sich um keine Loopback-Adresse und um eine lokale Adresse handelt. Unter Windows wird die lokale Adresse ausgegeben.

Für den `Scheduler` wurde ein Attribut hinzugefügt, um Informationen darüber zu haben, ob der Worker im Moment einen Job bearbeitet.

Experiment

Ein Experiment besteht aus einer Liste von zu bearbeitenden Jobs. Außerdem ist es möglich, spezifische Worker zur Bearbeitung des Jobs anzugeben. Dabei können mehrere Worker angegeben werden, von denen dann vom `Scheduler` einer zur Bearbeitung des Jobs ausgewählt wird. Alternativ kann das Attribut `singleWorker` gesetzt werden, wodurch nur ein Worker zur Bearbeitung aller Jobs genutzt wird. Das Attribut `priority` betrachtet der `Scheduler` bei der Auswahl eines Jobs. Dabei entsprechen niedrige `priority`-Werte einer hohen Priorität. Mit diesem Attribut kann die Reihenfolge der bearbeiteten Experimente gesteuert werden.

Ein Experiment kann in einem von vier Zuständen sein:

- *„new“*: Das Experiment wurde erzeugt.
- *„active“*: Ein Job des Experiments wird ausgeführt
- *„waiting“*: Kein dem Experiment zugewiesener Worker steht zur Bearbeitung eines Jobs zur Verfügung.
- *„terminated“*: Alle Jobs des Experiments wurden ausgeführt.

Ein Experiment wird durch einen Aufruf im `Scheduler` instanziiert und bekommt eine ID übergeben. Anschließend können `CSJobs` durch die Methoden `addJobs()` oder `addJob()` hinzugefügt werden. Die Methode `addJobs()` erwartet, im Gegensatz zu `addJob()`, neben dem `CSJob` noch die Anzahl, wie oft er hinzugefügt werden soll.

Zum Entfernen stehen die Methoden `removeJob()` und `removeJobs()` zur Verfügung. Als Parameter erwarten beide Methoden den zu löschenden Job. Mit der Methode `removeJobs()` können zusätzlich Kopien desselben Jobs gelöscht werden; dafür bekommt die Methode noch einen Parameter übergeben, der angibt, wie viele Kopien zu löschen sind. Um einen `WorkerDescriptor` hinzuzufügen, wird die Methode `assignWorker()` aufgerufen, welcher der hinzuzufügende `WorkerDescriptor` übergeben wird. Um einen Worker wieder zu entfernen, wird die Methode `unassignWorker()` genutzt. Bei der Instanziierung eines Experiments wird das Attribut `singleWorker` auf `false` gesetzt; zur Änderung dient die `setSingleWorker()`-Methode. Eine Änderung kann nur erfolgen, solange sich das Experiment im Zustand *„new“* befindet.

Scheduler

Der `Scheduler` dient der Bestimmung, welche Jobs ausgeführt werden. Dafür wird in ihm eine Liste von aktuell offenen Experimenten und den registrierten Workern gehalten. Um Experimente hinzuzufügen und zu löschen, gibt es die Methoden `newExperiment()` und `cancelExperiment()`. Bei der Methode `newExperiment()` wird das neue Experiment auch in die Bearbeitungsliste des `Schedulers` eingefügt. Für die Worker stehen die Methoden `addWorker()` und `deleteWorker()` zur Verfügung. Die Bestimmung, welcher Job als nächstes ausgeführt werden soll, erfolgt mittels `schedule()`. Dabei wird der Job des Experiments mit der niedrigsten Priorität ausgewählt. Als nächstes wird betrachtet, ob im Experiment ein Worker angegeben ist, der im Moment keinen Job ausführt. Wenn dies der Fall ist, wird der Worker zur Bearbeitung des Jobs ausgewählt und der Zustand des Experiment wird auf *„active“* gesetzt. Sollte für den `CSJob` kein Worker frei sein, wird der Zustand des Experiments auf *„waiting“* gesetzt und das nächste Experiment betrachtet.

BundleManager

Die entwickelten Algorithmen sind in Bundles gespeichert. Dabei wird eine *jar*-Datei mit einer *Manifest*-Datei zusammengefasst. Der `Bundlemanger` hat Zugriff auf die Datenbank, um Bundles zu speichern und sie bei Bedarf wieder zu laden.

Ebenso können Bundles gelöscht werden, z. B. falls diese veraltet oder fehlerhaft sein sollten.

Die Methode `uploadBundleOrBundlesInDirectory()` steht zum Hochladen von Bundles bereit. Sie bekommt entweder einen Ordner oder ein einzelnes Bundle zum Hochladen angegeben. Mittels `getFile()` kann ein Bundle wieder heruntergeladen werden. Dabei wird ein Deskriptor des Bundles als Parameter übergeben.

Zum Löschen eines Bundles wird die Methode `removeBundleFromDatabase()` genutzt. Diese bekommt ebenfalls einen Deskriptor des zu löschenden Bundles als Parameter übergeben.

ControlServer

Zur Netzwerkkommunikation zwischen dem Control Server und dem Worker dient die `Controlserver`-Klasse. Die Klasse `Controlserver` verwaltet die Liste der registrierten Worker und hat den `Scheduler` instanziiert.

Zum Start des Control Servers dient die Methode `run()` und zum Stoppen wird die Methode `stop()` genutzt. Um einen Job an einen Worker zu senden, wird die Methode `sendJob()` genutzt und über `handleMessages()` kommuniziert der Worker mit dem Control Server. Darüber wird auch die Information gesendet, wenn der Worker den Job beendet hat.

5.3. Kommunikation

Da die Module Worker und Control Server verschiedene Prozesse sind, ist eine Form von Kommunikation notwendig, mit der sie verschiedenste Nachrichten austauschen können. Hauptsächlich sind dabei zwei Aufgaben zu erfüllen, und zwar die Übermittlung von Aufträgen vom Control Server an einen Worker und Statusupdates vom Worker an den Control Server. Wir entschieden uns für eine String-basierte Netzwerkkommunikation, da sie einfach zu realisieren und zu erweitern ist und

zudem betriebssystem- wie programmiersprachenunabhängig ist, sodass wir uns die Möglichkeit offen halten, einen Worker in einer von Java verschiedenen Sprache zu realisieren. Zudem werden Messergebnisse wie Laufzeiten kaum beeinflusst, wenn Control Server und Worker auf unterschiedlichen Maschinen ausgeführt werden. Im Sinne der einfachen Erweiterbarkeit der Kommunikationsmöglichkeiten beginnt jede Nachricht, bestehend aus mehreren Strings, mit einem für den bezweckten Fall (Auftragsübermittlung, Statusupdate, sonstige Anweisungen) eindeutigen String, wie zum Beispiel *„newJob“* für die Auftragsübermittlung. Jede Nachricht endet zudem, vergleichbar mit einer E-Mail gemäß SMTP, mit einem String, der lediglich einen einzelnen Punkt enthält.

5.3.1. Übermittlung von Aufträgen

Die Übermittlung von Aufträgen ist einer der wichtigsten Teile der Kommunikation, der immer dann gebraucht wird, wenn der Control Server einem Worker einen Auftrag zuweist. Ein Auftrag besteht, wie zuvor beschrieben, aus einer Implementierung eines Algorithmus in der Datenbank, einem Graphen als Eingabeinstanz in der Datenbank, einem Parser, der den Graphen in das zum Algorithmus passende Format wandelt, und einem zur gewählten Implementierung passenden Parametersatz. Die Kommunikation beginnt in diesem Fall immer beim Control Server, da dieser alle Aufträge überblickt und auf die Worker verteilt. Stehen alle oben genannten Elemente fest, so sendet der Control Server den String *„newJob“* an den Worker, der den Auftrag ausführen sollte, gefolgt von einem String der Form *„<Komponente>: <Parameter>: <Wert>“* für jedes weitere spezifizierende Element des Auftrags. Der Teilstring *„<Parameter>“* stellt hier einen Platzhalter für einen von der angegebenen Implementierung erwarteten Bezeichner dar, analog steht *„<Wert>“* für den Wert, mit dem dieser Parameter belegt werden soll. Der Teilstring *„<Komponente>“* wurde hinzugefügt, da wir einen Mechanismus brauchten, um verschiedene Mengen von Parametern zu handhaben. So können nun Parameter für den Algorithmus und für den Parser getrennt übermittelt werden, und auch Datenstrukturen lassen sich parametrisieren, ohne die Datenstrukturen für die restlichen Parametermengen zu beeinflussen. Für die nicht-Parameter-Werte wie den zu verwendenden Parser oder der Job-ID wird nach wie vor das Format *„<Parameter>: <Wert>“* verwendet,

wobei als Parameterbezeichnung festgelegte Strings wie *„jobID“* verwendet werden. Die Reihenfolge, in der diese Werte übermittelt werden, ist dabei nicht relevant.

5.3.2. Übermittlung von Statusupdates

Es gibt vordefinierte Status, die den aktuellen Bearbeitungsstand des Workers bezeichnen, während er gerade nicht den eigentlichen Algorithmus gemäß dem durch ihn zu bearbeitenden Auftrag ausführt. Diese kennzeichnen zum Beispiel, dass der Worker aktuell die Eingabeinstanz aus einer gegebenen Textdatei parst. Zudem hat der Entwickler eines Algorithmus für MONET die Möglichkeit, eigene Status zu definieren, die der Worker im Laufe der Bearbeitung des Algorithmus annehmen kann. Im Falle einer Statusänderung schickt der Worker einen String der Form *„newState: <Bezeichnung>“* an den Control Server, gefolgt von einem String, bestehend aus einem Punkt, da der Benutzer auch Bezeichnungen wählen kann, die einen Zeilenumbruch enthalten. Eine weitere Möglichkeit für den Entwickler, die Ausführung seines Algorithmus zu beobachten, sind Logging-Nachrichten. Diese werden, ähnlich wie Aufträge, durch eine Reihe spezifizierender Strings übermittelt, allen voran der String *„logging“*. So kann der Nutzer beliebige Nachrichten – und bei Verwendung von `Errorlogs` sogar Stacktraces – und weitere Informationen übermitteln, um das Debugging so einfach wie möglich zu gestalten.

5.3.3. Ordentlicher Verbindungsaufbau und -erhalt

Bevor ein Worker einen Auftrag entgegennehmen kann, muss dieser dem Control Server bekannt gemacht werden. Hierzu geht der Verbindungsaufbau mit den von Java angebotenen Klassen `Socket` und `ServerSocket` vom Worker aus, der anschließend Systemspezifikationen wie CPU, RAM und einen Computernamen übermittelt. Dies geschieht, wie auch jede bisher beschriebene Kommunikation, mithilfe von fest vorgegebenen String-Formaten, wie zum Beispiel *„CPU: <Bezeichnung>“*.

Die Klassen `Communicator` und `Controlserver` wurden erweitert, sodass nun auf beiden Seiten (Worker und Control Server) erkannt werden kann, dass keine weitere

Kommunikation möglich ist, sei es durch einen Absturz oder Verbindungsverlust. Dies wird durch ein einfaches Protokoll realisiert: Empfängt eine Seite ein *„ping“*, sendet sie ein *„pong“* zurück. Beide Seiten führen dieses Protokoll aus, da beide Seiten darauf angewiesen sind, einen Verbindungsverlust zu erkennen. Erkennt der Worker einen Verbindungsverlust, so versucht er wiederholt, die Verbindung zum ihm bekannten Control Server wieder aufzubauen. Ein Control Server muss zudem verschiedene seiner Datenstrukturen, wie die Menge aller Worker und die sich in Ausführung befindlichen Jobs, aktualisieren. Verliert der Control Server seine Verbindung zu einem arbeitenden Worker, so geht dieser davon aus, dass der von diesem Worker bearbeitete Job fehlgeschlagen ist.

5.3.4. Weiteres

Das vorzeitige Abbrechen von Jobs und das ferngesteuerte Beenden von Workern über die GUI ist ebenfalls möglich. Auch hierfür verwenden wir vordefinierte Strings, die der Worker erkennt und entsprechend reagiert. Um einen Job vorzeitig abbrechen zu können, wird das Werfen einer `Exception` im Job Thread durch den Worker Thread ausgelöst. Wir entschieden uns für diese Variante, da jeder andere Ansatz von Interprozesskommunikation in Java verlangt, die Algorithmen umzuschreiben, sodass diese eine bestimmte Kontrolle wiederholen und in kurzen Zeitabständen ausführen, was sich auf ihr Laufzeitverhalten auswirkt. Zudem sind wir für die korrekte Arbeitsweise des Workers oder des Control Servers nicht darauf angewiesen, Algorithmen „ordentlich“ zu beenden.

Die Verwaltung von Dateien, also Graphen und Bundles, wurde mit MongoDB realisiert (siehe Kapitel 6.2). Da diese als Datenbank bereits von sich aus netzwerkfähig ist und Dienste zum Ablegen und Abrufen von Dateien zur Verfügung stellt, müssen wir uns nicht um das Übermitteln von Dateien kümmern, sondern können Dateien auf der Seite des Control Servers unter einem festen Schlüssel ablegen und auf der Seite des Workers mithilfe dieses Schlüssels auf die Datei zugreifen. Dies wird immer getan, um auf Dateien zu verweisen, sofern diese für einen Auftrag den Wert eines Parameters bilden. So haben wir die Möglichkeit, verschiedene Implementierungen von Datenstrukturen als Parameter zu wählen.

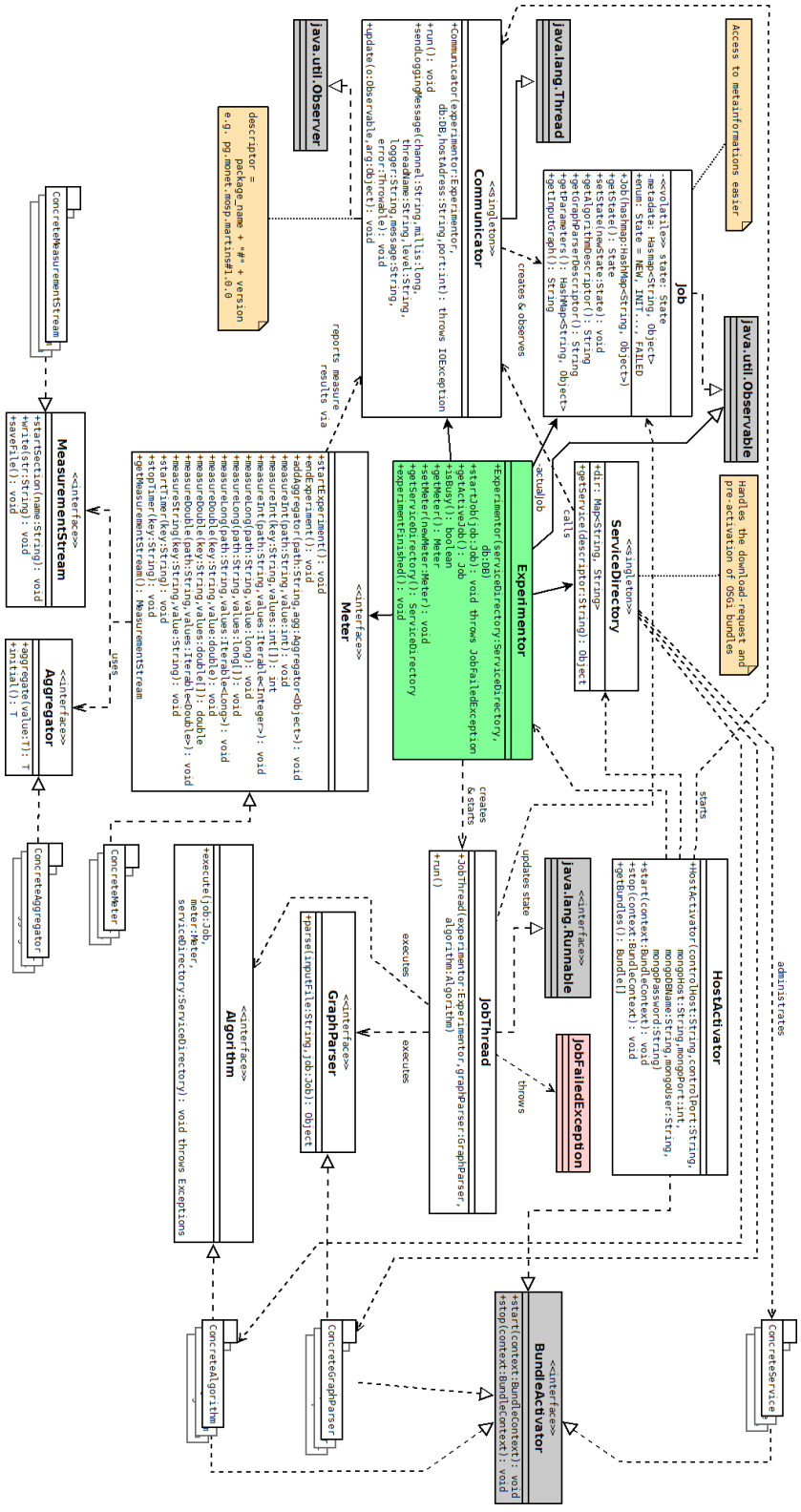

Abbildung 5.1.: Klassendiagramm Worker

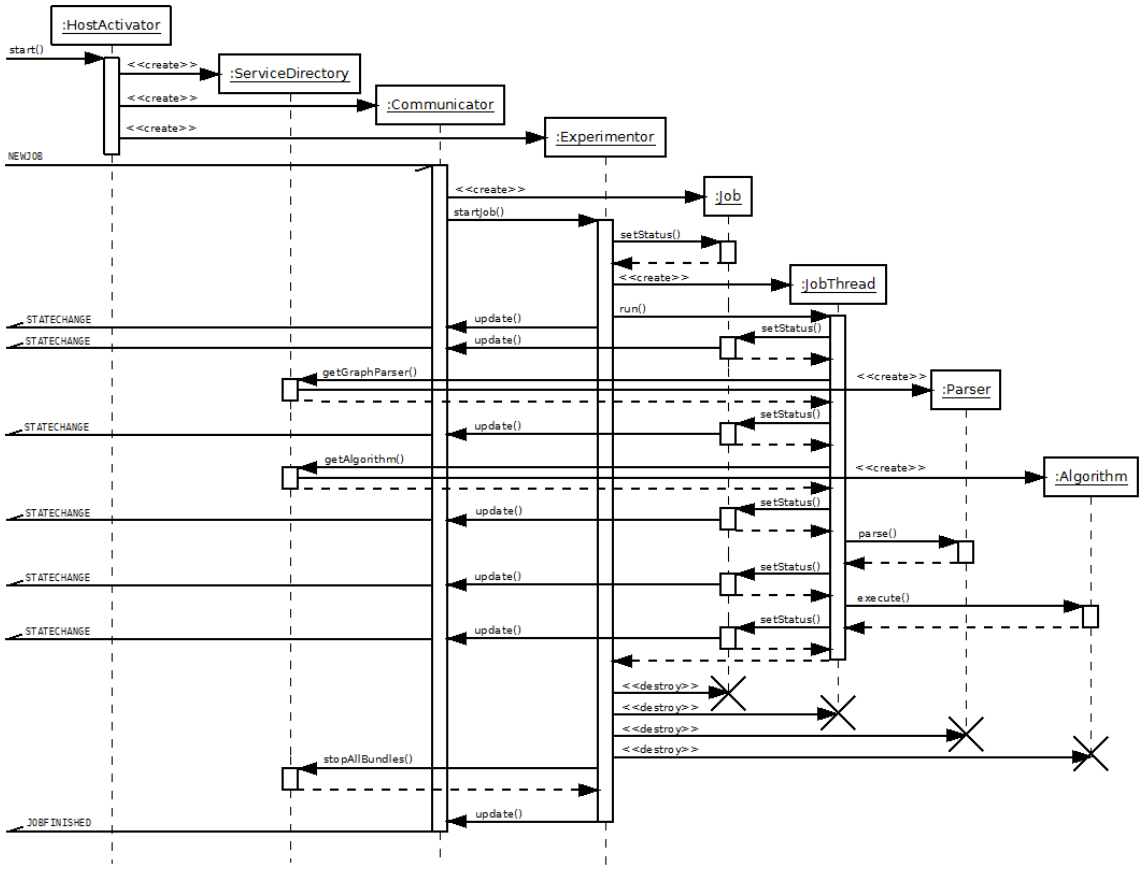

Abbildung 5.2.: Sequenzdiagramm Worker: Abarbeitung eines Auftrags

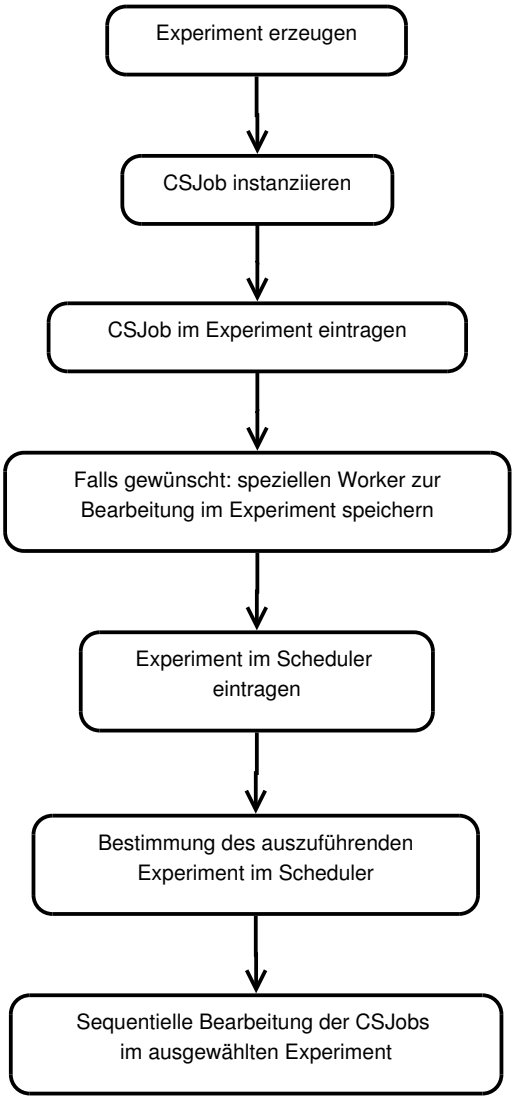

Abbildung 5.3.: Schritte zur Ausführung eines Algorithmus

6

Technische Aspekte

In diesem Kapitel sollen verschiedene technische Aspekte von MONET vorgestellt werden. Insbesondere soll gezeigt werden, wieso eine entsprechende Technologie (gegenüber anderen) gewählt wurde und wie diese in MONET eingesetzt wird. Diese Technologien sind: Java, MongoDB, OSGi und Apache Felix, Maven, verschiedene Graphformate, Git sowie eine Reihe von Webtechnologien, wie Apache Wicket, *HTML5*, *CSS3*, *JavaScript*, *jQuery* und *canvasExpress*.

6.1. Java

Java ist eine Programmiersprache, die Teil der *Java*-Technologie ist, welche aus dem Entwicklerwerkzeug *Java Development Kit (JDK)* und der Laufzeitumgebung *Java Runtime Environment (JRE)* besteht. Die JRE setzt sich aus einer virtuellen Maschine (Java Virtual Machine JVM) und einer Reihe von Standardbibliotheken zusammen. Diese Hierarchie ist in Abbildung 6.1 dargestellt.

Durch das gewählte Programmierparadigma der Objektorientierung weist Java viele Ähnlichkeiten zu älteren Sprachen wie *C++* auf. Die Entwickler der Sprache, ursprünglich *Sun Microsystems*, mittlerweile *Oracle*, hatten eine Reihe von Zielvorgaben, die durch *Java* erreicht werden sollten. Java sollte einfach, objektorientiert, verteilt,

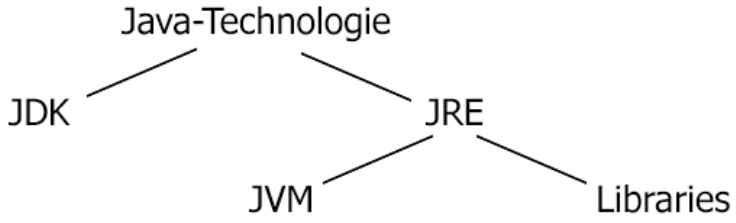

Abbildung 6.1.: Hierarchie Java-Technologie

robust, sicher, architekturneutral, portabel, leistungsfähig, interpretierbar, parallelisierbar und dynamisch sein. Insbesondere die Architekturneutralität und Portabilität sind wichtige Merkmale, die Java von vielen anderen Sprachen abhebt. Der Compiler, welcher Teil des JDK ist, übersetzt den Programmcode in Java-Bytecode, der in der JVM ausgeführt werden kann. Das heißt, die Ausführung des Programms geschieht in einer eigenen virtuellen Maschine und nicht direkt auf der darunterliegenden Hardware. Die JVM ist für alle gängigen Betriebssysteme und viele eingebettete Systeme erhältlich, was zu einer Architekturneutralität und damit umfangreichen Kompatibilität führt. Die Portabilität wird durch standardisierte primitive Datentypen erreicht, die unabhängig von der Architektur immer die gleichen Grenzen, arithmetischen Verhaltensweisen und die gleiche Darstellung haben. Ein Verzicht auf die aus den *C*-Sprachen bekannte Zeigerarithmetik sowie die starke Typisierung und Ausnahmebehandlung macht Java robuster als vergleichbare Sprachen, aber auch weniger umfangreich und mächtig. Ein weiterer Aspekt der Robustheit ist die *Garbage Collection*, welche nicht mehr benutzte Daten aus dem Speicher entfernt und unerwünschte Nebeneffekte wie Speicherlecks verhindert. Das Ziel der Einfachheit ist gleichzeitig auch ein Hauptkritikpunkt an Java, da der Sprachumfang im Vergleich zu den C-Sprachen als zu reduziert bemängelt wird. Es fehlen Möglichkeiten wie Operatorüberladung, Mehrfachvererbung und die bereits erwähnte Zeigerarithmetik. Durch Standardbibliotheken wird ein einfacher Zugriff auf häufig genutzte Systemfunktionen wie das Multithreading oder die Kommunikation über ein Netzwerk (z. B. mit TCP/IP) ermöglicht.

Unsere Wahl fiel auf Java, da die durchschnittliche Erfahrung der Projektgruppenmitglieder in dieser Sprache am höchsten war. Außerdem konnten wir so eine größtmögliche Portabilität sicherstellen, da MONET sowohl unter *Linux* als auch unter *Mac OS X* und *Windows* entwickelt, kompiliert und ausgeführt wird. Der

Nachteil, dass Java weniger performant als C oder C++ sein soll, ist nicht relevant, da die Laufzeiten der implementierten Algorithmen nicht möglichst niedrig, sondern lediglich vergleichbar sein sollen. Mit Java sind außerdem die inhärenten Eigenschaften der MONET Architektur, Modularisierung und ein verteiltes System, effizienter und weniger aufwendig zu realisieren.

