Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Semesterarbeit SA-2003.14

Routing and Wavelength Assignment in All-Optical Ring Networks

Wintersemester 2002/2003
7. Februar 2003

Florian Süss
René Greminger

Supervisors: Stamatis Stefanakos
Prof. Dr. Thomas Erlebach

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Inhaltsverzeichnis

1 Einführung **1**
1.1 Optische Ringnetzwerke . . . 1
1.1.1 Gerichtete und ungerichtete Ringe . . . 1
1.1.2 Requests und Routing . . . 1
1.2 Wavelength Division Multiplexing . . . 1
1.3 Färbungen . . . 1
1.4 Aufgabenstellung . . . 2
1.5 Übersicht über die Arbeit . . . 3

2 Grundlagen **4**
2.1 LEDA . . . 4
2.2 CPLEX . . . 4
2.3 Workbench . . . 5
2.3.1 Struktur . . . 5
2.3.2 Schnittstelle zur Workbench . . . 5
2.3.3 Anpassungen an die Färbungsalgorithmen . . . 5
2.3.4 Implementierung von Färbungsalgorithmen . . . 7
2.4 Testprogramm . . . 8

3 Path Coloring and Routing Algorithmus (Kumar) **9**
3.1 Einleitung . . . 9
3.2 Theoretische Grundlagen . . . 9
3.3 Der Algorithmus . . . 10
3.3.1 Routing . . . 11
3.3.2 Formulierung des Linearen Programmes (LP) . . . 11
3.3.3 Lösen des Linearen Programmes . . . 13
3.3.4 Recoloring-Techniken . . . 13
3.3.5 Randomisiertes Runden und Path Stripping . . . 14
3.3.6 Greedy-Coloring . . . 15
3.4 Optimale Lösung mittels Integer Problem . . . 15
3.5 Leistung . . . 16

4 Path Coloring Algorithmus (Kumar) **17**
4.1 Einleitung . . . 17
4.2 Theorie . . . 17
4.3 Der Algorithmus . . . 18
4.3.1 Vorbereitungen . . . 19
4.3.2 Formulierung des Linearen Programmes (LP) . . . 19
4.3.3 Lösen des Linearen Programmes . . . 20

4.3.4 Randomisiertes Runden 20
4.3.5 Greedy Coloring 20
4.4 Optimale Lösung mittels Integer Problem 21
4.5 Leistung 21

5 Färbungsalgorithmus von Karapetian 22
5.1 Übersicht 22
5.2 Initialisierung 22
5.3 Bestimmung der maximalen Last 22
5.4 Färbung im Uhrzeigersinn 22
5.5 Färbung im Gegenuhrzeigersinn 23
5.6 Wiederholung der Prozedur 23

6 Tests 24
6.1 Durchführung 24
6.2 Schlussfolgerungen 26

7 Ausblick 27

Vorwort

Dieser Bericht enthält die Ergebnisse unserer Semesterarbeit am Institut für Technische Informatik und Kommunikationsnetze (TIK) der Eidgenössischen Technischen Hochschule Zürich (ETHZ). Diese Arbeit ermöglichte uns, Erfahrungen im Bereich der objektorientierten Programmierung zu sammeln. Es war zugleich auch eine grosse Herausforderung. Es war nicht immer einfach, die bereits vorhandenen Schnittstellen, Methoden und Datenstrukturen zu erkennen und zu verstehen. Schlussendlich konnten wir die Algorithmen jedoch ohne grössere Probleme in die bestehenden Programme einbinden. Am Schluss erreichten wir die Ziele der Arbeit, nicht zuletzt dank der guten Betreuung durch Stamatis Stefanakos und Prof. Dr. Erlebach, die uns bei Fragen und Problemen stets weiterhelfen konnten.

Florian Süss und René Greminger

1 Einführung

In der Praxis kommen immer häufiger optische Netzwerke zum Einsatz. Diese weisen oft eine ringförmige Struktur auf. Im Folgenden geben wir eine Einführung in den Aufbau und die Funktionsweise dieser *optischen Ringnetzwerke.*

1.1 Optische Ringnetzwerke

1.1.1 Gerichtete und ungerichtete Ringe

Es gibt *gerichtete und ungerichtete Ringe.* Bei *ungerichteten Ringen* existiert zwischen zwei Knoten nur eine Verbindung, welche in beide Richtungen genutzt wird. Bei *gerichteten Ringen* hingegen gibt es zwei parallele Verbindungen zwischen den Knoten, die nur in je einer Richtung betrieben werden können. In dieser Arbeit werden nur ungerichtete Ringe betrachtet.

1.1.2 Requests und Routing

Wird eine Verbindung zwischen zwei Knoten gewünscht, wird das in Form einer Anfrage, einem sogenannten *Request* dargestellt. Da die Daten vom Sender sowohl im Uhrzeiger- als auch im Gegenuhrzeigersinn an ihr Ziel gelangen können, muss der *Request* geroutet werden. Das heisst es wird ihm eine Umlaufrichtung zugewiesen. Diesen Prozess nennt man *Routing.*

1.2 Wavelength Division Multiplexing

Um mehrere Requests gleichzeitig zu verarbeiten, müssen die optischen Verbindungen Informationen parallel übertragen können. Dies wird durch die Verwendung verschiedener Wellenlängen erreicht. Dieses Verfahren heisst *Wavelength Division Multiplexing*, kurz *WDM.* Alle Requests, die ein Verbindungsstück des Ringes gemeinsam haben, dürfen nicht die gleiche Frequenz benutzen, da es sonst zu Interferenz kommt.

1.3 Färbungen

Die Frequenzen entsprechen im Folgenden Farben. Das Zuweisen von Frequenzen in einem optischen Ringnetzwerk kann also als Ringfärbungsproblem verstanden werden. Im allgemeinen gilt es, unter Verwendung möglichst weniger Farben, den Ring mit einer *gültigen Färbung* zu versehen. Unter einer *gültigen Färbung* versteht man eine Färbung, bei der für jede Kante des Ringes jede der darüberlaufenden Anfragen eine eigene Farbe

besitzt (Abb. 1).

Es gibt zwei Arten von Algorithmen zur Färbung eines Ringes:

- Algorithmen, welche ein Routing voraussetzen. In diesem Fall müssen die Requests vor der Anwendung des Färbungsalgorithmus mit einem Routingalgorithmus geroutet werden. Wir bezeichnen diese Algorithmen im Folgenden als *reine Färbungsalgorithmen*.

- Algorithmen, welche Routing und Färbung gleichzeitig durchführen. In diesem Falle kann man den Färbungsalgorithmus direkt auf den ungerouteten Ring anwenden.

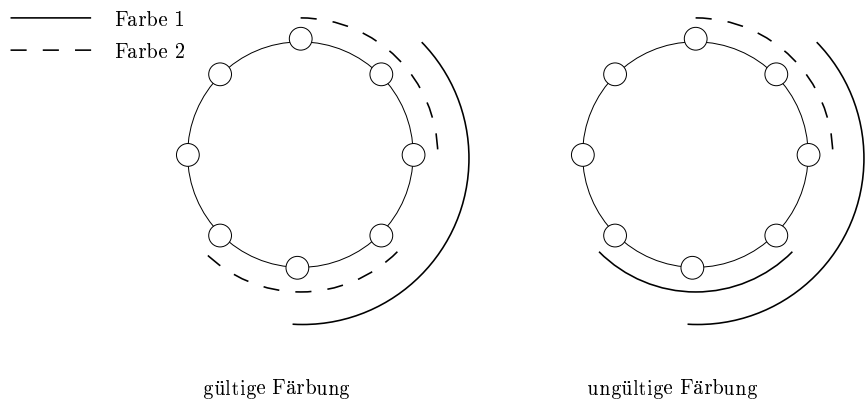

Abbildung 1: Gültige und ungültige Färbung

1.4 Aufgabenstellung

In Rahmen dieser Arbeit wurden folgende Aufgaben bearbeitet:

- Implementierung des *Path Coloring Algorithmus* von *Kumar* [4]. Dieser Algorithmus setzt ein vorgängiges Routing voraus.

