Technische Universität München
Institut für Informatik

Diplomarbeit am
Lehrstuhl für Effiziente Algorithmen

WWW-Visualisierung und Analyse von Push-Relabel-Flußalgorithmen

Klaus Holzapfel

Aufgabensteller: *Prof. Dr. Ernst W. Mayr*
Betreuer: *Dr. Thomas Erlebach*
Abgabedatum: *15. November 1999*

Inhaltsverzeichnis

Kapitel 1

Einleitung

Schon Goethes Zauberlehrling stand vor dem Problem, große Wassermengen zu transportieren. Er löste es mittels eines verzauberten Besens, der diese Aufgabe für ihn übernahm. Im alltäglichen Leben stehen uns leider keine solchen Zaubermittel zur Verfügung. Wir müssen uns daher selbst geeignete Verfahren ausdenken, wenn wir ein ähnliches Transportproblem effizient lösen wollen.
Ein solches alltägliches Problemen besteht beispielsweise darin, möglichst viel Öl in einem Pipelinesystem von der Quelle zum Zielort zu bringen. Auch für die Verkehrsplanung oder bei Massenveranstaltungen ist es von Interesse, ein ähnliches Problem zu lösen. Im Fall der Massenveranstaltungen geht es darum, zu ermitteln, wie alle Besucher in Gefahrensituationen am schnellsten aus dem Zentrum des Veranstaltungsortes durch die Gänge im Gebäude bzw. des Geländes zu den Ausgängen gelotst werden können.

In der Informatik können solche Probleme durch geeignete Übertragung auf gerichtete Graphen beschrieben werden. Die Aufgabe besteht jetzt darin, möglichst viele Flußeinheiten von einem Knoten des Graphens über dessen Kanten zu einem ausgezeichneten Zielknoten zu schicken. Die maximale Menge an Einheiten, die über einen realen Weg transportiert werden kann, wird durch die Verwendung von Kantenkapazitäten modelliert.
Bereits 1951 hat Dantzig einen Algorithmus vorgestellt, der dieses Problem lösen konnte. In zahlreichen Arbeiten wurde dieses Verfahren verbessert oder neue Methoden eingeführt. Besonders herauszuheben sind hierbei die Arbeiten von Ford und Fulkerson (1955) sowie Dinitz (1970). Diese verwendeten Methoden versuchen alle, immer wieder einen Pfad durch den Graphen zu finden, um noch mehr Einheiten in die Quelle zu verschieben. Erst im Jahre 1986 (publiziert 1988 [GT88]) entwickelten Goldberg und Tarjan einen neuen Algorithmus, der auf dem, 1974 von Karzanov eingeführten, Prinzip des Preflows arbeitet. Dieser sogenannte Push-Relabel-Algorithmus wurde später in vielen anderen Arbeiten aufgegriffen, modifiziert und weiterentwickelt. Bis heute dient er als Grundlage für die derzeit besten Algorithmen zur Bestimmung eines maximalen Flußes in Graphen.
In der Diplomarbeit wird dieser Algorithmus und einige mögliche Verbesserungen vorgestellt und analysiert. Im wesentlichen wird dabei die Auswahltechnik der Knoten während des Algorithmus untersucht. Die behandelten Verbesserungen ersetzen eine zufällige Auswahlreihenfolge durch Verfahren, die eine Warteschlange (*First In First Out*) bzw. eine durch Prioritäten geordnete Sortierung (*Highest Label First*) verwenden. Ebenfalls werden Strategien erläutert und untersucht, die die Komplexität nicht verringern, jedoch die Laufzeit des Algorithmus in vielen Fällen verkürzen.

Desweiteren wird noch ein im Rahmen dieser Diplomarbeit entwickeltes Client-Server-System zur Präsentation von Algorithmen im Internet vorgestellt. Mit diesem System ist es möglich, Algorithmen mittels eines Tutorials einem Benutzer zu erläutern und gleichzeitig Beispiele anzuzeigen. Desweiteren können die Abläufe der gegebenen Algorithmen visualisiert werden. Als ein Beispiel wird der Push-Relabel-Algorithmus in diesem System realisiert. Die Implementierung stellt neben der reinen Visualisierung noch Werkzeuge zum Generieren von Graphen sowie zum Vergleichen von mehreren durchgeführten Testläufen bereit.

Die Diplomarbeit ist in drei Teile gegliedert. Im ersten Teil (Kapitel 2) wird die theoretische Grundlage für den Algorithmus gelegt. Desweiteren werden der Algorithmus selbst sowie einige seiner Verbesserungen und Strategien erläutert. Der mittlere Teil (Kapitel 3) erläutert dann das implementierte System. Es werden dabei speziell die Möglichkeiten der Algorithmenvisualisierung, die Verwendung und Verwaltung des Systems sowie die Umsetzung des Push/Relabel-Algorithmus mit den erwähnten Werkzeugen beschrieben. Der letzte Teil (Kapitel 4) erläutert die vorgenommenen Testläufe zur Bestimmung eines optimierten Einsatzes der implementierten Verbesserungen und Strategien. Die Resultate aller Tests werden mit den Ergebnissen aus früheren Arbeiten über den Algorithmus verglichen.
Am Ende der Arbeit werden alle Kapitel nochmals zusammengefaßt und bewertet sowie Möglichkeiten zur Fortführung der Arbeit bzw. Erweiterung des Systems vorgeschlagen.

Kapitel 2

Push-Relabel-Algorithmus

In diesem Kapitel wird der Push-Relabel-Algorithmus von Goldberg und Tarjan [GT88] zur Bestimmung eines maximalen Flusses in einem Netzwerk beschrieben. Vor dieser Beschreibung werden zunächst einige Begriffe definiert (Abschnitt 2.1), anhand derer der Algorithmus erläutert wird (Abschnitt 2.2).
Dieser anfänglichen Darstellung folgen dann in Abschnitt 2.3 einige Verbesserungen des Algorithmus, mit denen eine bessere Komplexität erreicht werden kann. Abschließend werden noch einige Strategien vorgestellt, mit denen zwar keine Komplexitätsverbesserung möglich ist, bei geeigneter Verwendung jedoch eine Beschleunigung des Algorithmus erreicht werden kann.

2.1 Definitionen

Wie bereits oben erwähnt, dient der Push-Relabel-Algorithmus zur Bestimmung eines maximalen Flusses in einem Netzwerk. Bei einem Netzwerk handelt es sich um einen gerichteten Graphen mit gewichteten Kanten (Definition 2.1):

Definition 2.1 *Ein Netzwerk G besteht aus einem 5-Tupel (V, E, c, s, t). Die fünf Elemente haben folgende Bedeutung:*

- V — *die Knotenmenge des Netzwerks:* $|V| = n$
- E — *Menge von gerichteten Kanten:* $E \subset V \times V$, $|E| = m$
- c — *Kapazitätsfunktion, die jeder Kante e ein nicht-negatives ganzzahliges Gewicht* $c(e)$ *zuordnet:* $E \to \mathbb{N}_0$
- s — *die Quelle:* $s \in V$
- t — *die Senke:* $t \in V$

In Abbildung 2.1 ist ein Netzwerk dargestellt. Die Quelle und Senke sind durch die Buchstaben s und t gekennzeichnet. Die Kantenkapazitäten sind an den Kanten notiert. Es kann o.B.d.A. angenommen werden, daß jeder Knoten von der Quelle aus erreichbar ist und daß von jedem Knoten ein Weg zur Senke führt. Sollte dies nicht der Fall sein, so kann der Knoten und die zugehörigen Kanten aus dem Netzwerk entfernt werden, da er nicht zur Bildung des maximalen Flusses beitragen kann. D.h. für alle Komplexitätsbetrachtungen sollte man sich folgende Aussage vor Augen halten, $n - 1 \leq m \leq n^2$.

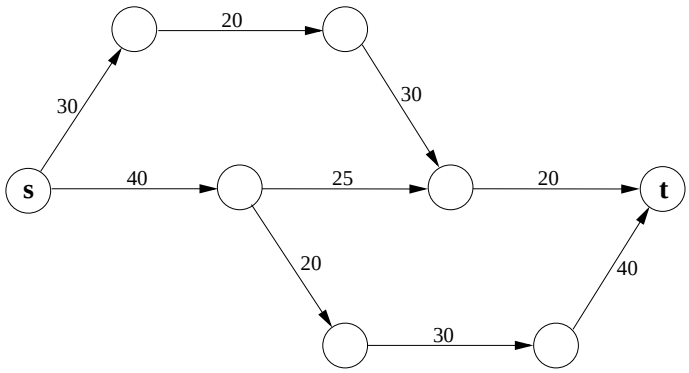


Abbildung 2.1 *Beispiel für ein Netzwerk*

Ein Fluß ordnet jeder Kante e einen Wert $f(e)$ (Fluß über die Kante e) zu, wobei der Wert die Kantenkapazität nicht übersteigen darf. Ferner muß für alle Knoten $v \in V \setminus \{s,t\}$ die Flußerhaltung gelten, d.h. die Summe der Flüsse über die Kanten, die in v endenden $\searrow(v) := E \cap \{(w,v) | w \in V\}$, muß gleich der Summe der Flüsse über die Kanten sein, die in v starten $\nearrow(v) := E \cap \{(v,w) | w \in V\}$. Formal kann dies wie folgt definiert werden:

Definition 2.2 *Ein (gültiger) Fluß f zu einem Netzwerk $G = (V, E, c, s, t)$ ist eine Funktion, für die gilt:*

- $f : E \to \mathbb{N}_0$
- $\forall\ e \in E\ .\ 0 \leq f(e) \leq c(e)$
- $\forall\ v \in V \setminus \{s,t\}\ .\ \sum_{e \in \searrow(v)} f(e) = \sum_{\bar{e} \in \nearrow(v)} f(\bar{e})$

Der Wert eines Flusses $|f_G|$ in einem Netzwerk G mit der Flußfunktion f_G ist definiert durch die Summe der Flüsse über die in der Senke t_G des Graphen ankommenden Kanten minus der Summe der Flüsse über die von der Senke weggehenden Kanten. Dieser Wert ist identisch mit der Summe der Flüsse der in der Quelle s_G des Graphen startenden Kanten minus der Summe der Flüsse der an der Quelle ankommenden Kanten. Formal ist dies in folgender Definition festgelegt.

Definition 2.3 *Der Wert eines gültigen Flusses f_G in dem Netzwerk $G = (V, E, c, s, t)$ berechnet sich wie folgt:*

$$|f_G| = \sum_{e \in \searrow(t_G)} f_G(e) - \sum_{\bar{e} \in \nearrow(t_G)} f_G(\bar{e}) = \sum_{e \in \nearrow(s_G)} f_G(e) - \sum_{\bar{e} \in \searrow(s_G)} f_G(\bar{e})$$

Da im folgenden immer eindeutig ist, von welchem Netzwerk die Rede ist, wird der Index G weggelassen. Auch wird meist vom Fluß statt vom Wert des Flusses gesprochen, wenn dies aus dem Zusammenhang klar erkennbar ist.
Nachdem nun die grundlegenden Definitionen gemacht wurden, kann der Begriff maximaler Fluß definiert werden. Ein maximaler Fluß in einem Netzwerk ist ein gültiger Fluß, so daß der Wert des Flusses unter allen gültigen Flüssen maximal ist.

Definition 2.4 *Ein Fluß f_{max} in einem Netzwerk G heißt maximal gdw.*

$$\forall\ f \in \text{Menge der gültigen Flüsse in G}\ .\ |f_{max}| \geq |f|$$

Der maximale Fluß eines Netzwerks muß nicht eindeutig sein. Zur Lösung des Problems des maximalen Flusses in einem Netzwerk genügt es, einen der maximalen Flüsse anzugeben, falls es mehrere solche gibt. In Abbildung 2.2 sind zwei mögliche maximale Flüsse zu dem bereits weiter oben beschriebenen Netzwerk zu sehen. An den Kanten sind zwei Werte angegeben, der erste Wert entspricht den Flußeinheiten und der zweite Wert der Kantenkapazität. Die zugeordneten Flußeinheiten sind ebenfalls durch die Dicke der Kanten symbolisiert, eine Kante, über die keine Flußeinheiten fließen, ist gepunktet gezeichnet.

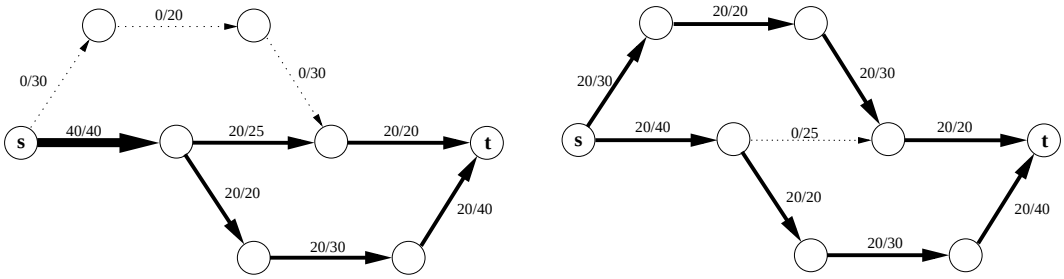


Abbildung 2.2 *Beispiel für maximale Flüsse*

Zur Lösung des maximalen Fluß Problems wurden viele Algorithmen gefunden [AMO93] [May98]. In Abbildung 2.3 werden eine Reihe von wichtigen Algorithmen mit zugehörigen Jahreszahlen, Referenzen und Entdeckern anhand der jeweiligen Zeitkomplexitäten gegenüber gestellt. Eine genauere Übersicht findet sich in den Arbeiten von Goldberg und Tarjan [GT88] und von Goldberg und Rao [GR98], denen auch die Daten der Abbildung entnommen sind.

Datum	Entdecker	Referenz	Komplexität
1970	Dinic	[Din70]	$O(n^2m)$
1974	Karzanov	[Kar74]	$O(n^3)$
1980	Sleator und Tarjan	[Sle80], [ST83]	$O(nm \log n)$
1986	Goldberg und Tarjan	[GT88]	$O(nm \log(n^2/m))$
1987	Ahuja und Orlin	[AO87]	$O(nm + n^2 \log U)$
1998	Goldberg und Rao	[GR98]	$O(m^{3/2} \log(n^2/m) \log U)$
			$O(n^{2/3}m \log(n^2/m) \log U)$

Abbildung 2.3 *Übersicht für Algorithmen für das maximale Fluß-Problem*

Viele der entwickelten Algorithmen zur Bestimmung des maximalen Flusses verwenden die Methode des Finden und Augmentierens von augmentierbaren Pfaden. Bereits 1974 hat Karzanov den Begriff des Präflusses eingeführt, mit dem sich eine andere Lösungsmethode realisieren läßt. Eine solche Methode wird in dem von Goldberg und Tarjan entwickelten Push-Relabel-Algorithmus verwendet.
Ein Präfluß $\tilde{f}$ ist eine Funktion, die den Kanten eines Netzwerks, wie bei der Definition des Flusses, einen nicht-negativen Wert $\tilde{f}(e)$ zuweist. Auch für einen Präfluß muß gelten,

daß der Wert, der einer Kante zugeordnet ist, nicht größer als die Kantenkapazität ist, also $0 \leq \tilde{f}(e) \leq c(e)$. Im Gegensatz zu einem normalen Fluß muß die Flußerhaltung (siehe Definition 2.2) nicht gelten. Es muß nur gewährleistet sein, daß in allen Knoten, bis auf die Quelle und die Senke, mindestens soviele Flußeinheiten ankommen wie aus dem Knoten herausfließen. Ein Präfluß definiert sich also wie folgt.

Definition 2.5 *Ein Präfluß $\tilde{f}$ zu einem Netzwerk $G = (V, E, c, s, t)$ ist eine Funktion, für die gilt:*

- $f : E \to \mathbb{N}_0$
- $\forall\ e \in E\ .\ 0 \leq \tilde{f}(e) \leq c(e)$
- $\forall\ v \in V \setminus \{s, t\}\ .\ \sum\limits_{e \in \searrow(v)} \tilde{f}(e) \geq \sum\limits_{\bar{e} \in \nearrow(v)} \tilde{f}(\bar{e})$

Durch diese Definition ist es möglich, daß in einen Knoten mehr Flußeinheiten fließen, als wieder abfließen. Es kann also eine weiter Funktion, die Überflußfunktion e (engl. excess) genannt wird, definiert werden.

Definition 2.6 *Sei $\tilde{f}$ ein Präfluß für das Netzwerk $G = (V, E, c, s, t)$, dann ist die Überflußfunktion e wie folgt definiert:*

- $e : V \to \mathbb{Z}$
- $e(v) = \sum\limits_{e \in \searrow(v)} \tilde{f}(e) - \sum\limits_{\bar{e} \in \nearrow(v)} \tilde{f}(\bar{e})$

Aus Definition 2.5 und 2.6 ergibt sich also, daß der Präfluß in keinem Knoten, abgesehen von Quelle und Senke, einen negativen Überfluß erzeugt. Ferner ist ein Präfluß, der in keinem Knoten, außer der Quelle und der Senke, einen Überfluß erzeugt, ein gültiger Fluß. Der Wert dieses Flusses ist dann gleich dem Wert $e(t) = -e(s)$.
Um den Algorithmus von Goldberg und Tarjan formalisieren zu können, müssen noch ein paar Begriffe eingeführt werden. Hierbei handelt es sich um den Begriff des Residuennetzwerkes und die Einführung einer später verwendeten Entfernungsfunktion.
Ein Residuennetzwerk G' wird zu einem Netzwerk G und einem gültigen Präfluß $\tilde{f}$ gebildet, indem für jeden Knoten eine Kopie im Residuennetzwerk angelegt wird und für jede Kante des Netzwerks maximal zwei Kanten im Residuennetzwerk eingefügt werden. Die erste Kante ist gleichgerichtet mit der ursprünglichen Kante und hat als Wert den Wert, der sich aus der Differenz der Kantenkapazität und der Flußeinheiten berechnet, die zweite Kante, antiparallel zur ursprünglichen Kante, hat den gleichen Wert wie die Flußeinheiten, die über die ursprüngliche Kante fließen. Wenn eine der Kanten den Wert 0 zugewiesen bekommt, wird die Kante nicht eingezeichnet. Dieses Prinzip ist in Abbildung 2.4 darstellt. Formal läßt sich ein Residuennetzwerk wie folgt definieren.

Definition 2.7 *Sei $G = (V, E, c, s, t)$ ein Netzwerk und $\tilde{f}$ ein Präfluß in diesem Netzwerk. Das zugehörige Residuennetzwerk $G' = (V', E', c', s', t')$ wird wie folgt gebildet.*

- *V' — für jeden Knoten aus V wird eine Kopie in V' angelegt: $|V| = |V'|$ und es existiert eine Bijektion $map: V \to V'$*

- E',c' — *Für jede Kante* $e = (v, w) \in E$ *werden folgende Kanten in* E' *gebildet:*
  $e_1 = (map(v), map(w))$ *mit* $c'(e_1) = c(e) - \tilde{f}(e)$, *falls* $\tilde{f}(e) < c(e)$
  $e_2 = (map(w), map(v))$ *mit* $c'(e_2) = \tilde{f}(e)$, *falls* $\tilde{f}(e) > 0$
- s' — $s' = map(s)$
- t' — $t' = map(t)$

Existieren in einem Netzwerk, das ja gerichtet ist, zwei Kanten zwischen zwei Knoten, dann kann es im zugehörigen Residuennetzwerk bis zu vier Kanten geben (siehe ebenfalls Abbildung 2.4). Im folgenden wird nicht mehr zwischen den Knoten im Netzwerk und den Kopien im Residuennetzwerk unterschieden. Die Bezeichnungen v und $map(v)$ werden als synonym betrachtet.
Die angesprochene Entfernungsfunktion d (engl. distance) ist eine Funktion, die jedem Knoten in einem Graphen einen Wert aus $\mathbb{N}_0$ zuweist. Die zulässigen Werte hängen von dem durch einen Fluß/Präfluß induzierten Residuennetzwerk ab (siehe Definition 2.8).