Die Erfahrung, die wir im Laufe des Projekts mit Java gemacht haben, sind zwiespältig. Die im vorangegangenen Text beschriebenen und von uns anvisierten Vorteile der Java-Technologie, wie zum Beispiel die einfache Anbindung von Multithreading, haben sich für uns ausgezahlt. Probleme gab es bei der Entwicklung des Graph-Frameworks und bei Algorithmen, die viel Arbeitsspeicher benötigen. Wenn es dazu kommt, dass ein Algorithmus mehr temporären Speicher benötigt, als die JVM zur Verfügung stellt, muss die Garbage Collection sehr oft aktiv werden, was sich stellenweise so deutlich negativ auf die Laufzeit ausgewirkt hat, dass es nicht mehr zu vernachlässigen war.

6.2. MongoDB

MongoDB[1] ist eine schemalose Datenbank, die anstatt mit SQL und Tabellen dokumentenorientiert arbeitet. Dabei werden *JSON*-Objekte[2] in *Collections* abgespeichert und indiziert. Es können Indizes zu jedem Feld in einem Dokument einer Collection erstellt und zum effizienten Suchen genutzt werden. Man kann aber auch in nicht indizierten Feldern suchen. Unterstützt werden außerdem das Speichern von Dateien (GridFS) und Features wie Replikation und Map-Reduce.

In MONET wird MongoDB eingesetzt, um die Messergebnisse einer Ausführung eines Algorithmus zu speichern. Dabei werden mehrere Ausführungen zu Experimenten zusammengefasst, die eine einheitliche Experiment-ID haben. Dem Nutzer ist es angeraten, für alle diese Ausführungen das gleiche Schema zu verwenden, denn dadurch ist es möglich, Suchanfragen über diese speziellen Messungen zu machen.

[1] `http://www.mongodb.org`

[2] Die sogenannte JavaScript Object Notation, kurz JSON (`http://json.org`), ist ein modernes, kompaktes und leichtgewichtiges Dateiformat, welches aufgrund seiner Einfachheit gerne für den Datenaustausch zwischen Anwendungen genutzt wird.

Zudem werden aber auch einige Dinge wie die Laufzeit automatisch gemessen und gespeichert.

Weiterhin wird MongoDB dazu genutzt, die Algorithmen und Parser in GridFS zu speichern und die Abhängigkeiten von Algorithmen aufzulösen. Darüber hinaus ist es möglich, bei jeder Ausführung eine Datei mit Messdaten anzulegen, falls sehr viele Daten (mehr als 16MB) gemessen werden.

MongoDB hat eine exzellente und ausführliche Dokumentation und ist gut erprobt in vielen Projekten wie SourceForge, Forbes und The New York Times[3]. Für MONET ist insbesondere der Aspekt der Schemalosigkeit wichtig, da den Entwicklern nicht bekannt ist, welche Daten die Implementierer eines Algorithmus messen möchten und welches Schema sie dabei verwenden möchten. Trotz dieser Schemalosigkeit erlaubt MongoDB leichte Suchoperationen auf beliebigen Feldern.

6.3. OSGi und Apache Felix

OSGi ist der de-facto-Standard für die dynamische Ausführung von Code in Java, also das Nachladen von Code in einem Programm während dieses bereits läuft. Dabei werden dynamisch auszuführende Code-Teile in sogenannten Bundles abgepackt und dann ausgeführt. Diese Bundles sind `jar`-Dateien, die neben dem compilierten Code ein Manifest über das Bundle enthalten. Besonders wichtig in diesem Manifest sind die importierten und exportierten Java-Packages, denn diese werden von einer OSGi-Plattform dazu genutzt, alle Abhängigkeiten von *Bundles* aufzulösen. Die OSGi-Plattform verwaltet den kompletten Lebenszyklus (siehe Abb. 6.2) eines Bundles.

OSGi selber ist lediglich ein Standard, für den es drei wichtige Implementierungen gibt: Eclipse Equinox[4], Apache Felix[5] und *Knopflerfish*[6]. Tatsächlich unterscheiden sich diese drei Implementierungen kaum. Für MONET wurde Apache Felix ausgewählt,

[3] `http://www.mongodb.org/about/production-deployments/`
[4] `http://www.eclipse.org/equinox/`
[5] `http://felix.apache.org/`
[6] `http://www.knopflerfish.org/`

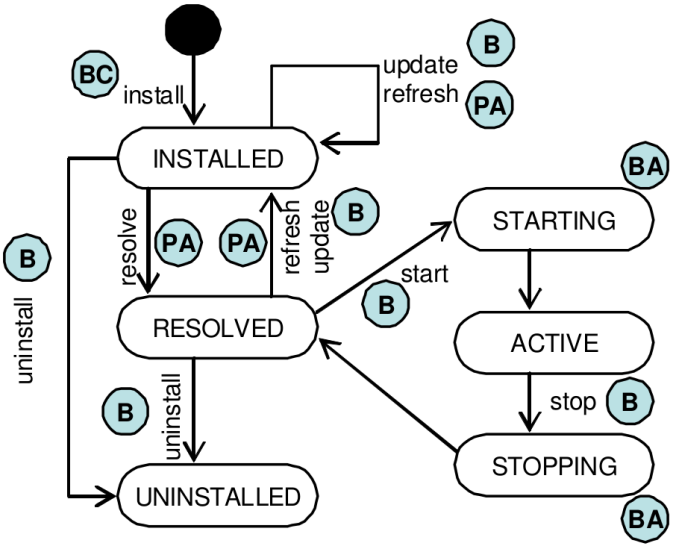

Abbildung 6.2.: Lebenszyklus von OSGi Bundles[8]

da dies die übersichtlichste und, zumindest in den Augen der Entwickler, beste Dokumentation der drei hat.

Eingesetzt wird OSGi in MONET im Worker, um Algorithmen und Parser dynamisch ausführen zu können. Dabei wird jeder Algorithmus und jeder Parser in ein OSGi-Bundle verpackt und über den Control Server an MongoDB hochgeladen. Ein Worker löst bei der Ausführung eines Algorithmus mittels des `ServiceDirectory` die Abhängigkeiten auf und holt sich den Algorithmus, den nötigen Parser und deren Abhängigkeiten von MongoDB. Anschließend wird der Algorithmus in dem im Worker eingebetteten Apache-Felix-Framework ausgeführt. Normalerweise kann OSGi alle Abhängigkeiten auflösen; hier gibt es aber einen Sonderfall, da in MONET dem Worker nicht alle Bundles bekannt sind. Deshalb ist es nötig, dass MONET selbst die Abhängigkeiten vorher auflöst. Apache Felix startet und installiert trotzdem nur die Bundles, die wirklich nötig sind. Nach jeder Ausführung eines Algorithmus werden alle Bundles gestoppt und deinstalliert, damit nicht benötigte Bundles bei der Ausführung des nächsten Algorithmus nicht im Speicher liegen.

6.4. Maven

Apache Maven, kurz Maven ist ein Management-Werkzeug für Softwareprojekte. Dabei wird für ein Softwareprojekt eine Datei namens `pom.xml` angelegt, die das Projekt und dessen Abhängigkeiten beschreibt. Maven kann dann mittels dieser `pom.xml` ein Projekt kompilieren, die Dokumentation erstellen, auf einem Server installieren und vieles mehr. Details wie das Auffinden und Installieren von Abhängigkeiten erledigt Maven selbstständig. Dadurch wird der Build-Prozess eines Projektes wesentlich einfacher und vor allem beschleunigt.

MONET besteht aus einer ganzen Reihe von *Maven-Modulen* und einem einzigen Maven-Projekt. Dies ist praktisch, da MONET aus einer Reihe von Teilen besteht, die weitestgehend unabhängig sind. Auch wenn sich die Konfiguration und das Schreiben der `pom.xml` häufig als schwierig erwiesen hat, ist Maven eine erhebliche Hilfe bei der Entwicklung von MONET. Sowohl die Schwierigkeiten als auch die Vorteile liegen an der enormen, aber notwendigen Zersplitterung von MONET in einzelne Projekte. Der vielleicht größte Vorteil ist der, dass Maven die OSGi-Bundles erstellt. Dies ist sonst ein mehrschrittiger und langwieriger Prozess: Kompilieren der Javaquelldateien, Erstellen des Manifestes und Verpacken der `jar`-Dateien.

Maven hat sich insgesamt sehr bewährt, lediglich das inkrementieren der Versionsnummer ist sehr mühselig. Dafür wurde kein Skript geschrieben, wie sonst häufig üblich, da dies ohnehin nur zweimal gemacht worden ist. Störend war die schlechte Integration der Maven Life Cycle in Eclipse mittels *m2e*: Nach verschiedenen git-Operationen war das Projekt scheinbar nicht mehr in Eclipse ausführbar und es musste ein "Maven Refresh" durchgeführt werden. Dies war aber für die Integration von *log4j2* nötig.

Ein Entwickler, der an MONET entwickeln will, muss nur wenige Änderungen an der `pom.xml` vornehmen. In Abschnitt 9.6 und Abschnitt 9.5 wird dies für die Entwicklung von Algorithmen und Parsern genauer beschrieben.

6.5. Webtechnologien

Die Oberfläche der MONET-Plattform ist webbasiert. Die Umsetzung erfolgte basierend auf einer Reihe von Technologien, Frameworks und Sprachen, welche in diesem Kapitel vorgestellt werden. Neben dem Java-Framework Wicket, welches auf Serverseite zum Einsatz kommt, zählen hierzu die Webstandards HTML, CSS sowie die Skriptsprache JavaScript, welche für die Darstellung und Interaktion mit dem Benutzer auf der Clientseite zum Einsatz kommen.

6.5.1. Wicket

Kernpfeiler der Plattform ist das komponentenbasierte Web-Framework *Apache Wicket*[7]. Es erlaubt eine einfache, strikt objekt-orientierte Entwicklung von Webapplikationen.

Wicket setzt im Vergleich zu der Vielzahl alternativer Java-basierter Web-Frameworks nur geringfügig auf XML-Konfigurationsdateien und führt überdies keine spezielle HTML-Syntax ein. Vielmehr wird die HTML-Syntax durch mit dem Webstandard verträgliche Namesspaces sinnvoll erweitert. Java und HTML kommunizieren dabei durch sogenannte *Wicket-Ids*. Dies sind Komponenteneigenschaften in Java und einfache HTML-Attribute in HTML. Auf diese Weise ist eine weitestgehende Unabhängigkeit von Applikationsentwicklung und Entwicklung der Oberfläche möglich.

6.5.2. Webstandards

Das Design der Weboberfläche entstand zunächst als einfaches Mockup in Photoshop[8] und wurde anschließend bis ins Detail ausgearbeitet. Besonderer Wert wurde dabei auf die Lesbarkeit und Einhaltung der Übersicht auf jeder Seite der Applikation gelegt. Nach dieser Konzeptions- und Designphase wurde das Design mittels

[7] `http://wicket.apache.org`
[8] http://www.adobe.com/de/products/photoshop.html

der Webstandards HTML5 (Hyper Text Markup Language) und CSS3 (Cascading Stylesheets) umgesetzt.

Hyper Text Markup Language HTML5 ist die aktuelle Version einer Textauszeichnungssprache, die 1989 am CERN (Europäische Organisation für Kernforschung) in der Schweiz zum Zwecke des Datenaustausches von Forschungsergebnissen mit anderen Forschern entwickelt wurde. Es handelt sich dabei wohlgemerkt um eine Auszeichnungssprache (Markup Language) mit einfacher Syntax und nicht um eine Programmiersprache. Der wesentliche Aufbau einer HTML-Datei besteht aus dem HTML-Kopf, welcher technische Aspekte wie z. B. die Sprachversion (document type oder kurz doctype), Import von JavaScript- oder CSS-Dateien oder diverse Metainforamtionen beinhaltet, sowie dem HTML-Rumpf, welcher den eigentlichen für den User sichtbaren Teil des Dokuments ausmacht. Der HTML-Rumpf besteht im Wesentlichen aus Text, welcher zum Teil von speziellen HTML-Elementen umschlossen ist. So wird etwa eine Überschrift erster Ordnung durch `<h1>Überschrift</h1>` dargestellt. Neben der visuellen Hervorhebung der Überschrift im Browser strukturiert HTML das Dokument dahingehend, dass die Inhalte von automatisierten Programmen besser verarbeitet werden können. Suchmaschinen gewichten etwa Schlüsselwörter, die in Überschriften stehen, stärker als Text, der im Fußbereich der Seite dargestellt wird.

In Zusammenwirkung mit Wicket existiert nicht für jede Seite ein eigenes HTML-Dokument mit Kopf- und Rumpf. Vielmehr werden sogenannte Templates/Vorlagen angelegt, welche ineinander eingebettet werden. Auf diese Weise wird Redundanz weitestgehend vermieden. Weiterhin sind Wicket-Pages in der Regel nicht statisch, sondern vielmehr dynamisch. D. h. die letztlich auf den Webseiten angezeigten Informationen werden bei Page-Requests aus Datenbanken gelesen und in die HTML-Templates eingefügt; der Browser erhält schließlich nur noch ein konstruiertes HTML-Dokument.

Cascading Style Sheets Frühere HTML-Versionen verfügten über spezielle Elemente und Attribute zur grafischen Gestaltung von HTML-Dokumenten. Mit diesen Elementen war man in der Lage, Einfluss auf die Darstellung des Dokuments im Browser zu nehmen. Dadurch wird jedoch Struktur und Präsentation vermischt. Um diesen

Missstand zu beheben, wurde die Stilsprache CSS (Cascading Stylesheets) entwickelt. Diese Sprache bietet die Möglichkeit, durch diverse Selektoren Mengen/Teilmengen von HTML-Objekten mit speziellen Eigenschaften auszuwählen und diesen spezifische Stileigenschaften zuzuweisen. Auf diese Weise kann man beispielsweise ganz leicht alle Überschriften erster Art orange setzen und die Schriftart abändern. Durch die namensgebende Kaskade bzw. Kaskadierung von Selektoren ist es überdies möglich, komplexe Selektoren aus einfachen Selektoren zusammenzusetzen. So kann man etwa alle Überschriften erster Ordnung, welche sich im Abschnitt mit der ID `experiments` befinden, anders gestalten als die Überschriften außerhalb dieses Bereiches.

HTML und CSS wurden im Rahmen der Projektgruppe eingesetzt, um das grafische Layout umzusetzen.

JavaScript Mit Hilfe von HTML wird die Struktur eines Webdokuments festgelegt. Mittels CSS wird die Präsentation des Dokuments im Browser gesteuert. JavaScript schließlich dient der dynamischen Veränderung von HTML-Seiten während der Anzeige im Browser. Die MONET-Plattform nutzt JavaScript für einige dynamische Aspekte wie z. B. die automatische Generierung von Filter-Buttons für Experimente, die Anzeige von Bestätigungsfenstern nach dem Klick auf kritische Aktionen wie dem Abbruch von Experimenten oder visuell ansprechende FadeIns bzw. FadeOuts von Zusatzinformationen. Grundlegend hierfür ist das jQuery-Framework[9], welches die Selektion von HTML-Objekten stark vereinfacht und überdies diverse Methoden für Animationen bereitstellt, die browserübergreifend funktionieren.

Die Entwicklung der grafischen Benutzeroberfläche erwies sich als unproblematisch. Dies ist hauptsächlich auf das hervorragende Zusammenwirken der eingesetzten Websprachen HTML, CSS und JavaScript zurückzuführen. Die Plattform wurde ausgiebig unter den neusten Versionen der Browser mit den höchsten Marktanteilen[10] getestet. Hierzu zählen die Browser Safari 7.0.2 (OS X), Google Chrome 33, Opera 12.16 (Windows und OS X), Firefox 27 (Windows und OS X) sowie Internet Explorer 10 (Windows). Ebenso wird die Plattform in den mobilen Varianten dieser Browser korrekt dargestellt, wobei auf Anpassungen der Ansicht an die Displaygröße und

[9] `http://jquery.com`
[10] Entsprechende Statistiken finden sich etwa auf `http://www.browser-statistik.de/`

Viewport mobiler Geräte verzichtet wurde. Ältere Versionen der genannten Browser wurden zu Testzwecken nicht herangezogen. Insbesondere kann nicht garantiert werden, dass die Oberfläche in älteren Versionen des Internet-Explorers fehlerfrei dargestellt wird[11].

6.6. Graph-Dateiformate

Um Graphen aus externen Quellen in MONET importieren bzw. berechnete Lösungen in Form von Graphen exportieren zu können, ist es notwendig, Dateiformate festzulegen oder zu spezifizieren, die gelesen bzw. geschrieben werden können. Im Folgenden werden das MONET-Graphformat – ein einfaches textbasiertes Format – und der existierende Standard GML für grafische Darstellungen von Graphen vorgestellt, die beide von MONET unterstützt werden.

6.6.1. MONET-Graphformat

Es existiert eine Vielzahl unterschiedlicher Formate für (gewichtete) Graphen. Prominente Beispiele sind etwa GML, GraphML[12] oder die dot-language[13]. Die meisten Formate setzen auf die Extensible Markup Language (XML) als Basis zur Graphenbeschreibung. Sie bieten die Möglichkeit, strukturgebende Eigenschaften sowie diverse Metainformationen mit anzugeben. Ein einheitlicher Standard existiert nicht, im Besonderen nicht für Graphen mit mehreren Kantengewichten.

Für das MONET-System wurde ein simples, textbasiertes Format entwickelt, welches sich für die Zwecke der Projektgruppe gut eignet. Sei $G = (V, E, c)$ ein kantengewichteter Graph mit Kostenfunktion $c\colon V \times V \to \mathbb{R}_+^p$, $p \in \mathbb{N}$. O.B.d.A. seien die Knoten durchnummeriert. Die ersten drei Zeilen des Formats enthalten die Anzahl Knoten

[11] Die notwendigen Anpassungen und Workarounds für eine korrekte Anzeige in den immer noch verwendeten älteren Internet Explorer Versionen stünden in keinem Verhältnis zum Nutzen. Wir gehen davon aus, dass versierte Nutzer der Plattform über einen modernen Browser verfügen, welcher Webstandards korrekt interpretiert.

[12] `http://graphml.graphdrawing.org/`

[13] `http://en.wikipedia.org/wiki/DOT_language`

$|V|$, die Anzahl Kanten $|E|$ sowie die Anzahl p der Zielfunktionen. Anschließend werden sämtliche Kanten zeilenweise aufgezählt. Dabei werden Start- und Zielknoten sowie die p Kantengewichte hintereinander notiert und durch ein Leerzeichen getrennt. Dieses einfache Format lässt sich sehr leicht parsen (siehe Kapitel 7.4) und enthält alle notwendigen Informationen. Zur grafischen Darstellung des Graphen wurde auf das GML-Format (siehe Kapitel 6.6.2) gesetzt. Für beide Formate wurden entsprechende *Exporter*-Klassen implementiert.

6.6.2. GML

Oftmals ist die grafische Darstellung von Netzwerken sinnvoll und hilfreich. So ist man etwa an der grafischen Darstellung des kürzesten Weges in einem Graphen interessiert. Dies ist nur sinnvoll machbar, wenn die Knoten des Graphen euklidische Koordinaten aufweisen und damit in der Ebene oder in einem dreidimensionalen kartesischen Koordinatensystem darstellbar sind.

Das MONET-Graphformat (siehe Kapitel 6.6.1) ist zwecks Simplizität nicht für die Speicherung von euklidischen Koordinaten ausgelegt. Es gibt eine Vielzahl von Graph-Formaten. Leider existiert jedoch kein einheitlicher Standard im Besonderen für Graphen mit mehrfach gewichteten Kanten. Für Visualisierungszwecke wurde das GML-Format (*Graph Modelling Language*) ausgewählt. Dieses erlaubt die Speicherung von Knoten mitsamt Koordinaten, (gewichteten) gerichteten Kanten und diversen weiteren strukturgebenden Metainformationen. Die Attribute der einzelnen Tags unterliegen dabei keinen strengen Vorgaben und können erweitert werden. Ein Beispiel für einen einfachen Graphen ist in Listing 6.3 gegeben.

Dadurch erreicht GML eine hohe Flexibilität. Editoren wie yEd[14] sind in der Lage, Dateien im GML-Format zu lesen und die Graphen visuell darzustellen.

[14] `http://www.yworks.com/de/products_yed_about.html`

6.7. Git

Der Quelltext eines jeden Softwareprojekts, an dem mehrere Entwickler gleichzeitig arbeiten, sollte mit Hilfe eines Versionskontrollsystems verwaltet werden. Diese bieten unter anderem eine komplette Historie der Quelltextentwicklung, ohne sehr viel Platz zu verschwenden. Dabei werden immer stückweise Änderungen am Projekt vorgenommen, die das Projekt erweitern, verbessern oder Fehler beheben. Bei den gängigen Versionskontrollsystemen wird dabei jede Änderung als sogenannter „Commit“ gespeichert. Anhand dieser Historie lässt sich unter anderem herausfinden, wer für welche Elemente des Projekts verantwortlich ist oder wann ein Fehler erstmals auftrat. Die (gängigen) Versionskontrollsysteme ermöglichen zu diesem Zweck den Stand des Projekts zu einem bestimmten Zeitpunkt wiederherzustellen. Am wichtigsten für die gemeinsame Arbeit ist aber wohl, dass man in den meisten Fällen nicht darauf achten muss, welche Änderungen ein anderer Entwickler eingebaut hat, da diese automatisch integriert werden, wenn man sein Projekt auf den neuesten Stand setzt. Falls an denselben Stellen im Quelltext gearbeitet wird, kann es jedoch passieren, dass die Änderungen sich nicht automatisch integrieren lassen und der Entwickler, der zuletzt seine Änderungen mit den anderen teilt, muss die Änderungen konfliktfrei miteinander vereinen.

Für uns standen im Wesentlichen nur zwei Versionskontrollsysteme zur Auswahl: *Subversion*[15] und Git[16]. Subversion ist ein älteres System, das sich in vielen Bereichen etabliert hat und deshalb auch noch heute an vielen Stellen verwendet wird. Git ist ein neueres Versionskontrollsystem, welches von Linus Torvalds ins Leben gerufen wurde, um den Quelltext des Linux-Kernels zu verwalten. Beide Systeme sind für die gemeinsame Entwicklung geeignet, jedoch haben wir uns einstimmig für Git entschieden. Dies lag zum einen daran, dass es einigen Entwicklern egal war, was verwendet wird, und zum anderen, dass andere sich stark für Git ausgesprochen haben. Da dieser Abschnitt kein Git-Tutorial werden soll, sei für weitere Informationen auf eine Übersicht verschiedener Git-Tutorials[17] verwiesen.

[15] `http://subversion.apache.org/`
[16] `http://git-scm.com/`
[17] `http://sixrevisions.com/resources/git-tutorials-beginners/`

Im Gegensatz zu Subversion ist Git ein dezentrales Versionskontrollsystem. Dies hat einige praktische Vorteile, die auf den ersten Blick nicht direkt erkennbar sind. Zum einen kann man beliebig viele *Branches* erzeugen, ohne dass die Historie auf dem Server landet oder im Hauptentwicklungszweig auftaucht. Im Gegensatz zu Subversion ist die Erstellung dieser Branches sehr leichtgewichtig und infolgedessen schnell. Deshalb bietet es sich an, für jede Aufgabe lokal einen *Branch* zu erzeugen und diesen in den Hauptentwicklungszweig zu integrieren. Diese Art der Entwicklung wird von erfahrenen Git-Benutzern bevorzugt und es hat sich deshalb das Mantra „Branch early and branch often!“ eingebürgert. Die Geschwindigkeit und das leichtgewichtige Branchen waren der Hauptgrund für die Verwendung von Git. Zudem ließe sich der Subversion-Workflow auch in Git abbilden mit der Besonderheit, dass man die lokalen Änderungen nicht nur „commiten“ muss, sondern auch auf den Server explizit hochladen muss. Dies führte bei einigen von uns dazu, dass zugeteilte Aufgaben zwar erledigt wurden, aber nicht im zentralen Git-Verzeichnis zu finden waren.

```
graph [
  creator "PG573 MONET"
  comment "This graph was exported by class monet.generator.GMLGraphExporter"
  directed 1
  node [
    id 0
    label "0"
    graphics [
      center [ x 0.65 y 0.09 ]
      fill "#cfcfcf"
    ]
  ]
  node [
    id 1
    label "1"
    graphics [
      center [ x 0.66 y 0.92 ]
      fill "#cfcfcf"
    ]
  ]
  node [
    id 2
    label "2"
    graphics [
      center [ x 0.88 y 0.99 ]
      fill "#cfcfcf"
    ]
  ]
  edge [
    source 1
    target 2
    label "[0.53]"
  ]
  edge [
    source 1
    target 0
    label "[0.91]"
  ]
  edge [
    source 2
    target 1
    label "[0.53]"
  ]
]
```

Abbildung 6.3.: Beispiel der Repräsentation eines gerichteten Graphen mit drei Knoten im GML-Format.

7

Graph-Framework

Die algorithmischen Problemstellungen der Projektgruppe sind auf Graphen formuliert. Da die Java-API Graphen und verwandte Datenstrukturen nicht anbietet, musste zunächst die Entscheidung getroffen werden, ob eine bereits vorhandene Bibliothek genutzt oder ein eigenes Framework entwickelt werden sollte.

7.1. Motivation einer eigenen Graph-Bibliothek

Oft verwendete, kostenlose Graph-Bibliotheken, die für die Verwendung in MONET in Frage kamen, sind JUNG [16] und JGraphT [15]. Die Bibliothek JUNG wird offenbar seit der Veröffentlichung der Version 2.0.1 im Jahr 2010 nicht mehr aktiv entwickelt. JGraphT besitzt zumindest noch eine moderat aktive Entwicklergemeinde. Schwerwiegender ist, dass JGraphT keine guten Voraussetzungen für mehrkriterielle Optimierungsprobleme bietet. So haben zum Beispiel Kanten immer ein skalares Gewicht. Die Dokumentation beider Bibliotheken besteht im Wesentlichen aus der generierten JavaDoc-Referenz. Zu JUNG existiert außerdem ein mehrseitiges Tutorial.

Für die Verwendung einer verfügbaren Bibliothek im Allgemeinen sprechen der reduzierte Entwicklungsaufwand sowie die geringere Fehlerdichte einer ausgereiften

Bibliothek. Dies trifft für die genannten Bibliotheken allerdings nicht in vollem Umfang zu, da die Fehlerdichte von JUNG aufgrund der niedrigen Minor-Version nicht einschätzbar ist und für JGraphT umfangreiche Anpassungen für mehrkriterielle Probleme notwendig gewesen wären. Für beide Graph-Bibliotheken wäre außerdem ein hoher Einarbeitungsaufwand zu erwarten gewesen, da keine Dokumentationen in wünschenswerter Form und Umfang vorhanden sind. Die Entwicklung einer eigenen Graph-Bibliothek hingegen ermöglicht die flexible Anpassung der Bibliothek auf die Anforderungen von MONET und mehrkriterieller Optimierung. Darüber hinaus ist die Planung einer eigenständigen Graph-Bibliothek eine interessante und anspruchsvolle Aufgabe. Die Entscheidung fiel daher für die Implementierung eines eigenen Frameworks.