- Implementierung des *Path Coloring and Routing Algorithmus* von *Kumar* [5]. Dieser Algorithmus führt Routing und Färbung gleichzeitig aus.

- Implementierung des *Path Coloring Algorithmus* von *Karapetian* [3]. Dieser Algorithmus setzt ebenfalls ein vorgängiges Routing voraus.

- Eingliederung der Algorithmen in die bestehende *Workbench* (Siehe Kap. 2.3) und Erweiterung derselben für die Verwendung von Färbungsalgorithmen.

- Verifizierung der Leistung der Algorithmen durch Experimente mit zufällig generierten Requests. Bei den reinen Färbungsalgorithmen interessiert auch die Abhängigkeit des Resultates vom gewählten Routingalgorithmus.

1.5 Übersicht über die Arbeit

- Im Kapitel 2 (Grundlagen) werden die Hilfsmittel und die bestehenden Programme erläutert, ebenso welche Änderungen nötig waren, um sie für die Färbungsalgorithmen kompatibel zu machen. Weiter enthält das Kapitel eine Anleitung zur Implementierung zusätzlicher Algorithmen.

- In den Kapiteln 3, 4 und 5 werden die einzelnen Algorithmen und deren Implementierung beschrieben.

- Das Kapitel 6 enthält Testresultate, welche die Leistungen der verschiedenen Algorithmen gegenüberstellen und vergleichen.

2 Grundlagen

2.1 LEDA

LEDA steht für *Library of Efficient Data types and Algorithms.* Wie der Name sagt, handelt es sich dabei um eine Bibliothek von Datenstrukturen und Algorithmen. Für unsere Algorithmen verwendeten wir die Pakete für lineare Listen, Iterationen und Graphen. Weitere Informationen dazu finden Sie in den LEDA-Handbüchern [7] [8].

2.2 CPLEX

CPLEX ist ein Programm zur Lösung linearer Optimierungsprobleme [9]. Das Problem muss folgendermassen angegeben werden:

$$
\begin{array}{lll}
\text{maximize (or minimize)} & c_1 * x_1 + c_2 * x_2 + \ldots + c_n * x_n & \\
\text{subject to} & a_{11} * x_1 + a_{12} * x_2 + \ldots + a_{1n} * x_n & \sim b_1 \\
 & a_{21} * x_1 + a_{22} * x_2 + \ldots + a_{2n} * x_n & \sim b_2 \\
 & \vdots & \vdots \\
 & a_{m1} * x_1 + a_{m2} * x_2 + \ldots + a_{mn} * x_n & \sim b_m \\
\text{with bounds} & l_1 \leq x_1 \leq u_1 & \\
 & \vdots & \\
 & l_i \leq x_i \leq u_i & \\
 & \vdots & \\
 & l_n \leq x_n \leq u_n &
\end{array}
$$

Dabei steht $\sim$ für $\leq$, $\geq$ oder $=$.

Das Problem gliedert sich in folgende drei Teile:

- In der ersten Zeile steht die zu minimierende respektive zu maximierende Funktion.
- Der Teil *subject to* enthält Bedingungen, wobei die Variablen links vom Vergleichsoperator, und die Konstante rechts davon stehen müssen.
- Der Teil *with bounds* enthält Grenzen, wobei für jede Variable eine obere und eine untere Grenze definiert werden kann.

Möchte man ein Problem ohne Funktion, also nur bezüglich der Bedingungen und Grenzen lösen, so kann für die Funktion eine Eins eingesetzt werden. Das Problem kann auf zwei Arten übergeben werden:

- In Form einer Textdatei, welche der *CPLEX*-Funktion übergeben wird.
- Direkt, das heisst die Variablen werden der *CPLEX*-Funktion als Werte übergeben.

2.3 Workbench

Die Workbench ist eine grafische Oberfläche, in der Ringnetzwerke dargestellt werden. Es können Requests eingezeichnet und anschliessend das Problem mit einem der implementierten Algorithmen gelöst werden. Das Resultat wird wiederum grafisch dargestellt. Die Workbench wurde im Rahmen von zwei früheren Semesterarbeiten [1] [2] erstellt.

2.3.1 Struktur

Die Workbench besteht aus drei Hauptelementen:

- Dem Hauptfenster (Abb.2), in welchem der Ring, wie auch die Requests und gegebenenfalls ihr Routing, grafisch dargestellt werden.
- Dem *Requestpanel* (Abb. 3), welches die Requests in aufgelisteter Form enthält. Man kann einzelne Requests hervorheben oder nicht anzeigen lassen. Zudem besteht die Möglichkeit, die Requests manuell zu routen.
- Dem *Algorithmpanel* (Abb. 3), welches die zur Verfügung stehenden Algorithmen auflistet. Der ausgewählte Algorithmus wird beim nächsten Ausführen der Run-Funktion angewendet.

2.3.2 Schnittstelle zur Workbench

Alle in die *Workbench* implementierten Algorithmen müssen von der abstrakten Klasse RPRAlgorithm abgeleitet sein. Die Funktion `int Run(RPRRing*)` dient als Schnittstelle zur Workbench. Sie wird als erstes ausgeführt, wenn der Algorithmus aufgerufen wird. Ihr wird ein Zeiger auf die Ringstruktur übergeben, mit dem auf alle Ringparameter und die Requests zugegriffen werden kann. Details zur Implementierung von Algorithmen in die Workbench findet man in [1]. Zusätzliche Hinweise zur Implementierung von Färbungsalgorithmen stehen in Kapitel 2.3.4.

2.3.3 Anpassungen an die Färbungsalgorithmen

Die Workbench unterstützte keine Farben, da sie ursprünglich für reine Routingalgorithmen konzipiert wurde. Dies erforderte einige Modifikationen, welche in den folgenden Abschnitten erläutert werden.

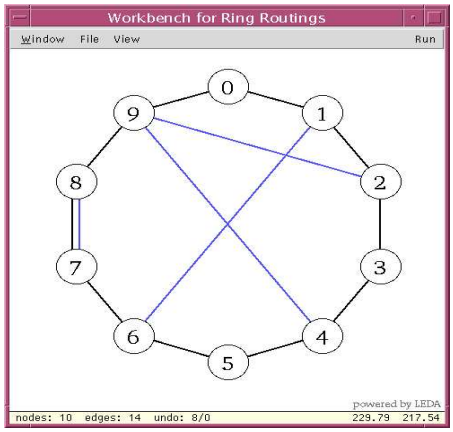

Abbildung 2: Die Workbench

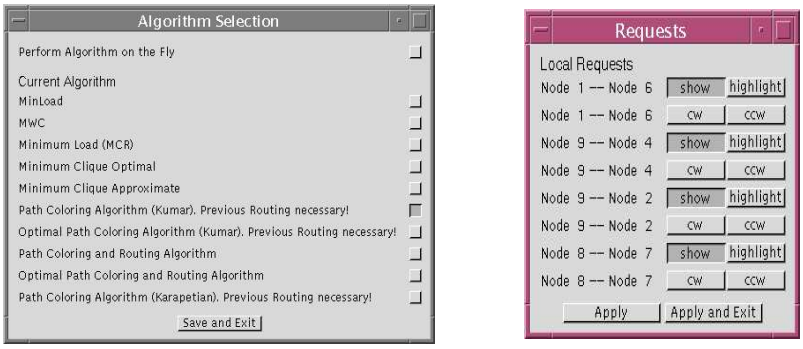

Abbildung 3: Algorithmpanel und Requestpanel

Erweiterung der Datenstrukturen Die Datenstruktur des Requests musste um eine Variable für die Farbe erweitert werden. Dies war nötig um den Requests überhaupt eine Farbe zuordnen zu können. Die Farben werden in Form von positiven Integerwerten gespeichert. -1 bedeutet, dass der Request ungefärbt ist.