Definition 2.8 *Sei* $G = (V, E, c, s, t)$ *ein Graph und* $G' = (V', E', c', s', t')$ *das zugehörige Residuennetzwerk. Eine Funktion* d *heißt Entfernungsfunktion gdw.*

- $d: V \to \mathbb{N}_0$
- $d(t) = 0$
- $\forall\ e = (v, w) \in E'\ .\ d(v) \leq d(w) + 1$

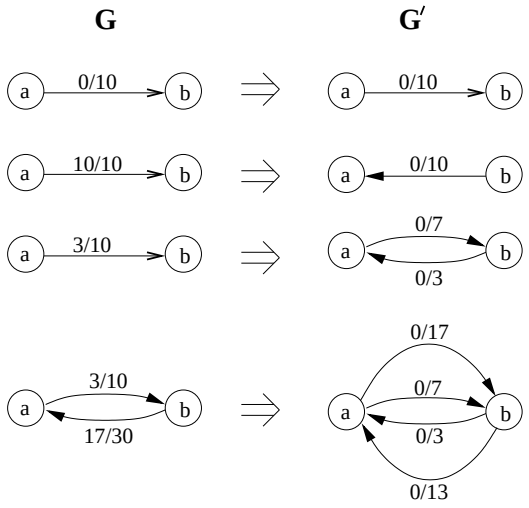


Abbildung 2.4 *Regeln zur Bildung des Residuennetzwerkes*

Eine gültige Entfernungsfunktion gibt für jeden Knoten eine untere Schranke für die Entfernung zum Zielknoten im zugehörigen Residuennetz an. Vor der genauen Beschreibung des Algorithmus sind noch zwei weitere Definitionen nötig:

Definition 2.9 *Sei* G *ein Graph,* G' *das zugehörige Residuennetzwerk und* d *eine Entfernungsfunktion. Eine Kante* $e = (v, w)$ *in* G' *heißt* zulässig *gdw.* $d(v) = d(w) + 1$ *und* $c'(e) > 0$.

Definition 2.10 *Sei $G = (V, E, c, s, t)$ ein Graph und f ein gültiger Präfluß in dem Graphen. Ein Knoten $v \in V \setminus \{s, t\}$ heißt* aktiv*, wenn f in v einen Überfluß hervorruft, d.h. $e(v) > 0$.*

Mit diesen Begriffen ist es nun möglich, den Algorithmus von Goldberg und Tarjan zu beschreiben.

2.2 Funktionsweise des Algorithmus

Bevor der Algorithmus nun formal beschrieben wird, soll der zugrundeliegende Gedanke dargestellt werden. Wie bereits weiter oben gesehen, kann ein Präfluß ein Fluß sein, wenn in keinem Knoten außer der Quelle und der Senke ein Überfluß existiert. Wandelt man also einen Präfluß so um, daß diese Eigenschaft erzielt wird, hat man einen Fluß gefunden.
Diese Erkenntnis wird in dem Algorithmus von Goldberg und Tarjan genutzt. Die Arbeit des Algorithmus kann man sich in zwei Wellen vorstellen. Bei der ersten werden von der Quelle maximal viele Flußeinheiten ausgegossen und möglichst weit in Richtung der Senke verschoben. Durch diesen Prozeß wird ein maximaler Präfluß erzeugt (der Wert eines Präflusses kann, analog zu dem Wert eines Flusses, als die Summe der in der Senke ankommenden Flußeinheiten definiert werden). Die zweite Welle schwappt zurück zur Quelle und bewirkt, daß nun alle überflüssigen Flußeinheiten, die sich noch im Netz befinden, wieder in die Quelle zurückgeschoben werden. Somit wird erreicht, daß der Überfluß in allen Knoten bis auf Quelle und Senke den Wert 0 annimmt, was nach obiger Aussage einer Umwandlung in einen Fluß gleichkommt. Da nach der ersten Welle der Überfluß in der Senke maximal war und in der zweiten Welle dieser Wert nicht verändert wurde, hat der entstehende Fluß den gleichen Wert wie der Präfluß.
Es ist offensichtlich, daß jeder Fluß auch ein Präfluß ist, woraus folgt, daß der Wert eines maximalen Präflusses mindestens so groß ist, wie der Wert eines maximalen Flusses. Andererseits kann jeder Präfluß leicht in einen Fluß mit gleichem Wert umgewandelt werden, indem alle überflüssigen Flußeinheiten in die Quelle zurückgeschoben werden. Hieraus folgt, daß der Wert eines maximalen Präflusses höchstens so groß ist wie der eines maximalen Flusses. Beide Aussagen ergeben zusammen, daß der Wert eines maximalen Präflusses gleich dem Wert eines maximalen Flusses ist. Für unseren Algorithmus bedeutet dies, daß es sich bei dem gefundenen Fluß um einen maximalen Fluß handelt.
Nach dieser ersten Intuition für den Algorithmus folgt nun die formale Beschreibung. In der Beschreibung wird auf die einzelnen Beweise verzichtet, um eine unnötige Länge zu vermeiden. Eine vollständige Erläuterung des Algorithmus mit allen Beweisen kann der Originalarbeit von Goldberg und Tarjan [GT88] entnommen werden.
Zur Initialisierung des Algorithmus wird zuerst das Residuennetzwerk gebildet. Der gesamte Algorithmus arbeitet auf diesem Netzwerk. Wenn der Fluß auf einer Kante verändert wird, muß also die Kapazität von zwei Kanten im Residuennetzwerk verändert werden. Nachdem das Residuennetzwerk aufgebaut ist, wird von der Quelle eine maximale Menge an Flußeinheiten ausgesendet, indem über alle ausgehenden Kanten die maximale Anzahl an Flußeinheiten gesendet wird. D.h. für den resultierenden Präfluß $\tilde{f}$ über diese Kanten e gilt $\tilde{f}(e) = c(e)$. Daß es sich um einen Präfluß handelt, ist offensichtlich. Zusätzlich wird noch die Entfernungsfunktion initialisiert. Durch eine Breitensuche, ausgehend von der Senke, kann jedem Knoten als Entfernung die Anzahl der Kanten auf einem kürzesten Weg zur Senke zugeordnet werden. Die Quelle erhält die Knotenanzahl

als Entfernungswert. Die so entstandene Entfernungsfunktion erfüllt die in Definition 2.8 angegebenen Bedingungen.
Nach dieser Initialisierung wird wie folgt verfahren. Durch den erzeugten Präfluß wird eine Menge von aktiven Knoten geschaffen. In der nun folgenden Hauptschleife wird jeweils ein aktiver Knoten herausgenommen, auf diesen Knoten eine sogenannte *Push-Relabel-Operation* angewendet (hierbei können neue aktive Knoten erzeugt werden, diese werden dann ebenfalls in die Menge aufgenommen). Falls am Ende der Operation der Knoten immer noch aktiv ist, wird er wieder zur Menge hinzugefügt. Diese Schleife wird solange ausgeführt, bis die Menge der aktiven Knoten leer ist. Das Grundgerüst des Algorithmus ist in Abbildung 2.5 in einem Pseudocode-Programm dargestellt.

```
Bestimme Knoten s und t;
Initialisieren des Residuennetzwerks;
Initialisieren des Praeflusses/Menge der aktive Knoten;
WHILE es existiert aktiver Knoten v DO
  push_relabel(v);
OD
```

Abbildung 2.5 *Grundgerüst des Push-Relabel-Algorithmus*

Während der *Push-Relabel-Operation* wird der Präfluß und die Entfernungsfunktion verändert, sie bleiben jedoch immer gültig. Wenn die Schleife terminiert, besitzt kein Knoten außer der Quelle und der Senke einen Überfluß, der Präfluß ist also in einen Fluß umgewandelt worden.
Die *Push-Relabel-Operation* auf einem Knoten v, auch für sie ist ein Pseudocode-Programm gegeben (siehe Abbildung 2.6), kann in zwei Fälle unterteilt werden:

1. Wenn im Knoten v Überfluß vorhanden ist und es eine ausgehende Kante $e' = (v, w)$ im Residuennetzwerk gibt, die zulässig ist, wird ein *Push* über eine dieser Kanten ausgeführt.
   Bei einem *Push* werden $\delta = \min\{e(v), c'(e')\}$ Flußeinheiten über die Kante geschoben. Wenn die Kante e' dabei parallel zur zugehörigen Kante im ursprünglichen Netzwerk verläuft, wird der Präfluß um δ Flußeinheiten erhöht, anderenfalls (Kanten verlaufen antiparallel) wird der Präfluß um δ Flußeinheiten erniedrigt.

2. Wenn im ersten Schritt keine *Push*-Operation ausgeführt werden konnte, wird ein *Relabel* auf den Knoten durchgeführt. Bei einer *Relabel*-Operation wird der Wert von $d(v)$ auf den neuen Wert $\min\{d(w) + 1\}$ erhöht, wobei w die Knoten sind, die im Residuennetz durch eine ausgehende Kante vom Knoten v erreicht werden können.

Aus obigen Definitionen folgt, daß für jeden aktiven Knoten entweder eine zulässige Kante im Residuennetzwerk existiert, also eine *Push*-Operation ausgeführt werden kann, oder durch eine *Relabel*-Operation der Wert der Entfernungsfunktion so abgeändert werden kann, daß eine zulässige Kante entsteht.

```
PROCEDURE push_relabel( Node v)
   Node w; Edge e';
BEGIN
```

```
  IF e(v) > 0 AND ex. zulaessige Residuenkante e' = (v,w) THEN
     delta := min( e(v),c(e') );
     CO fuehre push ueber die Kante e' aus OC
     e(v) := e(v) - delta;
     e(w) := e(w) + delta;
     Aktualisieren des Residuennetzwerks;
     Aktualisieren Menge der aktiven Knoten
  ELSE
     minDist = ininit;
     FORALL ueber Residuenkanten erreichbare Knoten w DO
       IF minDist > d(w) THEN
         minDist := d(w);
       FI
     OD
     d(v) := minDist + 1;
  FI
END push_relabel;
```

Abbildung 2.6 *Programmcode der Push-Relabel-Operation*

Eine erste Komplexitätsbetrachtung kann wie folgt vorgenommen werden. Für die Initialisierung des Algorithmus ist der Aufbau des Residuennetzwerks ($O(n+m)$) und die Initialisierung des Präflusses und der Entfernungsfunktion($O(m)$) nötig. Die Abarbeitung der aktiven Knoten wird in die *Relabel-* und die *Push*-Operationen unterschieden. Die *Push*-Operationen können ihrerseits in sättigende und nicht-sättigende Operationen aufgeteilt werden. Bei sättigenden Operationen wird im Gegensatz zu nicht-sättigenden Operationen die Restkapazität einer Kante voll ausgeschöpft. Über die Anzahl der drei Operationen kann man mit einigen Überlegungen folgende Theoreme beweisen:

Theorem 2.1 *Die Anzahl der* Relabel-*Operationen pro Knoten ist maximal* $2n - 1$. *Damit werden insgesamt höchstens* $(2n-1)(n-2) < 2n^2 = O(n^2)$ *solche Operationen ausgeführt.*

Beweis Eine Relabel-Operationen erhöht den Wert $d(v)$ mindestens um 1. Eine obere Schranke für $d(v)$ bildet also auch eine obere Schranke für die Anzahl der Relabel-Operationen pro Knoten. Die Idee des Beweises besteht darin zu zeigen, daß das Zurückfließen der Flußeinheiten in die Quelle auf Pfaden mit maximaler Länge $n-1$ stattfindet, daraus folgt $d(v) <= d(s) + (n-1)$. Mit dem Initialwert $d(s) = n$ ergibt sich darus die Behauptung für einen einzelnen Knoten. Ferner ist zu beachten, daß die Relabel-Operationen nicht auf die Quelle und die Senke ausgeführt wird. □

Theorem 2.2 *Die Anzahl der insgesamt ausgeführten sättigenden Push-Operationen ist maximal* $2nm = O(nm)$.

Beweis Über eine ursprüngliche Kante (v, w) kann über die zugehörigen Kanten im Residuennetzwerk nur dann ein sättigender *Push* verlaufen, wenn $d(v) = d(w) + 1$ bzw. $d(v+1) = d(w)$. Nach einem sättigenden *Push* muß also einer der Entfernungswerte mindestens um 2 steigen. Aus der Beschränktheit der Entfernungswerte kann man zeigen, daß für eine Kante im ursprünglichen Graphen maximal $2n$ sättigende *Push*-Operationen möglich sind. Mit insgesamt m Kanten folgt die Behauptung. □

Theorem 2.3 *Die Anzahl der insgesamt ausgeführten nicht-sättigenden Push-Operationen ist maximal* $4n^2m = O(n^2m)$.

Beweis Der Beweis wird über den Wert der Summe der Entfernungswerte der aktiven Knoten geführt. Jeder nicht-sättigende *Push* bewirkt eine Verminderung um 1. Eine sättigende *Push*-Operation erhöht den Wert um maximal $2n-1$. Hiermit und aus Theorem 2.2, sowie den Anfangsbedingungen der Entfernungsfunktion, kann die Behauptung gezeigt werden. □

Nimmt man also alle Theorem zusammen, so kann man für die Anzahl der Basis-Operationen des Algorithmus (d.h. *Relabel*- und *Push*-Operationen) eine obere Schranke von $O(n^2m)$ angeben. In dem nun folgenden Abschnitt werden nun einige Verbesserungen dieser oberen Schranke vorgestellt.

2.3 Verbesserungen

Die Verbesserung der Komplexität kann in zwei Schritten vorgenommen werden. Beim ersten Schritt wird die Reihenfolge bei der Auswahl der Knoten und Kanten, die bisher in beliebiger Reihenfolge stattfinden konnte, festgelegt (Teilabschnitt 2.3.1). Um eine weitere Verbesserung der Komplexität zu erzielen wird, die gewonnene Information über aufnahmebereite Kanten in einem Dynamic Tree gespeichert. Damit ist es möglich, die Push-Operation effizienter zu gestalten (Teilabschnitt 2.3.2).

2.3.1 Festlegung der Auswahlreihenfolge

Aus der obigen Beschreibung des Algorithmus geht keine Regel hervor, wie die aktiven Knoten und die zulässigen Kanten ausgewählt werden sollen. Der Algorithmus liefert das korrekte Ergebnis und terminiert für eine beliebige Auswahlreihenfolge. Im folgenden wird gezeigt, daß bei einer speziellen Wahl der Kanten bzw. Knoten die Laufzeit reduziert werden kann [GT88]. Eine spezielle Auswahl der Kanten und Knoten liefert nach obiger Aussage ebenfalls einen korrekten terminierenden Algorithmus.
Zuerst soll die Auswahl der Kanten betrachtet werden. Um eine geeignete Aussage treffen zu können, werden die Kanten, die von einem Knoten im Residuennetz ausgehen, in einer knoteneigenen Liste gespeichert. Zusätzlich wird jedem Knoten eine aktuelle Kante aus dieser Liste zugeordnet, anfänglich ist dies die erste Kante in der Liste. Bei einer *Push-Relabel*-Operation wird nun versucht, eine *Push*-Operation über die aktuelle Kante durchzuführen. Wenn die Kantenkapazität ausgeschöpft wird, wird die nächste Kante in der Liste zur aktuellen Kante bestimmt und falls sich noch immer Überfluß im Knoten befindet, wird diese Vorgehensweise solange fortgesetzt, bis entweder der Überfluß abgebaut ist oder die letzte Kante in der Kantenliste erreicht und gesättigt ist. Erst dann wird eine *Relabel*-Operation ausgeführt. Mittels dieser Vorgehensweise kann man nun zeigen, daß:

1. eine *Relabel*-Operation auf einen Knoten v nur dann ausgeführt wird, wenn keine *Push*-Operation mehr auf v anwendbar ist, und
2. die Kantenliste eines Knotens maximal $4n - 1 = O(n)$ mal durchlaufen wird.

Mit diesen Aussagen kann man nun eine andere Aussage über die Laufzeit des Algorithmus nachweisen, die jedoch in dieser Version noch keine Verbesserung der Komplexität bewirkt.

Theorem 2.4 *Der Push-Relabel-Algorithmus hat eine Laufzeit von $O(nm)$ plus $O(1)$ pro nicht-sättigender Push-Operation, also insgesamt $O(n^2m)$.*

Beweis Die Laufzeit abgesehen von den nicht-sättigenden *Push*-Operation berechnet sich aus der Summe der Anzahl der sättigenden *Push*-Operation ($O(nm)$) und der *Relabel*-Operationen ($O(n^2)$) sowie der Durchläufe der Kantenlisten. Die Kantenliste eines Knotens wird einmal vor und einmal während jeder der maximal $2n-1$ *Relabel*-Operationen des Knotens durchlaufen. Nach der letzten *Relabel*-Operation eines Knotens findet höchstens ein weiterer dieser Durchläufe statt. Hieraus ergeben sich die maximal Anzahl von $4n-1$ Durchläufe pro Knoten. Für alle Knoten folgt daraus eine obere Schranke der Größe $O(n^2)$. □

Aus diesem Theorem kann man ablesen, daß eine Verbesserung erzielt wird, wenn man die Anzahl der nichtsättigenden *Push*-Operationen weiter einschränkt. Im folgenden werden drei Möglichkeiten vorgestellt, wie diese Anzahl auf $O(n^3)$ zu beschränken. Die Methode der Verwaltung der aktiven Knoten in einer First-in-First-out (FIFO) Warteschlange wird näher beschrieben, die beiden anderen nur kurz erwähnt. Mit diesen Verbesserungen und obigem Theorem kann man dann folgendes Theorem zeigen:

Theorem 2.5 *Der Push-Relabel-Algorithmus kann durch Verwendung von FIFO-Warteschlangen oder der Highest-Label-First-Methode bzw. der Wellenmethode in einer Laufzeit von $O(n^3)$ implementiert werden.*

FIFO-Warteschlange

Um die aktiven Knoten in einer FIFO-Warteschlange zu verwalten, wird wie folgt verfahren. Bei der Initialisierung werden alle Knoten in die Warteschlange eingefügt. Bei einer *Push-Relabel*-Operation wird der erste Knoten aus der Warteschlange genommen und auf diesem die Operation ausgeführt. Alle hierbei entstehenden aktiven Knoten werden an das Ende der Warteschlange angehängt. Wenn die Operation beendet ist, ist entweder der Überfluß komplett abgebaut, oder es hat eine *Relabel*-Operation stattgefunden. Im ersten Fall wird der Algorithmus mit dem nächsten Knoten am Anfang der Warteschlange fortgesetzt. Im anderen Fall kommt es nun auf die Strategie an. Entweder kann der Knoten am Ende der Warteschlange angehängt werden, und dann wie gewöhnlich weitergemacht werden, oder es wird solange erneut die *Push-Relabel*-Operation fortgesetzt, bis der Überfluß vollständig abgebaut ist. Beide Varianten liefern die gleiche Schranke für den Algorithmus.
Für die Bestimmung der Schranke wird gezählt, wie viele Durchläufe höchstens auf der FIFO-Warteschlange stattfinden. Der erste Durchlauf dauert hierbei solange, bis alle Knoten, die nach der Initialisierung in der Warteschlange waren, aus dieser herausgenommen und abgearbeitet sind. Analog dazu dauert der Durchlauf $i+1$ solange, bis alle Knoten, die im Durchlauf i in die Warteschlange eingefügt wurden, wieder aus dieser herausgenommen wurden. Es kann gezeigt werden, daß höchstens $4n^2$ Durchläufe durch die Warteschlange stattfinden. Hierauf aufbauend kann folgendes Theorem gezeigt werden:

Theorem 2.6 *Die Anzahl der nicht-sättigenden Push-Operationen bei der Verwendung einer FIFO-Warteschlange zur Verwaltung der aktiven Knoten beträgt maximal $4n^3 = O(n^3)$.*

Beweis Für den Beweis wird der Wert $\Phi = \max\{d(v) | v \textit{ is active}\}$ nach jedem Durchlauf betrachtet. Der Wert von Φ ist zu Beginn und am Ende des Algorithmus 0. Der

Wert kann nur dann steigen oder gleich bleiben, wenn in dem Durchlauf eine *Relabel*-Operation stattfindet. Aus der Begrenzung Anzahl dieser Operationen in Theorem 2.1 ergeben sich also maximal $2n^2$ solcher Durchläufe. In allen anderen Durchläufen sinkt Φ mindestens um den Wert 1. Insgesamt finden also höchstens $2 \cdot 2n^2$ Durchläufe statt, wobei in jedem Durchlauf maximal n nicht-sättigenden Push-Operationen auftreten können. □

Highest Label First

Bei dieser Variante, die häufig *Highest Label First* (HLF) genannt wird, wird die *Push-Relabel*-Operation immer auf einen Knoten ausgeführt, der maximale Entfernung unter den aktiven Knoten besitzt. Ähnlich wie bei der Implementierung mittels einer FIFO-Warteschlange kann auch diese Variante mit einer speziellen Datenstruktur zur Verwaltung der aktiven Knoten implementiert werden. Es handelt sich dabei um eine gewichtete Warteschlange. D.h. es wird eine Menge von FIFO-Warteschlangen verwendet, wobei jeder möglichen Entfernung eine eigene Schlange zugeordnet wird. Wenn ein Knoten entnommen werden soll, wird er immer vom Anfang der Warteschlange entnommen, die unter den nicht leeren Warteschlangen dem größten Entfernungswert zugeordnet ist.
Ebenso wie bei der oben vorgestellten Variante kann gezeigt werden, daß maximal $O(n^3)$ nicht-sättigende *Push*-Operationen stattfinden können.