Das Graph-Framework besteht aus drei Modulen, die in den folgenden Abschnitten detailliert beschrieben werden. *MONET Graph* implementiert alle notwendigen Datenstrukturen für Graphen, Knoten, Kanten, Gewichte und so weiter. Der *Graph Generator* bietet parametrisierbare Methoden an, um Graphen bestimmter Typen zufällig zu erzeugen. Durch den *MONET Parser* können außerdem vorhandene Graphen aus Dateien bzw. Datenströmen eingelesen werden.

7.2. MONET Graph

Der wesentliche Teil von MONET Graph wurde im ersten Sprint entwickelt. Zunächst werden die zu Beginn dieses Sprints vereinbarten Anforderungen vorgestellt. Danach erfolgt ein Überblick über das Graph-Framework in seinem derzeitigen Zustand. Im Verlauf der weiteren Sprints wurden einige Änderungen vorgenommen, da sich Teile des Frameworks bei der Entwicklung von Algorithmen als noch nicht ausgereift herausstellten. Auf diese wird im letzten Abschnitt gesondert eingegangen.

Anforderungen

Die folgenden Punkte wurden von der Algorithmengruppe lose vereinbart:

- Auch wenn im Rahmen der Projektgruppe nur mehrkriterielle MST- und Kürzeste-Wege-Probleme bearbeitet werden, soll sich das entwickelte Framework möglichst für alle mehrkriteriellen Graphenprobleme eignen.

- Es gibt keine Graph-Repräsentation (z. B. Distanzmatrix-, Adjazenzlistendarstellung), die für alle Algorithmen optimal geeignet ist. Aufgrund der Architektur von MONET, die ein vom Algorithmus unabhängiges Einlesen der Eingabedaten vorsieht, muss es daher einfach möglich sein, eine Speicherstruktur einzubinden und die Daten entsprechend einzulesen.

- Ein Eingabegraph besteht in seiner einfachsten Form aus Knoten und Kanten. Für viele Algorithmen ist es allerdings erforderlich, Zusatzinformationen wie Start- und Endknoten oder Gewichte zu kodieren. Insbesondere mehrdimensionale Gewichte bzw. Kosten sind für alle zu implementierenden Algorithmen von großer Bedeutung.

Die tatsächliche Implementierung erfolgte zusammen mit diesen Anforderungen als Leitlinie agil.

Aufbau und Philosophie

Zu Beginn der Planung wurde entschieden, in MONET Graph intensiven Gebrauch von generischer Programmierung zu machen, um redundanten Code trotz der geforderten Flexibilität möglichst zu vermeiden. Generische Typen werden zur Verbesserung der Lesbarkeit im Folgenden nur angegeben, wenn sie zum Verständnis erforderlich sind.

`Graph<N extends Node, E extends Edge, G extends Graph<N,E,G>>` ist das zentrale Interface, welches Methoden zur Verfügung stellt, die sowohl für gerichtete als auch ungerichtete Graphen sinnvoll sind. Dazu zählen zum Beispiel Methoden

wie `getNumNodes()` zur Abfrage der Knotenanzahl oder `addNode()` zum Hinzufügen von Knoten. Letztere ist eine Konsequenz aus den im vorigen Abschnitt aufgeführten Anforderungen an die Verwendung beliebiger Speicherstrukturen. Da die Speicherstruktur einer Implementierung von `Graph` nicht vorgegeben werden soll, ist auch nicht bekannt, wie die entsprechende Speicherstruktur von Knoten und Kanten aussieht. Diese werden daher zunächst durch die generischen Typen `N` und `E` repräsentiert. Da generische Typen aber nicht ohne weiteres instanziiert werden können, muss die Erstellung von Knoten und Kanten von der entsprechenden `Graph`-Implementierung vorgenommen werden. Der Typ `G` existiert aus technischen Gründen, die später erläutert werden, und hat für die Funktion des Graph-Frameworks keine Bedeutung.

Die Interfaces `DirectedGraph` und `UndirectedGraph` erweitern das Interface `Graph` um spezifische Methoden für gerichtete bzw. ungerichtete Graphen. Dazu zählen zum Beispiel `getSource(E e)` und `getIncomingEdges(N n)` für die Abfrage gerichteter Kanten. Mit `Edge`, `DirectedEdge` und `UndirectedEdge` existieren entsprechende Interfaces für Kanten. Da über ihre Struktur kein allgemeingültiges Wissen vorhanden ist, sind sie leer und haben lediglich Marker-Funktion innerhalb der Klassenhierarchie.

Die (bislang einzigen) Implementierungen dieser Interfaces sind `SimpleDirectedGraph` und `SimpleUndirectedGraph` mit ihrer abstrakten Basisklasse `SimpleAbstractGraph` für Graphen in Adjazenzlistendarstellung. Sie verwenden die Implementierungen `SimpleNode` und `SimpleEdge`, die lediglich interne Identifikationsnummern für die im Graphen-Objekt hinterlegten Adjazenzlisten enthalten.

In Graphen bzw. Elementen eines Graphen sollten beliebige Zusatzinformationen wie Gewichte kodiert werden. Diese Zusatzinformationen heißen in MONET Graph *Annotationen* und müssen ebenfalls unabhängig von der Speicherstruktur des Graphen sein. Mit der Klasse `AnnotatedGraph<N extends Node, E extends Edge, G extends Graph<N, E, G>>` kann ein beliebiges Graph-Objekt zusammen mit einem `HeterogeneousHashAnnotatorContainer` gekapselt werden. Die Klasse `HeterogeneousHashAnnotatorContainer` fasst dabei mehrere Annotationsmengen zusammen, deren Wertemenge nicht gleich sein muss (zum Beispiel Annotationen von Integer- und String-Werten). Für die Annotation von Knoten und Kanten wird eine

solche Annotationsmenge bereitgestellt durch `GraphElementHashAnnotator`. Die Klasse `GraphElementReverseHashAnnotator` ermöglicht es zusätzlich, die Urbilder von Annotationen abzufragen.

Ein besonderer Annotationstyp ist `Weight`. Es handelt sich dabei um einen Vektor von Gleitkommazahlen zur Annotation von mehrdimensionalen Gewichten bzw. Kosten. Mit `GraphElementWeightAnnotator` existiert außerdem eine Spezialisierung von `GraphElementAnnotator`, die zusätzliche Methoden zur Bestimmung der Dimension sowie zur Skalarisierung und Summierung anbietet. Eine Menge von Vektoren kann durch die Klasse `LabelSet` abgebildet und annotiert werden. Die Klasse `WeightedEdgesCalculator` dient zur Berechnung der Kosten eines Graphen mit Kantengewichten, zum Beispiel eines minimalen Spannbaums.

Im Laufe der Entwicklung entstanden einige weitere Klassen, die als Verbindung zwischen MONET Graph und den einzelnen Algorithmen nötig wurden. `SimplePriorityQueue` kapselt eine `PriorityQueue` der Java-API, ermöglicht zusätzlich aber eine einfache Aktualisierung der Prioritäten bereits eingefügter Objekte. Das entsprechende Interface mit Methode zur Aktualisierung ist `FancyPriorityQueue`. `ParetoSet<N,E,G> extends Set<G>` implementiert eine Menge von Graphen mit Pareto-optimalen Zielfunktionswerten. `ParetoFront` implementiert dieses Interface für `Weight`-Objekte. `UniobjectiveAlgorithm` und `UniobjectiveAllAlgorithm` verallgemeinern Algorithmen zur Berechnung einer bzw. aller optimalen Lösungen eines unikriteriellen Optimierungsproblems. Solche Algorithmen werden oft von Algorithmen zur Lösung von mehrkriteriellen Problemen aufgerufen. Das Interface ermöglicht es, die Implementierungen einfach auszutauschen (z. B. Prim gegen Kruskal).

MONET Graph verfolgt für die Fehlerbehandlung in Debugging- und Produktionsszenarien zwei verschiedene Strategien. In der produktiven Nutzung werden Fehler durch Rückgabewerte (wie false oder null) kommuniziert, um den aufrufenden Code nicht durch *checked exceptions* unleserlich zu machen. Eine lokale Behandlung von *Ausnahmen* ist außerdem oft nicht sinnvoll. Sie sollten im Regelfall nicht auftreten, können bei einer tatsächlichen Auslösung allerdings mannigfaltige Gründe haben, die lokal oft nicht alle erfasst, geschweige denn beseitigt werden können. Beim Debugging ist diese Art der stillen Kommunikation allerdings hinderlich. Wenn bekannt ist, dass eine Ausnahme unter bestimmten Umständen auftritt, sollte dies

einfach nachzuvollziehen sein. Wenn die entsprechende Funktion beim Ausführen der *JVM* aktiviert wird, werden daher im Fall eines Fehlers zusätzlich Assertions ausgelöst, die in Java durch *unchecked exceptions* signalisiert werden. Diese können dann (z. B. von der Entwicklungsumgebung) abgefangen und die Situation mit gängigen Debugging-Werkzeugen analysiert werden.

Aufgetretene Probleme

Während der Implementierung von MONET Graph traten einige Probleme auf, die hauptsächlich durch die Grenzen der JVM entstanden und zu intensiven Entwicklungen in einzelnen Bereichen führten.

Ein wesentliches Problem für die Typsicherheit des Graph-Frameworks bereitete die *Type Erasure* von Java. Daher wurde zum Beispiel dem Interface `Graph<N extends Node, E extends Edge, G extends Graph <N, E, G>>` der selbstreflexive Typ `G` hinzugefügt. Obwohl `N` und `E` durch `Object` ersetzt werden, können zum Beispiel *Casts* und *Unboxings* typsicher durchgeführt werden.

Für die Definition von Gewichten durch die Klasse `Weight` bereitete das Interface `Number` Probleme. Zwar erben geboxte Primitive wie `Integer` und `Double` von diesem Interface, es ermöglicht allerdings keine arithmetischen Operationen. Das Unboxing von gekapselten Primitiven benötigt außerdem Zeit, was sich insbesondere bei aufwendig Rechnungen äußert. Dadurch ist es unmöglich, generische Methoden zum Rechnen mit Gewichten bereitzustellen. `Weight` ist daher nicht generisch und verwendet zur Speicherung der Werte den primitiven Typ `double`.

Standardmäßig überprüfen alle Methoden von Klassen des Graph-Frameworks ihre Parameter auf Gültigkeit. Soll zum Beispiel eine Kante in einen Graphen eingefügt werden, wird überprüft, ob die inzidenten Knoten existieren. Bei der Verwendung des Graph-Frameworks in der Praxis hat sich gezeigt, dass diese Überprüfungen einen wesentlichen Anteil an der Laufzeit der entsprechenden Methoden halten. Daher wurde für `SimpleAbstractGraph` und abgeleitete Klassen die Möglichkeit implementiert, diese Tests instanz-spezifisch zu deaktivieren. Für hinreichend getestete Algorithmen besteht so die Möglichkeit, die Laufzeit zu verbessern. Methodenaufrufe

mit ungültigen Parametern gefährden bei deaktivierter Überprüfung allerdings die strukturelle Integrität der entsprechenden Graph-Objekte.

7.3. Graph-Generator

Zu Testzwecken ist das Generieren von Zufallsnetzwerken hilfreich, da die Erzeugung von Graphen von Hand mühsam und langwierig ist. Daher wurde für das MONET-System ein Verfahren zur Erzeugung zufälliger Netzwerke implementiert. Dabei sind die Anzahl Knoten, die Anzahl der Zielfunktionen (d. h. die Anzahl der Gewichte pro Kante) sowie die Dichte des Graphen parametrisierbar und erlauben eine hohe Flexibilität.

Das Verfahren arbeitet in mehreren Schritten. Zunächst werden n Punkte in der euklidischen Ebene und im Einheitsquadrat $[0,1]^2$ zufällig gleichverteilt platziert. Diese Punkte entsprechen den euklidischen Koordinaten der n Knoten des erzeugten Graphen. Im Anschluss werden unter Nutzung einer geeigneten Metrik die paarweisen Distanzen zwischen den Knoten berechnet. Als Standardmetrik wurde die euklidische Metrik gewählt. Jedoch ist die Manhattan-Block-Metrik oder jede andere durch L_p-Normen induzierte Metrik ebenso denkbar und abhängig vom Anwendungsgebiet. Die erzeugten n Knoten ergeben mit den paarweisen Distanzen einen vollständigen Graphen $G = K_n$. Als nächstes wird mit dem Algorithmus von Prim ein minimaler Spannbaum $G' = MST(G)$ von G berechnet, welcher als Ausgangspunkt für die nachfolgenden Schritte dient. Die Spannbaumeigenschaft stellt sicher, dass der Graph zusammenhängend ist (eine in der Regel wünschenswerte Eigenschaft eines Netzwerks). Im nächsten Schritt wird G' um Kanten erweitert, deren Kantengewicht unter der Schranke $1.6\alpha\sqrt{n}$ liegt. Letztere lässt sich über einen reellwertigen Parameter $\alpha \in \mathbb{R}_+$ zur Steuerung der Dichte des Graphen regulieren. Niedrige Werte führen zu dünn besetzten, hohe zu dicht besetzten Graphen. Abschließend werden die Kanten um zufällige weitere Gewichte ergänzt, bis die Anzahl der gewünschten Zielfunktionen erreicht wird.

Die generierten Graphen können in verschiedene Formate exportiert werden. Zwecks Visualisierung bietet sich das GML-Format an (siehe Kapitel 6.6.2). Für die inter-

ne Verarbeitung in MONET und Anwendung von Algorithmen ist das einfache, textbasierte MONET-Graphenformat (siehe Kapitel 6.6.1) gedacht.

7.4. MONET Parser

Eine Software zum experimentellen Vergleich von Algorithmen auf Netzwerken muss die Möglichkeit bieten, Netzwerke aus Dateien oder Dateiströmen einzulesen. Zu diesem Zweck wurde das Parser-Interface `GraphParser` entwickelt. Klassen, die dieses Interface implementieren, müssen nur eine Methode `parse(String, Job)` anbieten. Diese erwartet neben dem Pfad zur Datei ein `Job`-Objekt mit Informationen über den auszuführenden Job. Intern wird das MONET-Format für die persistente Speicherung von Graphen benutzt. Zu diesem Zweck wurde die Klasse `MONETParser` verwendet, welche das Dateiformat einliest und einen annotierten Graphen zurückgibt.

8

Algorithmen

Zu Beginn der PG teilten wir uns in drei Gruppen auf. Zwei dieser Gruppen hatten den Auftrag, sich näher mit Algorithmen zur Lösung des mehrkriteriellen Kürzeste-Wege- bzw. Minimale-Spannbäume-Problems zu beschäftigen. Nach einer breit gefächerten Recherche wurde jeweils ein in sich stimmiges Portfolio von Algorithmen zusammengestellt und mit der Implementierung begonnen. Im zweiten Semester haben wir dies fortgesetzt und weitere Algorithmen fertig gestellt.

In diesem Abschnitt werden die Beweggründe innerhalb des Auswahlprozesses beschrieben und die ausgewählten Algorithmen näher erläutert. Insbesondere wird auch auf die Realisierung der implementierten Algorithmen eingegangen. Außerdem wird das ebenfalls erstellte und für die implementierten evolutionären Algorithmen verwendete EA-Framework diskutiert, das eine einfache Realisierung beliebiger evolutionärer Algorithmen ermöglicht.

8.1. Kürzeste-Wege-Problem

Eines der von uns betrachteten Probleme ist das Kürzeste-Wege-Problem. Es ist ein kombinatorisches Optimierungsproblem aus dem Bereich der Graphentheorie. Bei der hier betrachteten Variante ist es das Ziel, in einem kantengewichteten Graphen von

einem ausgezeichneten Startknoten den kürzesten Pfad zu einem festen Endknoten zu finden. Da das Thema der Projektgruppe mehrkriterielle Optimierung ist, sind die Kantengewichte hier allerdings Gewichtsvektoren. In diesem Fall muss es nicht immer einen Pfad geben, der bezüglich aller Kriterien der kürzeste ist, so dass stattdessen die Menge der Pareto-optimalen Pfade gesucht wird.

Dieses Kapitel beschreibt die Algorithmen, die im Verlauf der PG hinsichlich des Kürzeste-Wege-Problems implementiert wurden.

8.1.1. Label-Correcting

Aus den bekannten Algorithmen für das mehrkriterielle Kürzeste-Wege-Problem haben wir den Label-Correcting-Algorithmus als denjenigen ausgewählt, welcher als erstes implementiert werden sollte. Ausschlaggebend für diese Entscheidung war, dass der erste Algorithmus schnell und einfach zu implementieren sein sollte, um später implementierte Algorithmen im direkten Vergleich unkompliziert auf ihre Korrektheit überprüfen zu können.

Der Algorithmus wurde zunächst in der simpelsten Variante implementiert. Die von Skriver und Andersen [22] vorgestellten Änderungen sind in dieser Variante noch nicht enthalten und die `Merge`-Operation ist ein einfacher, paarweiser Vergleich aller Labels. `Merge` auf diese Weise zu realisieren hat zwar den Nachteil einer vergleichsweise hohen Laufzeit, dafür lässt sich der Algorithmus so aber theoretisch auf Eingabegraphen mit beliebig-dimensionalen Kantengewichten ausführen. Der nächste Schritt war die Implementierung eines komplexeren `Merge`-Verfahrens. In dieser Variante, die mit der von Brumbaugh-Smith und Shier [3] vorgestellten weitestgehend identisch ist, lässt sich der Algorithmus nur bei zweikriteriellen Eingaben verwenden, da in `Merge` Eigenschaften von Labelmengen ausgenutzt werden, die bei mehr als zwei Kriterien nicht mehr vorliegen. Anschließend wurden die Verbesserungen von Skriver und Andersen in den Algorithmus eingefügt.

Für den Algorithmus wurde zudem ein von Kung et al. [11] vorgestelltes Verfahren zur Bestimmung von dominierten Vektoren in einer Menge von Vektoren beliebiger Dimension implementiert, damit der Algorithmus auch auf Graphen ausgeführt werden

kann, die durch mehr als zwei Kriterien gewichtet sind. Die genauere Funktionsweise wird in Kapitel C beschrieben. Experimente mit unseren Implementierungen ergaben jedoch, dass dieses Verfahren sich nicht für die Anwendung im

8.1.2. Mehrkriterieller A*-Algorithmus

Ein weiterer Algorithmus für das Kürzeste-Wege-Problem, den wir implementiert haben, ist der NAMOA*-Algorithmus von Mandow und de la Cruz [12]. NAMOA* steht für „A New Approach to Multiobjective A* Search“. Im einkriteriellen Fall löst der A*-Algorithmus das Kürzeste-Wege-Problem, indem er eine Heuristik $h(x)$ verwendet, die für jeden Knoten die verbleibende Distanz zu einem festen Endknoten schätzt. Der Algorithmus verwaltet eine Liste von Knoten, von denen aus Wege verlängert werden können. Für jeden dieser Knoten wird ein Wert $f(x) = g(x) + h(x)$ gespeichert, wobei $g(x)$ die Länge des kürzesten bekannten Weges zu x ist. Es wird dann immer zuerst der Weg erweitert, dessen Endknoten den niedrigsten Wert in f hat.

Im mehrkriteriellen Fall ergeben sich einige Unterschiede. Die Werte der Heuristik sind Vektoren. Demzufolge kann ein Knoten mehrere undominierte Heuristik-Vektoren haben, so dass die Funktion h die Knoten auf eine Menge von Vektoren abbildet. Außerdem kann es mehrere Pareto-optimale Wege zu einem Knoten geben. Es genügt also nicht mehr eine Liste von Knoten zu verwalten von denen aus Wege erweitert werden können. Vielmehr werden die Wege selber in einer Liste gehalten und jeder von diesen kann mehr als einen Wert von f haben, da der Endknoten des Weges mehrere Werte in h haben kann. In diesem Fall ist es nicht mehr möglich, einen zu erweiternden Weg nach dem kleinsten Wert in f auszuwählen, da es ein eindeutiges Kleinstes nicht mehr gibt.

Für einen effizienten Ablauf benötigt der Algorithmus einige Datenstrukturen. Für jeden Knoten v werden zwei Listen mit offenen beziehungsweise geschlossenen Labels $G_{op}(v)$ und $G_{cl}(v)$ gehalten. Jedes Label entspricht einem bereits gefundenen Weg zu einem Knoten und enthält die Länge dieses Weges. Labels sind offen, wenn der zugehörige Weg noch nicht erweitert wurde, ansonsten sind sie geschlossen. Außerdem

wird eine Liste *OPEN* von noch zu erweiternden Wegen verwendet. Jeder dieser Wege wird durch ein Tripel $(v, g, f(v, g))$ repräsentiert. Hierbei ist v der Endknoten des Weges, g die Länge des Weges und $f(v, g)$ die geschätzte Länge des Weges, wenn er bis zum Endknoten fortgesetzt wird, welche sich aus der Länge g des bisher bestimmten Weges und den Werten der Heuristikfunktion für v zusammensetzt. Jeder Eintrag in *OPEN* entspricht einem offenen Label. Des Weiteren ist für das fehlerfreie Funktionieren des Algorithmus Voraussetzung, dass die Heuristik h die Entfernung zum Zielknoten nie überschätzt.

Solange *OPEN* nicht leer ist, entfernt der Algorithmus ein Element $(v, g, f(v, g))$ aus *OPEN* und verschiebt das entsprechende Label von $G_{op}(v)$ nach $G_{cl}(v)$. Anschließend erweitert er, falls nötig, den entsprechenden Weg. Hierbei sind mehrere Fälle möglich. Ist v der Endknoten muss der Weg nicht weitergeführt werden und es ist ein Lösungskandidat gefunden. In diesem Fall können alle Elemente aus *OPEN* entfernt werden, deren Werte von f durch die Länge des gefundenen Weges dominiert werden. Andererseits werden alle von v ausgehenden Kanten betrachtet. Für eine dieser Kanten (v, u) wird zunächst überprüft, ob $g + c((v, u))$ von einem Label von u dominiert wird. Ist dies nicht der Fall, werden alle Label von u, die von $g + c((v, u))$ dominiert werden, entfernt und ein neues Element $(u, g', f(u, g'))$ erstellt. Aus $f(u, g')$ werden diejenigen Vektoren entfernt, die von der Länge eines bekannten Weges von Start- zu Endknoten dominiert werden. Sollte $f(u, g')$ nicht leer sein, wird $(u, g', f(u, g'))$ in *OPEN* und g' in $G_{op}(u)$ eingefügt.

Ist *OPEN* leer, so befinden sich die Vektoren mit den Zielfunktionswerten der gefundenen Wege in G_{cl} des Endknotens. Die Wege selber lassen sich dann mit einer Backtracking-Methode aus den Labels bestimmen.

8.2. Minimale-Spannbaum-Problem

Ein weiteres kombinatorisches Optimierungsproblem, das im Rahmen der Projektgruppe untersucht wird, ist das MST-Problem. Ähnlich wie beim Kürzeste-Wege-Problem sind die Kanten hier mit Gewichtsvektoren versehen. Gesucht ist die Menge aller Spannbäume, die Pareto-optimal sind.

Im Folgenden werden Implementierungsdetails für die in der Projektgruppe für das MST-Problem implementierten Algorithmen angegeben.

8.2.1. Two-Phase-Methode für das bikriterielle Spannbaumproblem

Für das bikriterielle MST-Problem wurden zwei Two-Phase-Algorithmen implementiert. Beide verwenden eine gemeinsame erste Phase und unterscheiden sich damit nur in der Realisierung der zweiten Phase. Zunächst wurde der in [24] vorgeschlagene k-Best-Ansatz implementiert (siehe Abschnitt 3.3.3). Im zweiten Semester wurde dieser um den in [23] erläuterten Branch-and-Bound-Ansatz ergänzt (siehe Abschnitt 8.2.4).

8.2.2. Erste Phase

Um das einkriterielle MST-Problem zu lösen, wurden die Algorithmen von Prim und Kruskal implementiert. Die Implementierung von Prim nutzt die Prioritätswarteschlange `SimplePriorityQueue`, die Implementierung von Kruskal die Union-Find-Datenstruktur `TreeUnionFind`. Die Kruskal-Implementierung hat sich dabei im Laufzeitvergleich gegen Prim durchgesetzt und wurde in den Experimenten verwendet.

Ferner werden die Extrempunkte gefunden, indem jeweils der minimale Spannbaum bezüglich eines Kriteriums berechnet wird. Hierbei kann der Fall eintreten, dass die gefundene Lösungen durch eine andere Lösung dominiert wird. Diese wird dann allerdings im Verlauf der ersten Phase aufgefunden.

8.2.3. Zweite Phase: k-Best-Ansatz

Wie in Abschnitt 3.3.3 beschrieben, wird in [24] ein k-Best-Algorithmus für das Auffinden der verbleibenden Pareto-optimalen Lösungen verwendet.

In der Literatur finden sich eine Reihe von k-Best-Algorithmen für das einkriterielle MST-Problem, beispielsweise [7] mit einer Laufzeit von $\mathcal{O}(km\alpha(m,n) + m\log m)$ und [10] mit einer Laufzeit von $\mathcal{O}(km + \min\{n^2 m m \log\log n\})$. Hierbei bezeichnet α die inverse Ackermann-Funktion.

Sowohl unsere, als auch die Implementierung aus [24] verwendet den Algorithmus von Gabow [7] aufgrund seiner einfachen und effizienten Implementierbarkeit. Die Idee hinter dem Algorithmus von Gabow besteht darin, mit einem minimalen Spannbaum zu beginnen und bei jeder Iteration ein Paar von Kanten (e, f) zu finden, so dass e im aktuellen Spannbaum enthalten ist und f nicht, sowie gleichzeitig die Differenz $c(f) - c(e) \geq 0$ über alle solche Kantenpaare minimal ist. Ferner muss gelten, dass durch Austauschen der Kante e mit f weiterhin ein Spannbaum vorliegt.

8.2.4. Zweite Phase: Branch-and-Bound

Der in [23] vorgestellte und von uns implementierte Branch-and-Bound-Algorithmus partitioniert die Kantenmenge E an jedem Knoten im Branch-Baum in drei Mengen E_0, E_+ und E_-: Dabei enthält E_+ Kanten, die im Spannbaum enthalten sein müssen und E_- Kanten, die nicht im Spannbaum enthalten sein dürfen. E_0 enthält alle restlichen Kanten, also $E \setminus (E_+ \cup E_-)$. In jedem Knoten des Branch-Baums werden die extrem effizienten Spannbäume des Eingabegraphen unter einer Partitionierung (E_0, E_+, E_-) als Nebenbedingung bestimmt.

Die initiale Partitionierung der Kanten wird durch ein Preprocessing bestimmt. Zunächst gilt $E_0 := E$ und damit $E_+ := E_- := \emptyset$. Solange ein Kreis C in $E_0 \cup E_+$ mit einer Kante $e \in C$ im Graph existiert, deren Kantenkosten von den Kosten aller anderen Kanten aus $E_0 \cap C \setminus \{e\}$ dominiert werden, wird e aus E_0 entfernt und zu E_- hinzugefügt. Dies ist zulässig, da die Kosten jedes Spannbaums, der e enthält, wie folgt verbessert werden können: Der Spannbaum zerfällt durch das Entfernen von e in zwei Zusammenhangskomponenten. Da C ein Kreis ist, existiert eine andere Kante $e' \in C$, die diese Zusammenhangskomponenten ebenfalls verbindet und deren Kosten die Kosten von e dominieren.