Die Datenstruktur des Ringes musste um eine Boolean-Variable für den Färbungsstatus erweitert werden. Dies war nötig, um vor der grafischen Ausgabe des Ringes entscheiden zu können, ob die Prozedur zum Zeichnen des gefärbten oder des ungefärbten Ringes zu verwenden ist.

Darstellung von Farben Die Workbench war nicht dafür ausgelegt, gefärbte Requests anzuzeigen. Deshalb wurde die Funktion `RPRRingViewer::DrawArcs()` so erweitert, dass gefärbte Ringe dargestellt werden können. Beim Aufrufen der Funktion wird der Färbungsstatus des Ringes abgefragt. Ist der Ring gefärbt, so wird der neue Teil der Funktion ausgeführt, ansonsten der herkömmliche. Die gefärbten Requests werden so angezeigt, dass sich Requests mit der gleichen Farbe im gleichen Abstand vom Ring befinden.

Fehlerbehandlungen Es bedurfte einer neuen Fehlerbehandlung für den Fall, dass jemand versucht einen reinen Färbungsalgorithmus auf einem nicht gerouteten Problem auszuführen. Sollte dies geschehen, wird der Benutzer mit einem Dialogfenster darauf aufmerksam gemacht. Das Programm kann anschliessend normal weiterverwendet werden.

Manuelles Routing Bis anhin war es nur möglich, ein Problem mit einem der implementierten Routingalgorithmen zu routen. Es ist aber wichtig, reine Färbungsalgorithmen mit einem ganz bestimmten Routing testen zu können. Deshalb modifizierten wir das Requestpanel so, dass jeder Request manuell geroutet werden kann.

2.3.4 Implementierung von Färbungsalgorithmen

Bei Färbungsalgorithmen, welche ein Routing voraussetzen, muss zusätzlich eine if-Bedingung erweitert werden (siehe unten). Sie wird ausgeführt, wenn der ausgewählte Algorithmus ein vorgängiges Routing voraussetzt und überprüft, ob das Problem geroutet ist oder nicht. Sonst würde das Programm abstürzen, wenn der Algorithmus auf einem unvollständig gerouteten Problem ausgeführt wird.

Zur Implementierung des Algorithmus sind folgende Schritte nötig:

- Fügen Sie Ihren Algorithmus in die Liste der Algorithmen ein. Die Liste befindet sich in der Datei `RPRCactusBench.cpp` etwa bei Zeile 200.

- Geben Sie in der Datei `RPRCactusBench.h` die .h-Datei ihres Algorithmus an.

- In der Datei `RPRRingViewer.cpp`, in der Funktion `refresh()` finden Sie die Codezeile, wo die Algorithmen ausgeführt werden. Unmittel-

bar davor steht die if-Bedingung. Erweitern Sie diese um eine Oder-Verknüpfung mit der Nummer ihres Algorithmus.

2.4 Testprogramm

Um die Algorithmen effizient testen zu können, erstellten wir ein Testprogramm (`CACTest.cpp`), mit dem die Algorithmen unabhängig von der Workbench getestet werden können. Die Requests können zufällig generiert oder mit Hilfe eines Inputfiles eingelesen werden. Die reinen Färbungsalgorithmen können mit den Routingalgorithmen Minimum Load oder Minimum Clique kombiniert werden. Ansonsten werden sie auf einem zufällig gerouteten Problem ausgeführt. Die Resultate können in ein Textfile ausgegeben werden.

3 Path Coloring and Routing Algorithmus (Kumar)

3.1 Einleitung

Dieser Algorithmus routet und färbt ein Problem gleichzeitig und benötigt damit kein vorgängiges Routing. Das Problem des Routen und Färben in Ringen ist NP-vollständig. Es handelt sich hier um einen Approximationsalgorithmus, der maximal $1.5 + 1/2e + o(1)$ mal mehr Farben benötigt als die optimale Färbung. Der Algorithmus verwendet Wahrscheinlichkeitstheorien, randomisiertes Runden und Path Stripping. Das resultierende Problem wird als Lineares Programm beschrieben und mit *CPLEX* gelöst [9]. Dies kann in polynomieller Zeit erreicht werden. Entwickelt wurde dieser Algorithmus von Vijay Kumar [5] [6].

3.2 Theoretische Grundlagen

Dieser Abschnitt vermittelt einige Grundlagen, die später für das Verständnis des Algorithmus benötigt werden.

Ein *conflicting pair* von Bögen ist ein Paar von Bögen, in denen jeder Punkt des Ringes enthalten ist und die sich in mindestens einem Punkt überlappen (siehe Abbildung 4). Ein *Paralleles Routing* ist eine Sammlung von Bögen, die keine *conflicting pairs* enthält.

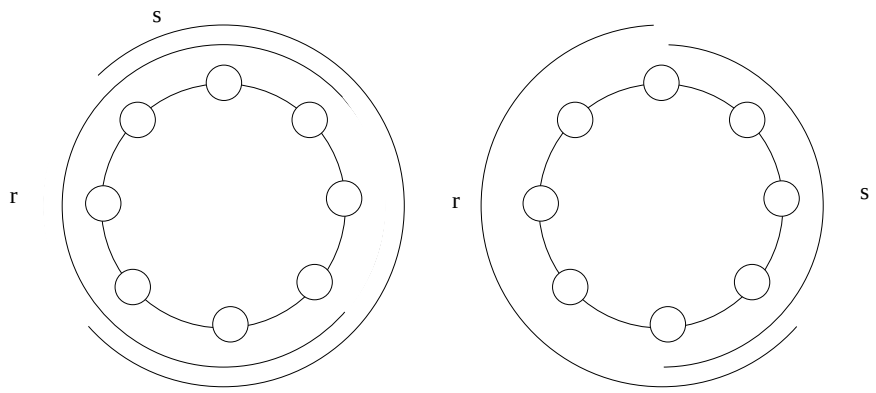

Abbildung 4: Requests r und s stellen ein *conflicting pair* von Bögen dar

Seien a und b zwei Requests aus einem parallelen Routing. Dann gibt es für jeden Link e auf dem Ring einen Link f, so dass weder a noch b beide

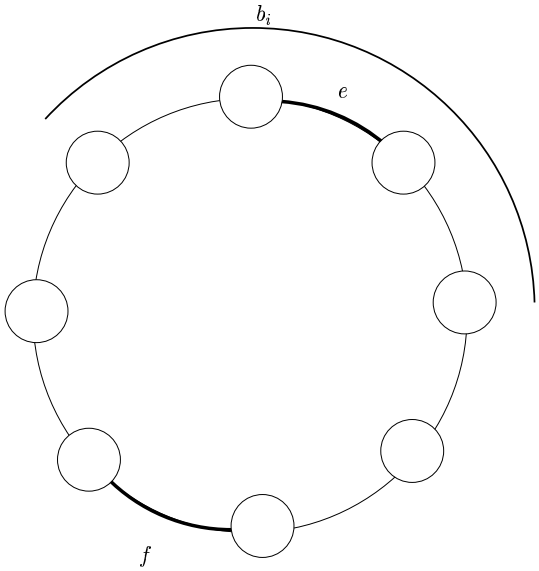

Abbildung 5: Links e und f, b_i: Crossover Request

Links e und f enthalten. Link e nennt man Komplement von Link f. Wenn man diese beiden Links entfernt, entstehen zwei Hälften des Ringes. Falls ein Request beide Endpunkte in einer Hälfte hat, wird er so geroutet, dass er komplett in dieser einen Hälfte liegt.