Wellen Methode

Als letzte Variante, die ebenfalls eine obere Schranke von $O(n^3)$ für die nicht-sättigenden *Push*-Operationen besitzt, ist die Wellen Methode zu nennen (siehe [GT90]). Ähnlich wie bei der FIFO-Methode existiert auch hier eine Liste von Knoten. Jetzt enthält die Liste allerdings nicht nur die aktiven Knoten sondern alle Knoten, ferner ist die Liste bezüglich der zulässigen Kanten topologisch geordnet (d.h. wenn eine zulässige Kante (v, w) existiert, dann befindet sich v in der Liste vor w). Nun wird in jeder "Welle" über die gesamte Liste gegangen. Findet man einen aktiven Knoten v, wird auf diesem solange eine *Push-Relabel*-Operation ausgeführt bis er entweder inaktiv wird, oder ein *Relabel* stattfindet. Im ersten Fall bleibt der Knoten an der gleichen Stelle und im zweiten Fall wird er an den Anfang der Liste gesetzt. Dies verletzt die Forderung nach der topologischen Sortierung nicht, da direkt nach einem *Relabel* keine zulässigen Kanten in einem Knoten ankommen können. Wurde in einer "Welle" kein Knoten verschoben, so wird der Algorithmus abgebrochen, anderenfalls wird eine neue Welle durchlaufen.
Analog zur FIFO-Methode kann gezeigt werden, daß maximal $O(n^2)$ "Wellen" möglich sind, da nur dann mit einer neuen Welle begonnen wird, wenn ein Knoten verschoben wird, was nur nach einem *Relabel* möglich ist. Somit können höchstens $O(n^3)$ nicht-sättigende *Push*-Operationen stattfinden.

2.3.2 Dynamic Trees

Nach dieser Verbesserung der Zeitkomplexität von $O(n^2m)$ auf $O(n^3)$ soll nun eine weitere Verringerung der Schranke beschrieben werden. In dieser Art der Implementierung kommen die von Sleator und Tarjan beschriebenen Dynamic Trees [ST83] zum Einsatz. Grundgedanke dieser Implementierung ist es, sich bereits gewonnene Information über aufnahmefähige Kanten eines Knotens zu merken. Es wird parallel zum Residuennetzwerk ein Wald von Dynamic Trees angelegt, in dem zu jedem Knoten ein Baumknoten

existiert. Im Laufe des Algorithmus werden nun Bäume aufgebaut, die folgende Eigenschaft besitzen. Von einem Blatt eines Baumes können entlang des Pfades vom Blatt bis zur Wurzel Flußeinheiten über zulässige Kanten verschoben werden. Wird nun im Algorithmus ein Knoten ausgewählt, werden mittels der Dynamic Trees die Flußeinheiten nicht nur zu einem Nachbarknoten, sondern falls möglich über mehrere Knoten hinweg in Richtung Wurzel verschoben. Um den Algorithmus zu implementieren, muß eine neue *Push-Relabel*-Operation verwendet werden, die die Einbindung der Dynamic Trees unterstützt. Der äußere Rahmen zur Verwaltung der Knoten in einer FIFO-Warteschlange oder einer ähnlichen Variante bleibt jedoch bestehen. Unter Ausnutzung der logarithmischen Komplexität von Operationen in den Dynamic Trees und einigen weiteren Überlegungen kann gezeigt werden, daß sich die Laufzeit des modifizierten Algorithmus wie folgt verhält:

Theorem 2.7 *Der Push-Relabel-Algorithmus mit Einsatz von FIFO-Warteschlangen und Dynamic Trees benötigt $O(nm \log k)$ Zeit plus $O(\log k)$ Zeit pro Einfügen in die Warteschlange.*

Beweis Der Beweis wird durch Abzählung der Operationen die im Lauf des Algorithmus auf dem Dynamic Trees geführt. Es kann gezeigt werden, daß $O(nm)$ plus Anzahl der Einfügungen in die Warteschlange mal eine solche Operation ausgeführt wird. Durch die Beschränkung der Größe der Bäume auf k, und der daraus folgenden amortisierten Komplexität von $O(\log k)$ pro Operation folgt die Behauptung. □

Mittels weiteren Überlegungen kann man zeigen, daß höchstens $O(nm + n^3/k)$ mal ein Knoten in die Warteschlange eingefügt wird. Hiermit kann folgende Laufzeit erzielt werden.

Theorem 2.8 *Der Push-Relabel-Algorithmus mit Einsatz von FIFO-Warteschlangen und Dynamic Trees läuft in $O(nm \log(n^2/m))$ Zeit, wenn die maximale Größe der Dynamic Trees k mit n^2/m gewählt wird.*

Wie man aus diesen Überlegungen ablesen kann, kann man durch den Einsatz der Dynamic Trees gerade bei leichten Netzwerken, d.h. wenn die Anzahl der Kanten nahe bei der Knotenanzahl liegt, eine deutliche Verbesserung der Komplexität gegenüber dem früher hergeleiteten Wert von $O(n^3)$ erzielen. Bei sehr dichten Netzen, d.h. $m \approx n^2$, kann keine Verbesserung erzielt werden, da in diesem Fall die maximale Größe der Dynamic Trees sehr klein ist. Im Extremfall $m = n^2$ wird die maximale Größe auf $k = 1$ festgelegt. In diesem Fall reduziert sich dann der Algorithmus auf den bereits früher beschriebenen Algorithmus mit Verwendung der FIFO-Warteschlangen.
Die in den obigen Theoremen gemachten Aussagen können auch für den Fall bewiesen werden, wenn anstelle der FIFO-Warteschlange die HLF-Variante verwendet wird.

2.4 Strategien

Nachdem im vorangegangenen Abschnitt einige Methoden vor Verbesserung der Komplexität besprochen wurden, sollen nun einige Strategien erläutert werden, die helfen sollen, die Abarbeitung des Algorithmus zu beschleunigen, ohne dabei jedoch die Komplexität zu verändern [CG94].

Im wesentlichen kann man zwei verschiedene Methoden unterscheiden: Erstens die Aufteilung der Hauptschleife in zwei Phasen (Teilabschnitt 2.4.1) und zweitens die Beschleunigung der entstehenden ersten Phase durch Optimierungen beim Verändern der Entfernungsfunktion (Teilabschnitt 2.4.2 und 2.4.3).

2.4.1 Einführung einer zweiten Phase

Im bisherigen Ablauf des Algorithmus wurde zum Überfluten des Netzwerks und zum Abziehen der überflüssigen Flußeinheiten eine einzige Schleife verwendet. Die in diesem Teilabschnitt vorgestellte Strategie beschreibt nun eine Möglichkeit, wie aus der Hauptschleife zwei Phasen erzeugt werden können. Die erste Phase hat zum Ziel, einen maximalen Präfluß zu finden, ohne dabei Flußeinheiten in die Quelle zurückzusenden. Die zweite Phase startet dann mit einem Präfluß, der in einen normalen Fluß umgewandelt werden soll. Diese Umwandlung kann effizienter gestaltet werden, als dies durch die herkömmliche Methode möglich war. Somit kann an der Gesamtkomplexität zwar keine Verbesserung erzielt werden, der konstante Faktor der oberen Schranke kann jedoch reduziert werden.

Erste Phase

Wie bereits erwähnt soll in der ersten Phase ein maximaler Präfluß gefunden werden. Die Idee dabei ist, festzustellen, wann ein aktiver Knoten nicht mehr zur Erhöhung des Wertes des Präflusses beitragen kann. Wie bereits bei den Definitionen (Abschnitt 2.1) erwähnt, bildet der Wert der Entfernungsfunktion für jeden Knoten eine untere Schranke für die Länge eines kürzesten Pfades von dem Knoten zur Senke. Wenn ein aktiver Knoten zur Erhöhung des Präflusses beitragen soll, muß er einen Teil seines Überflusses zur Senke verschieben. Dies muß über einen Pfad zur Senke geschehen. An dieser Stelle wird folgendes Theorem angewendet:

Theorem 2.9 *Wenn im Laufe des Push-Relabel-Algorithmus ein Knoten v einen Entfernungswert $d(v) \geq n$ zugewiesen bekommt, so existiert im Residuennetz kein Pfad mehr von v zur Senke.*

Beweis Der Beweis wird durch Widerspruch geführt. Angenommen es gäbe einen solchen Pfad, so müßte er aufgrund des Wertes der Entfernungsfunktion mindestens über $n+1$ Knoten laufen. Ein Pfad, der über mindestens $n+1$ Knoten verläuft, muß jedoch einen Zyklus besitzen und kann somit nicht nicht ein kürzester Pfad sein. □

Angewendet auf den Algorithmus bedeutet dies, daß die Hauptschleife nur für die aktiven Knoten v ausgeführt wird, die eine Entfernung $d(v) < n$ besitzen. Für die Implementierung bedeutet dies, daß ein Knoten, der nach einer *Relabel*-Operation einen in diesem Sinne zu großen Wert zugewiesen bekommt, nicht mehr in die Menge der aktiven Knoten aufgenommen wird. Nach dieser Änderung terminiert der Algorithmus, wenn ein maximaler Präfluß gefunden wurde.

Zweite Phase

Die zweite Phase hat nun zur Aufgabe, einen gültigen Präfluß so in einen gültigen Fluß zu verwandeln, daß der Wert des Flusses gleich dem des Präflusses ist. Der Grundgedanke hinter dem Vorgang ist sehr einfach. Es sollen alle überflüssigen Flußeinheiten in den

aktiven Knoten zur Quelle zurückgeschoben werden. Da alle Flußeinheiten in der ersten Phase von der Quelle zu den aktiven Knoten geflossen sind, ist für jede Flußeinheit eine Rückführung zur Senke auf den von ihr durchflossenen Wegen möglich. Um dies umzusetzen ist es sinnvoll, aus dem durch den Präfluß induzierten Graphen einen azyklischen Graphen zu erzeugen. Hierfür wird für alle Kanten eines Zyklus der Fluß um soviele Einheiten verringert, wie auf dem Zyklus zirkulieren. Dies bewirkt, daß die Kanten des Zyklus, der die wenigsten Flußeinheiten zugeordnet sind, alle Flußeinheiten verliert, was wiederum zur Folge hat, daß der Zyklus eliminiert wird. Die Präfluß- und die Überflußfunktion sind bei dieser Aktion entsprechend anzupassen. Bei dem so entstandenen DAG (directed acyclic graph) können nun alle überzähligen Flußeinheiten der aktiven Knoten in die Quelle zurückgeschoben werden. Diese Verschiebung erfolgt auf den Pfaden von der Quelle bis zum jeweiligen Knoten. Hierfür werden alle Knoten so numeriert, daß alle Kanten von einem Knoten mit kleinerem Index zu einem mit größerem Index verlaufen. Beginnend von den großen Indizes können dann die übrigen Flußeinheiten um jeweils eine Kante in Richtung Quelle verschoben werden. Unter dem Einsatz von Dynamic Trees kann die Bildung des azyklischen Graphen in $O(m \log n)$ durchgeführt werden [ST83]. Die Rückführung der Flußeinheiten kann dann in $O(m)$ durchgeführt werden. Alles in allem erhält man eine Komplexität von $O(m \log n)$ was im Vergleich zur ersten Phase keine Veränderung in der Gesamtkomplexität des Algorithmus bewirkt, jedoch eine deutliche Beschleunigung der zweiten Phase darstellt.

2.4.2 Global Relabeling

Eine weitere Strategie, die eine Verbesserung der Abarbeitung der aktiven Knoten ermöglichen soll, besteht darin, unnötige *Push*-Operationen zu vermeiden (ebenfalls [CG94]). Wie beschrieben, wird eine *Push*-Operation über die Kante (v, w) nur dann durchgeführt, wenn es die Entfernungswerte der beteiligten Knoten erlauben (d.h. $d(v) = d(w)+1$). Diese Regel soll sicherstellen, daß Flußeinheiten immer in Richtung Senke verschoben werden. Da die Entfernungswerte nur eine untere Schranke für die Länge eines kürzesten Pfades sind, kann es vorkommen, daß fälschlicherweise angenommen wird, daß eine Verschiebung in Richtung Senke möglich ist. In Abbildung 2.7 ist eine Situation dargestellt, in der der Überfluß zwischen zwei Knoten unnötigerweise zweimal hin- und hergeschoben wird, bis einer der Knoten einen Entfernungswert erhält, der es ermöglicht, einen Weg über einen anderen Knoten zu wählen. Der Knoten, der gerade den Überfluß enthält, ist dick umrandet. DenmKnoten sind zwei Werte zugeordnet, der erste ist der Überfluß und der zweite die Entfernung. An den Kanten sind die Werte für den aktuellen Fluß und die Kantenkapazität angetragen.
Die Idee beim *Global Relabeling* ist die, daß die Entfernungsfunktion stets eine möglichst gute untere Schranke für die Länge eines kürzesten Pfades ist. Im Idealfall sind beide Werte sogar gleich. Da der vorgestellte Algorithmus die gewünschte Eigenschaft nicht erfüllt, muß von außen in den Algorithmus eingegriffen werden. D.h. in gewissen Abständen wird die Entfernungsfunktion mittels einer BFS ausgehend von der Senke wieder auf den gewünschten Wert gebracht.

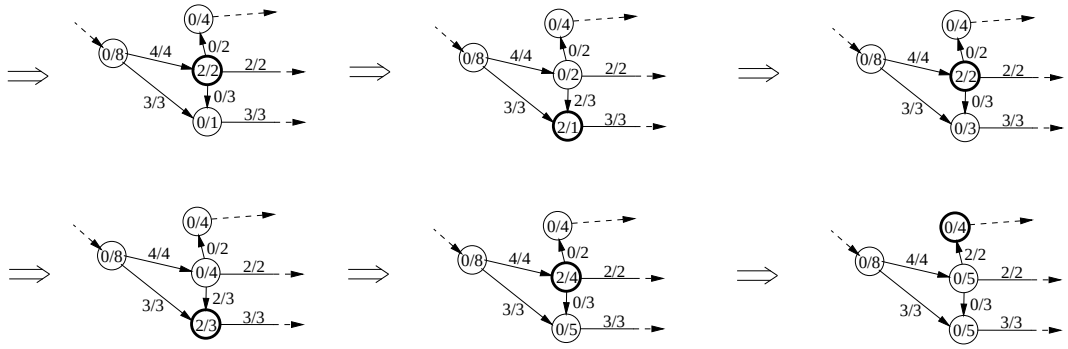


Abbildung 2.7 *motivierendes Beispiel für Global Relabeling*

Die Abstände, in denen dieses *Global Relabeling* stattfindet, sollten sinnvollerweise in Abhängigkeit von der Anzahl der einfachen (lokalen) *Relabel*-Operationen im Algorithmus, die seit dem letzten *Global Relabeling* stattgefunden haben, gewählt werden.

2.4.3 Gap Relabeling

Eine weitere Methode zur Vermeidung von unnötigen *Push*-Operationen nennt sich *Gap Relabeling* [CG94]. Der Grundgedanke dieser Strategie besteht darin, ähnlich wie bei der Aufteilung des Algorithmus in zwei Phasen, nur solchen Knoten eine *Push*-Operation zu ermöglichen, bei denen es überhaupt möglich ist, daß sie einen Pfad zur Senke besitzen, der durch *Push*-Operationen durchlaufen werden kann. In Abbildung 2.8 ist ein solches Beispiel zu sehen. (die Beschriftung der Knoten und Kanten ist wie in der vorherigen Abbildung gewählt). Wenn der Algorithmus in einen Zustand kommt, der im ersten Bild angegeben ist, muß er erst den dick gezeichneten Pfad durchlaufen, bevor zu erkennen ist, daß dieser am Ende nicht fortgesetzt werden kann (zweites Bild).

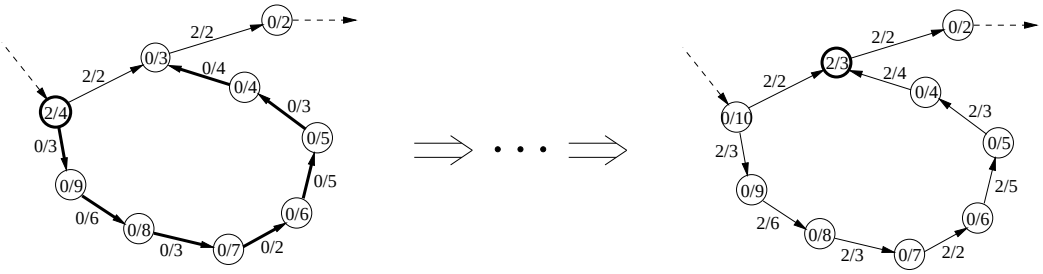


Abbildung 2.8 *motivierendes Beispiel für Gap Relabeling*

Die Strategie *Gap Relabeling* versucht, genau solche nicht fortsetzbare Pfade zu erkennen. Hierfür wird eine Regel aufgestellt, die sich auf das folgende Theorem stützt.