Im Branch-Schritt wird eine Kante aus E_0 für die Verzweigung gewählt. Die Auswahl

erfolgt heuristisch: Es wird die Kante e gewählt, deren Kosten bezüglich aller Kanten in E_0 in einer Komponente minimal sind, d. h. $e := \arg\min_{e' \in E_0} \min\{c_1(e'), c_2(e')\}$. Es werden zwei neue Probleminstanzen (E'_0, E'_+, E'_-) und (E''_0, E''_+, E''_-) aus (E_0, E_+, E_-) konstruiert, in die der Algorithmus verzweigen kann:

$$\begin{aligned} E'_0 &:= E_0 \backslash \{e\}, & E'_+ &:= E_+ \cup \{e\}, & E'_- &:= E_- \\ E''_0 &:= E_0 \backslash \{e\}, & E''_+ &:= E_+, & E''_- &:= E_- \cup \{e\} \ . \end{aligned}$$

Um eine untere Schranke für das Bounding zu erhalten, werden zunächst die beiden Extrempunkte unter Berücksichtigung der Mengen E_+ und E_- berechnet. Anschließend werden jeweils zwei benachbarte Spannbaume aus der unteren Schranke des Vater-Knotens des Branch-Baums betrachtet und mit der aus Abschnitt 2.2.2 bekannten ersten Phase des Two-Phase-Ansatzes unter Berücksichtigung der Mengen E_+ und E_- expandiert.

Beim Bounding werden zunächst die lokalen Nadirpunkte für zwei aufeinanderfolgende Lösungen der unteren Schranken berechnet. Wenn alle diese Nadirpunkte unterhalb der aktuellen Paretofront liegen, kann der aktuelle Branch-Knoten verworfen werden, da die bisher gefundenen nicht-dominierten Lösungen alle Lösungen dominieren, die in dem Unterbaum des Knotens noch zu finden wären.

8.3. Evolutionäre Algorithmen

Dieser Abschnitt beschreibt die Konzepte und Funktionsweise der im Rahmen der Projektgruppe implementierten evolutionären Algorithmen.

8.3.1. Prüfer-EA

Prüfer-EA: Der Prüfer-EA [28] ist nach der Prüfer-Nummer benannt, die als Genotyp für die einzelnen Individuen verwendet wird. Auf eine solche Kodierung lassen sich klassische Mutations- und Rekombinationsverfahren auf einfache Weise übertragen.

Bei dem Prüfer-EA, dem ersten bekannten evolutionären Algorithmus zur Lösung mehrkriterieller MST-Probleme, handelt es sich um einen leicht zu implementierenden Algorithmus mit vielen Parametern, die sich zum Testen eignen, was die Motivation für die Implementierung im Rahmen der Projektgruppe darstellt.

Ablauf: Zuerst wird eine Initialpopulation erzeugt. Die Größe und der 0-stellige Suchoperator sind dabei über Parameter vorgegeben. In jeder Generation werden Nachkommen durch Rekombination aus zufällig gewählten Eltern (mit Wiederholung) erzeugt und anschließend mutiert. Nach dem *Genotyp-Phänotyp-Mapping (GPM)*, also der Umwandlung des Genotyps in einen Spannbaum, folgt die Fitness-Evaluation und anschließend die Auswahl guter Individuen für die nächste Generation.

Nach Zhou und Gen [28] werden *Uniform Crossover* und *Uniform Mutation* als Suchoperationen verwendet. Zur Evaluation der Fitnesswerte wird zuerst jeder sukzessiven Pareto-Front der aktuellen Lösungsmenge eine geringere Dummy-Fitness zugewiesen. Anschließend wird diese Fitness durch die Anzahl ähnlicher Individuen in der Population geteilt (Sharing-Konzept). Als Selektionsstrategie wird die klassische Roulette-Wheel-Selektion verwendet. Die genannten Verfahren entsprechen der *Strategie II* des in [28] vorgestellten Verfahrens. Über die verschiedenen Parameter des EAs ist es jedoch auch möglich, andere Verfahren für den Algorithmus zu testen.

Tabelle 8.1 gibt einen Überblick über die möglichen Parameter. Obwohl der Prüfer-EA nach Zhou und Gen die Prüfer-Nummer als Genotyp verwendet, kann der grundlegende Algorithmus theoretisch auch mit einer anderen Kodierung ausgeführt werden. Aus diesem Grund sind `encodingName`, `mutatorName` und `recombinatorName` als Parameter vorhanden.

8.3.2. SMS-EMOA

SMS-EMOA: Der SMS-EMOA (S-Metrik-Selektion Evolutionärer Mehrziel-Optimier-Algorithmus) verwendet das von einem Individuum dominierte Hypervolumen als Grundlage für dessen Fitness. Die Motivation für die Implementierung dieses Algorithmus ist, dass die S-Metrik allgemein als vielversprechender Ansatz zur

Parameter	Bedeutung
popSize	Größe der Population (Integer)
offspringSize	Anzahl zu erzeugender Nachkommen pro Generation (Integer)
mappingName	Name des GPM
creatorName	0-stellige Suchoperation zur Erzeugung der Initialpopulation
mutatorName	Name des Mutations-Operators
recombinatorName	Name des Rekombinations-Operators
selectorName	Name des Selektions-Algorithmus
evaluatorName	Name der Evaluationsstrategie
terminatorName	Name der Strategie für Abbruchbedingungen

Tabelle 8.1.: Parameter für die Prüfer-EA-Implementierung.

Evaluation der Fitness angesehen wird und in empirischen Studien überwiegend gute Ergebnisse liefert [2].

Ablauf: In jeder Generation wird mit Hilfe von nicht fest vorgegebenen Suchoperationen ein neues Individuum erzeugt und in die Population aufgenommen. Existiert in der neuen Population mindestens ein dominiertes Individuum, so wird das Individuum mit der höchsten Dominanzzahl aus der Population entfernt. Sonst wird das Individuum mit dem kleinsten S-Metrik-Beitrag entfernt. Nach Überprüfung der Abbruchbedingung folgt die nächste Generation. Die Initialpopulation wird durch den gegebenen *Creator* erzeugt.

Berechnung des Hypervolumens: Die S-Metrik wird mit einer Implementierung des *HSO-Algorithmus (Hypervolume by Slicing Objectives)* [27, 1] berechnet. Eine denkbare Verbesserung wäre z. B. die Verwendung eines inkrementellen Algorithmus zur Berechnung der S-Metrik, um die Performanz zu verbessern.

Tabelle 8.2 gibt einen Überblick über die möglichen Parameter. Der Algorithmus ist unabhängig von der gewählten Kodierung.

Parameter	Bedeutung
popSize	Größe der Population (Integer)
mappingName	Name des GPM
creatorName	0-stellige Suchoperation zur Erzeugung der Initialpopulation
mutatorName	Name des Mutations-Operators
recombinatorName	Name des Rekombinations-Operators
terminatorName	Name der Strategie für Abbruchbedingungen

Tabelle 8.2.: Parameter für die SMS-EMOA-Implementierung.

8.3.3. SPEA2

SPEA2: Der SPEA2 (Strength Pareto EA 2) ist ein EA, der bei der Fitnessberechnung eines Individuums auf die Stärke derjenigen Individuen zurückgreift, die dieses dominieren [29]. Die Stärke eines Individuums i ist dabei definiert als die Anzahl der Individuen, die von i dominiert werden: $S(i) = |\{j \mid j \in P_t + \bar{P}_t \wedge i \text{ dominiert } j\}|$ mit Population P_t und Archiv $\bar{P}_t$ in Iteration t (s. u.).

Ablauf: SPEA2 verwendet neben der Population P noch ein initial leeres Archiv $\bar{P}$ in welchem eine Auswahl an nicht-dominierten Individuen gehalten wird. Falls nicht genug nicht-dominierte Individuen gefunden wurden, kann das Archiv zusätzlich dominierte Individuen enthalten. Der Ablauf des Algorithmus gliedert sich in die folgenden Schritte [13]:

1. Initialisiere P_0 mit N Individuen und setze $\bar{P}_0 = \emptyset,\ t = 0$.

2. Fitnessberechnung für jedes Individuum i:

   - Berechne einen vorläufigen Fitnesswert $R(i) = \sum_{j \in P_t + \bar{P}_t, j \prec i} S(j)$.

   - Berechne Distanz von Individuum i zu allen anderen Individuen j und sortiere aufsteigend.

   - Mit der k-ten Distanz σ_i^k aus der Liste wird die Dichte $D(i) = \frac{1}{\sigma_i^k + 2}$ mit $k = \lfloor \sqrt{N + \bar{N}} \rfloor$ berechnet, wobei $\bar{N} = |\bar{P}|$.

- Die Fitness ergibt sich zu $F(i) = R(i) + D(i)$.

3. *Environmental Selection*: Kopiere alle nicht-dominierten Individuen aus P_t und $\bar{P}$ nach $\bar{P}_{t+1}$.

   - Wenn $|\bar{P}_{t+1}| > \bar{N}$: entferne Individuen, die sich am nächsten zu anderen Individuen befinden. Die Distanzen wurden bereits in Schritt 2 berechnet.

   - Wenn $|\bar{P}_{t+1}| < \bar{N}$: fülle $\bar{P}_{t+1}$ mit den gemäß ihrer Fitness besten dominierten Individuen aus P_t und $\bar{P}$.

4. Terminiere, falls Abbruchkriterium erfüllt ist (die Ergebnisindividuen sind dann alle nicht-dominierten Individuen aus $\bar{P}_{t+1}$).

5. *Mating Selection*: Wende den Selektionsalgorithmus auf $\bar{P}_{t+1}$ an.

6. Führe Mutation und Rekombination auf ausgewählten Individuen druch.

7. Wiederhole ab Schritt 2 mit $t = t + 1$.

Tabelle 8.3 gibt einen Überblick über die möglichen Parameter. Der Algorithmus ist unabhängig von der gewählten Kodierung.

Parameter	**Bedeutung**
popSize	Größe der Population (Integer)
archiveSize	Größe des zu verwendenden Archivs
mappingName	Name des GPM
creatorName	0-stellige Suchoperation zur Erzeugung der Initialpopulation
mutatorName	Name des Mutations-Operators
recombinatorName	Name des Rekombinations-Operators
terminatorName	Name der Strategie für Abbruchbedingungen
selectorName	Name des Selektions-Algorithmus
evaluatorName	Name der Evaluationsstrategie

Tabelle 8.3.: Parameter für die SPEA2-Implementierung.

8.4. EA-Framework

Ziel des *EA-Frameworks* ist die Bereitstellung eines allgemeinen Frameworks, welches eine schnelle Implementierung neuer evolutionärer Algorithmen zur heuristischen Findung der Pareto-optimalen Lösungen mehrkriterieller MST- und Kürzeste-Wege-Probleme ermöglicht. Die Berechnung anderer Optimierungsprobleme ist zwar im Rahmen der Projektgruppe nicht vorgesehen, sollte jedoch ohne weitere Probleme mit dem Framework möglich sein.

Die Motivation für die Implementierung von evolutionären Verfahren ist es, einen Vergleich heuristischer Herangehensweisen mit exakten Algorithmen zu erhalten. Evolutionäre Algorithmen bieten zwar keine feste Approximationsgüte, versprechen aber in vielen Fällen eine schnellere Laufzeit als exakte Algorithmen. Es sei erwähnt, dass bereits Frameworks für evolutionäre Algorithmen existieren, wie z. B. *jMetal*[1], deren Anpassung an die Anforderungen der Projektgruppe jedoch so umfangreich wären, dass beschlossen wurde, ein eigenes Framework zu implementieren.

Im Folgenden wird zuerst in Abschnitt 8.4.1 auf die einzelnen Bestandteile des Frameworks eingegangen und anschließend in Abschnitt 8.4.2 der grobe Ablauf eines beliebigen EAs erklärt, wobei außerdem Aspekte des *Loggings* und *Messens* von Ergebnissen behandelt werden. Darauf folgt eine Anleitung zum Einbinden eines neuen Algorithmus in Abschnitt 8.4.3.

8.4.1. Komponenten

In diesem Abschnitt werden die einzelnen Komponenten des EA-Frameworks genannt und jeweils kurz beschrieben.

[1] `http://jmetal.sourceforge.net/` (letzter Zugriff: 17.10.2013)

Allgemeine Komponenten

EaRandom: Innerhalb evolutionärer Algorithmen spielen randomisierte Funktionen eine wichtige Rolle. Um trotzdem die Reproduzierbarkeit der erzielten Ergebnisse zu garantieren, wird durch statische Methoden dieser Klasse eine Möglichkeit bereitgestellt, einen zentralen Seed-Wert für den ausgeführten Algorithmus zu verwalten. Zur Reproduktion von Ergebnissen kann dieser Seed-Wert gespeichert, und bei einem späteren Durchlauf wiederverwendet werden.

Nameable: Aufgrund zahlreicher potenziell austauschbarer (Teil-)Algorithmen, die der Benutzer möglichst durch lesbare, von Klassennamen abstrahierende Parameternamen auswählen können soll, spezifiziert dieses Interface Methoden zur Benennung von Algorithmen. Zusätzlich benötigen verschiedene Algorithmen meist auch verschiedene Parameter, sodass eine Methode zur dynamischen Konfiguration unter Verwendung einer *String-Object-HashMap* spezifiziert wird. Auf diese Weise können Algorithmen, die an dieser Stelle neben konkreten evolutionären Algorithmen auch einzelne Suchoperatoren, Selektions-Algorithmen, usw. bezeichnen, einheitlich erzeugt und konfiguriert werden.

Functions: In der `Functions`-Klasse werden statische Funktionen gesammelt, die an verschiedenen Stellen bzw. in verschiedenen Algorithmen benötigt werden. Dazu gehört z. B. das Parsen von als Strings gegebenen Parametern, Funktionen, die das Logging betreffen, und Funktionen zur Verwaltung verfügbarer Algorithmen. Allgemeine Funktionen auf Graphen, wie z. B. die Erzeugung eines zufälligen Spannbaums oder eines zufälligen Weges zwischen zwei Knoten, sind auch vorhanden.

Kodierung und Individuen

Ein Individuum (Klasse `Individual`) setzt sich aus allen Informationen, die innerhalb eines klassischen EAs benötigt werden, zusammen: Individuen besitzen einen Genotyp (Klasse `Genotype`) und einen Phänotyp (Klasse `Phenotype`). Zur Erzeugung

des Phänotyps enthält ein Individuum zusätzlich einen Verweis auf die zu verwendende Genotyp-Phänotyp-Abbildung (Klasse `Phenotype-Mapping`). Des Weiteren besitzt jedes Individuum einen Fitnesswert und kann zur Berechnung von Dominanzbeziehungen auf die Zielfunktionswerte des Phänotyps zurückgreifen. Neben der Fitness stehen weitere numerische Variablen, z. B. für den Rang des Individuums, zur Verfügung, die bei Bedarf verwendet werden können. Als statische Methoden bereitgestellte Funktionalitäten der `Individual`-Klasse umfassen unter anderem die Extraktion nicht-dominierter Individuen aus einer Population und die Berechnung der Ränge von Individuen.

Zur Vermeidung unnötiger Redundanz werden Informationen, die von jedem Genotyp benötigt werden, in ein zentrales `Encoding`-Objekt ausgelagert. Beispiele für solche Informationen sind die Länge und die gültigen Symbole der Genotypen. `Encoding`-Objekte können gemäß des `Nameable`-Interfaces benannt und konfiguriert werden. Die Verwendung eines von konkreten Genotypen unabhängigen `Encoding`-Objekts bringt zusätzliche Vorteile bei der Erzeugung neuer Genotypen durch 0-stellige Suchoperatoren. Das zu benutzende `Encoding`-Objekt wird durch den verwendeten `Creator` bestimmt und ist daher kein zusätzlicher Parameter für den Benutzer.

Ist eine Genotyp-Phänotyp-Abbildung gegeben, so ist dadurch implizit der zu verwendende Phänotyp spezifiziert. Analog ist durch gegebene Suchoperationen der Genotyp spezifiziert.

Evolutionäre Algorithmen

Konkrete evolutionäre Algorithmen werden im EA-Framework von der abstrakten Oberklasse `EvolutionaryAlgorithm` abgeleitet. Nach der Konfiguration gemäß des `Nameable`-Interfaces kann die parameterlose `execute`-Methode ausgeführt werden, welche den Kern der Ausführung repräsentiert.

Operatoren und Strategien

Evolutionäre Algorithmen setzen sich häufig aus kleineren Operatoren bzw. Strategien, z. B. zur Mutation und Rekombination von Individuen, zusammen, die dynamisch ausgetauscht werden können. Im Folgenden werden die Oberklassen der wichtigsten Operatoren, die alle zur einfachen Konfiguration das `Nameable`-Interface implementieren, beschrieben. Die Verwendung dieser Klassen innerhalb eines konkreten evolutionären Algorithmus ist optional und wird über die entsprechende Konfiguration gesteuert.

Alle verwendeten Suchoperatoren werden von der abstrakten Oberklasse `SearchOperator` abgeleitet. Für die wichtigsten Fälle existieren speziellere Klassen: `Creator` für 0-stellige, `Mutator` für 1-stellige und `Recombinator` für 2-stellige Suchoperatoren.

Die Evaluation wird von einem Objekt des Typs `Evaluator` vorgenommen, welches allen Individuen einer gegebenen Population Fitnesswerte zuweist. Es sei darauf hingewiesen, dass die Fitness eines Individuums durch andere Individuen der Population beeinflusst werden kann und daher die gesamte Population betrachtet werden muss. Die Selektion von Individuen wird von Objekten des Typs `Selector` vorgenommen. Die Abbruchbedingungen des Algorithmus werden von einem `Termination`-Objekt verwaltet.

Abbildung 8.1 zeigt ein verkürztes UML-Klassendiagramm der wichtigsten von `Operator` abgeleiteten Klassen.

8.4.2. Ablauf der Ausführung

Die `execute`-Methode der `Evolution`-Klasse ist die im Rahmen des MONET-Programms die Eintrittsmethode des EA-Frameworks. Nach allgemeinen Operationen wie der Zuweisung und/oder Speicherung des Seed-Wertes wird ein `EAConfigurator`-Objekt erzeugt. Die von außen übergebenen Parameter werden von diesem Objekt interpretiert und, falls nötig, werden `Operator`-Objekte erzeugt. Schließlich wird

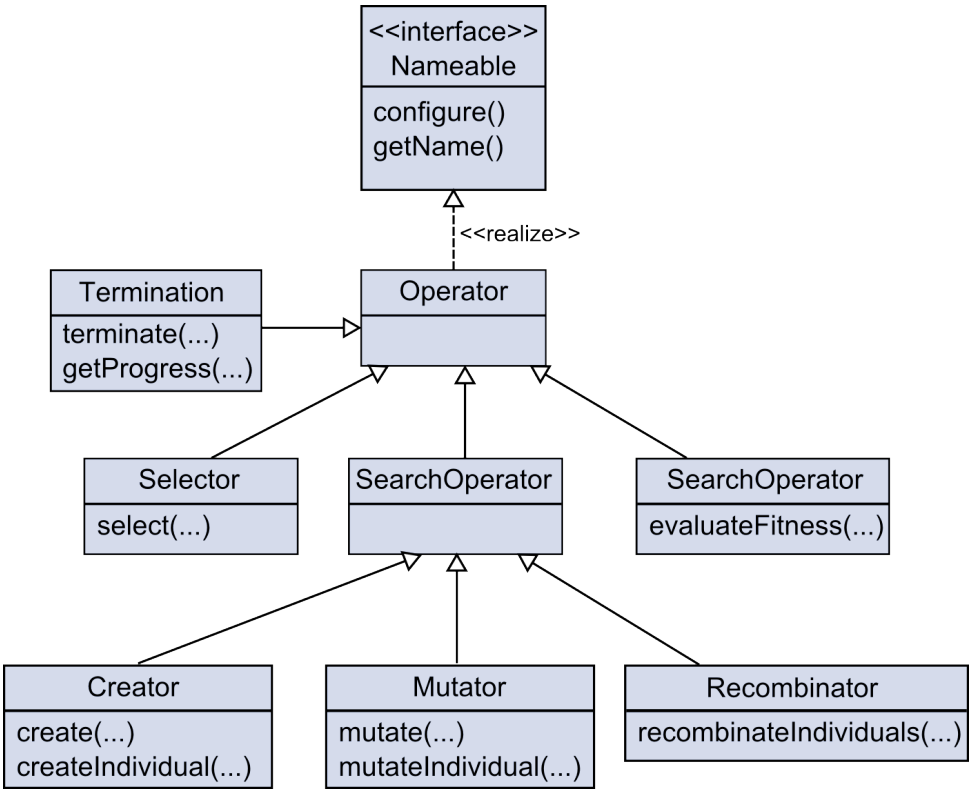

Abbildung 8.1.: Überblick über die wichtigsten Operatoren, die evolutionären Algorithmen innerhalb des Frameworks bei der Ausführung zur Verfügung stehen.

ein den Parametern entsprechender evolutionärer Algorithmus angelegt und nach erfolgreicher Konfiguration ausgeführt. Nach der Ausführung werden die Ergebnisse durch einen von außen gegebenen `Meter` an das Hauptprogramm zurückgegeben, um anschließend weiterverarbeitet bzw. gespeichert zu werden. Während der Ausführung können über den von der `Functions`-Klasse bereitgestellten Logger Debugging-Hinweise, Warnungen oder Fehler protokolliert werden.

Allgemeine Parameter: Über den Parameter `eaName` wird der Name des auszuführenden Algorithmus spezifiziert. Algorithmen können mittels der Parameter `creatorName`, `mutatorName`, `recombinatorName`, `selectorName`, `evaluatorName`, `terminatorName` sowie `mappingName` spezifiziert werden. Für diese Parameternamen wird automatisch versucht, einen entsprechenden Algorithmus zu finden, zu instantiieren, zu konfigurieren und anschließend als neuen Parameter `creator`, `mutator`, usw. in die Parameterliste aufzunehmen. Über den Parameter `seed` kann ein Random-Seed

gesetzt werden, um Ergebnisse zu reproduzieren. Ist der Parameter nicht gesetzt, so wird ein neuer Seed erzeugt und gespeichert. Für eine vollständige Auflistung aller verfügbaren Parameter des EA-Frameworks siehe Abschnitt 9.2.1.

Automatisch erzeugte Parameter: Der `EaConfigurator` erzeugt zu Beginn der Ausführung automatisch die folgenden Parameter, die von allen (Teil-)Algorithmen wie normale benutzerdefinierte Parameter verwendet werden können:

- `numNodes`: Anzahl der Knoten im Graphen.
- `numEdges`: Anzahl der Kanten im Graphen.
- `isGraphComplete`: Gibt an, ob der Graph vollständig ist.
- `numObjectives`: Dimension des Problems (Anzahl der Zielfunktionen).
- `problemGraph`: Annotierter Eingabe-Graph.
- `idNodeMap`: `HashMap` mit Zuordnung der Form *Integer* → *Knoten*; Zuweisungen von Knoten zu IDs.
- `nodeIdMap`: `HashMap` mit Zuordnung der Form *Knoten* → *Integer*; jeder Knoten hat eine eindeutige ID.
- `fitnessMaximization`: Dieser Parameter wird automatisch vom verwendeten `Evaluator` gesetzt und bestimmt, ob die Fitness maximiert oder minimiert werden muss. Der Parameter wird bei der Selektion benötigt.
- `startNode`: Erster Knoten (Start) des Weges bei kürzeste Wege Problemen. Der Wert wird aus einem mit dem Eingabe-Graphen gegebenen `Annotator` extrahiert.
- `endNode`: Letzter Knoten (Ziel) des Weges bei kürzeste Wege Problemen. Der Wert wird aus einem mit dem Eingabe-Graphen gegebenen `Annotator` extrahiert.

Die Verwendung der automatisch erzeugten Parameter ist optional, vermeidet jedoch Redundanz, da die Parameter zentral gespeichert werden und nicht in jedem Algorithmus erneut berechnet werden müssen.

8.4.3. Implementierung neuer Algorithmen

Um einen neuen evolutionären Algorithmus zu implementieren, muss eine neue Unterklasse von `EvolutionaryAlgorithm` erzeugt werden, die eine entsprechende `execute`-Methode implementiert. Werden für den Algorithmus Parameter benötigt, so ist die `configure`-Methode zu überschreiben, die eine *Map* aus allen zur Verfügung stehenden Parametern enthält. Damit der Algorithmus durch den `EAConfigurator` gefunden werden kann, muss er in dessen `initialize`-Methode als Operator registriert werden.

Weitere Selektionsverfahren, Evaluationsverfahren, usw. werden analog als Unterklassen von `Selector`, `Evaluator`, usw. implementiert und anschließend als Operatoren registriert.

Damit die Parameter dem Benutzer auch in der GUI zur Verfügung stehen, muss eine entsprechende XML-Datei angepasst werden (manuell oder durch ein dem EA-Framework beiliegendes Python-Skript).

8.4.4. Konkrete Operatoren

Prüfer-Kodierung: Die Prüfer-Kodierung wurde mit den bereits beschriebenen Operatoren implementiert. Die Operatoren wurden so angepasst, dass sie nicht nur auf vollständigen, sondern auch auf dichten Graphen fehlerfrei, jedoch langsamer, funktionieren. Im Wesentlichen ist dies dadurch erfolgt, dass Operatoren, die eine ungültige Lösung erzeugt haben, erneut ausgeführt werden.

Direkte MST-Kodierung: Neben der bereits beschriebenen Prüfer-Kodierung wurde auch die direkte Kodierung von Spannbäumen als Kantenliste mit entsprechenden Operatoren nach [17] implementiert. Die im folgenden vorgestellten Operatoren unterstützen einen Parameter, der den maximalen Knotengrad der erzeugten Lösungen angibt. Der `Creator` erzeugt zufällige Spannbäume mit einer Variante des Kruskal-Algorithmus, wobei Kanten jedoch nicht nach ihrem Gewicht, sondern in zufälliger Reihenfolge iteriert werden. Kanten, die zu einem zu großen Knotengrad führen würden, werden übersprungen. Das `PhenotypeMapping` ist in diesem Fall die Identität, da der Genotyp bereits aus einer einfachen Kantenliste besteht, die genau dem Phänotyp entspricht. Bei der Erzeugung des Phänotyps müssen somit nur noch die Zielfunktionswerte bestimmt werden, die bei der Evaluation benötigt werden. Bei der Mutation eines Genotyps wird eine zufällige Kante (i, j) des Graphen gewählt, die noch nicht Teil des Genotyps ist. Anschließend wird in dem bisherigen Genotyp bzw. Spannbaum der eindeutige Weg von i nach j gesucht, der mit der neuen Kante (i, j) einen Kreis bildet. Nachdem eine Kante des Pfades aus dem Genotyp entfernt wurde, wird die Kante (i, j) hinzugefügt, so dass das Ergebnis wieder ein Spannbaum ist. Bei der Rekombination zweier Spannbäume wird ein neuer Baum erzeugt, der in einem ersten Schritt genau die Kanten enthält, die beide Eltern enthalten. Anschließend werden zufällig gewählte Kanten hinzugefügt, die nur in einem der Eltern vorhanden sind. Da hierbei Kanten übersprungen werden, die den benutzerspezifizierten Knotengrad verletzten würden, ist das bisherige Ergebnis nicht immer ein Spannbaum. In einem letzten Schritt werden daher nach und nach Zusammenhangskomponenten durch Kanten verbunden, bis die Lösung ein Spannbaum ist.