Ein Request, dessen Endpunkte in verschiedenen Hälften liegen, muss also immer entweder über Link e oder Link f geroutet werden. Solche Requests werden im folgenden *Crossover Request* genannt (siehe Abbildung 5).

3.3 Der Algorithmus

Der Funktion `int BARAlgorithm::run(RPRRing* ring)` wird ein Zeiger auf die Ringstruktur übergeben. Daraus werden die Requests ausgelesen und in der Liste *origReq* gespeichert. Zuerst wird nun eine Kopie der Requests-Liste (*requests*) angelegt. Alle Operationen, welche die Requests betreffen, werden auf der Kopie ausgeführt und erst nach Ausführung des gesamten Algorithmus auf die Originalliste übertragen.

Die Funktion `int BARAlgorithm::run(RPRRing* ring)` enthält nun folgende Routine, bei der ein Link e fest gewählt wird und für alle möglichen Links f eine komplette Färbung berechnet wird:

for all edges f **do**

Route if possible

Duplicate crossover requests

write and solve the linear program

if solution is fractional **do**

recoloring and randomized rounding

color the uncolored arcs with greedy-algorithm

return solution.

Die einzelnen Funktionen werden in den folgenden Abschnitten näher betrachtet.

Für jeden Link f wird eine Färbung berechnet und zwischengespeichert. Dabei wird also alles n mal ausgeführt und auch der zeitaufwändige *CPLEX*-Teil muss n mal aufgerufen werden. Vor Beendigung der Funktion `int BARAlgorithm::run(RPRRing* ring)` wird die Lösung, welche die wenigsten Farben verwendet, auf die Originalliste *origreq* übertragen, mittels der Funktion `void BARAlgorithm::TestIfValid(RPRRing* ring)` auf ihre Gültigkeit überprüft und zurückgegeben.

3.3.1 Routing

Hier werden die Requests, wenn möglich, so geroutet, dass sie weder Link e noch Link f enthalten (siehe Kapitel 3.2). Dies wird mit der Funktion `void BARAlgorithm::RouteIfPossible()` gemacht.

Alle noch nicht gerouteten Requests sind Crossover Requests (enthalten bei jedem möglichen Routing entweder Link e oder Link f). Die Funktion `void BARAlgorithm::RouteOtherReq(RPRRing* ring)` dupliziert diese Requests und routet diese in jeweils einer Richtung.

Die Funktion `int BARAlgorithm::getStatus(RPRRequest* Req)` gibt an, ob der Request *Req* über Link e (Returnwert 1), über Link f (Returnwert -1) oder ohne die beiden Links zu überqueren (Returnwert 0) geroutet ist.

3.3.2 Formulierung des Linearen Programmes (LP)

Definition der Variablen:

- $r_1, r_2, \ldots, r_l$ sind die l Crossover Requests. Jeder von diesen kann entweder als b_i (über Link e) oder als $\bar{b_i}$ (über Link f) geroutet werden.

- $a_1, a_2, \ldots, a_m$ sind alle m Requests, also mit allen b_i und $\bar{b_i}$.
- x_i ist 1 wenn r_i als b_i und 0 wenn r_i als $\bar{b_i}$ geroutet wird. x_i ist auch die Menge der Farbe i, die insgesamt verwendet wird.
- $y_{i,j}$ sind Variablen, die angeben, welche Menge der Farbe i ein Request a_j erhält. Die Farbe 0 steht für ungefärbt.
- g ist ein Link des Ringes.

Nun wird die Datei *CPLEXout* erstellt, die ein von CPLEX lesbares Format besitzt. Dies wird in der Funktion `void BARAlgorithm::writeCPLEXfile(RPRRing* ring)` gemacht.

Die *objective function* lautet:

- Minimize $u + \sum_i x_i$

Dabei werden folgende Bedingungen benutzt:

1. $\sum_{j=0}^{l} y_{i,j} = y_{i,k} = x_k$ $\quad\forall i$ falls $a_i = b_k$
2. $\sum_{j=0}^{l} y_{i,j} = 1 - x_k$ $\quad\forall i$ falls $a_i = \bar{b_k}$
3. $\sum_{j=0}^{l} y_{i,j} = 1$ $\quad$für alle anderen i
4. $\sum_{i:g \in a_i} y_{i,j} \leq x_j$ $\quad\forall g, \forall j \geq 1$
5. $\sum_{i:g \in a_i} y_{i,0} \leq u$ $\quad\forall g$

und folgende Grenzen gesetzt:

- $x_i \in [0, 1]$
- $y_{i,j} \in [0, 1]$

Die Bedingungen 1 bis 3 bestimmen die maximale Menge Farbe, die auf einem Request vorkommen darf. Bedingung 4 sagt, dass auf jedem Link maximal die in x_i vorgegebene Menge der Farbe i vorkommen darf. Die Variable u wird nur für die Definition der *objective function* benötigt. Zusätzlich kommt noch folgende Bedingung hinzu: i ist keine gültige Farbe für $\bar{b_i}$.

3.3.3 Lösen des Linearen Programmes

In der Funktion `double* BARAlgorithm::CPLEX(char* filename, RPRRing* ring)` wird aus der Datei *filename* das *CPLEX* LP-Problem eingelesen und eine *CPLEX* Umgebung geöffnet. Da *CPLEX* ein lizenzpflichtiges Programm ist, wird eine Fehlermeldung ausgegeben, falls keine Lizenz zur Verfügung steht.

Die *CPLEX*-Funktion `int CPXprimopt(env, probptr)` löst nun das LP-Problem und gibt allfällige Fehler zurück. Danach werden die Lösungen aller Variablen in einen array (*CPLEXsol*) gespeichert und mittels der Funktion `int BARAlgorithm::ResultTest(double* CPLEXsol)` auf allfällige Unregelmässigkeiten getestet.

3.3.4 Recoloring-Techniken

Wenn die Lösung des LP-Problems fraktionale Werte ergibt, müssen diese speziell behandelt und in eine ganzzahlige Lösung umgewandelt werden. Falls $x_i = 1$ ist, wird i als volle Farbe bezeichnet, bei $x_i = 0$ als ungebrauchte Farbe. Partielle Farben haben einen x_i-Wert zwischen 0 und 1. Nach Ausführung dieser Recoloring Techniken muss die Lösung folgenden zwei Aussagen genügen:

- Wenn i und j zwei partielle Farben sind, dann ist b_j nicht in b_i enthalten.
- Für jede Farbe i, die gebraucht wird, muss der zugehörige Request $\bar{b_i}$ vollkommen ungefärbt sein.