Theorem 2.10 *Wenn im Laufe des Push-Relabel-Algorithmus ein Knoten v mit dem Entfernungswert $d(v)$ existiert, so daß es einen Entfernungswert x, mit $d(v) > x \geq 1$, gibt, der keinem Knoten zugeordnet ist, so kann es keinen Pfad von v zur Senke geben, auf dem Flußeinheiten zur Senke verschoben werden können.*

Beweis Der Beweis wird mittels Widerspruch geführt. Angenommen es gäbe einen solchen Pfad, dann gibt es im Residuennetzwerk einen durchgehenden gerichteten Pfad von

v zur Senke t mit den folgenden Knoten $v = w_0, w_1, \ldots, w_{k-1}, w_k = t$ und den Kanten (w_i, w_{i+1}). Mit $d(t) = 0$ und der Eigenschaft von gültigen Entfernungsfunktionen, daß für jede Kante (w_i, w_{i+1}) im Residuennetz gilt $d(w_i) \leq d(w_{i+1})+1$, folgt, daß es eine Folge von Indizes $j_0, j_1, \ldots, j_l$ geben muß, für die gilt $d(w_{j_0}) = d(v)$ und $d(w_{j_l}) = d(t) = 0$ und ferner allgemein $0 \leq j_i \leq k \ \wedge \ j_i < j_{i+1} \ \wedge \ d(w_{j_i}) = d(w_{j_{i+1}}) + 1$. In anderen Worten heißt dies, daß es für jeden Entfernungswert y mit $d(v) > y \geq 1$ mindestens einen Knoten auf dem Pfad gibt, dem dieser Wert zugewiesen ist. Dies ist jedoch ein Widerspruch zur Wahl von x. □

Dieses Theorem kann folgendermaßen auf den Algorithmus umgesetzt werden. Wenn nach einer *Relabel*-Operation eines Knotens v eine Lücke entsteht, d.h. wenn vor der *Relabel*-Operation v der einzige Knoten war, dem der alte Entfernungswert $d(v)_{alt}$ zugeordnet ist, dann können alle aktiven Knoten w mit einer Entfernung $d(w) > d(v)_{alt}$ keine Flußeinheiten mehr bis zur Senke verschieben. Auf diese Knoten muß in der ersten Phase also keine Operation mehr angewendet werden. Bei Verwendung der FIFO- bzw. HLF-Methode heißt dies, daß diese Knoten aus der Liste der aktiven Knoten entfernt werden können. Für die Implementierung zieht dieses Vorgehen nach sich, daß die Verwendung der einzelnen Entfernungswerte fortwährend gespeichert und aktualisiert werden muß.

Kapitel 3

Implementierung

In diesem Kapitel wird das in dieser Diplomarbeit implementierte System beschrieben. Dieses System baut auf zwei vorangegangene Arbeiten auf. Einerseits auf die Diplomarbeit von Ilia Dub [Dub98] und andererseits auf die Ergebnisse einer HiWi-Arbeit des Autors am Lehrstuhl des Aufgabenstellers. In der angesprochenen Diplomarbeit wurden Basisklassen zur Visualisierung von Algorithmen für ein stand-alone-System mittels Java Swing implementiert. Im Rahmen der Arbeit des Autors wurde ein grundsätzliches System zur Erstellung einer Web-basierten Client-Server-Struktur zur Verwaltung von Algorithmen, ebenfalls in Java, entworfen. In der hier beschriebenen Diplomarbeit wurden unter anderem die Basisklassen in das Client-Server-System eingearbeitet. Die vorhandenen Klassen mußten hierfür teilweise ergänzt oder modifiziert werden.
Basierend auf diesem neuen System wurde dann der Push-Relabel-Algorithmus von Goldberg und Tarjan (siehe Kapitel 2) implementiert. Diese Implementierung ermöglicht es, aus mehreren Heuristiken auszuwählen und die Ergebnisse visualisieren zu lassen. Ferner wurden noch einige Generatoren zum Erzeugen von speziellen Graphklassen implementiert.
In den folgenden Abschnitten wird zuerst das zugrundeliegende Konzept zur Visualisierung von Algorithmen dargestellt (Abschnitt 3.1). Dann wird das Grundprinzip der Client-Server-Anwendung erklärt (Abschnitt 3.2). Abschließend wird die Implementierung des Push-Relabel-Algorithmus in dem neuen System beschrieben (Abschnitt 3.3).

3.1 Algorithmenvisualisierung

In diesem Abschnitt wird der Grundgedanke zur Algorithmenvisualisierung des implementierten Systems beschrieben. Da im wesentlichen die von der oben angesprochenen Diplomarbeit implementierten Basisklassen verwendet werden, sei für eine ausführliche Beschreibung des Konzepts und der Methoden auf die zugehörige Ausarbeitung verwiesen [Dub98]. Im folgenden werden nur das wesentliche Gerüst und die für einen Benutzer vorhandenen Möglichkeiten beschrieben. Mit diesen Ausführungen sollte es möglich sein, die vorhandenen Klassen geeignet zu verwenden, um einen eigenen Algorithmus zu implementieren und dabei auf die vorhandenen Visualisierungsmöglichkeiten zurückzugreifen.

3.1.1 Konzept

Das wesentliche Konzept der Basisklassen besteht darin, daß eine Sammlung von Klassen zur Verfügung gestellt wird, die es erlaubt, daß der Entwickler sich nur noch um seinen Algorithmus kümmern muß, nicht jedoch um aufwendige Details wie Fenster zur Visualisierung oder die Ablaufsteuerung. Wenn er, der Entwickler, mit dem bereitgestellten Umfeld zufrieden ist, soll es ausreichend sein, wenn er seinen Algorithmus von einer bereitgestellten Basisklasse ableitet und zur Visualisierung nur entsprechende Methoden aufruft. Diese Aufrufe sollen ihrerseits möglichst intuitiv sein und ohne wesentliche Vorkenntnisse durchgeführt werden können.
Diese beschriebenen Anforderungen sollen nun umgesetzt werden. Um einen hohen Grad der Wiederverwendbarkeit der Software zu ermöglichen, werden die Basisklassen in drei große Teile aufgeteilt:

1. Editorfenster für die Benutzerschnittstelle
2. Datenstrukturen, die zur Visualisierung eingesetzt werden können
3. Algorithmenklassen, die diese Datenstrukturen verwenden und durch geeignete Methoden in einem Editorfenster darstellen können.

Diese Gruppen werden in den anschließenden Teilabschnitten vorgestellt. Da in der vorhandenen Implementierung besonderer Wert auf Graphalgorithmen gelegt wird, werden diese in einem eigenen Teilabschnitt behandelt. In den Beschreibungen werden des öfteren die Begriffe Entwickler und Benutzer vorkommen. Gemeint sind hiermit einmal die Personen (Entwickler), die die Basisklassen verwenden, um eigene Algorithmen zu implementieren. Die Entwickler werden also selbst Code implementieren und müssen daher die Struktur der Basisklassen kennen. Die Benutzer hingegen sollen von dieser Ebene losgelöst sein und bereits auf einen fertigen Algorithmus treffen. Sie sehen ihrerseits nur das Editorfenster und können keine Modifikationen, soweit dies nicht vom Entwickler extra vorgesehen ist, am Ablauf des Algorithmus vornehmen. In Abbildung 3.1 ist dargestellt, wer auf welche Bausteine zurückgreift und was derjenige seinerseits zur Verfügung stellt.

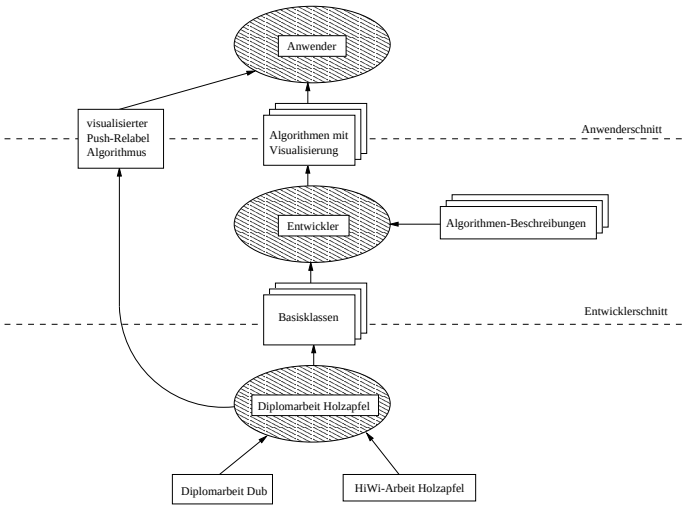


Abbildung 3.1 *Verwendung der einzelnen Bausteine in der Anwendung*

3.1.2 Editorfenster

Das Editorfenster stellt die Schnittstelle zwischen dem Benutzer und dem Algorithmus dar. Alle Eingaben müssen also mittels dieses Fensters vorgenommen werden. Im einfachsten Fall bedeutet dies, daß mit dem Fenster die Steuerung des Algorithmus zu ermöglichen ist. In der Abbildung 3.2 ist die Grundausführung des Editorfenster zu sehen.

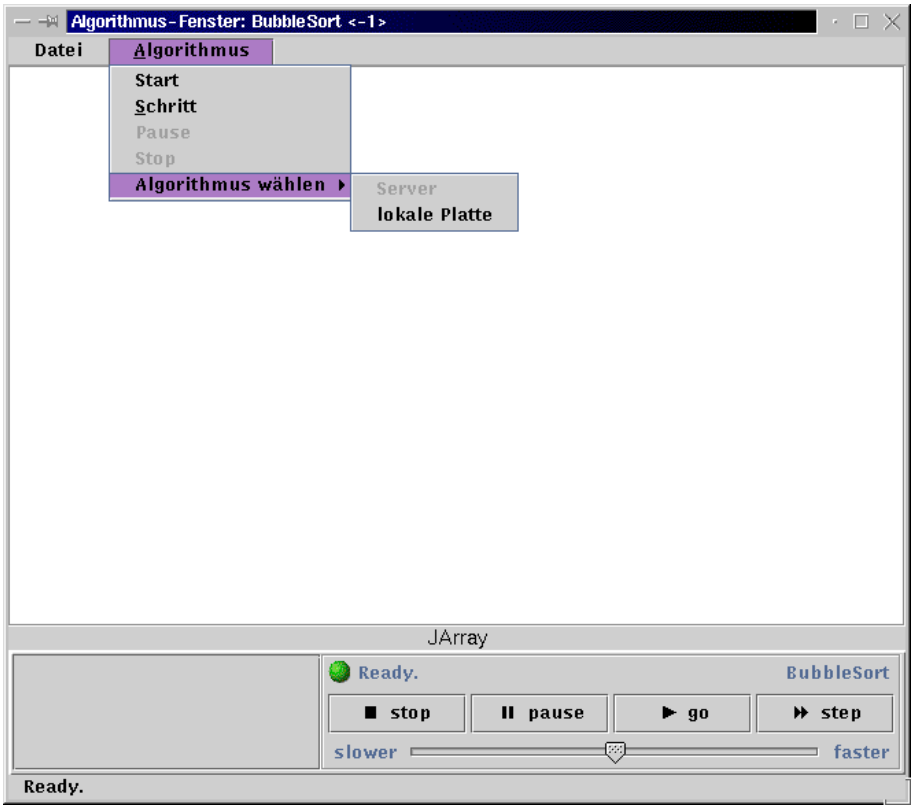


Abbildung 3.2 *Beispiel für ein Editorfenster*

Für die Steuerung befindet sich in jedem Fenster ein Kontrollblock. In diesem befinden sich Tasten "go" und "stop". Hierdurch kann ein Algorithmus gestartet und gestoppt werden. Ein Anhalten ist durch die Taste "pause" möglich. Ferner kann der Algorithmus, mittels der "step" Taste, schrittweise (die Haltepunkte sind vom Entwickler festzulegen) durchlaufen werden. Im "go"-Modus kann die Geschwindigkeit durch den Schiebebalken unter den Tasten modifiziert werden.
Die Ablaufsteuerung kann ebenfalls durch Auswahl der entsprechenden Menüpunkte in dem Menü "Algorithmus" durchgeführt werden. Ferner findet sich in diesem Menü noch eine Möglichkeit, einen anderen Algorithmus zu laden. Die Algorithmen können entweder von der lokalen Platte geladen werden, oder falls das Fenster in dem Client-Server-System läuft, können auch vom Server Algorithmen angefordert werden.
Das zweite Menü "Datei" beinhaltet nur einen Menüpunkt zum Schließen des Fensters, bei Erweiterungen des Fensters, z.B. für Graphen, können hier Menüpunkte zum Laden und Speichern von Datenstrukturen eingebunden werden.

Falls der Algorithmus dem Benutzer Mitteilungen machen möchte, so können diese in der Fußzeile des Fensters angezeigt werden. In der obigen Abbildung steht in diesem Feld die Nachricht "Ready.".
Mit diesen Mitteln ist eine Umgebung geschaffen, die die notwendigsten Schnittstellen zwischen Benutzer und Algorithmus zur Verfügung stellt. Es ist jede denkbare Erweiterung möglich. Für die Klasse der Graphen wurde eine solche Erweiterung implementiert. Im wesentlichen wurde das Fenster so erweitert, daß ihm nun ein eigener Graph zugeordnet ist. Dieser Graph kann durch verschiedene Werkzeuge verändert werden. Die Werkzeuge können entweder durch verschiedene Icons (auf der linken Seite des Anzeigefeldes) oder in einem eigenen Menü ausgewählt werden (siehe Abbildung 3.3).

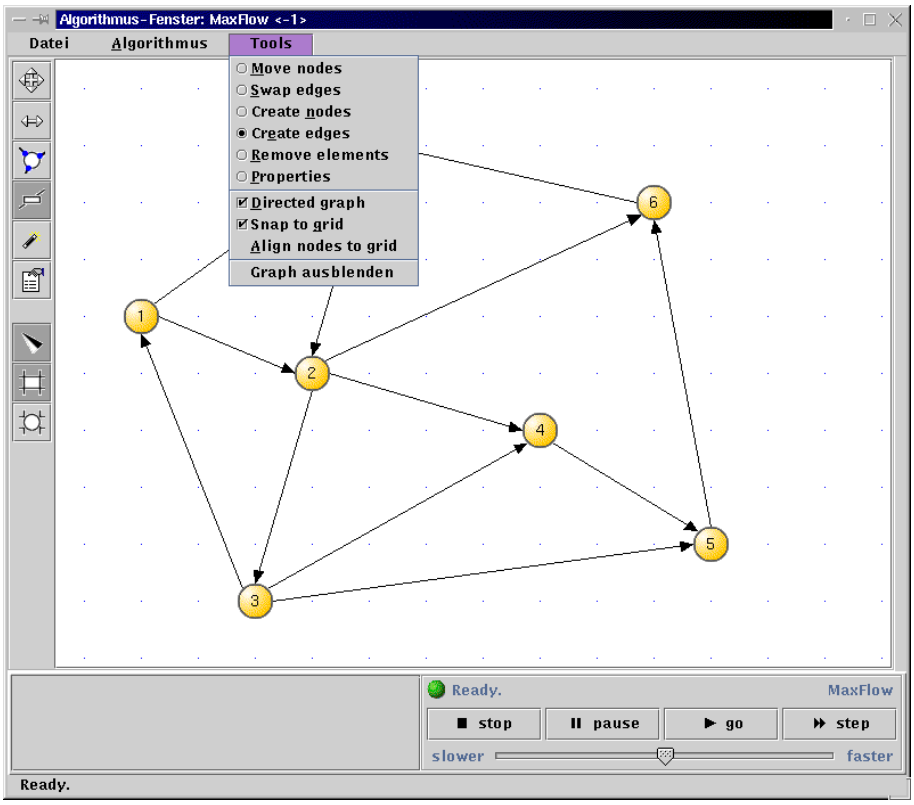


Abbildung 3.3 *Beispiel für ein Graph-Editorfenster*

Als Werkzeuge stehen zur Verfügung (in der Reihenfolge der Icons, von oben nach unten):

- Verschieben von Knoten
- Umdrehen der Kanten (falls diese gerichtet sind)
- Erzeugen von Knoten
- Erzeugen von Kanten
- Entfernen von Knoten oder Kanten
- Verändern von knoten- bzw. kanteneigenen Werten

- Wechsel von gerichteten zu ungerichteten Graphen
- An-/Ausschalten der Fixierung der Knoten auf die Gitterpunkte
- Positionieren der Knoten auf den jeweils nächsten Gitterpunkt

Neben diesen Möglichkeiten, einen Graphen zu manipulieren, wurden in das Menü "Datei" noch Einträge "Laden" und "Speichern" aufgenommen. Mit diesen Änderungen ist eine Umgebung geschaffen, die es einem Benutzer erlaubt, zu gegebenen Graph-Algorithmen beliebige Graphen zu zeichnen bzw. zu laden, den Algorithmus auf diesen Graphen auszuführen und ggf. die Ergebnisse oder die gezeichneten Graphen zu speichern, um sie zu einem späteren Zeitpunkt wiederverwenden zu können.
Bereits an dieser Stelle sei darauf hingewiesen, daß dem Entwickler jederzeit freigestellt ist, weitere Modifikationen am Editorfenster vorzunehmen. Ein Beispiel sei hierfür die Implementierung des Push-Relabel-Algorithmus (siehe Abschnitt 3.3). Ziel der Basisklassen ist es nur, eine Grundmenge von Eigenschaften des Fensters festzulegen, um für Algorithmen eine sinnvolle Schnittstelle bereitzustellen. Eine Beschreibung aller solcher Methoden der Editorfenster ist der oben erwähnten Diplomarbeit und der aus dem Code, mittels JavaDoc, erzeugten Dokumentation zu entnehmen.

3.1.3 Datenstrukturen

In dem System sind einige Datenstrukturen implementiert, die für die Implementierung verwendet werden können. Für diese Datenstrukturen ist auch die Visualisierung implementiert, was bedeutet, daß sie im Editorfenster angezeigt werden können. Wenn eine zusätzliche Datenstruktur zur Visualisierung implementiert werden soll, ist darauf zu achten, daß sie von der Basisklasse `edu.tum.avl.JAVLComponent` abgeleitet ist, damit sie korrekt in die Umgebung paßt. Eine genaue Beschreibung, was bei der Implementierung zu beachten ist, ist wiederum der bereits zitierten Diplomarbeit zu entnehmen. Die folgende Liste enthält eine kurze Übersicht über die wichtigsten vorhandenen Datenstrukturen. Diese Datenstrukturen befinden sich alle in dem *package* `edu.tum.avl` und sind sämtlich von der Klasse `JAVLComponent` abgeleitet:

- `JArray` — Komponente, die ein Array aus lauter von `JAVLComponent` abgeleiteten Instanzen aufnehmen kann.
- `JQueue` — Implementierung einer Warteschlange zur Verwaltung von Instanzen, die vom Typ `JAVLComponent` abgeleitet sind.
- `JPriorityQueue` — Warteschlange für gewichtete Instanzen, die ebenfalls vom Typ `JAVLComponent` abgeleitet sind. D.h. zu jedem Gewicht existiert eine eigene Warteschlange. Beim Einfügen wird ein Element in die, seinem Gewicht zugehörige, Warteschlange eingefügt. Beim Herausnehmen wird aus der Warteschlange mit größtem Gewicht, die nicht leer ist, ein Element herausgenommen.
- `JIntBar` — Visualisierungskomponente, die einen Integer-Wert als Balken repräsentiert. Die Länge des Balken wird durch den auszudrückenden Wert festgelegt.
- `JGraph` — Komponente zum Visualisieren eines Graphen. Sie besteht aus einer Sammlung von Knoten des Typs `JNode` und Kanten vom Typ `JEdge`. Ferner sind Informationen gespeichert, die z.B. angeben, ob der Graph gerichtet ist oder nicht.

- `JNode` — Repräsentation eines Knoten. Dem Knoten kann Farbe, Beschriftung und eine Sammlung von knotentypischen Werten (benutzerdefiniert) zugewiesen werden.
- `JEdge` — Analogon zu den Knoten. Auch diese Komponenten besitzen Farbe, Beschriftung und kantentypische Werte.

In der Diplomarbeit wurden noch einige andere Datenstrukturen zum Visualisieren implementiert, wie z.B. die Ablaufsteuerung. Diese Komponenten sind jedoch nicht direkt zur Implementierung von Algorithmen geeignet und daher nicht beschrieben.