Direkte SP-Kodierung: Die hier vorgestellte Implementierung einer direkten Shortest-Path-Kodierung funktioniert sowohl für gerichtete als auch für ungerichtete Graphen. Genotypen entsprechen bei dieser Kodierung geordneten Kantenmengen, die einen Pfad vom Start- bis zum Zielknoten bilden. Der `Creator` erzeugt einen zufälligen Pfad, indem, ausgehend vom Startknoten, immer ein zufällig gewählter noch nicht besuchter Nachbarknoten besucht wird. Existiert kein noch nicht besuchter Nachbarknoten, so wird der Pfad mittels Backtracking einen Knoten zurückverfolgt. Wie bei der direkten MST-Kodierung entspricht das `PhenotypeMapping` hier der Identität. Der Mutationsoperator wählt einen zufälligen Knoten aus dem gegebenen

Genotyp und ersetzt den Teilpfad von diesem Knoten bis zum Zielknoten durch einen neuen Pfad. Für die Generierung des neuen Pfades wird der gleiche Algorithmus verwendet, der bereits für den `Creator` beschrieben wurde. Bei der Rekombination wird ein zufälliger Knoten des einen Elter gewählt und überprüft, ob der Knoten auch im anderen Elter vorhanden ist. Ist dies der Fall, so werden die Teilpfade der beiden Eltern zum und vom gewählten Knoten miteinander kombiniert. Dafür gibt es zwei Möglichkeiten; in der Implementierung wird der Teilpfad vom Startknoten bis zum gewählten Zufallsknoten aus dem ersten Elter und der restliche Teilpfad aus dem zweiten Elter gewählt.

Evaluation: Zur Evaluation von Spannbäumen wird das bereits geschilderte Verfahren des Prüfer-EAs verwendet (auch bei anderer Kodierung). Wege werden entsprechend dem Verfahren von SPEA2 bewertet.

Selektion: Zwei gängige Selektionsverfahren wurden implementiert. Zum einen die einfache Roulette-Wheel-Selektion und zum anderen die Tournament-Selektion. Bei der Roulette-Wheel-Selektion ist die Wahrscheinlichkeit dafür, in die nächste Generation aufgenommen zu werden, für jedes Individuum proportional zu dessen Fitnesswert (soll die Fitness minimiert werden, so ist ein entsprechender Kehrwert zu bilden). Die Tournament-Selektion führt pro Individuum, das in die nächste Generation übernommen werden soll, genau ein Turnier durch. Pro Turnier wird eine vom Benutzer spezifizierte Anzahl an Individuen ausgewählt, von denen das mit der besten Fitness gewinnt und in die nächste Generation übernommen wird.

Terminierung: Als `Terminator` wurde ein Operator implementiert, welcher die Abbruchbedingungen Zeit, Anzahl an Generationen und Güte erreichter Fitness ermöglicht.

9

Handbuch

In diesem Abschnitt beschreiben wir die praktische Benutzung der Plattform MONET sowie die nötigen Schritte, um an MONET entwickeln zu können. Dabei zeigen wir in Abschnitt 9.1, wie die grafische Benutzeroberfläche zum Erstellen und Auswerten von Experimenten benutzt werden kann. In Abschnitt 9.3 beschreiben wir, wie man eine Entwicklungsumgebung aufsetzt, um dann die verschiedenen Komponenten zu erstellen. In Abschnitt 9.6 zeigen wir, wie man Algorithmen passend erstellt und in Abschnitt 9.5 wird in die Implementierung von Parsern eingeführt.

MONET steht zum größten Teil unter der GNU Affero General Public License (Version 3, 19. November 2007, kurz AGPL)[1]. Die lizensierbaren Interfaces, die bei der Entwicklung von Algorithmen, Parsern oder Graphen verwendet werden, stehen unter der GNU Lesser Public License (Version 3, 29. Juni 2007, kurz lGPL)[2]. Dies erlaubt es Algorithmen zu entwickeln und auf der Plattform zu testen, die nicht selbst unter der GNU Affero General Public License stehen. MONET kann auf GitHub[3] heruntergeladen werden.

[1] `http://www.gnu.org/licenses/agpl-3.0.html`
[2] `http://www.gnu.org/licenses/lgpl.html`
[3] `https://github.com/saep/MONET`

9.1. Bedienung der Weboberfläche

Die Weboberfläche dient als Schnittstelle zu der entwickelten Plattform. Mit ihr können Experimente angelegt, angemeldete Worker verwaltet und Bundles mit Algorithmen sowie Graphinstanzen hochgeladen werden. Die Startseite der Anwendung ist in Abbildung 9.1 abgebildet.

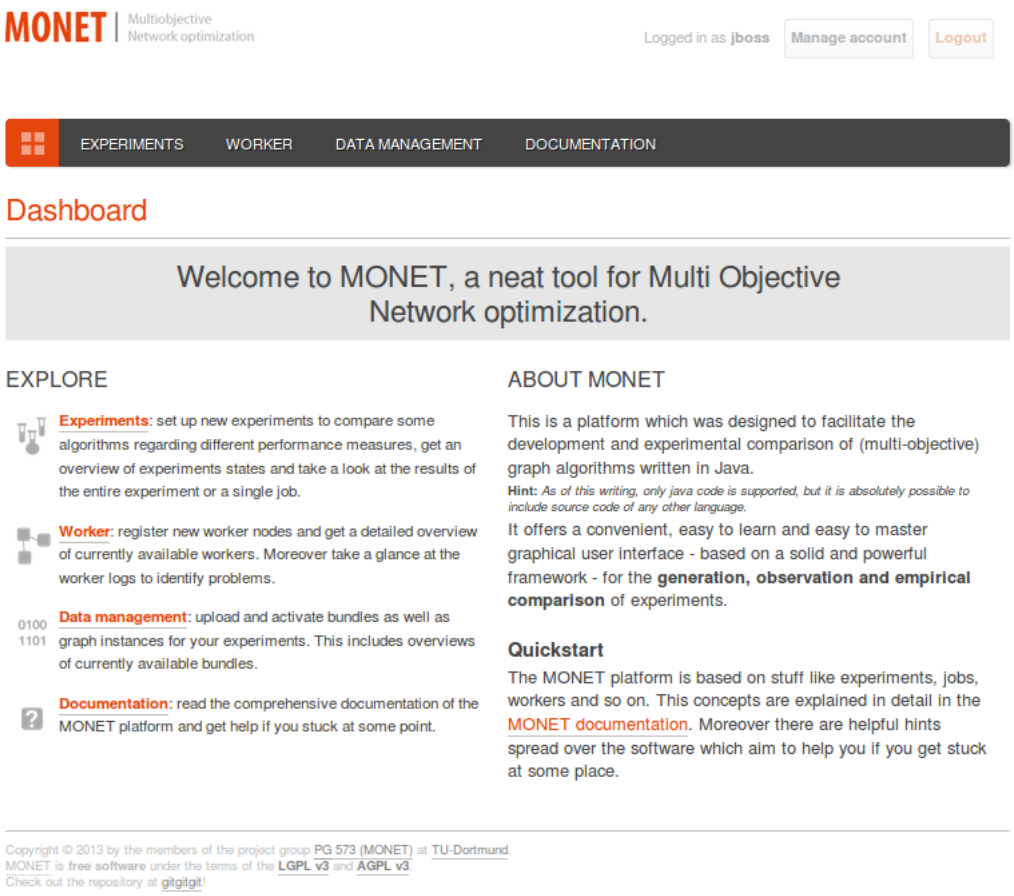

Abbildung 9.1.: Startseite von MONET

Experimente

Die Experimente dienen dazu, die Jobs zu gruppieren. Beim Anlegen eines Experiments wird der auszuführende Algorithmus ausgesucht und parametrisiert. Ein Job

enthält den Graphen, auf dem der Algorithmus ausgeführt wird, und einen Parser für den ausgesuchten Graphen.

Anzeige von Experimenten

Unter dem Menüpunkt Experiments (siehe Abbildung 9.1) gibt es die zwei Punkte *List of experiments* und *Add new experiment*.

Wenn man direkt auf Experiments oder List of experiments klickt, kommt man zu der Liste von Experimenten, welche aktuell auf dem Control Server vorliegen. Ein Experiment der Liste enthält den Namen, die Beschreibung, seine Priorität, die Anzahl der zugewiesenen Jobs, die zwei Buttons *Cancel* und *Repeat* und den aktuellen Zustand (*State*).

Die Liste kann nach den Zuständen gefiltert werden; dafür gibt es eine Leiste direkt über den eingetragenen Experimenten. Für die Suche nach einem einzelnen Experiment steht ein Eingabefeld zur Verfügung. Damit werden die Namen der eingetragenen Experimente durchsucht; dabei ist auf Groß- und Kleinschreibung zu achten.

Mit der roten Schaltfläche Cancel kann ein laufendes Experiment abgebrochen werden. Nachdem dieser Button gedrückt wurde, werden alle laufenden Jobs des Experiments beendet und das Experiment wird in den State *cancelled* gesetzt. Sollte das Experiment bereits durchgelaufen sein, kann es nicht mehr abgebrochen werden und die Schaltfläche wird grau eingefärbt, siehe hierfür Abbildung 9.2.

Um eine Kopie eines Experiments der Liste zu erzeugen, kann die Schaltfläche Repeat genutzt werden. Durch einen Klick darauf wird eine Kopie erzeugt, wobei der Name um das Präfix *Repetition i of* erweitert wird. Das i ist hierbei eine Nummer, die angibt, um die wievielte Kopie es sich handelt.

Anlegen von Experimenten

Über die Schaltfläche *Add experiment* oder über den Unterpunkt *Add new experiment* von Experiments können neue Experimente angelegt werden. Auf der folgenden Webseite werden die Metadaten des Experiments wie Name und Beschreibung eingegeben

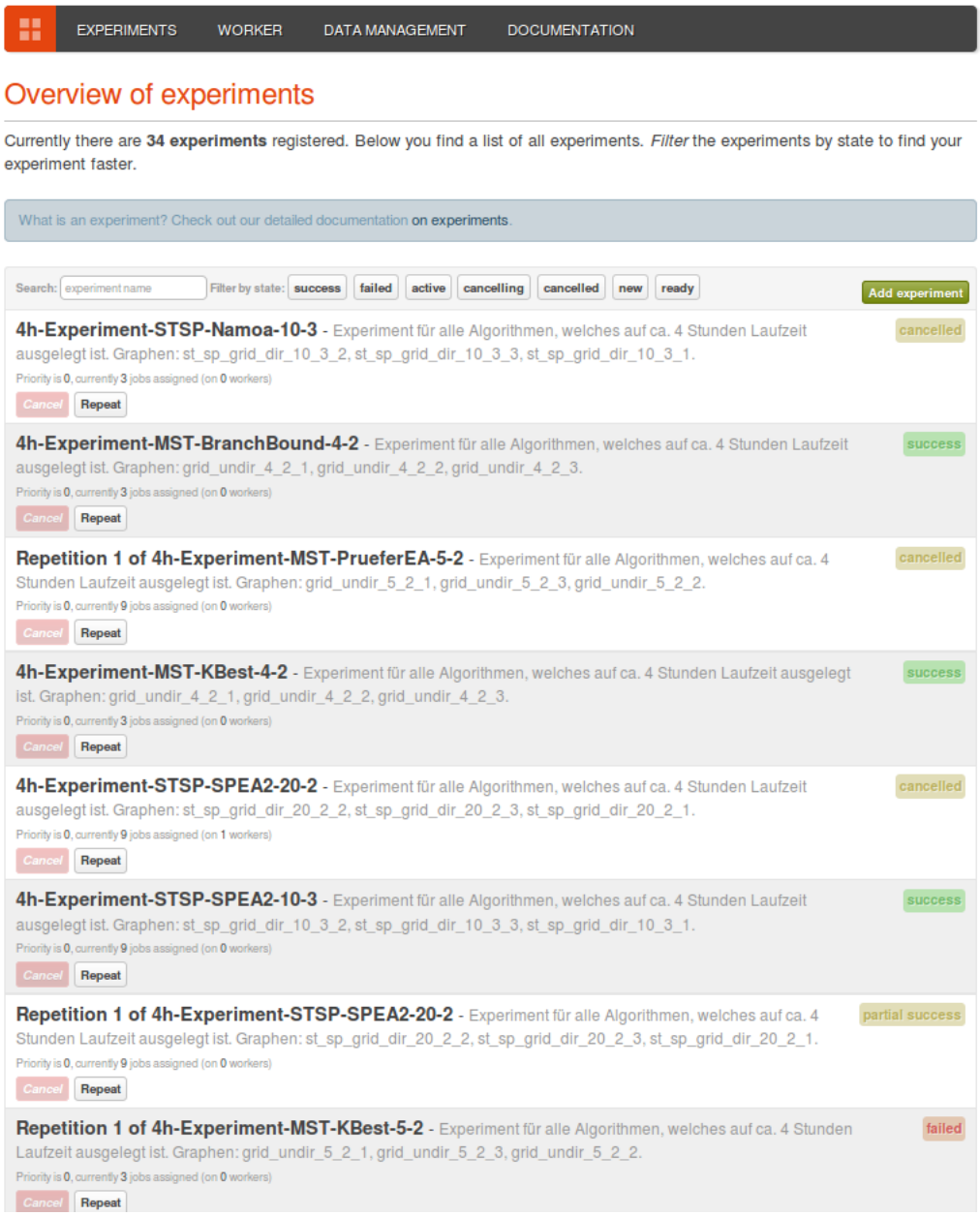

Abbildung 9.2.: Liste von angelegten Experimenten

und der auszuführende Algorithmus ausgewählt. Die Auswahl des Algorithmus erfolgt über die Auswahlliste bei *Algorithm*. Dabei ist zu beachten, dass pro Experiment nur ein Algorithmus ausgewählt werden kann. Nachdem die Wahl eines Algorithmus getroffen wurde, muss dieser parametrisiert werden. Dafür werden auf derselben Seite die Parameter angezeigt. In den meisten Fällen liegen Default-Werte vor. Zur Erklärung der Bedeutung der einzelnen Algorithmen liegen kurze beschreibende

Texte vor.

Bei manchen Algorithmen können die Parameter verschachtelt vorliegen, das heißt, dass ein Parameter erst zur Auswahl angezeigt wird, wenn der darüber liegende Parameter ausgewählt wurde. Wurde der Algorithmus *monet_ea*#0.0.2.*SNAPSHOT* ausgewählt, so liegen die drei Parameter *Preset*, *autoGenerateSeed* und *makeGraphComplete* vor. Wenn unter preset der Punkt *Custom* ausgewählt wird, erscheint am Ende der Liste der Parameter *eaName*. In Abhängigkeit von der Wahl, die bei eaName getroffen wird, erscheinen entsprechend weitere Parameter, siehe hierfür Abbildung 9.3 . Wenn die Parametrisierung des Algorithmus erfolgt ist, wird das Experiment angelegt, sobald auf *Create experiment* gedrückt wird.

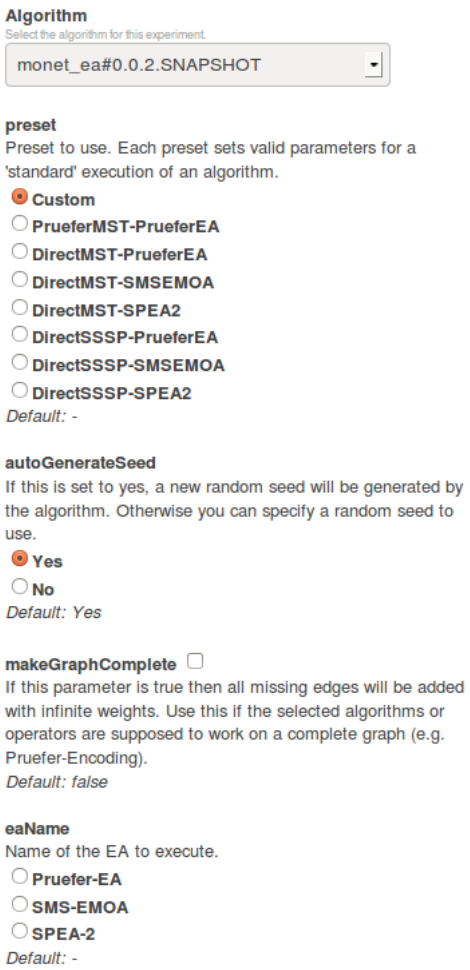

Abbildung 9.3.: Auswahl der Parameter

Darauf folgt die Detailansicht des angelegten Experiments. Unter *Experiment Details* werden die Metadaten und die Parameter des Experiments angezeigt. Bei *Job Summary* steht zu Anfang in allen Feldern der Wert 0, da jetzt erst die Jobs zum Experiment hinzugefügt werden. Der Algorithmus, der in den Jobs ausgeführt wird, ist beim Anlegen des Experiments festgelegt und parametrisiert worden. Während des Anlegens eines Jobs wird unter der Überschrift *Add a job* eine Graphinstanz beim Punkt *Graph file* und ein Parser bei *Graph parser* ausgesucht. Bei der Entwicklung eines Parsers ist es ebenfalls möglich, Parameter vom Benutzer auswählen zu lassen. Diese werden, falls welche einzustellen sind, nach der Wahl unterhalb der Auswahlliste angezeigt. Hier ist es ebenfalls möglich, die Parameter verschachtelt zu definieren. Falls es gewünscht ist, können Kopien des neuen Jobs direkt erzeugt werden. Dafür ist im Eingabefeld *Copies* ein positiver, ganzzahliger Wert einzutragen.

Falls es gewünscht wird, dass spezielle Worker das Experiment ausführen, sind diese unter *Assigned workers* auszuwählen. Damit wird ein Job des Experiments nur ausgeführt, wenn einer der ausgesuchten Worker zur Verfügung steht.

Nachdem ein Experiment erzeugt, die Jobs hinzugefügt, und, falls gewünscht, die Worker zur Bearbeitung ausgewählt wurden, wird das Experiment mit *Start this experiment* an den Scheduler übergeben. Die Jobs werden von der Plattform den Workern zugeteilt und bearbeitet. In Abbildung 9.4 ist ein Experiment abgebildet, welches einen Job zugewiesen bekommen hat und ausgeführt werden kann.

Ergebnisse

Nach der Ausführung der Experimente kann man sich die gelieferte Pareto-Front der einzelnen Jobs oder die Laufzeit aller Jobs anzeigen lassen. Zur Anzeige der Laufzeit aller Jobs muss man sich in der Detailansicht des Experiments befinden; dort gibt es oben den grünen Knopf *Show results*. Die Laufzeiten der Jobs des Experiments werden als Boxplot dargestellt; dabei ist ein roter Punkt die Laufzeit eines Jobs. Die Laufzeit wird in Minuten angegeben.

Für einen erfolgreich durchgeführten Job, d. h. einen Job, der sich im Zustand *success* befindet, ist es möglich, sich die Pareto-Front anzeigen zu lassen. Die Anzeige ist für jedes Problem möglich, erfolgt aber nur zweidimensional. Um sich die Pareto-Front

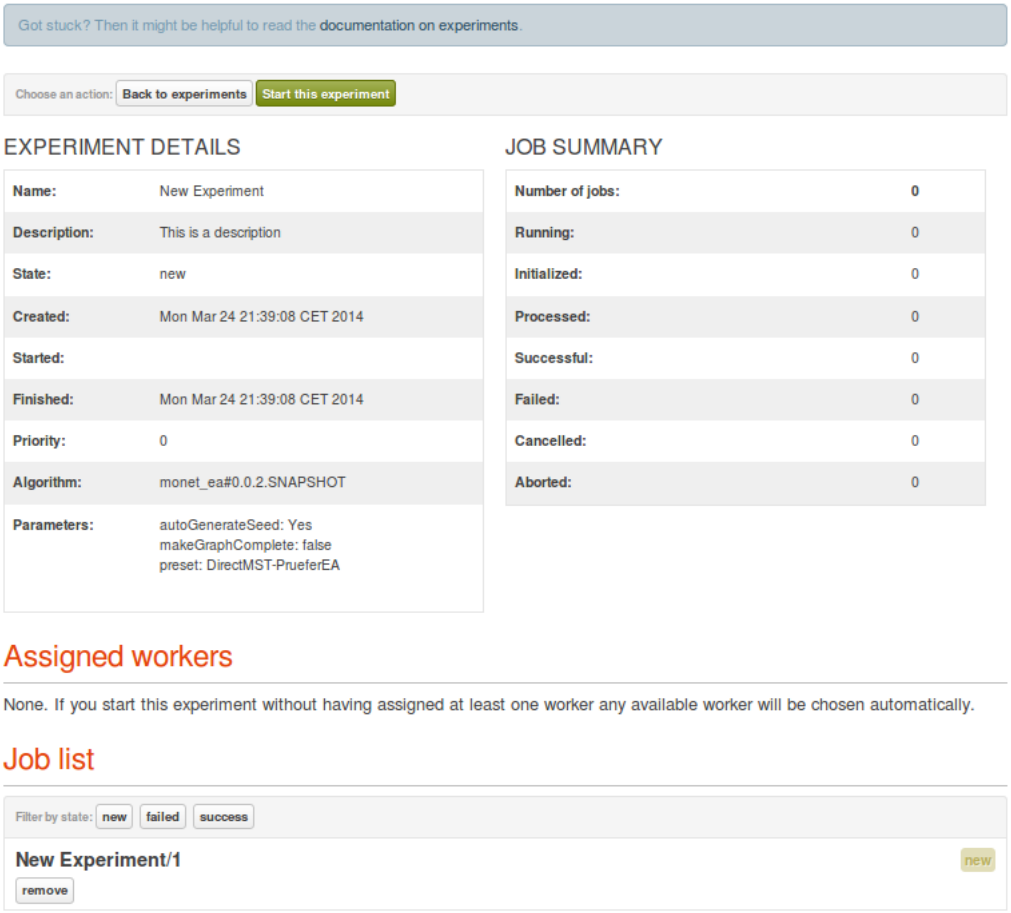

Abbildung 9.4.: Experiment mit einen zugewiesenen Job

anzeigen zu lassen, muss auf die Schaltfläche *Show results* bei einem einzelnen Job geklickt werden. Diese kann angezeigt werden, wenn der Job erfolgreich durchlief und das Problem zweidimensional ist, andernfalls erfolgt keine Ausgabe.

Die Plots können als *PNG*-Datei gespeichert werden. Dafür muss mit der Maus über die Grafik gefahren werden, wodurch oben rechts, innerhalb der Grafik, Menüpunkte erscheinen. Wenn man auf den dritten Punkt *Save as png* klickt, öffnet sich ein neues Fenster, in dem der Plot angezeigt wird, als *PNG*-Datei vorliegt und gespeichert werden kann.

Worker

Die Worker dienen dazu, die Jobs auszuführen. In der Liste der vorhandenen Worker werden die ausgelesenen Hardwareinformationen und der aktuelle Status des Workers angezeigt. In der Liste ist es möglich, über den Knopf *Terminate* einen Worker zu beenden und vom Control Server abzumelden. Mit der Schaltfläche *End Job* wird der aktuell auf dem Worker ausgeführte Job beendet.

In der Detailansicht ist zu sehen, welcher Algorithmus bzw. welches Bundle aktuell bearbeitet wird. Des Weiteren gibt es eine Liste von bisherigen Log-Nachrichten des Workers. Diese umfassen bearbeitete Algorithmen bzw. Bundles und ggf. vorliegende *Exceptions* und Verbindungsprobleme zwischen Worker und Control Server.

Data Management

Damit jeder Worker über die aktuellen Bundles verfügt, werden diese auf dem Control Server hochgeladen. Beim Start eines Workers kann dieser so einfach bestimmen, ob er ein oder mehrere Bundles aktualisieren oder nachladen muss. Zur Verwaltung dieser Punkte dient das *Data Management*.

Hochladen von Bundles

Für die Ausführung eines Jobs werden ein Algorithmus und ein Parser gebraucht. Beide müssen als Bundle auf dem Controlserver vorliegen. Bei *Bundle upload* kann über die Schaltfläche Durchsuchen das hinzuzufügende Bundle ausgesucht werden. Es öffnet sich ein Dialog zur Auswahl einer Datei, die hinzuzufügen ist. Ein Bundle ist eine spezielle *Jar*-Datei. Sollte keine Jar-Datei ausgewählt werden, wird beim Versuch des Hochladens eine Fehlermeldung ausgegeben. Im anderen Fall, also wenn eine Jar-Datei ausgewählt wurde, wird nach dem Hochladen eine Erfolgsmeldung ausgegeben.

Hochladen von Graphinstanzen

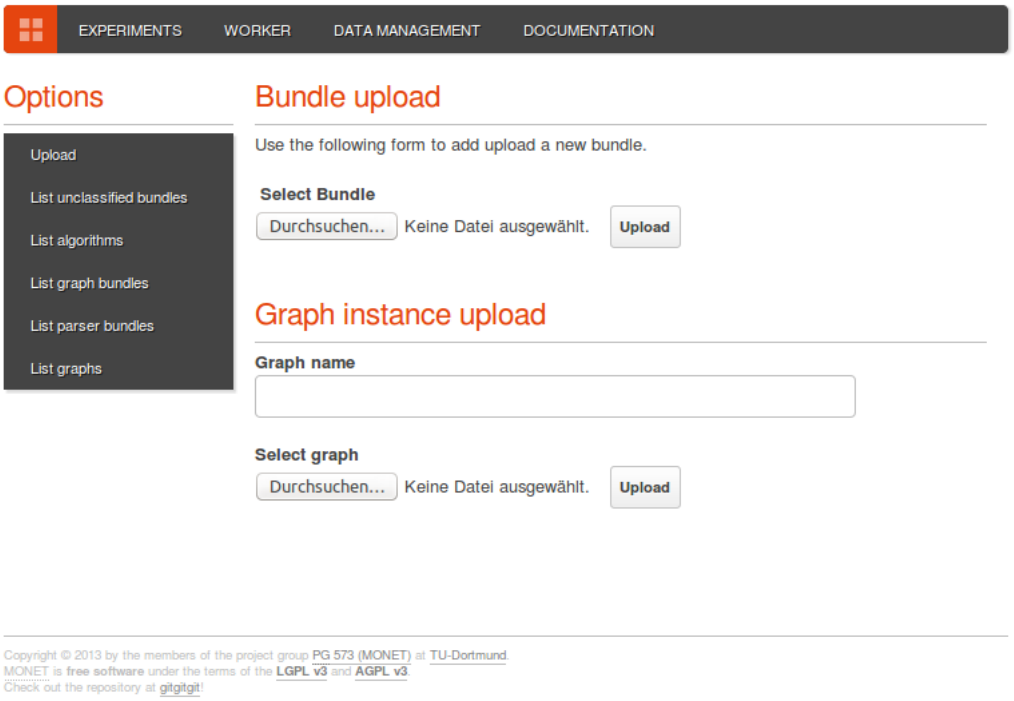

Abbildung 9.5.: Data Management von der MONET-Anwendung

Ein Graph liegt im Moment als Textdatei vor, aber das Format kann frei gewählt werden, wenn es einen entsprechenden Parser gibt. Im Eingabefeld *Graph name* muss ein Name angegeben sein. Unter diesem Namen wird der Graph abgespeichert. Die Datei wird unter *Select graph* ausgewählt und über das Klicken auf *Upload* hochgeladen. Sollte kein Name eingegeben worden sein, erfolgt eine Fehlermeldung, sonst eine Erfolgsmeldung. In den Menüpunkten unter *Options* können die Daten angezeigt werden, die im Moment in der Datenbank des Control Servers vorliegen.