Dazu wird die Funktion `void BARAlgorithm::recoloring(double* CPLEXsol, RPRRing* ring)` aufgerufen. In einem ersten Schritt wird für zwei partielle Farben i und j die Hilfsfunktion `int BARAlgorithm::isContained(RPRRequest* Request1, RPRRequest* Request2)` ausgeführt, die zurückgibt, ob der *Request1* in *Request2* enthalten ist (siehe Abbildung 6). Falls ja, werden folgende Requests umgefärbt:

- Die Menge der Farbe j, also x_j, wird um $\delta = min\{x_i, 1 - x_j\}$ erhöht. Damit wird entweder x_i auf Null oder x_j auf 1 gesetzt.
- x_i wird um δ verkleinert.
- Da nun nicht mehr die gleiche Menge der Farbe i zur Verfügung steht, muss auf dem gesamten Kreisbogen von p_2 zu p_1 die Menge der benutzten Farbe i um δ reduziert werden. Im Verhältnis zur benutzten Farbe

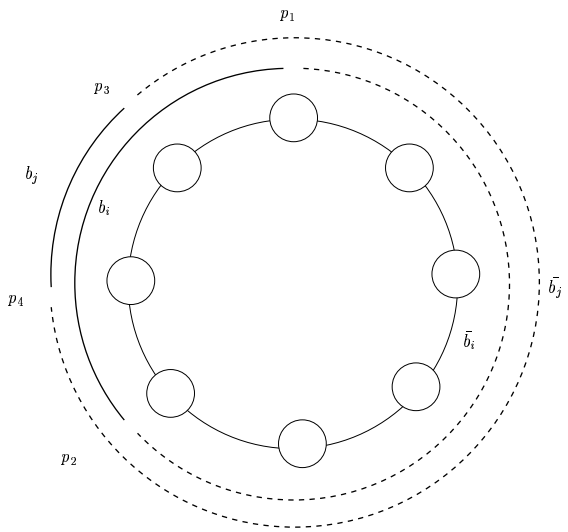

Abbildung 6: Request b_j ist in b_i enthalten

wird bei jedem Request, der die Farbe i verwendet, ein Teil dieser von Farbe i zu j getauscht. Damit wurde auch gleich die gebrauchte Menge der Farbe j auf dem Kreisbogen von p_2 zu p_1 um δ erhöht. Durch diese Vorgänge entstehen keinerlei Verletzungen der LP-Bedingungen.

- Nun müssen noch $\bar{b_i}$ und $\bar{b_j}$ angepasst werden: $\bar{b_j}$ wird um δ verkleinert. Die verschwindenden Farben können benutzt werden, um $\bar{b_i}$ um δ zu erhöhen.

Die aktuelle Lösung muss so umgewandelt werden, dass sie der 2. Bedingung genügt. α sei die totale Menge an Farbe auf dem Request $\bar{b_i}$. Es werden 2 Fälle unterschieden:

- $\alpha < x_i$: Jede Farbe j, die in $\bar{b_j}$ enthalten ist, wird um diese Menge reduziert.
- $\alpha \geq x_i$: Die Menge der Farbe i, also x_i, wird auf 0 gesetzt

Dies wird solange wiederholt, bis keine Farbe mehr die Bedingungen verletzt.

3.3.5 Randomisiertes Runden und Path Stripping

Als letzten Punkt der Anpassung muss aus der fraktionalen Lösung eine ganzzahlige gefunden werden, die möglichst nahe bei der optimalen Lösung liegt. Dazu wird die Funktion `void`

`BARAlgorithm::RamdomizedRounding(double* CPLEXsol, RPRRing* ring)` benötigt. Für jede Farbe i wird eine Zufallszahl *random* zwischen 0 und 1 bestimmt.

- Ist *random* grösser als x_i, wird die Farbe i nicht verwendet und auf allen Requests gelöscht.

- Andernfalls wird die verwendete Menge der Farbe i auf 1 gesetzt. Jetzt muss ein Pfad durch den Ring gefunden werden, der mit Farbe i gefärbt wird. Dazu wird der Endknoten von b_i betrachtet und mit diesem die Funktion `node BARAlgorithm::PathStripping(node end, int i)` aufgerufen. Diese liefert einen neuen Knoten *end* zurück. Damit wird die Funktion erneut aufgerufen, solange bis der berechnete Knoten *end* gleich dem Anfangsknoten von b_i ist.

Die Funktion `node BARAlgorithm::PathStripping(node end, int i)` bestimmt alle Requests, die mit Knoten *end* oder rechts davon beginnen und Farbe i verwenden. Daraus wird proportional zur Menge der Farbe i auf jedem Request zufällig einer ausgewählt, der dann die Farbe i erhält (falls er noch nicht gefärbt ist) und der Endknoten *end* dieses Requests wird zurückgegeben.

3.3.6 Greedy-Coloring

Nachdem die angepasste Lösung des LP-Problems auf die Requests übertragen wurde, verbleiben meistens einige Requests ungefärbt, welche mit einem einfachen Greedy-Algorithmus gefärbt werden. Dazu werden zusätzliche Farben verwendet. Man beginnt also beim Link e, weil darüber keine ungefärbten Requests laufen. Es wurden ja alle Crossover Requests entweder gefärbt oder in die andere Richtung geroutet. Man durchläuft den Ring in eine Richtung und gibt jedem ungefärbten Request die kleinste freie der zusätzlichen Farben.

3.4 Optimale Lösung mittels Integer Problem

Mit ein paar kleinen Änderungen am *CPLEX*-Inputfile und der Routine, die das Problem löst, kann das Färben der Requests optimal gelöst werden. Dabei müssen bei der Funktion `void BARAlgorithm::writeCPLEXfile(RPRRing* ring)` lediglich die Grenzen so geändert werden, dass nur noch binäre Zahlen als Lösung vorkommen können. Diese Lösung ist in der Klasse `BARopt` implementiert.

Das so entstehende Problem ist dann ein *Integer Problem* und kann mit der Funktion `CPXmipopt(env, probptr)` gelöst werden. Dann müssen die Recoloring-Techniken, Randomisiertes Runden und Path Stripping nicht mehr ausgeführt werden. Weiterhin nötig ist aber das Färben der verbleibenden ungefärbten Requests mittels Greedy-Coloring.

3.5 Leistung

Dieser Algorithmus benötigt, wie aus der Arbeit von Vijay Kumar [5] ersichtlich ist, mit grosser Wahrscheinlichkeit nicht mehr als $OPT(1.5+1/2e+o(1)) + O(\sqrt{OPT \ln n})$ Farben. OPT steht hier für die Anzahl benötigter Farben, wenn man das Problem optimal löst. Dies ist im Moment der beste bekannte Algorithmus für das Problem Routen und Färben in Ring-Netzwerken.

4 Path Coloring Algorithmus (Kumar)

4.1 Einleitung

Im Gegensatz zum Path Coloring and Routing Algorithmus in Kapitel 3 handelt es sich hier um einen reinen Färbungsalgorithmus. Alle Requests müssen also vorgängig geroutet werden. Entwickelt wurde auch dieser Algorithmus von Vijay Kumar [4] [6]. Dieses Problem ist ebenfalls NP-vollständig und kann nach gegenwärtigem Wissensstand in polynomieller Zeit nicht optimal gelöst werden. Bei einer genügend grossen Anzahl verwendeter Farben braucht eine Färbung mit diesem Algorithmus höchstens $1 + 1/e$ mal mehr Farben als die optimale Lösung.