3.1.4 Algorithmen

In diesem Teilabschnitt soll kurz dargestellt werden, wie die Klasse `JAlgorithm` verwendet werden kann, um einen Algorithmus zu implementieren und gleichzeitig das Editorfenster zur Visualisierung zu verwenden.
Um einen Algorithmus zu implementieren, muß eine neue Klasse erzeugt werden, die von der Klasse `JAlgorithm` abgeleitet ist. Diese neue Klasse muß drei Methoden bereitstellen bzw. Methoden der Basisklasse überschreiben. Um sicherzustellen, daß die Klasse in einem Editorfenster verwendet werden kann, müssen entsprechende Konstruktoren vorhanden sein. In diesen Konstruktoren müssen die zugehörigen Konstruktoren der Basisklasse aufgerufen werden, abgesehen davon können die Konstruktoren beliebig gestaltet werden. Bei den beiden Konstruktoren handelt es sich um den Konstruktor mit leerer Parameterliste und um den mit einem Editorfenster als einzigen Parameter. Ferner muß in der Methode `run()` der Algorithmus selbst implementiert werden. Diese Methode wird aufgerufen, wenn der Algorithmus gestartet wird. In Abbildung 3.4 ist für einen Algorithmus mit Namen `MyAlgorithm` ein Rahmen für die zu implementierende Klasse gegeben. Die Stellen, an denen eigener Code eingefügt wird, sind durch '...' gekennzeichnet. Wie in jeder anderen Klasse auch können beliebige weitere Methoden, Subklassen etc. implementiert werden. Der dargestellte Rahmen soll nur die Mindestanforderungen darstellen.

```
public class MyAlgorithm extends JAlgorithm
{
  ////////////////////
  // Konstruktoren //
  ////////////////////

   public MyAlgorithm()
  {
     super();
     ...
  }
  public MyAlgorithm(EditorWin editor)
  {
     super(editor);
     ...
  }
```

```
    /////////////////
    // Algorithmus //
    /////////////////

    public void run() throws Throwable
    {
        ...
    }
}
```

Abbildung 3.4 *Rahmen für einen Algorithmus*

Wird der Konstruktor ohne Parameter aufgerufen, wird der Algorithmus erzeugt, ohne daß ihm ein Editorfenster zugeordnet wird, anderenfalls wird das als Parameter übergebene Editorfenster zugeordnet. Innerhalb der Klasse kann das zugehörige Fenster, so es denn gesetzt ist, über die Instanzvariable `editor` referenziert werden. Soll im Fenster eine Datenstruktur (abgeleitet von der Basisklasse `JAVLComponent`) angezeigt werden, so kann dies durch die Methode `setMainDataStructure(...)` des Editorfensters geschehen. Nachrichten an den Benutzer, in Form eines Strings, können durch die Methode des Fensters `setStatusBarText(...)` in der Fußzeile angezeigt werden. Für eine detailliertere Beschreibung der Klassen `EditorWin` bzw. `JAlgorithm` sei auf die bereits mehrfach angesprochene Diplomarbeit [Dub98] und auf die Dokumentation der Klassen (JavaDoc) verwiesen.

3.1.5 Graph-Algorithmen

Wie bereits erwähnt, wurde für die Datenstruktur `JGraph`, also zur Visualisierung von Graphen, eine Erweiterung des Editorfensters realisiert. Analog dazu wurde auch eine eigene Klasse `JGraphAlgorithm` implementiert, die einen größeren Bedienungskomfort bietet, um Graph-Algorithmen zu visualisieren. Da es sich bei dieser Klasse um eine Subklasse von `JAlgorithm` handelt, bleiben alle in Teilabschnitt 3.1.4 beschriebenen Anforderungen und Schnittstellen erhalten. Im folgenden werden nur die Erweiterungen erläutert.
Dem Algorithmus ist automatisch ein Graph zugeordnet. Dieser Graph kann durch die Instanzvariable `graph` angesprochen werden. Da der Graph mit dem Graphen im Editorfenster gleichgesetzt ist, sollte das Zuweisen eines neuen Graphen nur mit der Methode `setGraph(...)` durchgeführt werden. Anderenfalls kann es zu Inkonsistenzen kommen, so daß der Graph nicht angezeigt wird. Der Graph wird automatisch im zugeordneten Editorfenster angezeigt. Alle Änderungen am Graphen mit den Werkzeugen des Editorfensters werden also direkt auf den Graphen des Algorithmus übertragen und sind somit für den Benutzer sichtbar. Ebenfalls werden somit alle Manipulationen auf dem Graphen durch den Algorithmus (z.B. Färben von Knoten und Kanten, Änderung von Beschriftungen etc.) sofort sichtbar.
Da es für einige Algorithmen sinnvoll ist, zusätzliche Datenstrukturen wie Listen oder ähnliches anzuzeigen, kann mit der Methode `addDataStruct(...)` eine von der Klasse `JAVLComponent` abgeleitete Instanz im Editorfenster angezeigt werden.
Bei machen Algorithmen ist es sinnvoll, nicht die zur Verfügung gestellte Klasse `JGraph` zu verwenden, sondern eine effizientere Repräsentation des Graphen zu wählen. Wenn

nun der `JGraph` im Editorfenster angezeigt wird, kann es zu ungewollten Anzeigen führen, da die Referenz auf einen, jetzt nicht gültigen, `JGraphen` immer noch besteht. Daher wird vor jedem Anzeigen des Graphen (z.B. nach dem Setzen des Graphen) die Methode `prepareGraphForVisualization()` aufgerufen, um dem Algorithmus die Möglichkeit zu geben, die eigene Datenstruktur entsprechend umzusetzen oder sonstige Schritte zu unternehmen. Wird also eine soche Sonderbehandlung vom Entwickler gewünscht, ist die Methode geeignet zu überschreiben.

3.2 Client-Server-Anwendung

In diesem Abschnitt wird die Client-Server-Anwendung beschrieben. Beginnend bei den Anforderungen und der daraus resultierenden konzeptionellen Gestaltung (Teilabschnitt 3.2.1) wird ein erster Eindruck für das System gegeben. Im nächsten Teilabschnitt (3.2.2) werden einige Fragen der Sicherheit behandelt, die durch das Verwenden von Java in verteilten Systemen auftreten. In den folgenden Teilen wird dann eine detaillierte Beschreibung gegeben, wie das System zu benutzen ist. Hierbei wird zuerst die Sicht eines Benutzers auf das System (Teilabschnitt 3.2.3) beschrieben. Hierauf aufbauend wird im letzten Teilabschnitt (3.2.4) erläutert, was ein Verwalter des Systems für Schritte vornehmen kann, um eigene Algorithmen, evtl. mit einem eigenen Tutorial, anzubieten. Da das System, wie zu Beginn des Kapitels bereits erwähnt, auf zwei vorangegangene Arbeiten aufbaut, ist die Wahl der Programmiersprache bereits vorgegeben. Das System soll in der Sprache Java unter Einsatz der Java-Swing Bibliothek implementiert werden. Ein wichtiger Baustein für die Visualisierung stellt die Diplomarbeit von Ilia Dub [Dub98] dar, deren Ergebnisse so weit wie möglich eingearbeitet werden sollen. Der daraus entstehende Editor (siehe Abschnitt 3.1) soll in dem System zur Visualisierung der Algorithmen verwendet werden.

3.2.1 Konzept

Die Darstellung des Konzepts der Client-Server-Anwendung wird in folgenden Schritten vorgenommen. Zu allererst werden die Anforderungen an das System aufgelistet. Hieran knüpft eine grundlegende Frage für Client-Server-Anwendungen an. Dies ist die Frage der Aufteilung der Aufgaben des Systems auf den Client und auf den Server. Erst jetzt ist es möglich, die einzelnen Komponenten zu betrachten, um ausführlich zu beschreiben, welche Anforderungen wie implementiert werden.
Nun also zu den Anforderungen. Das in dieser Diplomarbeit entwickelte System soll es ermöglichen, mehrere Algorithmen im Internet anzubieten, um diese plattformunabhängig mit einem Web-Browser, die Wahl fiel hierbei auf den Netscape Navigator, abzurufen. Neben dem reinen Laden und Starten des Algorithmus soll es möglich sein, ein Tutorial für den Algorithmus zu erstellen, um dieses dann zusammen mit dem Algorithmus zu präsentieren. Ferner soll es möglich sein, dem Benutzer für die Algorithmen eine gewisse Vorauswahl der Parameter zu präsentieren, damit dieser nicht erst eigene Parameter wählen muß, sondern bereits einen Algorithmus antrifft, den er nur noch zu starten braucht. Neben dem parallelen Arbeiten mit Tutorial und dem Algorithmus selbst soll es auch möglich sein, dynamisch zwischen den Algorithmen, die im System präsentiert sind, und eigenen, d.h. vom Benutzer selbst implementierten Algorithmen, zu wechseln. Die folgende Auflistung enthält die eben beschriebenen Anforderungen nochmals in komprimierter Form:

1. plattformunabhängige Benutzung des Systems mit dem Netscape Navigator
2. Bereitstellung von Algorithmen, die vom Benutzer ausgewählt werden können
3. Integration eines Tutorials für die Algorithmen
4. Möglichkeit zur Auswahl von vordefinierten Parameterbelegungen für die Algorithmen
5. paralleles Arbeiten mit dem Tutorial und den Algorithmen
6. dynamisches Einbinden von benutzereigenen Algorithmen

Im folgenden ist nun beschrieben, wie diese Anforderungen umgesetzt und in einzelne Komponenten aufgeteilt werden. Diese Komponenten werden dann auf den Client und den Server aufgeteilt. Da die Beschreibungen nur das Konzept darstellen, können keine Details aufgeführt werden. Diese Details sind dann im Teilabschnitt 3.2.4 genauer beschrieben.

- Durch die vorgegebene Programmiersprache Java, unter Einsatz des JavaBean-Konzepts, ist eine Plattformunabhängigkeit des Systems bereits gegeben. Lediglich die passenden Java Bibliotheken müssen auf der jeweiligen Zielplattform vorhanden sein. Im Fall der Diplomarbeit wird Java Swing verwendet. Diese Klassen befinden sich erst ab der Java-Version 1.2 standardmäßig in den bereitgestellten Bibliotheken. Die Klassen für die JavaBeans sind ebenfalls ab Version 1.2 enthalten. In Fällen, in denen das System bei einer älteren Java-Version verwendet werden soll, müssen diese Bibliotheken gesondert auf die Zielplattform abgelegt werden und die entsprechenden Java-Archive in den CLASSPATH aufgenommen werden. Der Code für das entwickelte System ist in einem eigenen Java-Archiv abgelegt und wird beim Laden der entsprechenden Startseite der Anwendung vom Web-Browser automatisch geladen. Der Wunsch, daß das System mit dem Netscape Navigator benutzt werden kann, kann durch die Verwendung eines Applets realisiert werden. Eine Kompatibilität zu anderen Browsern wäre mit diesem Ansatz denkbar, ist jedoch nicht möglich, ohne daß hierfür Änderungen an den implementierten Klassen vorgenommen werden müssen. Diese Anpassungen werden durch das Sicherheitskonzept in verteilten Systemen mit Java-Applets hervorgerufen (siehe Teilabschnitt 3.2.2).

- Das Bereitstellen von Algorithmen wird in einer kleinen Datenbank umgesetzt. In dieser Datenbank sollen die Algorithmen gruppiert werden. Als Gruppierung ist die Datenstruktur vorgesehen, auf der der Algorithmus läuft. Somit ist es also z.B. möglich, alle Graph-Algorithmen zusammenzufassen. Soll also im Editorfenster der Algorithmus gewechselt werden, so kann die jeweilige Gruppe aus der Algorithmensammlung angefragt werden. Das System kann dann direkt in dieser Gruppe nachschlagen, ohne neue Sortierungen vorzunehmen.

- Die Integration eines Tutorialsystems wird wie folgt umgesetzt. Von der Startseite des Systems, also der Seite, die im Browser geladen wird, kann man auf eine Seite des Tutorials gelangen. Diese Seite und die darin enthaltenen Tutorials für die einzelnen Algorithmen können beliebig in einer HTML-Struktur gespeichert werden. Durch Aufrufe von JavaScript-Befehlen können einzelne Beispiele in einem Editorfenster geladen werden.

- Um zu ermöglichen, daß dem Benutzer eine vordefinierte Parameterbelegung für die Algorithmen geliefert wird, kann für jeden Algorithmus ein sogenanntes Parameterfenster implementiert werden. Dieses Parameterfenster wird angezeigt, wenn der Benutzer im Browser einen Algorithmus auswählt. In diesem Fenster kann bereits eine Vorauswahl der Parameter dargestellt werden. Der Benutzer kann diese Auswahl abändern oder direkt übernehmen.

- Wie oben formuliert soll es dem Benutzer möglich sein, parallel mit dem Tutorial und den einzelnen Algorithmen zu arbeiten. Um dies zu ermöglichen, wurde die Abtrennung des Tutorials von den Editorfenstern, in denen die Algorithmen laufen, vorgenommen und das Tutorial in den Browser ausgelagert. Beispiele in Form von Datenstrukturen können in Editorfenstern angezeigt werden. Das System ist so konzipiert, daß mehrere Editorfenster gleichzeitig geöffnet sein können. Somit können nun in einem oder mehreren Editorfenstern Beispiele geladen werden, und unabhängig davon in anderen Editorfenstern mit Algorithmen gearbeitet werden.

- In den bisher formulierten Anforderungen an das System ist es nur möglich, die durch das System bereitgestellten Algorithmen zu verwenden. Es ist jedoch abzusehen, daß ein Anwender nicht nur diese, sondern auch andere Algorithmen verwenden möchte. Durch die Bereitstellung von Basisklassen (siehe Abschnitt 3.1) ist es möglich einfach Algorithmen zu implementieren, die von diesen Klassen abgeleitet sind. Diese Ableitung ermöglicht es jedoch, die Algorithmen im Editorfenster einzubinden und zu starten. Ein wünschenswertes dynamisches Modifizieren, erneutes Übersetzen und erneutes Laden ist aus Sicherheitsgründen leider nicht möglich. Diese Problematik wird genauer in Teilabschnitt 3.2.2 beschrieben.

Nachdem beschrieben ist, wie die Anforderungen im System umgesetzt werden, kann nun die Aufteilung auf den Client und den Server vorgenommen werden. Ziel bei dieser Aufteilung ist es, ein möglichst einfaches Verwalten der bereitgestellten Daten zu erreichen. Ferner soll es einem Benutzer möglich sein, ohne viel Vorarbeit die Startseite des Systems in seinem Browser zu laden und schnell alle Dienste zu nutzen. Gleichzeitig soll jedoch die Kommunikation möglichst gering gehalten werden.
Diese Anforderungen sind natürlich widersprüchlich. Wenn ein Benutzer keine Vorarbeiten erledigen soll, d.h. keine Klassendefinitionen laden soll, das System jedoch schnell verfügbar haben möchte, bedeutet dies, daß viele Teile des Systems nicht auf dem Client sondern auf dem Server ablaufen. Bei dem Einsatz mehrerer Editorfenster ist dies jedoch nicht geeignet umzusetzen. Im implementierten System wird daher ein Mittelweg gewählt. Beim Starten der Hauptseite wird eine Java-Bibliothek (ein sog. Java Archive kurz `jar`) vom Server auf den Client geladen. Dieses Laden benötigt etwas Zeit, erlaubt es jedoch später, ohne Nachladen von Klassendefinitionen zu arbeiten. Ferner können jetzt alle Instanzen auf dem Client initiiert werden, was ein schnelles Ausführen bei geringer Kommunikation ermöglicht. Anderenfalls müßten die Instanzen auf dem Server erzeugt werden und mittels verteilter Referenzen über das Netz angefragt werden. Auch für den Ausführungsort der Algorithmen wird der Client gewählt. Entgegen dem sonst üblichen Prinzip, daß die Berechnungen meistens durch den Server durchgeführt werden und dann nur die Ergebnisse dem Client mitgeteilt werden, wird bewußt diese Variante gewählt. Der Grund liegt auf der Hand. Bei dem umgesetzten System sollen Algorithmen visualisiert werden. Die Visualisierung impliziert viele Anweisungen des Algorithmus an das jeweilige Editorfenster (z.B. zum Färben von Elementen, Ändern von Beschriftungen

etc.). Würde der Algorithmus also auf dem Server ablaufen, würde dies eine Vielzahl an einzelnen Paketen im Netz nach sich ziehen. Abgesehen von der ungewollten Netzlast würde dies auf Grund der Netzverzögerung ein träges und oft ruckhaft erscheinendes System zur Folge haben. Aus diesen Gründen werden also alle Algorithmen auf dem Client ausgeführt.
Es verbleiben die Komponenten, die die Daten bzgl. Tutorial, Beispielbelegungen und damit verbundenen Datenstrukturen organisieren. Um ein leichtes Verwalten zu ermöglichen, müssen diese Komponenten auf dem Server angesiedelt sein. Auch ein Übertragen von Teilmengen beim Start des Systems auf den Client erscheint nicht sinnvoll. Gerade bei großen Datenmengen kann nicht gezielt vorhergesagt werden, welche Daten voraussichtlich vom Benutzer angefordert werden. Es wird daher also folgende Strategie verwendet. Alle Daten für das Tutorial, Beispieldatenstrukturen und Parameterbelegungen liegen auf dem Server und müssen gesondert angefordert werden.
Um die Kommunikation zwischen den einzelnen Komponenten auf dem Client und dem Server zu koordinieren, werden auf Client- und Serverseite sogenannte Controller aufgesetzt. Der Controller empfängt alle Anfragen der Editor- und Parameterfenster, sendet entsprechende Anfragen und verteilt die Antworten an die entsprechenden Komponenten. Neben dem Empfangen und Bearbeiten von Anfragen und Antworten, die auf beiden Seiten durchgeführt werden müssen, dient der Controller im Client auch zur Koordination der einzelnen Komponenten untereinander. Hierunter fällt neben dem reinen Weiterleiten von Daten auch das Initiieren von Algorithmen und Parameterfenstern. Um das Arbeiten mit mehreren Editorfenstern etwas zu vereinfachen, wird zwischen den Editorfenstern und dem Controller eine zusätzliche Komponente, der sogenannte Editor, eingeführt. Dieser steuert u.a. das Erzeugen und Entfernen der einzelnen Fenster sowie das Zuordnen von Algorithmen zu den einzelnen Fenstern. Dies ermöglicht, daß der Controller die Fenster an einem zentralen Punkt, d.h. dem Editor, nur durch ihre Identifikatoren ansprechen kann.
Die so erstellte Aufteilung in Komponenten und ihre Verteilung auf den Client und den Server ist in Abbildung 3.5 nochmals dargestellt.