Dokumentation

Unter dem Punkt *Documentation* kann die aktuell vorliegende Dokumentation der Plattform eingesehen und benötigte Informationen nachgelesen werden.

9.2. Existierende Algorithmen

In erster Linie ist MONET darauf ausgelegt, verschiedene Algorithmen hochzuladen und anschließend Experimente zu starten, um diese Algorithmen mithilfe der mitgelieferten Werkzeuge zu vergleichen. Im Rahmen der Projektgruppe MONET haben wir bereits einige Algorithmen für das Kürzeste-Wege- sowie das Minimaler-Spannbaum-Problem implementiert. Diese können ausgeführt und untereinander sowie mit weiteren Implementierungen verglichen werden. Im Folgenden werden die bereits implementierten Algorithmen und insbesondere ihre Parameter kurz vorgestellt und erläutert.

9.2.1. EA-Framework

Das EA-Framework umfasst die Algorithmen SPEA2, Prüfer-EA und SMS-EMOA. Die folgende Liste gibt eine vollständige Übersicht über alle dem Benutzer zur Verfügung stehenden Parameter des EA-Frameworks. Manche der Parameter werden nur unter bestimmten Bedingungen angezeigt, die hier nicht näher spezifiziert werden sollen (z. B. wird der Parameter `numTournamentRounds` sinnvollerweise nur dann angezeigt, wenn als `selectorName` der Tournament-Selection-Algorithmus gewählt ist).

- `preset`: Zu verwendendes Preset. Jedes Preset bestimmt gültige Standardparameter, die verwendet werden, solange sie nicht explizit überschrieben werden.

- `eaName`: Name des EAs, der ausgeführt werden soll.

- `autoGenerateSeed`: Wenn dieser Parameter auf wahr gesetzt ist, wird ein neuer Random-Seed vom Algorithmus erzeugt. Sonst kann der Benutzer selber einen Seed angeben.

- `makeGraphComplete`: Wenn dieser Parameter gesetzt ist, werden alle im Graphen fehlende Kanten automatisch hinzugefügt und deren Gewichte auf unend-

lich gesetzt. Dies kann z. B. verwendet werden, wenn ein bestimmter Operator nur auf vollständigen Graphen funktioniert, der gegebene Graph jedoch nicht vollständig ist.

- `seed`: Random-Seed, um Ergebnisse reproduzieren zu können. Wenn dieser Parameter nicht gesetzt wird, dann wird ein neuer zufälliger Random-Seed generiert und in die Logs geschrieben.

- `creatorName`: Name der Creator-Strategie, die zur Erzeugung neuer Genotypen verwendet werden soll. Dadurch wird implizit der zu verwendende Genotyp spezifiziert, wodurch ein Genotyp-Parameter entfällt.

- `mutatorName`: Name der zu verwendenden Mutations-Strategie. Die Auswahlmöglichkeiten dieses Operators hängen von dem verwendeten Genotyp ab.

- `recombinatorName`: Name der zu verwendenden Rekombinations-Strategie. Die Auswahlmöglichkeiten dieses Operators hängen von dem verwendeten Genotyp ab.

- `selectorName`: Name der zu verwendenden Selektions-Strategie. Durch die ausgewählte Evaluations-Strategie wird automatisch bestimmt, ob der hier eingetragene Selektionsmechanismus die Fitness während der Ausführung maximieren oder minimieren muss.

- `mappingName`: Name des Genotyp-Phänotyp-Mappings. Durch die Auswahl wird implizit der zu verwendende Phänotyp bestimmt. Der hier ausgewählte Operator muss den ausgewählten Genotyp unterstützen.

- `evaluatorName`: Name der zu verwendenden Evaluations-Strategie. Dieser Operator definiert automatisch, ob die Fitness während der Ausführung maximiert oder minimiert werden muss.

- `terminatorName`: Name der Strategie, die zum Abbruch der Ausführung führt. Je nach Auswahl stehen verschiedene weitere Parameter wie ein Zeitlimit zur Verfügung, um das Abbrucherhalten des ausgewählten Algorithmus zu steuern.

- `popSize`: Größe der Population.
- `offspringSize`: Anzahl der Kinder, die pro Generation erzeugt werden.
- `archiveSize`: Größe des Archivs.
- `mutatorProbability`: Mutationswahrscheinlichkeit.
- `recombinatorProbability`: Rekombinationswahrscheinlichkeit.
- `numTournamentRounds`: Anzahl der Runden, die bei Tournament-Selection pro Wettkampf ausgeführt werden.
- `maxGenerations`: Festlegung eines Abbruchkriterium bezüglich der Anzahl durchlaufener Generationen.
- `maxTime`: Zeitlimit als Abbruchkriterium für die Ausführung des Algorithmus.
- `fitnessThreshold`: Abbruchkriterium bezüglich der besten bisher gefundenen Fitness.
- `maxDegree`: Maximaler Grad eines MST, damit dieser eine zulässige Lösung darstellt. Der Parameter wird nicht von jedem Algorithmus unterstützt.
- `constantFitnessGenLimit`: Terminiere, nachdem sich die Summe aller Fitnesswerte der Population über die gegebene Anzahl an Generationen nicht mehr geändert hat.
- `createInitialPopFromST`: Erzeugung der initialen Population mittels eines zufälligen Spannbaumes. Wenn diese Variable false ist, werden zufällige Prüfer-Nummern genutzt.

9.2.2. NAMOA*

Dies ist der in 8.1.2 beschriebene Algorithmus für das mehrkriterielle Kürzeste-Wege-Problem. Der Algorithmus hat keine Parameter, allerdings muss ihm eine Heuristik zur Verfügung gestellt werden, die für jeden Knoten die Entfernung zum Zielknoten abschätzt und dabei nie überschätzt. Dies kann auf mehrere Arten geschehen. Zum einen wird der Algorithmus sich selbst eine Heuristik erstellen, sollte ihm keine gestellt werden. In diesem Fall enthält die Heuristik für jeden Knoten die Gewichtsvektoren seiner ausgehenden Kanten. Zum anderen kann die Heuristik in die Eingabedatei geschrieben werden. Dazu wird das MONET-Graphformat erweitert, indem nach der Liste der Kantengewichte zunächst eine Zeile mit dem Inhalt START_HEURISTIC eingefügt wird. Darauf folgt die Heuristik. In jeder Zeile wird ein Vektor der Heuristik vermerkt, indem zunächst die Nummer des zugehörigen Knotens und dann die Einträge des Vektors eingetragen werden. Für diese Variante muss der `monet_heur_parser` verwendet werden. Außerdem kann mit dem `monet_heur_grid_parser` automatisch eine Heuristik für Gittergraphen erstellt werden.

9.2.3. Label-Correcting

Dies ist der in 8.1.1 beschriebene Algorithmus für das mehrkriterielle Kürzeste-Wege-Problem. Der Algorithmus benötigt nur einen Parameter `MERGE_MODE`, der angibt, ob beim Zusammenfügen von zwei Labelmengen ein einfacher paarweiser Vergleich aller Labels verwendet wird, oder Prozeduren benutzt werden, die Strukturen der Labelmengen ausnutzen.

9.2.4. MST-Algorithmen

Im folgenden werden die Parameter für die in 8.2.1 beschriebenen Two-Phase-Algorithmen für das bikriterielle MST-Problem erläutert. Tabelle 9.1 gibt eine vollständige Übersicht über alle dem Benutzer zur Verfügung stehenden Parame-

ter.

Parameter	Beschreibung
weightAnnotationName	Gibt die Zeichenkette an, die auf die Kantengewichte abbildet. (Standardwert: “edges”)
secondPhase	Auswahl des Algorithmus für die zweite Phase. (Mögliche Werte: “kbest” oder “branchbound”)

Tabelle 9.1.: Parameter für die Two-Phase-Algorithmen für das MST-Problem.

9.3. Aufsetzen einer Entwicklungsumgebung

In diesem Abschnitt wird beschrieben, wie MONET auf einem POSIX[4]-konformen System eingerichtet werden kann, da die von uns verwendeten Technologien, welche in Kapitel 6 vorgestellt wurden, diesen Prozess stark vereinfachen. Die in diesem Kapitel beschriebenen Schritte entsprechen den üblichen Standards. Da aber nicht jeder mit diesen vertraut ist, wird jeder Schritt auch beispielhaft anhand der Kommandozeilenbefehle gezeigt, die auf allen Betriebssystemen funktionieren sollten. Der Stil des Source-Codes sollte weitestgehend den üblichen Java-Standards entsprechen, die von Sun verfasst wurden und jetzt von Oracle weitergetragen werden[5].

9.3.1. Benötigte Programme

Um MONET weiterentwickeln zu können, benötigt man Git (vgl. Abschnitt 6.7) und Maven (vgl. Abschnitt 6.4). Wenn diese beiden Programme installiert sind, kann man den Quelltext von MONET mittels des folgenden Befehls erhalten[6]:

```
git clone git@projekte.itmc.tu-dortmund.de:in/pg573repository.git
```

[4] `http://www.opengroup.org/austin/papers/posix_faq.html`
[5] `http://www.oracle.com/technetwork/java/codeconv-138413.html`
[6] Die Adresse des Repsoitories wird sich vermutlich noch einmal ändern, da das Projekt unter einer freien Lizenz steht und das Repository nicht frei zugänglich ist

Hierbei wird der Quellcode in den Ordner `pg573repository` des derzeitigen Arbeitsverzeichnisses kopiert. Nun sollte ein Ordner vorhanden sein, in dem die Projekte `controlserver`, `worker`, `graph`, `parser` und `dummy` liegen. Der Control Server ist das Projekt, das die gesamte Logik der Plattform beinhaltet und die Weboberfläche zur Verfügung stellt. Der Worker ist das Programm, das sich mit dem Control Server verbindet und die Algorithmen der Experimente durchführt. Für eine genauere Beschreibung sei auf das Kapitel 5 verwiesen. Der Graph ist eine Implementierung des Graph-Interfaces, die von MONET benötigt werden. Der Parser ist eine Implementierung für einen Parser und wird in Abschnitt 9.5 genauer beschrieben. Der Dummy-Algorithmus ist eine beispielhafte Implementierung für einen Algorithmus und kann als Vorlage zum Entwickeln eigener Algorithmen verwendet werden. Eine genaue Beschreibung hierzu ist in Abschnitt 9.6 zu finden.

Falls man keinen Zugriff auf eine bereits eingerichtete MongoDB-Datenbank (vgl. Abschnitt 6.2) hat, muss man diese jedoch noch zusätzlich installieren[7] und starten. Starten kann man eine MongoDB-Instanz beispielsweise mit dem POSIX-kompatiblen Shell-Skript aus Code-Ausschnitt 9.1, das eine MongoDB-Instanz mit Standardparametern startet und die Daten in `/tmp/mongo` ablegt. Diese Parameter sind kompatibel mit den Standarddatenbankparametern, die im Unterabschnitt 9.3.2 beschrieben werden.

```
1  #!/bin/sh
2
3  dbpath="/tmp/mongo"
4  [ -d "$dbpath" ] || mkdir -p "$dbpath"
5  mongod --dbpath "$dbpath" --smallfiles
6  unset dbpath
```

Code-Ausschnitt 9.1: Shell-Skript zum Starten einer MongoDB-Instanz

Abgesehen vom Import der Maven-Projekte in die verwendete IDE muss noch eine Konfigurationsdatei angelegt werden.

[7] `http://docs.mongodb.org/manual/installation/`

9.3.2. Konfigurationsdatei

Die Konfigurationsdatei ist eine einfache Java-Properties-Datei[8]. Dieses Format ist ein einfaches *Key-Value*-Format, das von der Java-Standard-Bibliothek bestens unterstützt wird und auch gut von Hand veränderbar ist. Die Einträge werden einfach in der Form `configurationoption=value` zeilenweise in die Datei geschrieben. Die folgende Liste der Optionen erläutert die einzelnen Einstellungsmöglichkeiten.

- `dbusername` – Standardwert: `""`
  Der Benutzername, der zur Authentifizierung mit der Datenbank benötigt wird, wird mit dieser Option gesetzt. Standardmäßig ist dieser Parameter leer, da MongoDB keine solche Authentifizierung erfordert, sofern man dies nicht explizit ändert.

- `dbpassword` – Standardwert: `""`
  Das Passwort, das zur Authentifizierung mit der Datenbank benötigt wird, wird mit diesem Parameter gesetzt.

- `dbname` – Standardwert: `"monet"`
  Dieser Parameter entspricht dem Namen der *Collection* (vgl. Abschnitt 6.2), die von der Datenbank verwendet wird.

- `dbhost` – Standardwert: `"localhost"`
  Dieser Wert gibt die Netzwerkadresse des Computers an, auf dem die Datenbank zu erreichen ist.

- `dbport` – Standardwert: `27017`
  Dieser Parameter gibt den Port an, auf dem die Datenbank erreichbar ist.

- `host` – Standardwert: `"localhost"`
  Diese Option gibt die Netzwerkadresse des Control Servers an. Diese Option ist nur für den Worker wichtig.

- `controlport` – Standardwert: `33380`

[8] `http://en.wikipedia.org/wiki/.properties`

Dieser Wert definiert den Port, über den die Worker mit dem Control Server kommunizieren können.

- `cache` – Standardwert: `"."` (d. h. das Verzeichnis, in dem der Worker oder Control Server gestartet werden.)
  Das Verzeichnis, in dem die verschiedenen Graph-Dateien, Algorithmen-Bundles und so weiter gespeichert werden, wird mit diesem Parameter konfiguriert.

- `documentation` – Standardwert: `"./doc"`
  Dieser Wert gibt den Pfad zur Plattformdokumentation an. Diese erzeugt aus den Dateien im Verzeichnis, das hier angegeben wird, die Hilfeseite in der Weboberfläche der Plattform (vgl. Unterabschnitt 9.3.5).

Je nachdem, wie man die entsprechenden Komponenten startet, hat man auch andere Möglichkeiten die Parameter an das Programm weiterzureichen. Wenn man die Komponenten mittels Maven startet, dann kann man auch in der `pom.xml` des Projekts ein Profil[9] anlegen oder verändern. Hierbei werden im `properties`-Feld die gewünschten Optionen gesetzt.

```
1  <profiles>
2    ...
3    <profile>
4      <id>example</id>
5      <properties>
6        <dbpassword>secret</dbpassword>
7        <deploy>deployment</deploy>
8        ...
9      </properties>
10     ...
11   </profile>
12   ...
13 </profiles>
```

[9] `http://maven.apache.org/guides/introduction/introduction-to-profiles.html`

Es besteht auch die Möglichkeit die entsprechenden Parameter mit zwei Bindestrichen als Präfix als Kommandozeilenparameter anzugeben und gar keine Konfigurationsdatei zu erstellen. Das nächste Unterkapitel führt diese Möglichkeiten anhand von Beispielen vor.

9.3.3. Ausführen der Plattform in der Entwicklungsumgebung

Es gibt drei grundsätzliche Möglichkeiten die Plattform auszuführen. Empfohlen ist die Variante mit dem in der IDE eingebetteten Application Server, da dies die sicherste Alternative ist.

Application Server in der IDE

Eclipse bietet in seiner *Enterprise Edition* Variante[10] die Möglichkeit Java Application Server innerhalb der IDE zu verwalten und zu starten. Unter Umständen muss man den gewünschten Application Server noch manuell in Eclipse einrichten[11]. Dann kann man den Control Server starten, indem man auf das `controlserver`-Projekt mit der rechten Maustaste klickt und beim Menüpunkt `Run As` den Unterpunkt `Run on Server` wählt. Es sollte sich jetzt ein Browser mit der passenden Webseite öffnen.

Maven-Ziel `jetty:run`

Man kann die Plattform mittels eines Maven-Ziels bequem von der Kommandozeile oder aus der IDE heraus starten. Als Vorarbeit hierfür empfiehlt es sich ein Maven-Profil in der `pom.xml` des Control Servers zu erstellen oder zu bearbeiten. Dann kann man einen lokalen Control Server mittels des Befehls

```
mvn -P profilname jetty:run
```

[10] `https://www.eclipse.org/`

[11] `https://docs.jboss.org/author/display/AS7/Starting+JBoss+AS+from+Eclipse+with+JBoss+Tools`

starten und im Browser unter der Adresse `https://localhost:8443` aufrufen.

Ausführen der Klasse `Start`

Die `main`-Methode der Klasse `Start` im Paket `monet.controlserver` bietet eine weitere Möglichkeit zum Starten des Control Servers. Diese startet den Control Server und lässt sich am zuverlässigsten über die Kommandozeilenparameter aus Unterabschnitt 9.3.2 konfigurieren. Zusätzlich gibt es hier den Parameter `-w`, der den gestarteten Control Server dazu veranlasst, einen Worker direkt mit zu starten.

9.3.4. Ausführen des Workers in der Entwicklungsumgebung

Man muss nur die `main`-Methode der Klasse `monet.worker.WorkerMain` ausführen und den Kommandozeilenparameter `-c` für die Konfigurationsdatei übermitteln oder alle Parameter auf der Kommandozeile manuell setzen.

9.3.5. Dokumentation in der Weboberfläche

Die Dokumentation, die in der Weboberfläche angezeigt wird, wird zur Laufzeit des Control Servers immer dann neu generiert, wenn sich etwas an der Datei geändert hat. Da man den Pfad zu diesem Dokumentationsverzeichnis frei wählen kann, lässt sich diese auch getrennt von der Entwicklung an der Plattform selbst verwalten. Man kann die Dateien zum Beispiel im Netzwerk freigeben und dort einfach bei Bedarf editieren. Es sind nur wenige Regeln zu beachten, damit die Dokumentation auch so aussieht, wie man es gerne hätte.

- Es werden noch keine Unterverzeichnisse unterstützt, deshalb darf man nicht einfach Ordner erstellen, um die Dokumentation zu organisieren.

- Die Dateieendung muss `.md` sein. Dementsprechend müssen die Dateien auch

mit *Markdown*-Syntax geschrieben werden.[12]

- Die ersten zwei Zeichen des Dateinamens werden zur Sortierung der Dokumentationsdateien verwendet.

- Alles zwischen den zwei Sortierzeichen und dem Dateiendungssuffix wird als Linkname in der Navigationsleiste der Dokumentationsseite verwendet.

9.4. Deployment

In diesem Abschnitt wird eine mögliche Art des Deployments des Workers und des Control Servers vorgestellt.

Worker

Für das Deployment des Workers benötigt man derzeit das „statisch gelinkte“ Jar des Workers, welches die benötigten Abhängigkeiten mit sich bringt. Falls man dies nicht zur Hand hat, muss man das Repository klonen und im Projekt `worker` das Maven-Ziel `package` ausführen. Dann wird eine `worker-VERSION-with-dependencies.jar` im Verzeichnis `target` generiert. Diese kann man dann mit Shell-Skript aus Code-Ausschnitt 9.2 ausführen.

```
1 #!/bin/sh
2
3 WORKER_JAR=worker.jar
4 WORKER_CONFIG=monetrc
5 STACK_SIZE_IN_MB=512
6 HEAP_SIZE_IN_GB=12
7
8 while true ; do
```

[12] `http://daringfireball.net/projects/markdown/syntax`

```
9     java -Xss"$STACK_SIZE_IN_MB"m -Xmx"$HEAP_SIZE_IN_GB"g -jar "
          $WORKER_JAR" -c "$WORKER_CONFIG"
10 done
```

Code-Ausschnitt 9.2: Skript zum Starten des Workers

9.4.1. Control Server

Um die Plattform als Webservice anbieten zu können, muss man zunächst einen Java Application Server installieren und einrichten. Da MONET keine besonderen Features nutzt, sollte jeder gängige Application Server funktionieren. Getestet wurde es bisher jedoch nur mit dem JBoss Application Server[13]. Wenn der Application Server eingerichtet ist, muss man die `pom.xml` des Control-Server-Projekts an die realen Bedingungen anpassen. Hierfür sollte man das Maven-Profil `production` als Vorlage nehmen und die entsprechenden Parameter anpassen. Diese Parameter sind fast ausschließlich diejenigen, die auch im Unterabschnitt 9.3.2 erläutert werden. Dann empfiehlt es sich die Zugangsdaten für das `jboss-as`-Maven-Plugin anzupassen. Danach kann man den Server dann mit dem Befehl

```
mvn jboss-as:deploy -P production
```

auf dem JBoss Application Server *deployen*. Für andere Application Server muss man ein passendes Maven-Plugin finden und konfigurieren. Es gibt auch diverse Möglichkeiten diesen Prozess stärker zu automatisieren. Diese übersteigen aber die Reichweite dieses Handbuchs bei weitem und werden hier nicht weiter erwähnt.

9.5. Parser entwickeln

Graphen können in beliebigen Dateiformaten vorliegen. Um nicht unterstützte Dateiformate für die MONET-Plattform lesbar zu machen, ist die Implementierung

[13] `http://www.jboss.org/jbossas`

eines entsprechenden Parsers notwendig. Parser werden wie auch Algorithmen mittels *OSGi* in Bundles gespeichert.

Jeder Parser implementiert das Interface `GraphParser`. Dieses schreibt nur eine einzige Methode `parse(String, Job)` vor, welche als erstes Argument den Pfad zur Datei erwartet und als zweiten Parameter das `Job`-Objekt. Das `Job`-Objekt enthält *Parser-Parameter*, welche für das erfolgreiche Einlesen von Graphinstanzen essentiell sein können. In Abhängigkeit vom Format können diese Parameter jedoch auch irrelevant sein.

9.6. Algorithmen entwickeln

Die für MONET entwickelten Algorithmen werden mittels OSGi in Bundles gespeichert. Innerhalb eines Bundles können Abhängigkeiten von anderen Bundles definiert werden, welche ebenfalls durch OSGi aufgelöst werden. Um ein Bundle zu erzeugen, wird Maven genutzt.

Ordnerstruktur

Zur Implementierung eines Algorithmus mit Java ist eine mögliche Ordnerstruktur wie folgt:

Algorithm:

- *src/*
  - *my/private/package*
    - *Activator.java*
    - *MyAlgo.java*
  - *test/*

  - *pom.xml*

- *ressources*

  - *parameters.xml*

Implementierungshinweise

Um ein Bundle mit OSGi zu starten, muss eine Klasse innerhalb des Projekts das Interface `BundleActivator` implementieren. Das Interface `BundleActivator` stellt die zwei Methoden `start()` und `stop()` zur Verfügung. Die Registrierung des Algorithmus erfolgt mittels `ServiceDirectory.registerAlgorithm(context, new MyAlgo())` innerhalb der `start()`-Methode. Der Parameter `context` ist vom BundleActivator vorgegeben.

Als Startpunkt für einen Algorithmus dient die Methode `execute()` des Interfaces `Algorithm`. Die Methode `execute()` hat als Parameter den zugehörigen `Job`, der `Meter` und das `ServiceDirectory`. Im Job liegen die Graphinstanz, die Parameter des auszuführenden Algorithmus, ein Logger und ein Statusattribut zur Anzeige auf dem Controlserver vor.

Die Ergebnisse des ausgeführten Algorithmus werden in einer Datenbank auf dem Control Server gespeichert. Mit den Methoden des Meters wird auf die Datenbank zugegriffen und die Ergebnisse gespeichert. Der Meter bietet für die elementaren Datentypen `Integer`, `Long`, `Double` und `String` jeweils eine Methode zum Speichern an, welche z. B. die Form `measureDouble()` oder `measureInt()` haben. Als Parameter bekommen alle Methoden den Pfad, unter dem das Ergebnis in der Datenbank zu liegen hat, und den zu speichernden Wert. Beim Wert kann es sich um ein einzelnes Datum oder eine Datenstruktur handeln, welche das Interface `Iterable` implementiert.

In der *pom.xml* sind Abhängigkeiten des Bundles einzutragen. Daran schließen sich Informationen zum Bauen des Bundles an. Dabei sind die Ordner mit den Quellcode-Dateien, den Testklassen und den Ressourcen zu spezifizieren. Mittels OSGi kann spezifiziert werden, welche Pakete des Algorithmus nach außen angeboten, welche nur

innerhalb des Projekts genutzt und welche von außen importiert werden. Des Weiteren ist die Klasse anzugeben, welche das Interface BundleActivator implementiert.

Die Parameter eines Algorithmus werden über die Webseite eingegeben. Für die Anzeige wird die *parameter.xml* geparst, welche beim Implementieren eines Algorithmus geschrieben werden muss. Ein Parameter hat einen Namen, eine Beschreibung und kann entweder über ein Eingabefeld eingegeben werden oder über eine Auswahlliste ausgewählt werden. Ein Parameter wird dabei wie folgt in der *parameter.xml* angegeben:

```
1  <algorithm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
       noNamespaceSchemaLocation="bundle-description.xsd">
2      <name><!-- Algorithm Name --></name>
3      <description><!-- Description Text --></description>
4      <parameters>
5        <parameter name="<!-- Param Name -->">
6          <description><!-- Parameter Description --></description>
7            <choices>
8              <choice value=<!-- Value Name -->></choice>
9              <choice value=<!-- Value Name -->></choice>
10           </choices>
11       </parameter>
12       <parameter name="<!-- Param Name -->">
13         <description><Parameter Description></description>
14         <integer default="<!-- Default Integer Value -->"/>
15       </parameter>
16   </parameters>
17 </algorithm>
```

Folgend wird eine simple *pom.xml* angegeben, welche durch ein paar Änderungen so eingesetzt werden kann, um einen Algorithmus für MONET zu entwickeln.