Zusätzlich zu den schon in Kapitel 3 verwendeten Techniken wird neu das Multicommodity Flow Problem zur Lösung verwendet.

Aufgrund der Ähnlichkeit zum in Kapitel 3 vorgestellten Algorithmus werden einzelne Vorgänge nur noch kurz beschrieben.

4.2 Theorie

Um den Algorithmus zu beschreiben, werden einige theoretische Grundlagen benötigt.

Als Last bezeichnet man die Anzahl Requests, die über einem einzelnen Link des Ringes vorhanden sind. Mit *MaxLoad* ist die maximale Last auf einem Link bezeichnet. Wenn nun ein Set von Requests mit MaxLoad L gegeben ist, kann man auf jedem Link, der von weniger als L Requests verwendet wird, Einheitsrequests von der Form *(Startknoten: node, Endknoten: node+1)* hinzufügen, bis auch dieser Link die Last L besitzt. So bekommt man einen Ring mit einheitlicher Last L. Die Anzahl der benötigten Farben, um den Ring zu färben, wird dadurch nicht verändert.

Wird der Ring an einem Knoten P_0 zerschnitten und alle Requests, die über diesen Knoten laufen, zu zwei neuen Requests der Form $(start, P_0)$ und (P_0, end) zerlegt, wird aus dem Ring eine Kette, die sich konfliktfrei mit *MaxLoad* Farben färben lässt. Da dies aber für den Ring nicht zutrifft, wird für das neue Set von Requests das *Multicommodity Flow Model* erstellt (siehe Abbildung 7).

Jetzt wird jeweils die Farbe i bei s_i ins Netzwerk eingespeist und durchläuft dieses von s_i zu t_i. Dabei kann jeweils am Endknoten eines Requests auf einen anderen gewechselt werden, der bei diesem Knoten beginnt. Falls es keine gültige Lösung mit diesen Vorgaben gibt, muss das Modell um eine Ebene erweitert werden, was mittels Einfügen von Einheits-Requests geschieht. Dadurch steht eine zusätzliche Farbe zur Verfügung. Dies wird

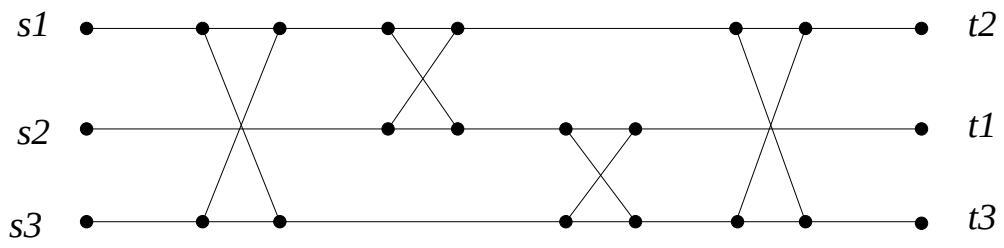

Abbildung 7: Multicommodity Flow Model

solange wiederholt, bis eine gültige Färbung entsteht.

Kumars Idee ist es, im Multicommodity Flow Model auch fraktionale Werte für die Menge der Farben (zwischen 0 und 1) zuzulassen, damit man das Problem mit linearer Programmierung in polynomieller Zeit lösen kann. Die so erhaltene Färbung ist dann aber nicht mehr optimal und es wird meistens einige Requests geben, die keine Farbe zugewiesen erhalten. Diese werden mit einem einfachen *Greedy*-Algorithmus mit zusätzlichen Farben gefärbt.

4.3 Der Algorithmus

Zuerst wird wieder eine Kopie der Requests-Liste erstellt (wie in 3.3). Um einfacheren Zugriff auf die Requests zu haben, werden diese zu Beginn alle im Uhrzeigersinn geroutet.

Vorbereitungen

while (true)

    write and solve the linear program

    if solution is valid **break**

    add a new color

if solution is fractional **do**

    randomized rounding

color the uncolored arcs with greedy-algorithm

return solution.

In den folgenden Kapiteln werden die einzelnen Schritte näher beschrieben.

4.3.1 Vorbereitungen

Um zu wissen, wieviele Farben mindestens gebraucht werden, muss die maximale Last *MaxLoad* auf einem Link des Ringes bestimmt werden (`int CACAlgorithm::FindMaxLoad()`).

Wie in Kapitel 4.2 erklärt, wird ein Schnittknoten *CutNode* bestimmt, und zwar dort, wo die Last minimal ist, damit auch die Anzahl der zusätzlich zu verwendenden Farben minimiert wird.

Die Funktion `void CACAlgorithm::Cut(node CutNode, RPRRing* ring)` führt folgende Aktionen aus:

- alle Requests *(StartNode, EndNode)* bestimmen, die über *CutNode* geroutet sind,
- diese durch die Requests *(StartNode, CutNode)* und *(CutNode, EndNode)* ersetzen,
- jeweils den zwei zusammengehörenden Requests die gleiche Farbe zuordnen, beginnend mit Farbe 0.

Nun wird *CutNode* von keinem Request mehr überquert. Die Requests, die zerschnitten wurden, haben die gleiche Farbe und können am Ende wieder zusammengesetzt werden. Als letzter vorbereitender Schritt wird der Ring mit Einheits-Requests von der Form *(Node, Node+1)* ergänzt, so dass überall auf dem Ring die gleiche Last *MaxLoad* vorhanden ist.

4.3.2 Formulierung des Linearen Programmes (LP)

Definition der Variablen:

- $x_{i,j}$ ist die Menge von Farbe j auf Request i.
- Es werden l verschiedene Farben verwendet.
- s bezeichnet den Startknoten und e den Endknoten eines Requests R.

Die *objective function* wird auf 1 gesetzt, weil in diesem Fall keine Funktion minimiert wird, sondern nur eine Lösung gefunden werden muss, die den folgenden Bedingungen genügt:

- Die totale Menge an Farbe auf jedem Request muss 1 sein.

  $\sum_{j=1}^{l} x_{i,j} = 1$

- Bei jedem Knoten muss die Menge Farbe, die auf dort endenden Requests frei wird, auf jene Requests verteilt werden, die dort beginnen.

  $\forall j, \forall nodeN : \sum_{\forall R:e=N} x_{R,j} = \sum_{\forall R:s=N} x_{R,j}$

- Alle Requests, die bei *CutNode* beginnen, bekommen eine unterschiedliche Farbe.

Folgende Grenzen werden gesetzt:

- $x_{i,j} \in [0, 1]$

4.3.3 Lösen des Linearen Programmes

Das beschriebene lineare Programm wird exakt auf die gleiche Weise gelöst wie in Kapitel 3.3.3 beschrieben.

Der Unterschied ist aber, dass das aktuelle Problem nicht immer lösbar ist, darum wird zuerst ermittelt, ob es eine gültige Lösung gibt. Wenn ja, wird diese berechnet und die **while**-Schlaufe in `int CACAlgorithm::run(RPRRing* ring)` verlassen. Gibt es keine Lösung, wird ein Satz von n Einheits-Requests hinzugefügt, was zur Folge hat, dass nun eine Farbe mehr zum Lösen des Problems zur Verfügung steht. Jetzt wird erneut das LP-Problem formuliert und gelöst. Dieser Vorgang wird solange wiederholt, bis eine gültige Färbung gefunden wird.