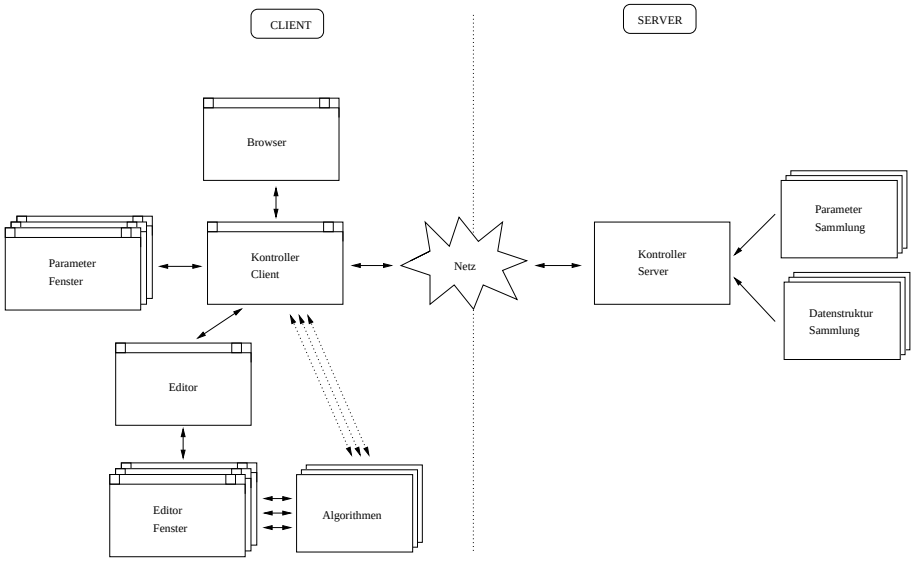


Abbildung 3.5 *Strukturierung und Aufteilung des Systems*

Nachdem das Konzept für die Client-Server-Anwendung ausführlich dargestellt ist, kann nun dessen Umsetzung in der gegebenen Umgebung erläutert werden. Hierfür werden zuerst allgemeine Bemerkungen zu Sicherheitsfragen in verteilten Systemen mit Java besprochen, da diese einige Einschränkungen bzw. zusätzliche Ergänzungen erfordern. Darauf folgend wird eine Übersicht gegeben, wie das System durch einen Benutzer gesehen wird und welche Möglichkeiten ihm zur Auswahl stehen, um die angebotenen Dienste zu nutzen. Abschließend wird das System dann aus der Sicht eines Programmierers, der das System nutzen möchte, um seine Algorithmen im Netz zu präsentieren, geschildert. Hierfür wird eine Anleitung der notwendigen Schritte gegeben.

3.2.2 Sicherheitsfragen

Da, wie bereits weiter oben angesprochen, der Benutzer nicht die Javaklassen aus dem Netz lädt, diese auf seiner lokalen Platte speichert und sie dann von dort auf seinem Rechner startet, sondern die Klassen durch ein in einer Web-Seite eingebundenes Applet verwendet, ergeben sich einige sicherheitsrelevante Probleme. Auch beim Einsatz von JavaScript zum Aufrufen von Methoden eines Applets finden Sicherheitsüberprüfungen statt.

Allgemeines

Ein Problem beim Verwenden von Code aus einem Netz besteht darin, sicherzustellen, daß durch das Ausführen des Codes nicht ungewollte Abläufe stattfinden können. Selbst wenn man Vertrauen in denjenigen hat, der den Code erzeugt hat, kann man bei Übertragungen, die nicht verschlüsselt oder anderweitig geschützt sind, nicht sichergehen, daß der empfangene Code wirklich der Code ist, der publiziert werden sollte. Erwähnt sei hierfür das sogenannte "Hacken von Web-Seiten". Um diese potentielle Gefahrenquelle einzuschränken, ist es Java-Applets normalerweise nicht gestattet, Aktionen auszuführen, die Informationen des Benutzers entweder publizieren, verfremden oder gefährden können. Typische Beispiele hierfür sind z.B. das Senden von e-mails mit Daten der Festplatte, Löschen von Daten, Installieren oder Manipulieren von Programmen etc.
Gerade in dem vorgestellten System bedeutet dies jedoch Einschränkungen, die nicht akzeptabel sind. Man denke nur an die Fähigkeit, Datenstrukturen auf der Platte abzulegen um diese später wiederzuverwenden. Ebenfalls wäre das Aufbauen einer Kommunikationsverbindung zum Server nicht möglich. Um dennoch Anwendungen zu implementieren, die solche Aktionen ermöglichen, sieht Java eine Methode vor, Code speziell zu schützen. Durch ein einfaches *public key* Verfahren kann Java Code signiert werden. Ein so signierter Code kann vom Browser gelesen werden und mittels des *private key* auch authentifiziert werden. Im Bereich des WWW gibt es sogenannte Schlüsselzentren, bei denen man seinen *public key* ablegen kann. Dieser kann dann von einem Netzbenutzer verwendet werden, um den Code zu überprüfen. Wenn der Code überprüft ist, können nun vom Benutzer verschiedene Rechte an das Programm gegeben werden.
Im aktuellen Fall wird ein Benutzer beim Laden der Hauptseite der Anwendung gefragt, ob er dem Programm erlauben möchte, alle relevanten Rechte für Netzanwendungen (dies beinhaltet z.B. Lese-/Schreiberechte, Aufbauen von Netzverbindungen etc.) zu erhalten. Wenn der Benutzer dies bestätigt, läuft das Programm ab dieser Stelle so, als ob es lokal auf dem Rechner gestartet wurde. Werden die Rechte verweigert, tritt an

sicherheitsrelevanten Stellen eine Exception auf und das Programm kann an dieser Stelle nicht arbeiten.
Im Fall der Implementierung der Diplomarbeit wird nur ein selbstgenerierter Testschlüssel verwendet, da das offizielle Erzeugen und Ablegen des Schlüssels mit finanziellen Leistungen verbunden ist. Der Testschlüssel kann auf der Startseite der Anwendung bezogen werden.

Einschränkungen

In der dargestellten Implementierung konnten zwei wünschenswerte Erweiterungen nicht durchgeführt werden, da diese aus Sicherheitsgründen nicht erlaubt sind:

1. Dynamisches Wiedereinbinden von Code — Wie beschrieben kann ein eigener Algorithmus, so er denn von den entsprechenden Basisklassen abgeleitet ist, in das Programm eingebunden werden. Wenn dies stattfindet, wird die Klasse dynamisch hinzugeladen, d.h. die Klassendefinition im sogenannten ClassLoader verfügbar gemacht. Wird nun ein Konstruktor der Klasse aufgerufen, so wird vom ClassLoader die Definition verwendet, um die Instanz zu erzeugen. Wenn nun also ein Benutzer seinen Algorithmus einbindet und feststellt, daß er noch fehlerhaft ist, wird er den Fehler beheben wollen und den Algorithmus erneut vom ClassLoader laden lassen. Für diesen Vorgang wäre es notwendig, bereits geladene Klassendefinitionen zu entfernen bzw. durch andere zu ersetzen. Da jedoch auch die Sicherheitsüberprüfungen durch eine geladene Klasse stattfinden, könnte man also auch diese Klasse ersetzen und so alle Überprüfungen ausschalten. Da dies nicht gewollt ist, kann man im Standard-ClassLoader beim Einsatz von Applets keine Klassen dynamisch wiedereinbinden. Das Erzeugen eines eigenen ClassLoader ist ebenfalls nicht gestattet.
   Im Fall der Client-Server-Anwendung heißt dies also, daß ein Benutzer einen eigenen Algorithmus zwar laden und verwenden kann, im Falle von Änderungen des Algorithmus das System aber erneut laden muß. Dies bedeutet natürlich auch den Verlust von allen Einstellungen, die im Laufe der Anwendung gemacht wurden. Da dies zeitaufwendig und nicht akzeptabel ist, kann das System also nur zum Präsentieren von fertigen Algorithmen verwendet werden. Die Entwicklung eines Algorithmus selbst wird am Besten in einem *stand alone* System vorgenommen. In solchen Systemen benötigt das Kompilieren und Starten des Algorithmus weniger Zeit und kann somit sinnvoll durchgeführt werden. Auch die Verwendung eines Debuggers ist in diesem Umfeld möglich.

2. Methodenaufrufe aus lokalen Web-Seiten — Ein weiteres Problem tritt auf, wenn man dem Benutzer ermöglichen möchte, eigene Tutorials zu erstellen und in das System zu integrieren. Es ist zwar möglich, einen eigenen HTML-Baum zu erzeugen und diesen im Browser einzubinden, aber es ist leider nicht erlaubt, aus einer von der lokalen Platte gelesenen Seite auf ein Applet zuzugreifen, das aus einer anderen Domain (also über das Netz) gelesen wurde.
   Will nun also ein Benutzer ein eigenes Tutorial anbieten, so kann er zum Anzeigen von Beispielen nicht auf ein Editorfenster zurückgreifen, sondern kann lediglich die herkömmliche Methode zum Einbinden von Bildern verwenden.

Diese beiden Einschränkungen haben leider dazu geführt, daß zwei wünschenswerte benutzerfreundliche Ergänzungen nicht umgesetzt werden konnten.

3.2.3 Benutzersicht

In diesem Teilabschnitt soll nun beschrieben werden, welche Möglichkeiten zur Bedienung des Systems einem Benutzer gegeben sind. Die Startseite des Systems ist in Abbildung 3.6 dargestellt.

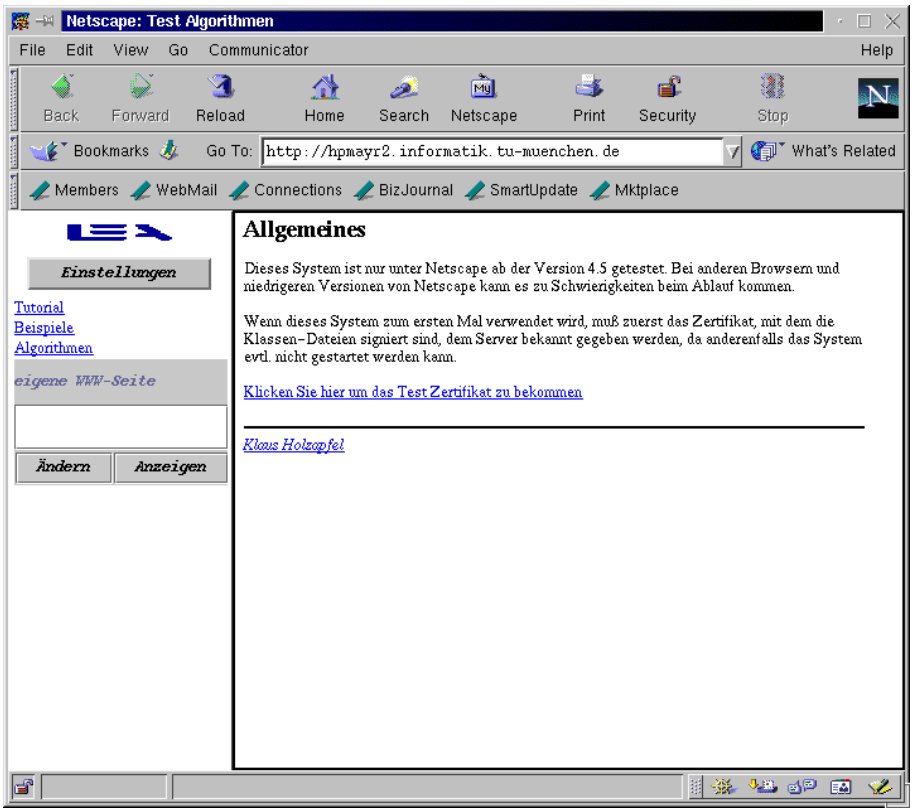


Abbildung 3.6 *Startseite des Systems*

Die Seite ist in zwei Frames unterteilt. Diese Unterteilung wird immer beibehalten. Der schmale linke Frame dient der Steuerung des Systems und der große rechte Frame steht dem Verwalter des Systems zur Verfügung. In diesem Frame können beliebige HTML-Seiten angezeigt werden.

Steuerungsframe

Der Steuerungsframe besteht im wesentlichen aus drei Teilen. Erstens beinhaltet er das Applet, hinter dem sich der Controller auf der Clientseite verbirgt. Dieses Applet ist durch den Knopf mit der Aufschrift *Einstellungen* zu erkennen. Darunter folgen drei Links, die zu den jeweiligen Wurzeln der HTML-Bäume für Tutorial, Beispielkonfigurationen von Algorithmen und der Algorithmenübersicht führen. Der letzte Teil in dem Frame ist das Applet, in dem eine lokale Webseite ausgewählt werden kann, um diese im Anzeigeframe zu laden.
Wenn der Knopf *Einstellungen* ausgewählt wird, erscheint ein Fenster, in dem die Systemeinstellungen vorgenommen werden können (siehe Abbildung 3.7). In der derzeitigen Version ist dies einmal die Sprache, die das Client-Server-System verwenden soll, und die

Anzahl der Editorfenster. Als Sprachen werden Deutsch und Englisch unterstützt. Die Wörter werden aus dem Wörterbuch, das sich in der Klasse `edu.tum.dal.Dictionary` befindet, abgefragt. Es kann so erweitert werden, daß auch andere Sprachen unterstützt werden.
Für die Einstellung zur Festlegung der Anzahl der Editorfenster sei folgendes zu bemerken. Wie in den Systemanforderungen in Teilabschnitt 3.2.1 gefordert, soll ein paralleles Arbeiten mit dem Tutorial und den Algorithmen möglich sein. Dies zieht jedoch nach sich, daß sehr viele Editorfenster auf einmal am Bildschirm zu sehen sind. Als Standard werden alle Tutorialbeispiele in einem Fenster angezeigt, wohingegen für jeden Algorithmus ein neues Fenster erzeugt wird. Um dies zu variieren kann der Benutzer die Anzahl der Fenster einschränken.

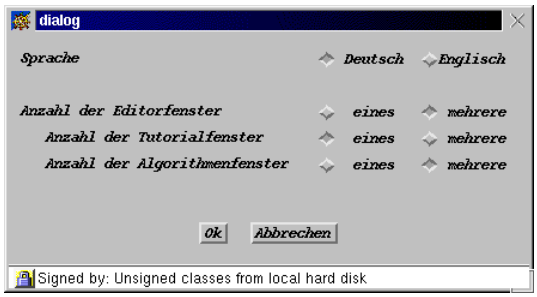


Abbildung 3.7 *Fenster zum Ändern der Systemeinstellungen*

Hierbei kann prinzipiell gefordert werden, daß nur ein einziges Fenster sowohl für das Tutorial, als auch die Algorithmen existiert. In diesem Fall muß beim Laden eines Tutorialbeispiels ein laufender Algorithmus beendet werden. Dies wird überprüft, gemeldet und ggf. auf Wunsch durchgeführt.
Es kann auch getrennt für das Tutorial und die Algorithmen entschieden werden, ob eines oder mehrere Fenster angezeigt werden. Wenn nur ein Algorithmenfenster angezeigt wird, muß der darin laufende Algorithmus beendet werden, bevor ein neuer Algorithmus gestartet wird.
Um eine eigene HTML-Seite im Anzeigeframe zu laden, steht ein zweites Applet zur Verfügung. In diesem Applet kann direkt der absolute Dateiname der Datei, die die Seite enthält, eingetragen werden. Es besteht jedoch auch die Möglichkeit, in einem eigenen Fenster eine Datei auszuwählen, hierzu ist der Knopf *Ändern* im Applet zu wählen. Um die angegebene Seite anzuzeigen, muß abschließend in beiden Fällen der Knopf *Anzeigen* gewählt werden. Die Seite wird daraufhin im Anzeigeframe geladen.

Anzeigeframe

Der Anzeigeframe wird dazu verwendet, Daten für den Benutzer zu präsentieren. Dies wird hauptsächlich das Tutorial mit seinen Unterseiten sein, ferner die vorbelegten Parametereinstellungen und eine Sammlung von Algorithmen. Wie bereits erwähnt, können die Einstiegspunkte zu diesen drei Seiten jederzeit durch die entsprechenden Links im Steuerungsframe ausgewählt werden.
Auf diesen Seiten können sich Knöpfe befinden, die dann einen entsprechenden Methodenaufruf im Client hervorrufen, um die gewünschte Aktion zu starten. Dies kann einerseits das Laden eines Beispielparameters sein, wie es meist im Tutorial vorkommen wird, und andererseits das Starten eines Algorithmus. In diesem Fall wird eventuell ein Parameterfenster angezeigt.

3.2.4 Programmierersicht

Nachdem beschrieben wurde, welche Auswahlmöglichkeiten ein Benutzer im System hat, soll erklärt werden, welche Schritte vorgenommen werden müssen, um eigene Algorithmen und zugehörige Tutorials im Netz zu präsentieren. Im folgenden wird unter anderem erläutert, wo welche Informationen auf dem Server abgelegt werden. Die Erklärungen beziehen sich hierbei auf ein Basisverzeichnis. In diesem Verzeichnis bzw. in einem seiner Unterverzeichnisse befinden sich alle relevanten Daten. Beim Starten des Servers wird dem System ein Verzeichnisname bekanntgeben. In diesem Verzeichnis muß sich ein Unterverzeichnis mit dem Namen "`Data`" befinden. Dieses Unterverzeichnis wird als Basisverzeichnis bezeichnet.

Algorithmen

Es können prinzipiell nur solche Algorithmen eingebunden werden, die von der erwähnten Basisklasse `JAlgorithm` abgeleitet sind. Wenn einem Entwickler ein solcher Algorithmus vorliegt, kann er ihn so im Server einbinden, daß er über das Netz verfügbar ist. Hierfür muß der Algorithmus einerseits an geeigneter Stelle des Servers abgelegt sein und andererseits durch entsprechende HTML-Seiten verfügbar gemacht werden. Die Algorithmen werden dann in das Java-Archive eingebunden und beim Laden der Startseite übertragen.
Es wird nun zuerst beschrieben, welche Klassen für einen neuen Algorithmus implementiert werden müssen. Zuallererst muß der Algorithmus selbst implementiert werden. Ein Algorithmus muß von der Basisklasse `JAlgorithm` aus dem *package* `edu.tum.avl` bzw. einer seiner Subklassen (z.B. `JGraphAlgorithm`) abgeleitet sein. Zur Gruppierung der Algorithmen wird die Bezeichnung der Basisklasse verwendet, auf der der Algorithmus im wesentlichen arbeitet. Für Graph-Algorithmen ist dies z. B. die Klasse `JGraph` und für Sortieralgorithmen die Klasse `JArray`. Damit ein Algorithmus korrekt instantiiert werden kann, muß er sich im richtigen *package* befinden. Der *package*-Name setzt sich wie folgt zusammen. Als Präfix besitzen alle Namen den Pfad `edu.tum.dal.algorithms`, hierauf folgt der Bezeichner der Basisklasse, also z.B. `JGraph` oder `JArray`. Wird eine entsprechende Klassendatei so abgelegt, kann sie vom Client verwendet werden. Für einen Graph-Algorithmus ergibt sich der *package*-Name `edu.tum.dal.algorithms.JGraph`.
Bei der Instantiierung einer Klassendatei aus dem Browser heraus wird zuerst versucht, ein Parameterfenster für den Algorithmus zu erzeugen. Ein Parameterfenster für einen Algorithmus muß sich im gleichen *package* wie der Algorithmus befinden und von der Klasse `ParameterWindow` im *package* `edu.tum.dal` abgeleitet sein. Der Name des Parameterfensters setzt sich aus dem Namen der Klasse und dem Suffix `ParameterWindow` zusammen. Für einen Algorithmus `BFS` ergibt sich für das Parameterfenster also der Name `BFSParameterWindow`. Eine genaue Beschreibung, wie ein Parameterfenster zu implementieren ist, kann dem Anhang B.1 entnommen werden. In dem Parameterfenster wird eine Belegung der Parameter des Algorithmus vorgenommen werden. Wird eine Beispielbelegung verwendet, wird diese beim Instantiieren des Fensters eingelesen und angezeigt. Für das Auslesen von Parameterbelegungen aus einer Datei muß sich im gleichen *package* ebenfalls ein sogenannter ExampleLoader befinden. D.h. zu jedem Parameterfenster muß noch eine zusätzliche Klasse implementiert werden. Der Bezeichner der Klasse setzt sich zusammen aus dem Namen des Algorithmus und dem Suffix `ExampleLoader`, für das obige Beispiel würde sich also der Name `BFSExampleLoader` ergeben. Beim Beenden des Parameterfensters werden die ausgewählten Parameter an

die neue Algorithmen-Instanz übergeben, die diese dann auswertet.
Um also einen Algorithmus zur Verfügung zu stellen, sollte daher neben dem Algorithmus noch das Parameterfenster und der ExampleLoader implementiert werden. Da es jedoch vorkommen kann, daß für einen Algorithmus keine Parameter einzustellen sind oder dies nicht gewünscht wird, kann auf das Implementieren dieser beiden Klassen verzichtet werden. Eine ausführliche Beschreibung zu den Anforderungen an die Implementierung der einzelnen Klassen kann in Anhang B gefunden werden.