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
       www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
       apache.org/xsd/maven-4.0.0.xsd">
```

```
3   <modelVersion>4.0.0</modelVersion>
4   <parent>
5     <groupId>monet</groupId>
6     <artifactId>monet</artifactId>
7     <version>0.0.2-SNAPSHOT</version>
8   </parent>
9   <packaging>bundle</packaging>
10  <artifactId><!-- Name --></artifactId>
11
12  <dependencies>
13    <dependency>
14      <groupId>org.OSGi</groupId>
15      <artifactId>org.OSGi.core</artifactId>
16      <version>5.0.0</version>
17      <scope>provided</scope>
18      <optional>true</optional>
19    </dependency>
20    <dependency>
21      <groupId>junit</groupId>
22      <artifactId>junit</artifactId>
23      <version>[4.0,)</version>
24      <scope>compile</scope>
25      <optional>true</optional>
26    </dependency>
27    <dependency>
28      <groupId>monet</groupId>
29      <artifactId>monet_worker</artifactId>
30      <version>0.0.2-SNAPSHOT</version>
31      <scope>compile</scope>
32      <optional>false</optional>
33    </dependency>
34
35    <!-- specific Dependency -->
36    <dependency>
```

```
37        <groupId>monet</groupId>
38        <artifactId>monet_graph</artifactId>
39        <version>0.0.2-SNAPSHOT</version>
40          <type>bundle</type>
41          <optional>true</optional>
42      </dependency>
43    </dependencies>
44    <build>
45      <sourceDirectory>src</sourceDirectory>
46      <testSourceDirectory>tests</testSourceDirectory>
47      <plugins>
48        <plugin>
49          <artifactId>maven-compiler-plugin</artifactId>
50          <version>3.0</version>
51          <configuration>
52            <source>1.7</source>
53            <target>1.7</target>
54          </configuration>
55        </plugin>
56
57        <plugin>
58          <groupId>org.apache.felix</groupId>
59          <artifactId>maven-bundle-plugin</artifactId>
60          <version>2.3.7</version>
61          <extensions>true</extensions>
62          <configuration>
63            <instructions>
64              <Bundle-Name>${pom.name}</Bundle-Name>
65              <Private-Package>
66                <!-- Internal Package -->
67              </Private-Package>
68              <Bundle-Activator><!-- Class that implements
                    BundleActivator --></Bundle-Activator>
69              <Import-Package>
```