4.3.4 Randomisiertes Runden

Falls *CPLEX* eine Lösung mit fraktionalen Werten liefert, müssen diese durch Randomisiertes Runden in ganzzahlige umgewandelt werden.

Dabei wird, ähnlich wie in Kapitel 3.3.5, für jede Farbe ein Pfad, der den ganzen Ring beinhaltet, gewählt und allen Requests auf diesem, die noch nicht gefärbt sind, die jeweilige Farbe zugeteilt. Für die Auswahl des Pfades werden Zufallszahlen verwendet, so dass jeder Request, der als Fortsetztung in Frage kommt, im Verhältnis zu seinem fraktionalen Wert $x_{i,j}$ der *CPLEX*-Lösung berücksichtigt wird.

4.3.5 Greedy Coloring

Da die Auswahl der Requests auf Zufallszahlen basiert, bleiben meist einige der Requests ungefärbt. Diese werden mit einem Greedy-Algorithmus, der in Kapitel 3.3.6 beschrieben ist, gefärbt. Als Startknoten, über den keine ungefärbten Requests laufen dürfen, wird der Knoten *CutNode* verwendet.

4.4 Optimale Lösung mittels Integer Problem

Analog zu Kapitel 3.4 kann auch dieses Problem optimal gelöst werden. Die *CPLEX*-Lösung besteht dann nur noch aus ganzzahligen Werten, und damit werden alle Requests mit der optimalen Anzahl Farben gefärbt. Randomisiertes Runden und Greedy-Coloring müssen nicht mehr ausgeführt werden.

4.5 Leistung

Mit einer Wahrscheinlichkeit von mindestens $1 - 2/n$ braucht dieser Algorithmus nicht mehr als $OPT + d/e + 2\sqrt{d \ln n}$ Farben, wobei OPT die Anzahl Farben bei der optimalen Lösung, n die Anzahl Knoten und d die maximale Last bedeutet.

5 Färbungsalgorithmus von Karapetian

5.1 Übersicht

Der Karapetian-Algorithmus ist ein reiner Färbungsalgorithmus und setzt ein vorgängiges Routing voraus. Wir implementierten den ganzen Algorithmus als `Class Karapetian`. Die Schnittstelle zur Workbench bildet die Funktion `int run(RPRRing*)`. Sie wird als erste ausgeführt, wenn der Algorithmus gestartet wird. In ihr ist der Grobablauf des Algorithmus gegeben. Deshalb werden die meisten Unterfunktionen aus der Run-Funktion aufgerufen. Details zur Implementierung von Algorithmen in die Workbench finden Sie in [1]. Der Algorithmus ist in vier zeitlich getrennte Phasen gegliedert. Im Folgenden werden wir die vier Phasen erläutern, und jeweils auf deren Implementierung eingehen.

5.2 Initialisierung

Vor dem Ausführen des eigentlichen Algorithmus sind einige Initialisierungen notwendig. Die Requests müssen zwingend alle im Uhrzeigersinn geroutet sein, ansonsten funktioniert der Algorithmus nicht. Das Routing eines Requests kann umgekehrt werden, wenn man den Startknoten und den Endknoten vertauscht. Die Funktion `void SetAllClockwise()` setzt die Routings aller Requests auf 1 (Uhrzeigersinn).

5.3 Bestimmung der maximalen Last

Auf dem Ring gibt es eine oder mehrere Kanten mit maximaler Last. Das heisst über diese Kanten laufen am meisten Requests. Gibt es mehrere solche Kanten, wird eine davon zufällig ausgewählt. Dies sei *Kante* A mit *Last* G. Die Funktion `void FindMaxLoadNode(RPRRing*)` bestimmt Kante A und speichert ihre Position sowie die Last.

5.4 Färbung im Uhrzeigersinn

Requests, welche die Kante A nicht enthalten, heissen *Crossrequests*. In dieser Phase werden keine Crossrequests gefärbt. Der Ring wird $G/2$ mal im Uhrzeigersinn umlaufen. Ist G ungerade, so wird abgerundet. Für jeden Umlauf wird eine neue Farbe verwendet, beginnend bei Farbe 0. Während des Umlaufs werden bei jedem Knoten die Requests betrachtet, die dort ihren Startpunkt haben, noch nicht gefärbt sind und keine Crossrequests sind. Von diesen Requests wird einer zufällig ausgewählt und gefärbt. Danach wird der

Umlauf beim Endknoten des soeben gefärbten Requests fortgesetzt. Diese Phase ist in der Funktion `void cwColoring()` implementiert.

5.5 Färbung im Gegenuhrzeigersinn

In dieser Phase werden die restlichen Requests gefärbt. Der Ring wird diesmal im Gegenuhrzeigersinn umlaufen, und zwar G Mal. Wiederum wird für jeden Umlauf eine neue Farbe verwendet. Jeder Umlauf beginnt mit der Färbung eines Requests, der Kante A enthält. Und zwar mit demjenigen, der seinen Endpunkt am weitesten von A entfernt hat. Danach wird analog vorgegangen wie bei der Färbung im Uhrzeigersinn. Diesmal werden bei jedem Knoten die Requests betrachtet, die dort ihren Endpunkt haben, noch nicht gefärbt sind und den Crossrequest nicht überlappen. Wiederum wird von allen Requests, welche diese Kriterien erfüllen, zufällig einer ausgewählt und gefärbt. Danach wird der Umlauf beim Startknoten des soeben gefärbten Requests fortgesetzt. Diese Phase ist in der Funktion `void ccwColoring` implementiert.

5.6 Wiederholung der Prozedur

Möglicherweise reichen $3G/2$ Umläufe nicht aus, um das Problem vollständig zu färben. In diesem Falle wird dem Ring ein Satz von Elementaren Requests angefügt, welche lediglich über eine Kante laufen. Die maximale Last G erhöht sich dadurch um 1. Dann werden die Schritte zwei bis vier wiederholt, bis das Problem vollständig gefärbt werden konnte. Die Funktion `bool ColorCheck()` überprüft, ob das Problem vollständig gefärbt ist. Die Funktion `void addRequests()` fügt dem Ring einen Satz Elementar-Requests hinzu, welche durch die Funktion `void removeRequests()` am Schluss wieder entfernt werden.

6 Tests

6.1 Durchführung

Im Folgenden geht es darum, die Leistungsfähigkeit der Algorithmen zu vergleichen. Als Vergleichsgrösse dient dabei die Anzahl benötigter Farben, um eine bestimmte Anzahl Requests zu färben. Die Resultate sind von der Anzahl Knoten n des Ringes unabhängig, da wir die Tests mit zufällig generierten Requests durchführen. Diese weisen im Mittel die Länge $n/2$ auf. Wäre die mittlere Länge der Requests konstant, so dürfte man bei einer Erhöhung der Knotenzahl n eine Verminderung der Anzahl benötigter Farben erwarten. Jeder Wert in den Graphen ist ein Mittelwert von 10 durchgeführten Versuchen. Wir vergleichen folgende Algorithmen:

- Die drei reinen Färbungsalgorithmen mit zufällig gerouteten Requests. Siehe auch Kapitel 4 und 5.
- Den Path Coloring Algorithmus von Kumar [4] mit MinLoad [1] oder MinClique [2] als vorgängige Routingalgorithmen, sowie den Path Coloring and Routing Algorithmus [5].

Zudem vergleichen wir die Laufzeiten der Algorithmen von Kumar und Karapetian.