Organisation der Information auf dem Server

Wie in den Anforderungen an das System gefordert, soll es möglich sein, den Algorithmus, der einem Editorfenster zugeordnet ist, dynamisch zu ändern. Damit der Benutzer weiß, welche Algorithmen ihm hierfür zur Verfügung stehen, muß diese Information auf dem Server abgelegt werden. Hierfür befindet sich im Basisverzeichnis auf dem Server ein Verzeichnis `Algorithms`. In diesem Verzeichnis finden sich Informationsdateien, welche Algorithmen existieren. Für jeden Datentypen, für den ein eigenes Editorfenster implementiert ist (z.B. `JGraph`), existiert ein Verzeichnis mit gleichem Namen wie der Datentyp. In all diesen Verzeichnissen befindet sich eine Datei mit dem Namen `info`. Diese Dateien sind wie folgt aufgebaut. In der ersten Zeile steht die Anzahl der implementierten Algorithmen, in den folgenden Zeilen stehen jeweils die Bezeichner der Klassen, die diese Algorithmen implementieren. Die Algorithmen die im Standard-Editorfenster (z.B. Sortieralgorithmen) laufen können, befinden sich in der Informationsdatei, die sich im Verzeichnis `Algorithms` befindet. In Abbildung 3.8 ist ein Beispiel für einen solchen Verzeichnisbaum dargestellt.

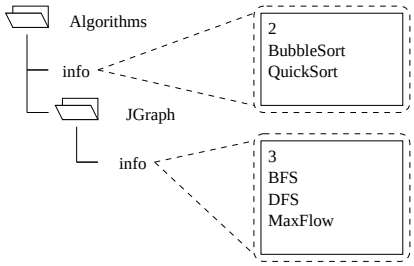


Abbildung 3.8 *Verzeichnisbaum für die Algorithmeninformation*

Mit den bisherigen Schritten ist es also möglich, einen Algorithmus zu starten und Algorithmen im Editorfenster zu wählen. Nun soll noch beschrieben werden, wie einzelne Datenstrukturen auf dem Server bereitgestellt werden können, um diese in einem Parameterfenster auszuwählen. Hierfür existiert im Basisverzeichnis ein Unterverzeichnis mit dem Namen `Parameter`, um die einzelnen Instanzen der Datenstrukturen aufzunehmen. Die Datenstrukturen werden analog zu den Algorithmen gruppiert. D.h. für jeden Datentypen ist ein eigenes Unterverzeichnis angelegt, in dem die Instanzen abgespeichert sind. Diese Instanzen können dann vom Client angefordert werden.
Um bei der Auswahl der Instanzen im Parameterfenster schnell auf die Namen der verfügbaren Instanzen zurückgreifen zu können, wird ein sogenanntes Repository verwednet. Hierfür wird eine eigene Datei angelegt, in der die entsprechenden Namen zu jeder Datenstruktur gespeichert werden. Die Informationen dazu liegen in einer Datei `info` im Verzeichnis `Parameter`. In Abbildung 3.9 ist wieder ein möglicher Verzeichnisbaum dargestellt.

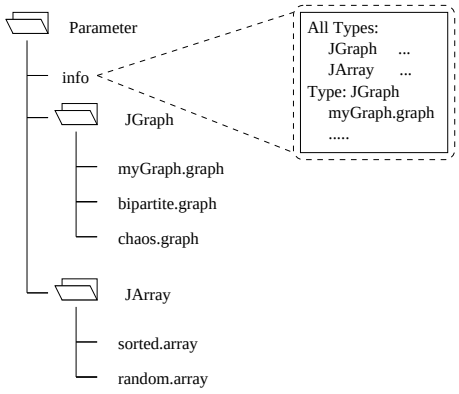


Abbildung 3.9 *Verzeichnisbaum für die Beispiel-Datenstrukturen*

Die Datei `info` wird automatisch von einer Instanz der Klasse `edu.tum.dal.Repository` generiert. Eine Instanz der Klasse `edu.tum.dal.TypeList`, die sich zur Laufzeit im Server befindet, liest diese Datei und wird dazu verwendet, auf die gespeicherten Informationen zurückzugreifen. Der genaue Gebrauch des Repository ist dem Anhang B.3 zu entnehmen.
Mit diesen Datenstrukturen können nun vordefinierte Parameterbelegungen für die einzelnen Algorithmen definiert und dem Benutzer zur Auswahl angeboten werden. Diese Beispiele werden in einen Unterverzeichnis des Basisverzeichnisses abgelegt, das den Namen `Examples` hat. In diesem Verzeichnis befindet sich für jede Basisklasse, für die ein Algorithmus definiert ist (vergleiche *package*-Namen für Algorithmen), ein eigenes Verzeichnis. In diesen Verzeichnissen findet sich dann für jeden zugehörigen Algorithmus nochmals ein eigenes Verzeichnis, in dem die vordefinierten Belegungen gespeichert sind. Auch diese Verzeichnisstruktur ist wieder in einer Abbildung verdeutlicht (Abbildung 3.10).

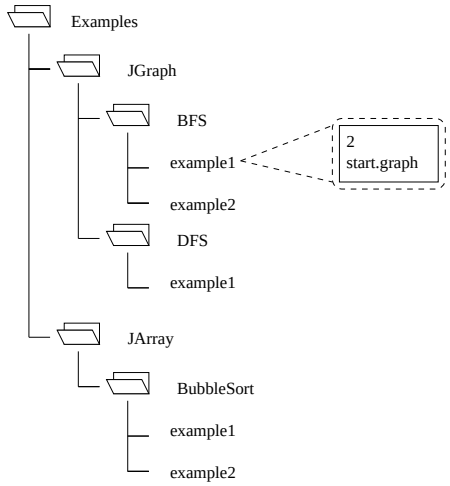


Abbildung 3.10 *Verzeichnisbaum für die vordefinierten Parametereinstellungen*

Durch diese anfänglich etwas unübersichtliche Verzeichnisstruktur ist sichergestellt, daß jedem Algorithmus auch ein Verzeichnis zugeordnet ist. In diesem Verzeichnis befindet sich für jede Parameterbelegung eine eigene Datei, die den Namen `example` gefolgt von der Beispielnummer trägt, z.B. `example3`. Diese Datei wird bei Bedarf von

einem sogenannten ExampleLoader gelesen und entsprechend kodiert an das jeweilige Parameterfenster weitergeleitet. Eine Beschreibung der Implementierung eines solchen ExampleLoader findet sich wieder im Anhang (siehe B.2).
Die letzte Gruppe von Informationen, die auf dem Server abgelegt sind, enthält die Beispiele, die im Rahmen der Tutorials angezeigt werden können. Die Instanzen der Datenstrukturen, die in den Tutorials angezeigt werden sollen, werden auf die gleiche Weise wie die Parameterbelegungen für einen Algorithmus gruppiert. Im Basisverzeichnis existiert also ein Verzeichnis `Tutorial`, in dem sich für jede Datenstruktur, auf der ein Algorithmus implementiert ist, ein Unterverzeichnis befindet. In diesen Unterverzeichnissen ist dann für jeden Algorithmus, der auf der entsprechenden Datenstruktur definiert ist, das Verzeichnis für die Tutorialbeispiele angelegt. Auch dieser Verzeichnisbaum ist wieder exemplarisch in Abbildung 3.11 veranschaulicht.

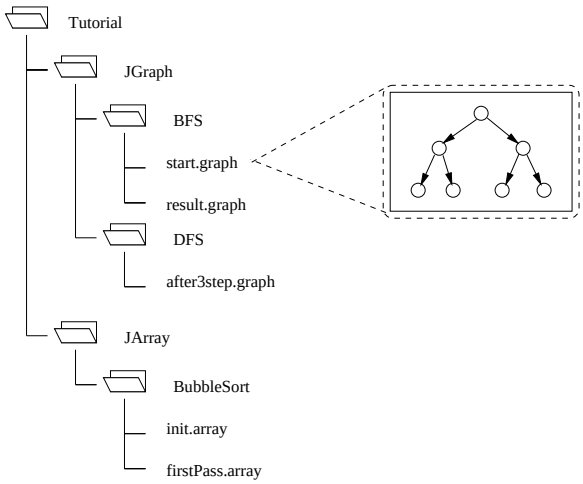


Abbildung 3.11 *Verzeichnisbaum für die Tutorialbeispiele*

In dem Basisverzeichnis befinden sich also vier Unterverzeichnisse. Diese Unterverzeichnisse sind zur Übersicht nochmals aufgelistet:

1. `Algorithms` Beschreibungen der Algorithmen
2. `Examples` vordefinierte Beispielbelegungen
3. `Parameter` Datenstrukturen, die als Parameter verwendet werden können
4. `Tutorial` Beispieldatenstrukturen für die Tutorials

Sollte das System erweitert werden, ist es sinnvoll, eventuelle weitere Daten auf dem Server ebenfalls in diesem Basisverzeichnis anzusiedeln.

HTML-Struktur

Nachdem alle Schritte erklärt wurden, die ein Entwickler vornehmen muß, um einen Algorithmus mit allen durch das System bereitgestellten Möglichkeiten (Tutorial und vordefinierten Beispielaufrufen) im Netz zu präsentieren, soll jetzt betrachtet werden, wie die einzelnen Aufrufe auf den HTML-Seiten angeordnet und verwendet werden. Die Beschreibung wird anhand der vorgegebenen Dreiteilung der HTML-Seiten durch die Links im Steuerungsframe durchgeführt.

1. Tutorial — Der Link "Tutorial" im Steuerungsframe bewirkt das Laden einer Startseite für die Tutorials. Im allgemeinen wird es sich dabei um eine Übersichtseite handeln, von der aus man dann die einzelnen Tutorials durch Links erreichen kann. Diese Startseite muß sich in einem Unterverzeichnis `Tutorial` des Verzeichnisses befinden, in dem die Startseite abgelegt ist, und den Namen `tutorial.html` tragen.
   Wenn in einem der Tutorials ein Beispiel, in Form einer Instanz einer Datenstruktur, in einem Editorfenster dargestellt werden soll, kann dies durch einen Knopf, der mit einem JavaScript-Befehl hinterlegt ist, durchgeführt werden. Der Befehl bewirkt das Aufrufen einer Methode des Controller-Applets mit dem Namen `showTutorialExample`. Der Code in der HTML-Seite hat folgende Gestalt:

   ```
   <form>
   <input type=button value="Abbildung 1"
    onclick="parent.index.document.controller.showTutorialExample
             ('BFS','JGraph','bfs1.graph');">
   </form>
   ```

   Dieser Code bewirkt das Anzeigen eines Knopfes mit der Aufschrift "Abbildung 1". Beim Drücken des Knopfes wird ein `JGraph` aus der Beispieldatei "bfs1.graph" für den Algorithmus "BFS" vom Server angefordert und der empfangene Graph dann in einem Editorfenster angezeigt. Da für den `JGraph` ein eigenes Editorfenster implementiert ist, würde in diesem Fall dieses auch verwendet werden, als *default*-Fenster wird das Standard-Editorfenster instantiiert. Bei dem Aufruf ist wichtig, daß der erste Parameter den Namen des Algorithmus und der zweite dem Bezeichner der Datenstruktur entspricht, da sonst das entsprechende Tutorialexample auf dem Server nicht gefunden bzw. die Instanz nicht geladen werden kann (vgl. Verzeichnisstruktur für Tutorialexample weiter oben in diesem Teilabschnitt).

2. Beispiele — Dieser Link führt auf die Startseite für die vordefinierten Parametereinstellungen (Unterverzeichnis: `Examples`, Datei:`examples.html`). Auch diese Seite kann wieder Links auf andere Seiten enthalten, um eine geeignete Gruppierung der Aufrufe zu erhalten. Die Auswahl eines Algorithmus mit vordefinierten Parametern bewirkt einen Aufruf von `initAlgorithmParameter`, einer Methode des Controller-Applets. Auch dieser Aufruf ist durch einen Knopf möglich:

   ```
   <form>
   <input type=button value="Beispielbelegung 1"
    onclick="parent.index.document.controller.initAlgorithmParameter
             ('BFS','JGraph',1);">
   </form>
   ```

   In diesem Fall wird die Parameterbelegung mit dem Name `example1` des Algorithmus "BFS" auf der Datenstruktur "JGraph" vom Server geladen. Soll eine Beispielbelegung von der lokalen Platte geladen werden, so kann an Stelle des Integerwertes, der die Beispielnummer angibt, auch der absolute Dateiname angegeben werden.

3. Algorithmen — Auch hier wird eine entsprechende Startseite geladen (Unterverzeichnis: `Algorithms`, Datei: `algorithms.html`). Auf dieser oder einer durch einen Link erreichbaren Seite soll es möglich sein einen Algorithmus zu starten, ohne eine vordefinierte Parametereinstellung zu verwenden. Der Aufruf ist analog zu dem, der eine vordefinierte Parameterbelegung verwendet, nur daß der Zahlwert diesesmal auf den Wert "0" gesetzt wird. Ein Aufruf, der sich wieder hinter einem Knopf versteckt, könnte also beispielsweise wie folgt aussehen:

```
<form>
<input type=button value="Breitensuche"
 onclick="parent.index.document.controller.initAlgorithmParameter
          ('BFS','JGraph',0);">
</form>
```

Mit diesen drei Möglichkeiten zum Aufruf von Methoden im Controller soll der Teil zur Beschreibung des Systems abgeschlossen werden. Es wurde gezeigt, welche Informationen auf dem Server abgelegt werden müssen und wie auf diese Informationen zurückgegriffen werden kann.

Algorithmen mit mehreren Datenstrukturen

Algorithmen, die mehrere Datenstrukturen als Parameter benutzen, können ebenfalls implementiert werden. Der Algorithmus wird dann der Datenstruktur zugeordnet, in deren Editorfenster er anzuzeigen ist. Soll er in dem *default*-Fenster angezeigt werden, da kein spezielles Editorfenster existiert, wird er einer der verwendeten Datenstrukturen zugeordnet. Eine analoge Gruppierung ergibt sich auch für die vordefinierten Parameterbelegungen.
Die Datenstrukturen für die Tutorials hingegen müssen in dem Verzeichnis abgelegt werden, das dem Namen der Datenstruktur entspricht. In diesem Verzeichnis findet sich dann das Unterverzeichnis mit dem Namen des Algorithmus. Es ist also möglich, daß sich in mehreren Verzeichnissen, die jeweils zu verschiedenen Datenstrukturen gehören, Unterverzeichnisse mit gleichem Namen (dies ist der Bezeichner des Algorithmus) existieren.

3.3 Push-Relabel-Algorithmus

Nachdem in den vorangegangenen Abschnitten das implementierte System dargestellt wurde, soll nun ein darin realisierter Algorithmus beschrieben werden. Es handelt es sich dabei um den Push-Relabel-Algorithmus von Goldberg und Tarjan (siehe Kapitel 2).
Bei der Implementierung des Algorithmus wurden, neben der reinen Umsetzung des Algorithmus, einige Erweiterungen vorgenommen:

- So wurden u.a. die in Abschnitt 2.4 vorgestellten Strategien Gap- und Global-Relabeling umgesetzt.
- Um ein schnelles Arbeiten auf großen Graphen zu ermöglichen, wurde eine eigene effiziente Graphklasse implementiert. Der Algorithmus läuft also nicht auf dem angezeigten Graphen, sondern auf einer Instanz dieser neuen Datenstruktur.

Hierdurch ist es möglich, die zeitaufwendige Visualisierung zu umgehen. Soll der Graph angezeigt werden, so müssen nun alle Änderungen im Graphen zur Laufzeit umgesetzt werden. Da eine Visualisierung jedoch meist nur bei kleinen Graphen sinnvoll eingesetzt werden kann, ist die dadurch bedingte Mehrarbeit gering und somit kein "Rucken" des Algorithmus zu bemerken. Ferner kann die maximale Anzahl von Knoten eingestellt werden, bis zu der die Graphen anzuzeigen sind. Mit diesen beiden Erweiterungen ist es möglich, auch große Graphen als Parameter des Algorithmus zu verwenden und diese nicht visualisieren zu lassen. Hierdurch wird die Laufzeit des Algorithmus deutlich verringert, was den Gebrauch der großen Graphen erst möglich macht.

- Zur Erzeugung von Graphen wurden vier Generatoren des *First DIMACS Implementation Challenge* [JM93] implementiert, die auch in der Arbeit von Cherkassky und Goldberg [CG94] zu Testzwecken verwendeten werden. Die erzeugten Graphen sind durch speziell codierte Textdateien beschrieben. Solche Dateien können ebenfalls geladen und gespeichert werden.

- Um die Verbesserungen beim Einsatz der implementierten Heuristiken einfacher untersuchen zu können, wurde, zusätzlich zu einer einfachen Ausgabe von Meßwerten eines Algorithmus, noch die Möglichkeit geschaffen, die Meßwerte von mehreren Algorithmenläufen (auch für verschiedene Eingabegraphen) graphisch darstellen zu lassen.

In den folgenden Teilabschnitten wird der Gebrauch des Algorithmus, sowie der Erweiterungen, erläutert.

3.3.1 Parametereinstellung

Zuerst soll beschrieben werden, welche Parametereinstellungen der Benutzer vornehmen kann. Der Ablauf des Algorithmus hängt wesentlich von der Wahl der Quelle und der Senke ab. Diese beiden ausgezeichneten Knoten sind vom Benutzer zu wählen. Falls der Graph nicht zu groß ist und noch im Editorfenster angezeigt werden kann, wird der Benutzer nach dem Starten des Algorithmus aufgefordert, diese beiden Knoten auszuwählen. Bei generierten Graphen wird die empfohlene Vorauswahl durch die Beschriftungen "s" und "t" für die Quelle und die Senke angezeigt. Dem Benutzer steht es aber frei, andere Knoten zu wählen. Die Wahl der Knoten wird nochmals in einem Auswahlfenster angezeigt, das nach der Selektion der Knoten erscheint. In den Fällen, in denen der Graph nicht mehr angezeigt werden kann, wird dieses Auswahlfenster sofort eingeblendet. Eine Vorauswahl der Quelle und Senke wird dann in diesem Fenster angezeigt. Das Fenster ist in Abbildung 3.12 dargestellt.
Der Algorithmus ist so implementiert, daß die Auswahl der Knoten für die Push/Relabel-Operation entweder durch eine FIFO-Warteschlange bzw. mit der HLF-Methode durchgeführt wird. Im Auswahlfenster kann entschieden werden, welche der beiden Varianten zu benutzen ist. Desweiteren kann ausgewählt werden, ob die zugehörige Datenstruktur (`JQueue` bzw. `JPriorityQueue`) ebenfalls im Editorfenster angezeigt wird.
Auch die in Abschnitt 2.4 vorgestellten Strategien sind implementiert. Die Aufteilung in zwei Phasen wird immer vorgenommen. Der Benutzer kann jedoch wählen, ob Gap- bzw. Global-Relabeling verwendet werden soll. Beim Einsatz von Global-Relabeling kann ausgewählt werden, nach wievielen Standard-Relabel-Operationen diese Strategie angewendet wird. Als Default-Wert ist die Knotenanzahl des aktuellen Graphen eingesetzt.

Falls die Strategie für das Gap-Relabeling verwendet wird, kann man auch die Liste, in der die Entfernungen der Knoten gespeichert sind, anzeigen lassen.