```
70            <!-- Imported Packages -->
71              org.OSGi.framework,
72              monet.interfaces,
73              monet.worker,
74              org.apache.logging.log4j,
75              monet.graph,
76              monet.graph.interfaces,
77              monet.graph.weighted,
78            </Import-Package>
79            <Export-Package>
80              <!-- Exported Packages -->
81            </Export-Package>
82            <Bundle-ClassPath>${maven-dependencies}</Bundle-ClassPath>
83          </instructions>
84        </configuration>
85      </plugin>
86    </plugins>
87  </build>
88 </project>
```

Installationshinweise

Aus diesem Projekt lässt sich mit *mvn install* ein Bundle erzeugen. Das Bundle ist eine *Jar*-Datei, wobei der Name die Form *name-version.jar* hat. Dieses Bundle kann anschließend auf dem Control Server hochgeladen werden.

10

Experimente

Dieser Abschnitt befasst sich mit den im Rahmen der Projektgruppe durchgeführten Experimenten zur Auswertung der implementierten Algorithmen. Zuerst wird beschrieben, welche Algorithmen mit welchen Parametern ausgewählt wurden, welche Graph-Instanzen verwendet wurden und wie die Einteilung in Jobs und Experimente erfolgte (Abschnitt 10.1). Bevor die Experimente in Abschnitt 10.3 ausgewertet werden, wird in Abschnitt 10.2 der Ablauf der eigentlichen Ausführung erläutert.

10.1. Aufbau der Experimente

Die Dauer der Experimentreihe wurde auf etwa vier Stunden festgelegt. Für MST-Probleme wurden die Algorithmen *k-Best*, *Branch-and-Bound*, *Prüfer-EA* und *SMS-EMOA* herangezogen. Kürzeste-Wege-Probleme wurden mit *SPEA2*, *Label Correcting* und *NAMOA** gelöst. Als Probleminstanzen wurden gerichtete (Kürzeste-Wege-Probleme) und ungerichtete (Spannbaum-Probleme) zweidimensionale Gittergraphen verschiedener Größen mit zufälligen zwei- oder dreidimensionalen Gewichtsvektoren erzeugt. Aus der Anzahl der Algorithmen und der festgelegten Zeitdauer ergibt sich, dass ein Algorithmus pro Probleminstanz eine Laufzeit von fünf Minuten nicht überschreiten sollte.

MST-Experimente: Als MST-Instanzen wurden Gittergraphen der Größe 4×4 (16 Knoten) und 5×5 (25 Knoten) mit jeweils zweidimensionalen Gewichtsvektoren generiert. Für jede Größe wurden drei Instanzen unabhängig voneinander mit verschiedenen Gewichten aus $[0; 300)$ erzeugt, die jeweils zu einem Experiment zusammengefasst wurden. Insgesamt ergibt dies sechs Instanzen.

Kürzeste-Wege-Experimente: Für diese Problemklasse wurden Graphen der Größe 20×20 (400 Knoten) und 40×40 (1600 Knoten) mit zweidimensionalen Gewichten und Graphen der Größe 10×10 und 20×20 mit dreidimensionalen Gewichten erzeugt. Wie bei den MST-Experimenten existieren je drei Instanzen mit verschiedenen Gewichten aus $[0; 300)$. Insgesamt ergibt dies zwölf Instanzen.

Da die Algorithmen *Prüfer-EA*, *SMS-EMOA* und *SPEA2* nicht deterministisch ablaufen, werden diese jeweils dreimal auf jeder einzelnen Graph-Instanz ausgeführt. Um die Ergebnisse reproduzieren zu können, werden die dabei verwendeten Random-Seeds in den entsprechenden Jobs geloggt.

Die folgende Tabelle gibt eine Übersicht über die einzelnen Experimente, dazugehörigen Parameter und Graph-Instanzen. Die angegebene Größe der Graphen bezieht sich auf das Gitter; so steht z.B. eine Größe von 40 für einen Graphen von 40×40 Knoten.

Experiment (Bundle)	**Parameter**	**Graphen** (Größe/Dim.)
STSP-Namoa-40-2 (monet_sssp_namoa)	keine	40/2
STSP-SPEA2-10-3 (monet_ea)	preset = DirectSSSP-SPEA2, makeGraphComplete = false, autoGenerateSeed = Yes	10/3
STSP-Namoa-20-3 (monet_sssp_namoa)	keine	20/3
STSP-Namoa-20-2 (monet_sssp_namoa)	keine	20/2
STSP-Namoa-10-3 (monet_sssp_namoa)	keine	10/3
STSP-LabelCorrectingMerge-40-2 (monet_sssp_labelcorrecting)	MERGE_MODE = true	40/2

Experiment (Bundle)	**Parameter**	**Graphen** (Größe/Dim.)
STSP-LabelCorrectingMerge-20-3 (monet_sssp_labelcorrecting)	MERGE_MODE = true	20/3
STSP-LabelCorrectingMerge-20-2 (monet_sssp_labelcorrecting)	MERGE_MODE = true	20/2
STSP-LabelCorrectingMerge-10-3 (monet_sssp_labelcorrecting)	MERGE_MODE = true	10/3
STSP-LabelCorrecting-40-2 (monet_sssp_labelcorrecting)	MERGE_MODE = false	40/2
STSP-LabelCorrecting-20-3 (monet_sssp_labelcorrecting)	MERGE_MODE = false	20/3
STSP-LabelCorrecting-20-2 (monet_sssp_labelcorrecting)	MERGE_MODE = false	20/2
STSP-LabelCorrecting-10-3 (monet_sssp_labelcorrecting)	MERGE_MODE = false	10/3
MST-BranchBound-5-2 (monet_mst_twophase)	secondPhase = branchbound	5/2
MST-BranchBound-4-2 (monet_mst_twophase)	secondPhase = branchbound	4/2
MST-KBest-5-2 (monet_mst_twophase)	secondPhase = kbest	5/2
MST-PrueferEA-5-2 (monet_ea)	preset = DirectMST-PrueferEA, makeGraphComplete = false, autoGenerateSeed = Yes	5/2
MST-KBest-4-2 (monet_mst_twophase)	secondPhase = kbest	4/2
MST-PrueferEA-4-2 (monet_ea)	preset = DirectMST-PrueferEA, makeGraphComplete = false, autoGenerateSeed = Yes	4/2
MST-SMSEMOA-5-2 (monet_ea)	preset = DirectMST-SMSEMOA, makeGraphComplete = false, autoGenerateSeed = Yes	5/2
MST-SMSEMOA-4-2 (monet_ea)	preset = DirectMST-SMSEMOA, makeGraphComplete = false, autoGenerateSeed = Yes	4/2
STSP-SPEA2-40-2 (monet_ea)	preset = DirectSSSP-SPEA2, makeGraphComplete = false, autoGenerateSeed = Yes	40/2

Experiment (Bundle)	**Parameter**	**Graphen** (Größe/Dim.)
STSP-SPEA2-20-3 (monet_ea)	preset = DirectSSSP-SPEA2, makeGraphComplete = false, autoGenerateSeed = Yes	20/3
STSP-SPEA2-20-2 (monet_ea)	preset = DirectSSSP-SPEA2, makeGraphComplete = false, autoGenerateSeed = Yes	20/2

Tabelle 10.1.: Übersicht durchgeführter Experimente.

Die ausgewählten Presets der EAs definieren Parameter für eine Terminierung nach 400 (SPEA2, Prüfer-EA) bzw. 4000 Generationen (SMS-EMOA). Alle weiteren Abbruchbedingungen wurden deaktiviert.

10.2. Durchführung

Die Experimente und Jobs wurden automatisiert über ein Skript angelegt. Aufgrund der Erzeugung der Experimente im Zustand *ready* werden diese, falls ein Worker zur Verfügung steht, sofort ausgeführt. Alle Messungen beziehen sich auf einen Worker mit Intel Core i7 CPU 940 mit 2,93 GHz und etwa 12,3 GB RAM.

Die Durchführung erfolgte nach dem Anlegen der Experimente automatisch durch den *Control Server*. Die Reihenfolge, in der die Experimente durchgeführt wurden, war zufällig, weil die Priorität für alle Experimente gleich war.

10.3. Auswertung

Bei der Ausführung der Experimente werden die Laufzeit und eine Lösungsmenge erfasst. Dabei ist zu beachten, dass die evolutionären Algorithmen nicht die gleiche Lösungsmenge finden wie die exakten Algorithmen. Diese sind in den meisten Fällen schlechter. Zur Bewertung der Algorithmen wird einerseits die Laufzeit herangezogen, welche direkt zwischen den Algorithmen verglichen werden kann. Zur weiteren Bewertung der Algorithmen gibt es verschiedene weitere Ansätze, abhängig von dem betrachteten Algorithmus und Problem. In [24] wird z.B. für die beiden Two-Phase-Ansätze vorgeschlagen, die in der zweiten Phase gefundenen effizienten Lösungen und die Suchbaumgröße in Abhängigkeit von der Anzahl an Knoten zu erfassen. Die Ansätze sind nicht auf die beiden evolutionären Algorithmen Prüfer-EA und SMS-EMOA übertragbar, da keine zweite Phase existiert und kein Suchbaum berechnet wird, wie es beim Branch-and-Bound-Verfahren der Fall ist. In [22] wird ebenfalls die Laufzeit gemessen und zusätzlich die Anzahl einzelner Schritte des Algorithmus, z.B. Merge. Damit könnten Algorithmen, die ähnlich zueinander sind, verglichen werden, aber wenn der Ablauf sich deutlich unterscheidet, wie z.B. SPEA2 und Label Correcting, entfällt diese Möglichkeit. Eine Möglichkeit, die Algorithmen zu vergleichen, ist die S-Metrik für jeden Algorithmus zu bestimmen und ins Verhältnis zu einem Referenzergebnis zu setzen. Bei den exakten Algorithmen sollte sich hierdurch zeigen, dass das dominierte Hypervolumen identisch ist. Zusätzlich zeigt sich, wie stark die evolutionären Algorithmen vom exakten Ergebnis abweichen. Des Weiteren kann die S-Metrik mittels MONET einfach berechnet werden, wodurch die Ergebnisse einfach nachvollzogen werden können. Als Referenzalgorithmen für die Kürzeste-Wege-Experimente wurde Label Correcting ausgewählt und bei den MST-Experimenten fiel die Wahl auf den Two-Phase-Ansatz mit Branch-and-Bound.

Kürzeste-Wege-Experimente

Zur Lösung des Kürzeste-Wege-Problems wurden die exakten Algorithmen Label Correcting, Label Correcting mit der Merge-Methode nach Kung, NAMOA* und der evolutionäre Algorithmus SPEA2 implementiert. Die Wahl des Referenzergebnisses fiel auf Label Correcting, da die Experimente auf allen Graphen erfolgreich durchliefen und sowohl Label Correcting als auch Label Correcting mit der Merge-Methode nach

Kung dasselbe Ergebnis lieferten. Bei dem Vergleich zwischen den Label Correcting Verfahren zeigten sich keine großen Unterschiede in der Laufzeit. Label Correcting war minimal schneller, was ein Vorteil ist, da die Lösungen identisch waren. Im Vergleich dazu zeigte sich bei NAMOA*, dass die Laufzeiten deutlich schlechter ausfielen und das dominierte Hypervolumen im Dreidimensionalen deutlich geringer ausfiel. Der evolutionäre Algorithmus SPEA2 fiel beim Vergleich der Laufzeit deutlich schlechter aus als Label Correcting. Da zur Berechnung von kürzesten Wegen schnelle, exakte Algorithmen bekannt sind, sollte die Wahl auf diese fallen, hier Label Correcting. Die evolutionären Algorithmen sind langsamer und haben eine geringere Güte als die exakten Algorithmen.

MST-Experimente

Für die MST-Experimente wurden im Rahmen der PG zwei Two-Phase-Ansätze implementiert: Branch-and-Bound und der k-Best-Algorithmus. Bei diesen Experimenten kamen zwei evolutionäre Algorithmen zum Einsatz: erstens der Prüfer-EA und zweitens der SMS-EMOA. Als Referenz wurde hier der Two-Phase-Ansatz mit Branch-and-Bound gewählt, da mit dem k-Best-Algorithmus nicht alle Experimente erfolgreich verliefen und für den Graphen mit 5×5 Knoten keine Ergebnisse vorlagen, da Probleme wegen zu geringem Speicher auftraten. Des Weiteren war das dominierte Hypervolumen für die Graphen mit 4×4 Knoten erwartungsgemäß bei beiden exakten Algorithmen identisch. Bei den gegebenen Implementierungen sollte für die Lösung des Problems auf den Two-Phase-Ansatz mit Branch-and-Bound zurückgegriffen werden. Die Laufzeit mit dem k-Best Algorithmus ist wenigstens um den Faktor zehn schlechter. Die Ausführung der evolutionären Algorithmen dauerte im Vergleich zum Verfahren mit Branch-and-Bound immer länger. Als besonders langsam hat sich der SMS-EMOA für die Graphen mit 4×4 Knoten herausgestellt. Hier wird häufig die S-Metrik berechnet, um das zu entfernende Individuum zu bestimmen, was in einer langen Laufzeit mündet. Dies trat bei den Graphen mit 5×5 Knoten nicht auf. Bei den Graphen mit 4×4 Knoten erzielten die evolutionären Algorithmen gute Ergebnisse; in einigen Fällen war das dominierte Hypervolumen identisch. Der Prüfer-EA erzielte bei den Graphen mit 5×5 Knoten schlechtere Ergebnisse als der SMS-EMOA, sowohl bei der Güte als auch bei der Laufzeit. Allerdings schneidet der SMS-EMOA immer noch schlechter ab als der Two-Phase-Ansatz mit Branch-

and-Bound; von daher sollte auf den Two-Phase-Ansatz mit Branch-and-Bound zurückgegriffen werden.

Fazit

Zur effizienten Lösung der MST-Probleme wurden zwei exakte Algorithmen implementiert, die nur auf zweidimensionale Zielfunktionen anwendbar sind und teilweise schon auf Graphen mit 5×5 Knoten nicht fehlerfrei liefen. Kürzeste Wege konnten zwar für größere Graphen mit mehr als zwei Zielfunktionen bestimmt werden, aber hier bedarf es noch weiterer Untersuchungen, bis zu welcher Größe das noch effizient und fehlerfrei funktioniert. Die implementierten evolutionären Algorithmen fanden zwar immer Lösungen zu den gegebenen Problemen, aber in längerer Zeit und mit Abweichungen des dominierten Hypervolumens.

Auflistung der Ergebnisse

In Tabelle 10.2 sind die über die Graphinstanzen gemittelten Ergebnisse dargestellt. Die vollständigen Ergebnisse der Experimente sind im Anhang in Anhang E zu finden.

Algorithmus Name	Graph Größe/ Dim	S-Metrik	Laufzeit [s]	Güte
Branch-and-Bound	4/2	245794.3	0,77	1,00
Branch-and-Bound	5/2	951365	11,57	1,00
k-Best	4/2	245794.3	17,75	1,00
Prüfer-EA	4/2	198886.7	15,32	0,8092
Prüfer-EA	5/2	408364.6	31,12	0,4292
SMS-EMOA	4/2	219656	624,34	0,8937
SMS-EMOA	5/2	660492.8	25,36	0,6943
Label Correcting	20/2	10568291	0,22	1,00
Label Correcting	40/2	39358937	1,99	1,00
Label Correcting	10/3	5532498648	0,14	1,00
Label Correcting	20/3	48691467331	8,85	1,00
Label Correcting Merge	20/2	10568291	0,25	1,00
Label Correcting Merge	40/2	39358937	2,30	1,00
Label Correcting Merge	10/3	5532498648	0,12	1,00
Label Correcting Merge	20/3	48691467331	9,34	1,00
NAMOA*	20/2	10568291	1,59	1,00
NAMOA*	40/2	39357947	390,28	0,9999
NAMOA*	10/3	5532414078	0,44	0,9999
NAMOA*	20/3	23454204392	160,78	0,4817
SPEA2	20/2	2099624	51,39	0,1987
SPEA2	40/2	4105265	58,19	0,1043
SPEA2	10/3	1245168875	57,49	0,2251
SPEA2	20/3	13940925333	60,06	0,2863

Tabelle 10.2.: Die über die Graphinstanzen gemittelten Ergebnisse

11

Fazit

Insgesamt war die Projektgruppe MONET erfolgreich. Alle wesentlichen Anforderungen konnten erfüllt werden.

Die Plattform, also das webgestütze Analysewerkzeug, wurde insgesamt zu groß angelegt, weil alle Beteiligten keine Erfahrungen mit der Planung und Aufwandseinschätzung großer Projekte hatten. Dies hat sich leider erst sehr spät herausgestellt, was dazu geführt hat, dass die Qualität und die Anzahl der implementierten Algorithmen und durchgeführten Experimente kleiner war als ursprünglich erhofft. Dazu kam, dass verschiedene Bugs und Probleme beim Deployment die Experimente verzögert haben, so dass diese sehr oft wiederholt werden mussten.

Die Entscheidung für ein eigenes Graph-Framework hat sich insbesondere durch die so auf ihre Kernideen reduzierte Implementierung der damit entwickelten Algorithmen bewährt. Alle wesentlichen Funktionen, die von den implementierten Algorithmen im Umgang mit Graphen und Annotationen – insbesondere mehrkriteriellen Kantengewichten – benötigt wurden, konnten in das Framework ausgelagert werden. Darüber hinaus besitzt das Graph-Framework noch weitgehend ungenutztes Potential, so dass auch die Implementierung weiterer Algorithmen gut vorstellbar ist. Die Wahl von Java als Programmiersprache für das Graph-Framework und die Algorithmen hat – abgesehen von einigen softwarearchitektonischen Einschnitten im Graph-Framework – vor allem das Laufzeitverhalten an einigen Stellen beeinträchtigt. Bei der Verwen-

dung einer nicht virtualisierten Programmiersprache wie C++ wäre dieses Problem vermutlich nicht so stark aufgetreten; es hätte allerdings die gemeinsame Arbeit in der Projektgruppe auf Grund der Komplexität der Sprache selbst erschwert. Da die Projektgruppe für die meisten von uns das erste größere, selbstkoordinierte Softwareprojekt war, erweist sich die Wahl von Java auch im Nachhinein als guter Kompromiss.

Die in Kapitel 10 beschriebenen, auf eine Laufzeit von vier Stunden angelegten Experimente geben einen guten Überblick über die Instanzgrößen, die von den Algorithmen behandelt werden können. Während sich die exakten Algorithmen wegen ihrer Laufzeit auf kleinere Graphinstanzen mit wenigen Dimensionen beschränken, liefern evolutionäre Algorithmen nur heuristische Lösungen, eignen sich jedoch auch für größere Instanzen und mehr Dimensionen. Die Auswertung der Experimente hat ergeben, dass die Lösungsqualität der evolutionären Algorithmen, zumindest auf kleinen Instanzen, denen der exakten Algorithmen sehr nahe kommt. Eine weitere Reihe von Experimenten war zwar angedacht, wurde jedoch aus Zeitmangel und der Erkenntnis, dass auch leicht größere Instanzen die Laufzeit stark verlängern, nicht durchgeführt.

Literaturverzeichnis

[1] N. Beume. „Hypervolumen-basierte Selektion in einem evolutionären Algorithmus zur Mehrzieloptimierung“. Diplomarbeit. Technische Universität Dortmund, Fachbereich Informatik, 2006.

[2] N. Beume, B. Naujoks und G. Rudolph. „SMS-EMOA: Effektive evolutionäre Mehrzieloptimierung“. In: *Automatisierungstechnik* 56.7 (2008), S. 357–364.

[3] J. Brumbaugh-Smith und D. Shier. „An empirical investigation of some bicriterion shortest path algorithms“. In: *European Journal of Operational Research* 43.2 (1989), S. 216–224.

[4] M. W Carlyle und R. K. Wood. „Near-shortest and K-shortest simple paths“. In: *Networks* 46.2 (2005), S. 98–109.

[5] M. Chimani und K. Klein. „Algorithm Engineering: Concepts and Practice“. In: *Experimental Methods for the Analysis of Optimization Algorithms*. Hrsg. von T. Bartz-Beielstein, M. Chiarandini, L. Paquete und M. Preuss. Berlin: Springer, 2010, S. 131–158.

[6] M. Ehrgott. „A discussion of scalarization techniques for multiple objective integer programming“. In: *Annals of Operations Research* 147.1 (2006), S. 343–360.

[7] H. Gabow. „Two Algorithms for Generating Weighted Spanning Trees in Order“. In: *SIAM Journal on Computing* 6.1 (1977), S. 139–150.

[8] K. Gama und D. Donsez. „Applying dependability aspects on top of “aspectized” software layers“. In: *Proceeding of the 10th international conference on Aspect-oriented software development*. Hrsg. von P. Borba und S. Chiba. ACM, 2012, S. 177–189.

[9] H. W. Hamacher und G. Ruhe. „On spanning tree problems with multiple objectives“. In: *Annals of Operations Research* 52.4 (1994), S. 209–230.

[10] N. Katoh, T. Ibaraki und H. Mine. „An Algorithm for Finding k Minimum Spanning Trees“. In: *SIAM Journal on Computing* 10.2 (1981), S. 247–255.

[11] H. T. Kung, F. Luccio und F. P. Preparata. „On Finding the Maxima of a Set of Vectors“. In: *Journal of the ACM* 22.4 (1975), S. 469–476.

[12] L. Mandow und J. L. P. De La Cruz. „Multiobjective A* Search with Consistent Heuristics“. In: *Journal of the ACM* 57.5 (2008), 27:1–27:25.

[13] J. Maria, A. Pangilinan und G. Janssens. „Evolutionary algorithms for the multiobjective shortest path planning problem“. In: *International Journal of Computer and Information Science and Engineering*. 2007, S. 54–59.

[14] J. Mote, I. Murthy und D. L. Olson. „A parametric approach to solving bicriterion shortest path problems“. In: *European Journal Of Operational Research* 53.1 (1991), S. 81–92.

[15] B. Naveh. *Welcome to JGraphT – A free Java Graph Library*. 8.11.2012. URL: `http://jgrapht.org/` (besucht am 28. 09. 2013).

[16] T. Nelson, D. Fischer und J. O'Madadhain. *JUNG - Java Universal Network/Graph Framework*. 24.01.2010. URL: `http://jung.sourceforge.net/` (besucht am 28. 09. 2013).

[17] G.R. Raidl. „An Efficient Evolutionary Algorithm for the Degree-Costrained Minimum Spanning Tree Problem“. In: *Proceedings of the 2000 Congress on Evolutionary Computation* (2000), S. 104–111.

[18] A. Raith und M. Ehrgott. „A comparison of solution strategies for biobjective shortest path problems“. In: *Computional Operational Research* 36.4 (2009), S. 1299–1331.

[19] R.M. Ramos, S. Alonso, J. Sicilia und C. González. „The problem of the optimal biobjective spanning tree“. In: *European Journal of Operational Research* 111.3 (1998), S. 617–628.

[20] P. Sanders, K. Mehlhorn, R. Möhring, B. Monien, P. Mutzel und D. Wagner. *Description of the DFG algorithm engineering priority programme*. 2005.

[21] K. Schwaber. *Agile Project Management with Scrum*. Redmont: Microsoft Press, 2004.

[22] A. J. V. Skriver und K. A. Andersen. „A label correcting approach for solving bicriterion shortest-path problems“. In: *Computers & Operations Research* 27.6 (2000), S. 507–524.

[23] F. Sourd und O. Spanjaard. „A multiobjective branch-and-bound framework: Application to the biobjective spanning tree problem“. In: *INFORMS Journal on Computing* 20.3 (2008), S. 472–484.

[24] S. Steiner und T. Radzik. *Solving the biobjective minimum spanning tree problem using a k-best algorithm*. Technical Report TR-03-06. Department of Computer Science, King's College London, Okt. 2003.

[25] J. Sutherland und K. Schwaber. *The Scrum Guide*. 2011. URL: `https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/Scrum_Guide.pdf`.

[26] A. Warburton. „Approximation of Pareto optima in multiple-objective, shortest-path problems“. In: *Operations Research* 35.1 (1987), S. 70–79.

[27] L. While, P. Hingston, L. Barone und S. Huband. „A faster algorithm for calculating hypervolume“. In: *IEEE Transactions on Evolutionary Computation* 10.1 (2006), S. 29–38.

[28] G. Zhou und M. Gen. „Genetic algorithm approach on multi-criteria minimum spanning tree problem“. In: *European Journal Of Operational Research* 114.1 (1999), S. 141–152.

[29] E. Zitzler und L. Thiele. *Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach*. 1999.

A

Arbeitsaufteilung für den Endbericht

Die folgende Liste gibt einen Überblick über die Arbeitsaufteilung, die bei der Erstellung des Endberichts vorgenommen wurde. Die genannten Abschnitte (und, falls nicht anders aufgelistet, alle dazugehörigen Unterabschnitte) wurden von den jeweils in Klammern eingetragenen Personen erstellt.

1. Einleitung (Sven Selmke)
2. Übersicht (David Mezlaf)
3. Seminar (Jeder sein eigenes Thema)
4. Organisation (Marco Kuhnke)

4.4. Redmine (Max Günther)

5. Architektur (Sebastian Witte)

5.1. Worker (Johannes Kowald)

5.2. Control Server (Andreas Pauly)

5.3. Kommunikation (David Mezlaf)

6. Technische Aspekte (Johannes Kowald)

6.1. Java (Johannes Kowald)

6.2. MongoDB (Max Günther)

6.3. OSGi und Apache Felix (Max Günther)

6.4. Maven (Max Günther)

6.5. Webtechnologien (Jakob Bossek)

6.6. Graph-Dateiformate (Jakob Bossek)

6.7. Git (Sebastian Witte)

7. Graph-Framework (Hendrik Fichtenberger)

7.1. Motivation einer eigenen Graph-Bibliothek (Hendrik Fichtenberger)

7.2. MONET Graph (Hendrik Fichtenberger)

7.3. Graph-Generator (Jakob Bossek)

7.4. MONET Parser (Jakob Bossek)

8. Algorithmen (Hendrik Fichtenberger)

8.1. Kürzeste-Wege-Problem (Michael Capelle)

8.1.1. Label-Correcting (Michael Capelle, David Mezlaf)

8.1.2. Mehrkriterieller A*-Algorithmus (Michael Capelle)

8.2. Minimale-Spannbaum-Problem (Christopher Morris, Hendrik Fichtenberger)

8.3. Evolutionäre Algorithmen (Sven Selmke)

9. Handbuch (Sebastian Witte)

9.1. Bedienung der Weboberfläche (Andreas Pauly)

9.2. Existierende Algorithmen (David Mezlaf)

9.2.1. EA-Framework (Sven Selmke)

9.2.2. NAMOA* (Michael Capelle)

9.2.3. Label-Correcting (Michael Capelle)

9.2.4. MST-Algorithmen (Christopher Morris, Hendrik Fichtenberger)

9.3. Aufsetzen einer Entwicklungsumgebung (Sebastian Witte)

9.4. Deployment (Sebastian Witte)

9.5. Parser entwickeln (Jakob Bossek)

9.6. Algorithmen entwickeln (Andreas Pauly)

10. Experimente (Andreas Pauly, Sven Selmke)

10.1. Aufbau der Experimente (Sven Selmke)

10.2. Durchführung (Andreas Pauly, Sven Selmke)

10.3. Auswertung (Andreas Pauly)

11. Fazit (Max Günther, Sven Selmke)

A. Arbeitsaufteilung für den Endbericht (Sven Selmke)

B. Ausblick nach dem Sommersemester 2013 (Andreas Pauly)

C. Merge-Methode nach Kung (David Mezlaf)

D. Konsole (Johannes Kowald)

B

Ausblick nach dem Sommersemester 2013

Für das kommende Semester ist eine große offene Aufgabe die Erstellung einer graphischen Benutzerschnittstelle. Dies soll durch eine Weboberfläche realisiert werden, die wahrscheinlich mit *HTML5* erstellt wird. Eine Beschreibung des Aufbaus befindet sich in Kapitel 9.1.

Eine weitere Baustelle stellt die Analyse von Ergebnissen dar. Geplant ist, dass die Ergebnisse von Algorithmen miteinander verglichen werden können. Dabei sollen Diagramme und einfache statistische Eigenschaften genutzt werden.

Die Konfigurierbarkeit der Parameter eines Algorithmus sollen vereinfacht werden. Ziel ist hier eine XML-Datei im *Bundle* zu speichern, welche Informationen über die Namen und Wertebereiche der Parameter enthält. So soll eine falsche Parametrisierung ausgeschlossen werden.

Ein Algorithmus kann Fehler in der Ausführung verursachen und es nötig machen, dass er händisch vom Benutzer beendet wird; dies ist noch zu realisieren.

Ein weiterer Punkt ist das Auslesen der Hardwareinformation von *Workern*. Aktuell ist dies auf Microsoft Windows- und Linux-Systemen möglich. Eine Erweiterung hier ist die Portierung auf Betriebssysteme wie Mac OS X oder Solaris.

Um weitere Tests bei den SSSP-Algorithmen durchzuführen, soll im nächsten Semester

ein zweiphasiger Algorithmus und ein evolutionärer Algorithmus implementiert werden. Bei den MST-Algorithmen ist im nächsten Semester ebenfalls ein weiterer evolutionärer Algorithmus geplant, siehe. Hier ist neben den bisher implementierten Algorithmen der NSGA II (Non-Dominated Sorting Genetic Algorithm II) von Interesse.

C

Merge-Methode nach Kung

Wir beschreiben hier das von Kung et al. in [11] vorgestellte Verfahren zum Finden aller nicht-dominierten Vektoren in einer Menge von Vektoren. Sei $\mathcal{V} = \{v_1, \ldots, v_n\} \subset \mathbb{R}^d$ eine Menge d-dimensionaler Vektoren, und seien diese Vektoren aufsteigend nach erster Komponente sortiert. Das Verfahren, wie wir es implementierten, partitioniert nun diese Menge $\mathcal{V}$ in die Teilmengen $\tilde{\mathcal{S}} = \{v_1, \ldots, v_{n/2}\}$ und $\tilde{\mathcal{T}} = \{v_{n/2+1}, \ldots, v_n\}$, und wird für diese Teilmengen aufgerufen. Seien $\mathcal{S} = \{s_1, \ldots, s_k\}$ und $\mathcal{T} = \{t_1, \ldots, t_l\}$ die so gefundenen Mengen der Pareto-Optima aus $\tilde{\mathcal{S}}$ und $\tilde{\mathcal{T}}$. Da $\mathcal{V}$ aufsteigend nach erster Komponente sortiert ist, müssen wir nun lediglich die Vektoren aus $\mathcal{T}$ finden und entfernen, die von mindestens einem Vektor aus $\mathcal{S}$ dominiert werden. Die Mengen $\mathcal{S}$ und $\mathcal{T}$ müssen für die folgenden Schritte nach der ersten Komponente der enthaltenen Vektoren sortiert sein, was jedoch durch die vorgegebene Sortierung der Menge $\mathcal{V}$ keinen Mehraufwand bedeutet.

Der Algorithmus verfährt nun weiter, indem er $\mathcal{S}$ mit Hilfe eines Pivot-Elements $s_p = s_{k/2}$ weiter in die Mengen $\mathcal{S}_1 = \{s_1, \ldots, s_p\}$ und $\mathcal{S}_2 = \{s_{p+1}, \ldots, s_k\}$ zerlegt. Ebenso wird die Menge $\mathcal{T}$ durch das Pivot-Element s_p, genauer durch seine erste Komponente $r \in \mathbb{R}$, in die Mengen $\mathcal{T}_1$ der Elemente aus $\mathcal{T}$, deren erstes Element kleiner oder gleich r ist, und $\mathcal{T}_2 = \mathcal{T} \setminus \mathcal{T}_1$ zerlegt. So folgt direkt, dass kein Element der Menge $\mathcal{S}_2$ ein Element der Menge $\mathcal{T}_1$ dominieren kann, denn die erste Komponente jedes Vektors aus $\mathcal{S}_2$ ist bereits echt größer als diejenigen aus $\mathcal{T}_1$. Wir müssen nun also lediglich für die Paarungen $(\mathcal{S}_1, \mathcal{T}_1)$, $(\mathcal{S}_1, \mathcal{T}_2)$ und $(\mathcal{S}_2, \mathcal{T}_2)$ dominierte Vektoren finden. Ferner können wir bei der Paarung $(\mathcal{S}_1, \mathcal{T}_2)$ die erste Komponente der enthaltenen Vektoren

außer Acht lassen, da jeder Vektor aus $\mathcal{T}_2$ in seiner ersten Komponente echt größer ist als diejenigen aus $\mathcal{S}_1$. Die Dimension des durch diese Paarung gegebenen Teilproblems reduziert sich also um 1. Die gesamte Lösung entsteht durch die Vereinigung von $\mathcal{S}$, der Lösung von $(\mathcal{S}_1, \mathcal{T}_1)$ und dem Schnitt der Lösungen für die Paarungen $(\mathcal{S}_1, \mathcal{T}_2)$ und $(\mathcal{S}_2, \mathcal{T}_2)$. Sind die letzten beiden dieser Mengen konstruiert, entsteht die nach erster Komponente sortierte Lösung durch einfaches Aneinanderhängen. Es fehlt nun noch lediglich, den Rekursionsschluss zu behandeln.
Es gibt aufgrund der verschiedenen rekursiven Teilprobleme, die gelöst werden, verschiedene Rekursionsschlüsse, die unterschiedlich gelöst werden. Einfach ist es, den Rekursionsschluss zu lösen, wenn in einer der Mengen $\mathcal{S}$ oder $\mathcal{T}$ nicht mehr als ein Element liegt. Ist eine der Mengen leer, wird die jeweils andere zurückgegeben. Enthält eine der Mengen höchstens ein Element, wird dieses mit jedem Element der anderen Menge verglichen, um die Lösungsmenge zu konstruieren. Zudem muss ein Rekursionsschluss für eine ausreichend kleine Dimension geschehen. Für genau zwei Dimensionen ist dies durch Sortieren lösbar. Sortiert man eine Menge von zweidimensionalen Vektoren nach der ersten Komponente aufsteigend, so sind die Vektoren nach der zweiten Komponente absteigend sortiert, sofern alle Vektoren in dieser Menge Pareto-optimal sind. An jeder Stelle, an der diese absteigende Sortierung der zweiten Komponente gebrochen wird, findet man einen dominierten Vektor. In [11] wird ein Verfahren für drei Dimensionen vorgestellt, welches wir jedoch in modifizierter Form implementierten.
Das Verfahren erhält nun zwei Mengen $\mathcal{S}$ und $\mathcal{T}$ dreidimensionaler Vektoren, die nach ihrer ersten Komponente aufsteigend sortiert sind. Wir gehen nun davon aus, dass wir eine Menge $V = \{v_1, \ldots, v_n\}$ der Vektoren aus $\mathcal{S} \cup \mathcal{T}$ haben, die aufsteigend nach erster Komponente sortiert ist. Diese Reihenfolge kann in Linearzeit bezüglich der Größe der Mengen hergeleitet werden. Wir verwenden nun einen AVL-Baum τ, in dem Vektoren hinsichtlich der zweiten Komponente gehalten werden. Beim Einordnen eines Vektors in den AVL-Baum wird ein Vektor also rechts von der Wurzel eingeordnet, wenn seine zweite Komponente echt größer ist als die des Elements in der Wurzel selbst. Mithilfe des AVL-Baums können wir für eine Menge R von zweidimensionalen Vektoren schnell herausfinden, ob einer der Vektoren aus R einen gegebenen Vektor r dominiert. Der Zusammenhang der Menge R zu unserer Menge V wird später erläutert. Um in der Menge R einen Vektor zu finden, der r dominiert, kontrollieren wir, ob der Vektor in der Wurzel von τ den Vektor r dominiert. Ist dies

der Fall, sind wir fertig. Rekursiv testen wir, ob ein Vektor aus dem linken Teilbaum von τ den Vektor r dominiert. Falls r rechts von der Wurzel von τ angeordnet werden sollte, kontrollieren wir zudem rekursiv, ob im rechten Teilbaum von τ ein Vektor ist, der r dominiert.
Wir bearbeiten nun die Vektoren $v_1, \ldots, v_n$ nun nacheinander in der Reihenfolge der aufsteigenden Sortierung nach ihrer ersten Komponente, und verwenden einen AVL-Baum wie oben beschrieben für die zweite und dritte Komponente der Vektoren. Stammt der Vektor $v_i, 1 \leq i \leq n$ aus der Menge $\mathcal{S}$, wird dieser nicht dominiert, also in τ eingefügt. Stammt v_i aus $\mathcal{T}$, wird kontrolliert, ob τ einen Vektor enthält, der v_i dominiert. Ist dies der Fall, wird v_i nicht in die Lösungsmenge aufgenommen, da er auch in seiner ersten Komponente größer ist als jeder bisher betrachtete Vektor. Wird v_i nicht dominiert, so wird er in die Lösungsmenge aufgenommen, da jeder später betrachtete Vektor $v_j, j \gneq i$ in seiner ersten Komponente größer ist als v_i und diesen damit nicht dominieren kann. Genauer wird die Korrektheit in [11] mitsamt einer Laufzeitanalyse erläutert.

D

Konsole

Der *Control Server* besaß eine Konsole, um eine Steuerung ohne grafisches Interface zu ermöglichen. Diese wurde nicht mehr weiter entwickelt, als die Weboberfläche eingeführt wurde. Zum einen wurde sie nicht mehr benötigt, zum anderen fehlte die Zeit, um sie nebenher auf dem neusten Stand zu halten. Dennoch war sie ein Teil der Software und soll deshalb dokumentiert werden. Sie benutzte die Standardbibliotheken für Ein- und Ausgabe von Java und war somit weder abhängig vom eingesetzten Betriebssystem, noch von einer grafischen Darstellung einer Konsole in einem Fenster.

Die Architektur der Konsole wird in Abbildung D.1 dargestellt. Die zentrale Klasse ist `Console`, welche das *Java-Standardinterface* `Runnable` implementiert, um als eigener *Thread* ausgeführt werden zu können. Das Starten des *Threads* der Konsole geschah mittels der `run`-Methode, welche wiederum in einer Dauerschleife die Methode `handleInput` aufrief. Diese ermöglichte in einer Endlosschleife Eingaben durch den Benutzer. Die Eingabe wurde anhand von Leerzeichen getrennt und als *String Array* `signature` an die Methode `executeCommand` übergeben. Der erste *String* im *Array* war der gewählte Befehl, alle anderen waren die Parameter. Hier wurde nun die Validität des eingegebenen Befehls überprüft und bei Erfolg die zugehörige Implementierungsreferenz mittels der `getCommand`-Methode des `wie ConsoleCommandRegisters` geholt. Danach reichte ein Aufruf der `execute`-Methode mittels der `ConsoleCommand`-Referenz inklusive Parameter.

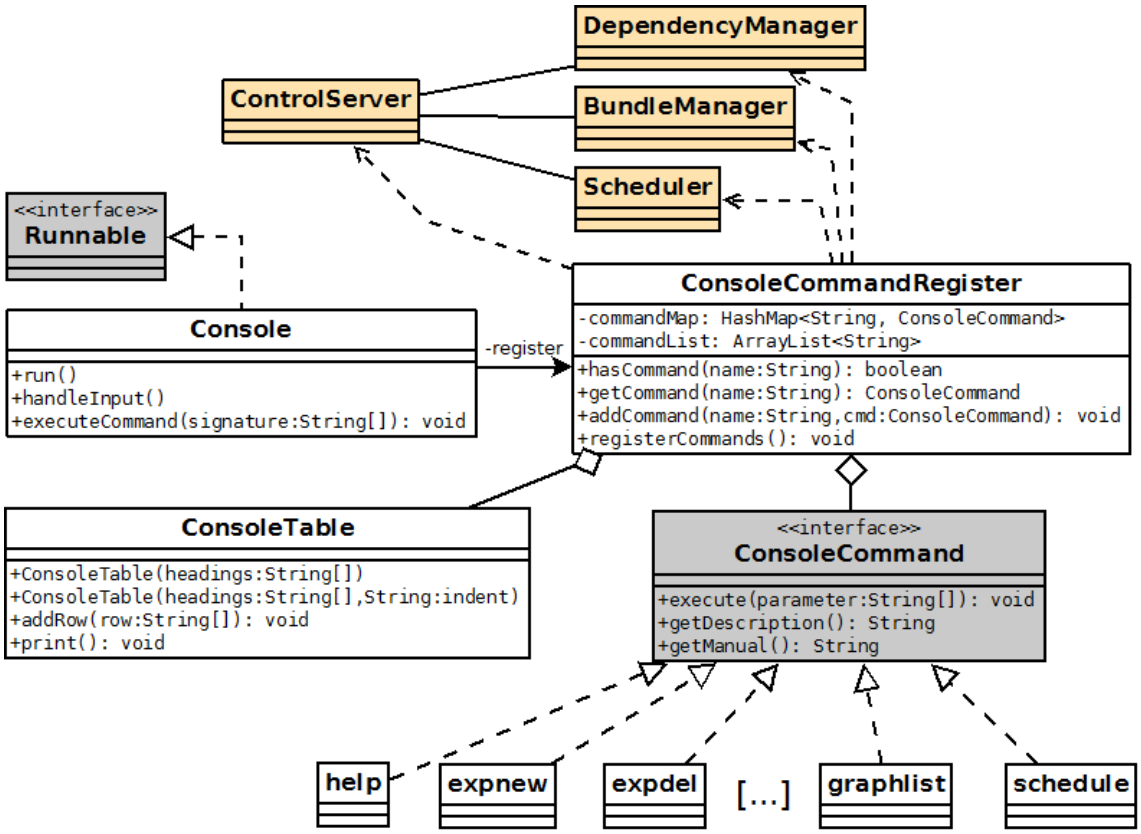

Abbildung D.1.: Klassendiagramm Konsole

Das `ConsoleCommandRegister` beinhaltete eine *Hash Map* `commandMap`, um die Stringrepräsentation eines Befehls effizient auf seine Implementierung abbilden zu können. Ein konkreter Befehl musste deshalb das Interface `ConsoleCommand` implementieren. Die damals vorhandenen Befehle werden unterhalb von `ConsoleCommand` in der Abbildung angedeutet. Außerdem pflegte das Register eine Liste mit der Bezeichnung `commandList`, welche eine alphabetisch sortierte Sammlung aller Befehle (beziehungsweise deren Bezeichner) beinhaltete. Mittels `has-`, `get-` und `addCommand` konnten Befehle geprüft, geholt oder hinzugefügt werden. In der Methode `registerCommands`, die beim Erstellen der Klasse mit ausgeführt wurde, wurden alle Standardbefehle erstellt und eingetragen. Da die Befehle stellenweise auch Auswirkungen auf andere Komponenten des *Control Servers* hatten, konnten die Befehle des Registers deren Schnittstellen nutzen (siehe ockerfarbene Klassen im Diagramm).

Das Interface `ConsoleCommand` schrieb drei Methoden vor, die zu implementieren waren. Während die Methode `execute` das Verhalten beinhaltete, gab `getDescription`

eine kurze Beschreibung und `getManual` eine ausführliche Erklärung der Nutzung des Befehls zurück. Die Nutzung eines Interfaces für die Befehle ermöglichte zum einen, dass die Implementierungen verschiedener Befehle klar getrennt und in Klassen gekapselt werden konnten. Zum anderen konnten so auch andere Teile des *Control Servers* eigene Befehle realisieren und mittels der `addCommand`-Methode anmelden. Hier war eine Verwandtschaft zu dem *Listener Pattern* von Java zu erkennen.

Die Klasse `ConsoleTable` stellte eine vereinfachte Möglichkeit dar, Informationen in der textbasierten Konsole in strukturierten, sich selbst in der Breite justierenden Tabellen darzustellen. Als Erstes musste ein Tabellenobjekt durch Übergabe eines *String Arrays*, welches die Überschriften beinhaltete, erzeugt werden. Danach konnte eine Zeile als *String Array* an `addRow` übergeben werden, wobei die Länge der Inhalte betrachtet und die Tabelle automatisch angepasst wurde. Zu kurze Arrays wurden als leere Zelle interpretiert, zu lange Arrays wurden abgeschnitten. Dadurch blieb die Tabelle robust in der Nutzung. Die Methode `print` stellte die Tabelle dann in der Konsole dar.

E

Ergebnisse der Experimente

Algorithmus Name	Graph Größe/ Dim	Graph-instanz	S-Metrik	Laufzeit [s]	Güte
Branch-and-Bound	4/2	1	260306	0,63	1,00
Branch-and-Bound	4/2	2	334505	1,36	1,00
Branch-and-Bound	4/2	3	142572	0,32	1,00
Branch-and-Bound	5/2	1	1039875	10,16	1,00
Branch-and-Bound	5/2	2	1062559	19,03	1,00
Branch-and-Bound	5/2	3	751661	5,52	1,00
k-Best	4/2	1	260306	17,49	1,00
k-Best	4/2	2	334505	22,03	1,00
k-Best	4/2	3	142572	13,74	1,00
Prüfer-EA	4/2	1	254198	18,78	0,9765
Prüfer-EA	4/2	1	251702	16,73	0,9669
Prüfer-EA	4/2	1	183184	15,12	0,7037

Algorithmus Name	Graph Größe/ Dim	Graph-instanz	S-Metrik	Laufzeit [s]	Güte
Prüfer-EA	4/2	2	334127	13,93	0,9989
Prüfer-EA	4/2	2	208322	14,65	0,6228
Prüfer-EA	4/2	2	185326	13,10	0,5540
	4/2	3	142572	16,55	1,00
PrueferEA	4/2	3	92848	12,47	0,6512
PrueferEA	4/2	3	137701	16,59	0,9658
PrueferEA	5/2	1	346531	30,16	0,3332
PrueferEA	5/2	1	121051	34,70	0,1164
PrueferEA	5/2	1	740493	33,76	0,7120
PrueferEA	5/2	2	352680	30,54	0,3319
PrueferEA	5/2	2	676051	21,79	0,6362
PrueferEA	5/2	2	636403	32,26	0,5989
PrueferEA	5/2	3	113855	32,20	0,1514
PrueferEA	5/2	3	522822	33,22	0,6956
PrueferEA	5/2	3	165395	31,48	0,2200
SMS-EMOA	4/2	1	217908	233,22	0,8371
SMS-EMOA	4/2	1	258878	489,62	0,9945
SMS-EMOA	4/2	1	259297	538,90	0,9961
SMS-EMOA	4/2	2	207621	995,00	0,6207
SMS-EMOA	4/2	2	334505	667,14	1,00
SMS-EMOA	4/2	2	273929	738,32	0,8189
SMS-EMOA	4/2	3	139622	878,20	0,9793
SMS-EMOA	4/2	3	142572	639,19	1,00
SMS-EMOA	4/2	3	142572	439,51	1,00
SMS-EMOA	5/2	1	669991	9,56	0,6442
SMS-EMOA	5/2	1	835054	7,19	0,8030
SMS-EMOA	5/2	1	1070726	7,44	1,0296
SMS-EMOA	5/2	2	616402	9,11	0,5801
SMS-EMOA	5/2	2	830235	7,76	0,7813
SMS-EMOA	5/2	2	592767	8,73	0,5578
SMS-EMOA	5/2	3	413605	26,89	0,8191

Algorithmus Name	Graph Größe/ Dim	Graph-instanz	S-Metrik	Laufzeit [s]	Güte
SMS-EMOA	5/2	3	299943	43,40	0,5502
SMS-EMOA	5/2	3	615712	108,15	0,3990
Label-Correcting	10/3	1	6174345183	0,16	1,00
Label-Correcting	10/3	2	3896190317	0,07	1,00
Label-Correcting	10/3	3	6526960445	0,21	1,00
Label-Correcting	20/2	1	11168898	0,25	1,00
Label-Correcting	20/2	2	9522736	0,18	1,00
Label-Correcting	20/2	3	11013240	0,25	1,00
Label-Correcting	20/3	1	33791212623	2,64	1,00
Label-Correcting	20/3	2	63565847285	9,60	1,00
Label-Correcting	20/3	3	48717342086	14,30	1,00
Label-Correcting	40/2	1	35099916	1,65	1,00
Label-Correcting	40/2	2	40467643	1,98	1,00
Label-Correcting	40/2	3	42509252	2,34	1,00
Label-Correcting-Merge	10/3	1	6174345183	0,16	1,00

Algorithmus Name	Graph Größe/ Dim	Graph-instanz	S-Metrik	Laufzeit [s]	Güte
Label-Correcting-Merge	10/3	2	3896190317	0,06	1,00
Label-Correcting-Merge	10/3	3	6526960445	0,16	1,00
Label-Correcting-Merge	20/2	1	11168898	0,29	1,00
Label-Correcting-Merge	20/2	2	9522736	0,23	1,00
Label-Correcting-Merge	20/2	3	11013240	0,24	1,00
Label-Correcting-Merge	20/3	1	33791212623	2,54	1,00
Label-Correcting-Merge	20/3	2	63565847285	10,83	1,00
Label-Correcting-Merge	20/3	3	48717342086	14,66	1,00
Label-Correcting-Merge	40/2	1	35099916	2,67	1,00
Label-Correcting-Merge	40/2	2	40467643	1,93	1,00

Algorithmus Name	Graph Größe/ Dim	Graph-instanz	S-Metrik	Laufzeit [s]	Güte
Label-Correcting-Merge	40/2	3	42509252	2,29	1,00
NAMOA	10/3	1	6174091473	0,53	0,9999
NAMOA	10/3	2	3896190317	0,20	1,00
NAMOA	10/3	3	6526960445	0,58	1,00
NAMOA	20/2	1	11168898	2,01	1,00
NAMOA	20/2	2	9522736	1,51	1,00
NAMOA	20/2	3	11013240	1,27	1,00
NAMOA	20/3	1	24399861303	235,55	0,7220
NAMOA	20/3	2	20751568643	71,42	0,3264
NAMOA	20/3	3	25211183230	175,36	0,5174
NAMOA	40/2	1	35099916	425,16	1,00
NAMOA	40/2	2	40464673	349,45	0,9999
NAMOA	40/2	3	42509252	396,24	1,00
SPEA2	10/3	1	2086590889	56,25	0,3379
SPEA2	10/3	1	3362360689	55,99	0,5445
SPEA2	10/3	1	1921665687	56,77	0,3112
SPEA2	10/3	2	148090530	58,16	0,0380
SPEA2	10/3	2	136231237	58,01	0,0349
SPEA2	10/3	2	178188911	59,32	0,0457
SPEA2	10/3	3	1286644346	59,70	0,1971
SPEA2	10/3	3	1242364122	56,63	0,1903
SPEA2	10/3	3	844383460	56,59	0,1293
SPEA2	20/3	1	5484380370	57,52	0,1623
SPEA2	20/3	1	10810270333	60,87	0,3199
SPEA2	20/3	1	9749322911	59,97	0,2885
SPEA2	20/3	2	10910016918	61,73	0,1716
SPEA2	20/3	2	25701035420	60,36	0,4043
SPEA2	20/3	2	12776520134	60,53	0,2009
SPEA2	20/3	3	19762769847	59,58	0,4056

Algorithmus Name	Graph Größe/ Dim	Graph-instanz	S-Metrik	Laufzeit [s]	Güte
SPEA2	20/3	3	14597585529	59,86	0,2996
SPEA2	20/3	3	15676426531	60,11	0,3217
SPEA2	20/2	1	2100924	50,98	0,1881
SPEA2	20/2	1	1968319	52,08	0,1762
SPEA2	20/2	1	2888045	52,96	0,2585
SPEA2	20/2	2	1985629	52,88	0,2085
SPEA2	20/2	2	2926647	51,59	0,3073
SPEA2	20/2	2	2324766	50,29	0,2441
SPEA2	20/2	3	968891	51,43	0,0879
SPEA2	20/2	3	2247487	51,17	0,2040
SPEA2	20/2	3	1485904	49,10	0,1349
SPEA2	40/2	1	2471887	58,30	0,0704
SPEA2	40/2	1	4384019	58,33	0,1249
SPEA2	40/2	1	2761943	59,31	0,0786
SPEA2	40/2	2	3115993	57,99	0,0769
SPEA2	40/2	2	4693879	57,94	0,1159
SPEA2	40/2	2	4411326	58,49	0,1090
SPEA2	40/2	3	8763814	56,02	0,2061
SPEA2	40/2	3	3670767	57,43	0,0863
SPEA2	40/2	3	2673754	59,87	0,0628

Tabelle E.1.: Laufzeit aller Algorithmen