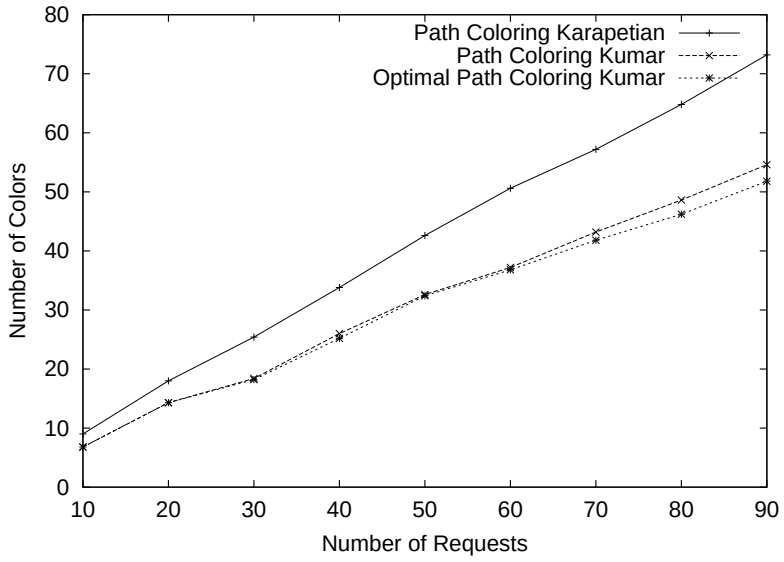

Abbildung 8: Färbungsalgorithmen mit zufälligem Routing

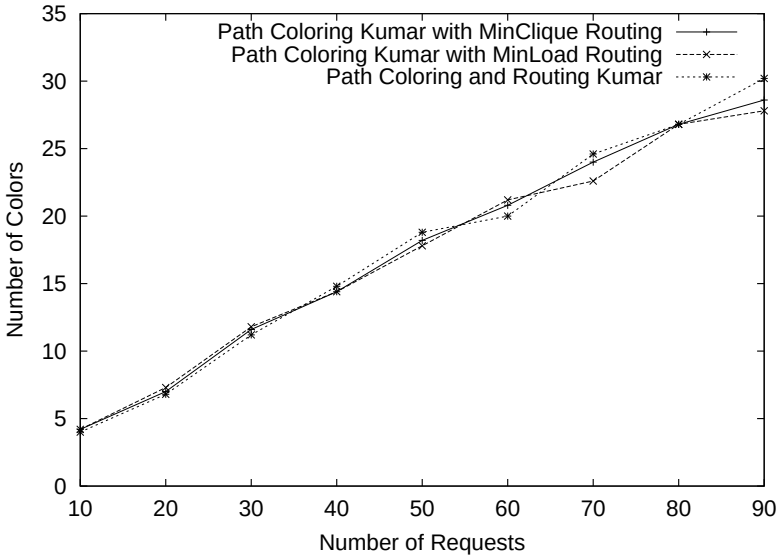

Abbildung 9: Algorithmen mit Routing

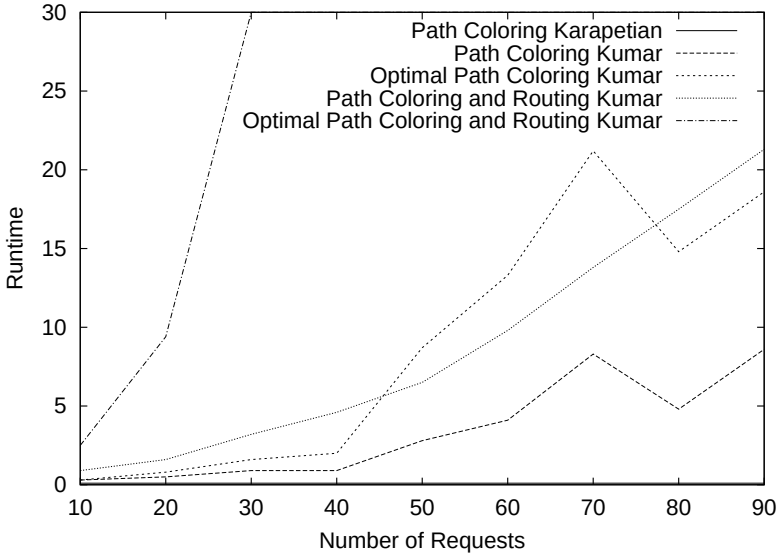

Abbildung 10: Laufzeiten der Färbungsalgorithmen

6.2 Schlussfolgerungen

- Der Algorithmus von Karapetian ist im Vergleich der schlechteste in Bezug auf die Anzahl benötigter Farben. Allerdings bleibt die Laufzeit mit zunehmender Anzahl Requests beinahe konstant bei etwa 0.1 Sekunden.

- Der optimale Path Coloring Algorithmus von Kumar benötigt leicht weniger Farben um das selbe Problem zu lösen als die normale Version, hat allerdings eine etwas grössere Laufzeit. Die Laufzeitdifferenz nimmt mit zunehmender Anzahl Requests zu, so dass für eine kleine Anzahl von Requests die optimale Version verwendet werden kann.

- Der Path Coloring and Routing Algorithmus weist in etwa die gleiche Leistung auf wie der Path Coloring Algorithmus von Kumar zusammen mit den reinen Routing Algorithmen MinLoad und MinClique.

- Die Laufzeit des optimalen Path Coloring and Routing Algorithmus steigt mit zunehmender Anzahl Requests so stark an, dass er bereits bei 30 Requests nicht mehr innert nützlicher Frist terminiert und somit nur auf kleine Probleme angewendet werden kann.

- Die beste Leistung wäre also zu erwarten, wenn man den MinClique Routing Algorithmus zusammen mit dem optimalen Path Coloring Algorithmus von Kumar verwendet.

7 Ausblick

Alle bis heute bekannten Algorithmen für Ringfärbungen sind nun in der Workbench implementiert. Nach wie vor offen bleibt aber die Implementierung und Betrachtung der drei Algorithmen für gerichtete Ringe. Weiter könnte man untersuchen, ob sich die Leistung der reinen Färbungsalgorithmen in Kombination mit anderen Routingalgorithmen verbessert.

Literatur

[1] Roman Vollenweider and Egon Burgener. *Routing Problems in Rings and Trees of Rings*, Semester project, ETH Zurich, SS 2002.

[2] Marcel Strittmatter and Mirjam Killer. *Minimum Clique Routing in Ring Networks*, Semester project, ETH Zurich, SS 2002.

[3] I.A. Karapetian. *On coloring of arc graphs.* Dokladi of the Academy of Science of the Armenian SSR, 70(5):306-311, 1980.

[4] Vijay Kumar. *An Approximation Algorithm for Circular Arc Coloring*, unpublished manuscript.

[5] Vijay Kumar. *Efficient Bandwith Allocation in Ring Networks*, unpublished manuscript.

[6] Vijay Kumar. *Approximating Circular Arc Coloring and Bandwith Allocation in All-Optical Ring Networks.* In *Proceedings of the International Workshop on Approximation Algorithms for Combinatorial Optimization APPROX'98*, LNCS 1444, pages 147-158, 1998.

[7] K. Mehlhorn and S. Näher. *LEDA: A platform for combinatorial and geometric computing*, Cambridge University Press, 1999.

[8] Algorithmic Solutions Software GmbH. *The LEDA User Manual*, Version 4.3, 2002. http://www.algorithmic-solutions.com.

[9] ILOG. *ILOG CPLEX 6.5 User's Manual, Reference Manual and Getting Started*, March, 1999. http://www.ilog.com/products/cplex/.