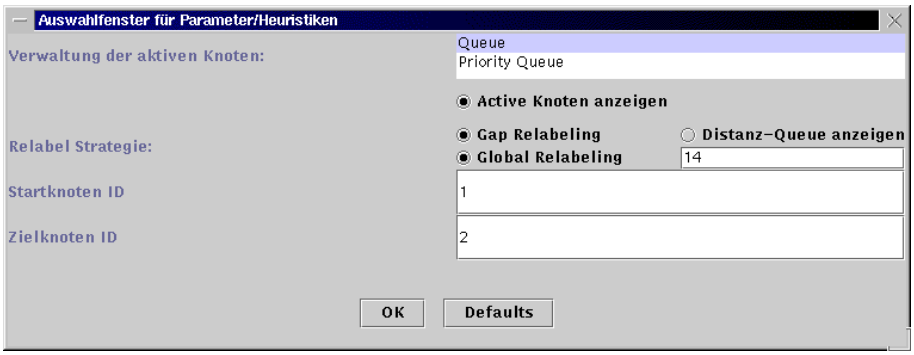


Abbildung 3.12 *Auswahlfenster des Push-Relabel-Algorithmus*

3.3.2 Generatoren

Wie bereits erwähnt, handelt es sich bei den Generatoren um diejenigen, die auch beim *First DIMACS Implementation Challenge* verwendet wurden. Insgesamt sind vier Generatoren implementiert. Der erste, im folgenden *Genrmf* genannt, wurde von Goldfarb und Grigoriadis [GG88], der zweite (*Washington*) von Anderson und Studenten in einem Seminar und der dritte (*AC*) von Sutubal (eine C-Version eines Generators von Waissi) entwickelt. Der vierte der Generatoren (*AK*) ist in der Arbeit von Cherkassky und Goldberg [CG94] vorgestellt.
Diese Generatoren wurden, ausgehend von den Source-Codes, die in der Programmiersprache C geschrieben sind, in die Sprache Java übertragen und stehen im Editor als eigene Menüpunkte zur Verfügung. Die Generatoren erzeugen einen Graphen, der in dem bereits weiter oben erwähnten *DIMACS*-Format gespeichert werden kann. Für kleine Graphen ist eine Layout-Methode implementiert, die die erzeugten Graphen automatisch in einen `JGraph` umsetzt, der dann angezeigt wird. Für größere Graphen hingegen ist keine übersichtliche Darstellung mehr möglich. Es folgt nun eine kurze Beschreibung der jeweils generierten Graphen:

1. *Genrmf (a,b,c_1,c_2)* — Dieser Graph besteht aus insgesamt b hintereinandergeschalteten Teilgraphen mit a^2 Knoten. Je zwei aufeinanderfolgende Teilgraphen sind durch a^2 Kanten miteinander verbunden. Hierfür ist jeder Knoten des einen Teilgraphen mit genau einem Knoten aus dem anderen Teilgraphen verbunden. Die Zuordnung wird zufällig bestimmt. Der Zufallsgenerator kann mit einem sogenannten *seed*-Wert, der optional vom Benutzer festgelegt ist, initialisiert werden.
   In den einzelnen Teilgraphen sind die Knoten in Gitterform angeordnet (Quadrat mit der Kantenlänge a), wobei jeder Knoten mit seinen maximal vier Nachbarn (die Knoten an den Rändern haben entsprechend weniger Nachbarn) verbunden ist. Da der Graph gerichtet ist, verlaufen also immer zwei Kanten zwischen zwei benachbarten Knoten.
   Die Gewichte der Kanten zwischen den Teilgraphen werden jeweils zufällig aus dem Intervall, das durch zwei auszuwählende Startwerte c_1 und c_2 (mit $0 < c_1 \leq c_2$) begrenzt ist, ausgewählt. Allen Kanten in den Teilgraphen wird das Gewicht $a^2 \cdot c_2$ zugeordnet.

Als Quelle wird der Knoten in der "linken oberen" Ecke des ersten Teilgraphen und als Senke der Knoten in der "rechten unteren" Ecke des letzten Teilgraphen verwendet. In Abbildung 3.13 ist ein Graph für die Werte $a = 3$ und $b = 2$ dargestellt. Die Gewichte der Kanten sind aus Gründen der Übersichtlichkeit nicht eingetragen, ferner sind Quelle und Senke durch die Bezeichner s und t hervorgehoben.

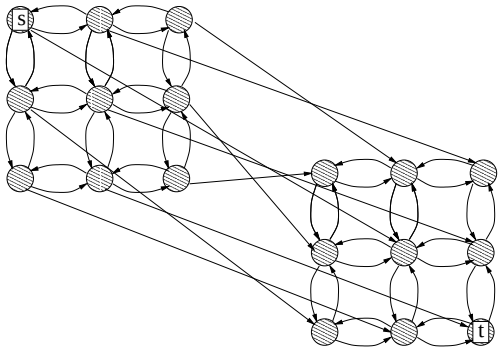


Abbildung 3.13 *Graphen des Generators Genrmf*

2. *Washington (m,n,r)* — Mit diesem Generator können zwei verschiedene Graphklassen erzeugt werden, die beide $2 + m \cdot n$ Knoten (mit $m \geq 3$ und $n \geq 1$) besitzen.

   (a) *Random Level* — Bei diesem Graph werden $m \cdot n$ Knoten in Form eines Gitters mit der Höhe m und der Breite n angeordnet. Die beiden verbleibenden Knoten bilden die Quelle und die Senke. Die Quelle ist mit allen Knoten der ersten und die Senke mit allen Knoten der letzten Spalte durch eine Kante mit der Kapazität $3 \cdot r$ verbunden. In dem Gitter ist jeder Knoten mit drei zufällig (wieder ist eine Initialisierung durch einen *seed*-Wert möglich) gewählten Knoten der nächsten Spalte in Richtung Senke, falls diese existiert, verbunden. Die Kapazitäten dieser Kanten werden zufällig aus dem Intervall $[0, m]$ gewählt. In Abbildung 3.14 ist ein solcher Graph für die Werte $m = 4$ und $n = 3$ dargestellt.

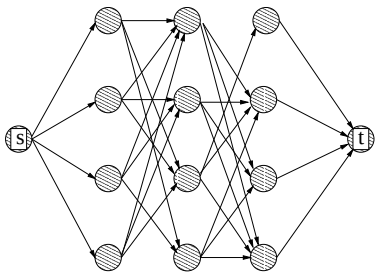


Abbildung 3.14 *Graphen des Typs Random Level des Generators Washington*

   (b) *Basic Line* — Bei diesem Graphtyp sind die $n \cdot m$ Knoten nicht in einem Gitter, sondern in einer Linie angeordnet. Dieses Mal ist die Quelle mit den ersten m

und die Senke mit den letzten m Knoten der Linie durch Kanten verbunden. Die Kapazität berechnet sich aus dem Produkt der im System festgelegten Obergrenze für den Wert r und und einem ebenfalls im System definierten Wert C_{max}. Die linear angeordneten Knoten sind wie folgt verbunden. Für jeden Knoten i werden zufällig (ein *seed*-Wert ist wieder möglich) r andere Knoten der Linie ausgewählt, aber es wird nur dann eine Kante vom Knoten i zu einem ausgewählten Knoten gezogen, wenn sich dieser näher an der Senke befindet als der Knoten i selbst. Es ist also möglich, daß ein Knoten weniger als r ausgehende Kanten besitzt. Im Extremfall kann er sogar keine ausgehenden Kanten haben. Für den Eingangsgrad eines Knotens gilt analog, daß er zwischen 0 und $n \cdot m$ liegen kann. Als Kapazität für diese Kanten wird zufällig ein Wert aus dem Intervall $[0, C_{max}]$ gewählt. Auch für diesen Graphtyp ist in Abbildung 3.15 ein Graph dargestellt ($m = 3$, $n = 3$ und $r = 2$).

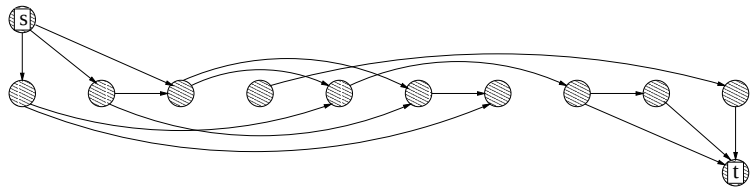


Abbildung 3.15 *Graphen des Typs Basic Line des Generators Washington*

3. *AC (n)* — Dieser Generator dient dazu, einen möglichst dichten azyklischen Graphen zu erzeugen. Hierfür werden n Knoten linear angeordnet. Der erste Knoten in der Reihe wird zur Quelle und der letzte zur Senke bestimmt. Nun wird für jeden Knoten eine Kante zu allen Knoten eingefügt, die in der Reihe näher an der Senke liegen. Die Kapazität der Kanten wird zufällig (unter Verwendung eines *seed*-Wertes) aus dem Intervall $[1{,}10^7]$ ausgewählt. In Abbildung 3.16 ist ein solcher Graph für 5 Knoten gezeigt.

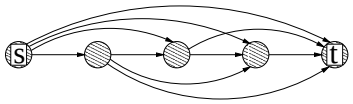


Abbildung 3.16 *Graphen des Generators AC*

4. *AK (r)* — Dieser Generator erzeugt einen Graphen, der im wesentlichen aus zwei parallelgeschalteten Teilgraphen besteht, d.h. von der Quelle geht je eine Kante zum Startknoten des ersten Graphen und zum Startknoten des zweiten Graphen. Analog sind zwei Kanten zur Senke vorhanden. Diesen vier Kanten wird das Gewicht 10^6 zugeordnet. Nun zur Beschreibung der beiden Teilgraphen:

   (a) Dieser Teilgraph besteht aus $2r+2$ Knoten. Zur Beschreibung wird angenommen, daß diese Knoten in zwei horizontalen übereinanderliegenden Linien mit je $r+1$ Knoten angeordnet sind. Der Startknoten des Teilgraphen ist der linkeste Knoten in der unteren, und der Endknoten der rechteste Knoten der

oberen Linie. Die einzelnen Knoten werden wie folgt durch Kanten verbunden. In der unteren Reihe wird der i-te Knoten von links mit dem $i+1$-ten Knoten, so er denn existiert, verbunden. Die Kapazität der Kante wird auf $r+2-i$ festgelegt. Ferner wird jeder Knoten der unteren Reihe mit dem ersten Knoten der oberen Reihe durch eine Kante mit der Kapazität 1 verbunden. Zusätzlich wird der letzte Knoten der unteren Reihe durch eine weitere Kante, die ebenfalls Gewicht 1 hat, mit dem letzten Knoten der oberen Reihe verknüpft. Abschließend wird in der oberen Zeile jeder Knoten, falls möglich, mit seinem rechten Nachbarn durch eine Kante mit der Kapazität $r+1$ verbunden. In diesem Teilgraph können also maximal $r+2$ Flußeinheiten vom Start- zum Zielknoten fließen.

(b) Auch dieser Teilgraph besteht aus $2r+2$ Knoten, die diesmal in zwei vertikalen Reihen zu je $r+1$ Knoten angeordnet sind. Der oberste Knoten in der linken Reihe ist der Start- und der oberste Knoten in der rechten Reihe ist der Zielknoten des Teilgraphen. Die Knoten der linken Reihe werde von oben nach unten, und die Knoten der rechte Reihe von unten nach oben durch Kanten mit der Kapazität r zu zwei Ketten verbunden. Eine Kante, die ebenfalls Kapazität r besitzt und zwischen dem untersten linken und dem untersten rechten Knoten verläuft, verbindet diese beiden Ketten miteinander. Desweiteren werden die ersten r Knoten der beiden Reihen so miteinander verbunden, daß jeweils zwischen dem i-ten Knoten der linken und der rechten Reihe eine Kante mit der Kapazität 1 verläuft. In diesem Teilgraphen können maximal $r+1$ Flußeinheiten vom Start- zum Zielknoten fließen.

Es ergibt sich also insgesamt ein Graph mit einem maximalen Fluß von $2r+3$ Einheiten (vorausgesetzt, der Parameter r ist nicht zu groß, d.h. $r \leq 10^6 - 2$). In Abbildung 3.17 ist wieder ein Beispielgraph für den Wert $r = 2$ dargestellt. Da die Kantenkapazitäten jetzt deterministisch sind, sind die Werte zum besseren Verständnis an den Kanten angetragen.

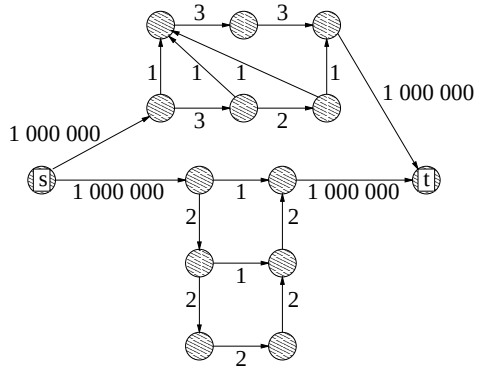


Abbildung 3.17 *Graphen des Generators AK*

Diese vier Generatoren werden auch verwendet, um eine Analyse der verschiedenen Strategien für den Algorithmus durchzuführen. Die Ergebnisse der Untersuchungen werden in Kapitel 4 ausführlich erklärt. In diesem Kapitel werden auch die Eigenschaften der vier Graphfamilien besprochen.

3.3.3 Analysewerkzeug

Abschließend soll nun noch das zum Editorfenster hinzugefügte Analysewerkzeug beschrieben werden. Um die verschiedenen Strategien gut miteinander vergleichen zu können ist es wichtig, die Ergebnisse von verschiedenen Abläufen des Algorithmus gegenüberzustellen. Nach der Termination des Algorithmus werden hierfür einige Meßwerte am Bildschirm ausgegeben. Hierbei handelt es sich um die folgende Werte:

- Laufzeit
  - Gesamtlaufzeit
  - Laufzeit der 1. Phase
  - Laufzeit der 2. Phase
- Push-Operationen
  - Anzahl der nichtsättigenden Push-Operationen
  - Anzahl der sättigenden Push-Operationen
  - Anzahl aller Push-Operationen
- Relabel-Operationen
  - Anzahl der Standard-Relabel-Operationen
  - Anzahl der Global-Relabel-Operationen
  - Anzahl der gefundenen Gaps beim Gap-Relabeling

Die Parameter, die die Laufzeit angeben, hängen sehr stark von dem Verlauf der Visualisierung ab. Da jedoch der Algorithmus auch durchgeführt werden kann, ohne daß eine Visualisierung stattfindet (siehe unten), ist es sinnvoll, auch diese Werte zu messen. Das angezeigte Ergebnisfenster ist in Abbildung 3.18 dargestellt.

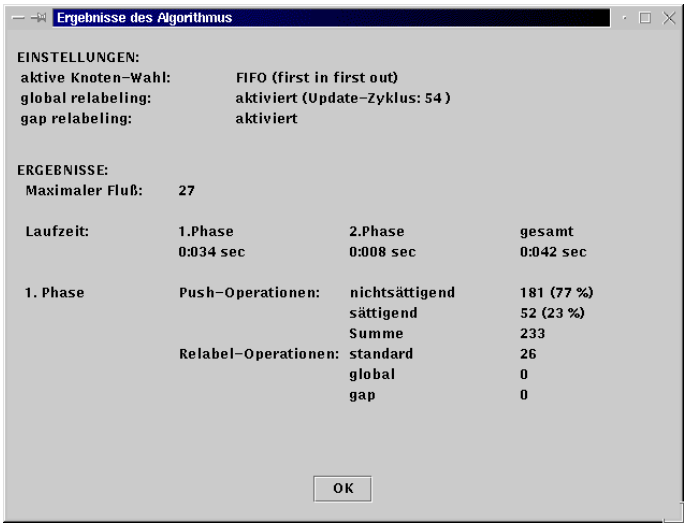


Abbildung 3.18 *Ergebnisfenster des Algorithmus*

Da es für den Benutzer jedoch umständlich ist, die einzelnen Meßwerte von verschiedenen Abläufen zu vergleichen, wenn er diese nach jedem Testlauf in einem Fenster präsentiert bekommt, wurde eine zusätzliche Analyseeinheit implementiert. Das Ziel dieser Einheit ist es, die Ergebnisse von mehreren Abläufen graphisch, also in Form von Funktionspunkten bzw. -kurven, gegenüberzustellen. Um ein benutzerfreundliches Anwenden des Werkzeugs zu ermöglichen, wurde es dreigeteilt. Der erste Teil ermöglicht das Auswählen von verschiedenen Graphen, der zweite das Festlegen der Parametereinstellungen über diesen Graphen und der letzte Teil die Definition der anzuzeigenden Werte. Die einzelnen Teile können durch Menüpunkte des Menüs "Statistik" ausgewählt werden. Im folgenden werden diese erläutert:

- **Graphauswahl** — Da es oft wünschenswert ist, die Ergebnisse von Testläufen auf verschiedenen Graphen anzeigen zu lassen, im Editorfenster aber maximal ein Graph geladen ist, können in diesem Auswahlfenster (siehe Abbildung 3.19) mehrere Graphen gespeichert werden. Für diese Graphen können dann, unabhängig davon, ob sie angezeigt sind oder nicht, Parametereinstellungen ausgewählt werden.

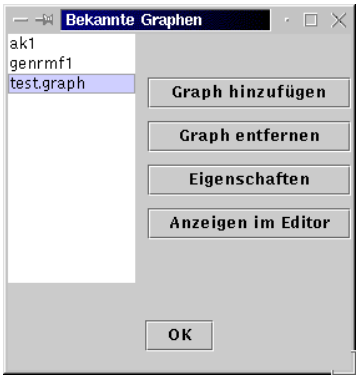


Abbildung 3.19 *Fenster für die Graphauswahl in der Statistikeinheit*

Graphen können zur Menge der gespeicherten Graphen hinzugefügt werden, indem sie im Editor geladen werden und anschließend im Auswahlfenster der Knopf "Graph hinzufügen" gedrückt wird. Ein in der Liste des Auswahlfensters markierter Graph kann mittels der Taste "Graph entfernen" wieder aus dieser Menge gelöscht werden. Der Knopf "Eigenschaften" zeigt die Eigenschaften eines Graphen in einem separaten Dialogfenster an (siehe Abbildung 3.20).

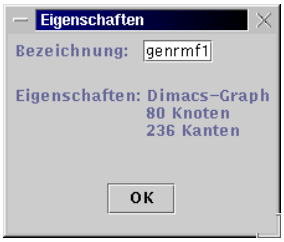


Abbildung 3.20 *Fenster für die Grapheigenschaften in der Statistikeinheit*

In diesem Fenster werden neben dem Typen des Graphen (`JGraph` oder `Dimacs`) noch die Knoten- und Kantenanzahl dargestellt. Ferner kann der Bezeichner des Graphen (*default*: Dateiname oder Name des Generators) geändert werden.

Durch Auswahl des Knopfes "anzeigen im Editor" wird der zur Zeit dargestellte Graph im Editorfenster durch den im Auswahlfenster selektierten Graphen ersetzt.

- **Parameterwahl** — In diesem Fenster (siehe Abbildung 3.21) können nun die Parametereinstellungen vorgenommen werden. Es stehen die beiden Knöpfe "Parameter hinzufügen" und "Parameter entfernen" zur Verfügung. Beim Hinzufügen von Parametern erscheint ein Fenster, das dem Paramterfenster des Algorithmus (siehe Abbildung 3.12) sehr ähnlich ist. Nur ist es in diesem Fenster zusätzlich möglich, den Graphen zu wählen. Wie bereits erwähnt, stehen hierfür die in dem Fenster Graphauswahl bestimmten Graphen zur Verfügung. Die getroffenen Parametereinstellungen werden in dem Übersichtsfenster in Kurzform dargestellt. So steht z.B. `genrmf1,FIFO,gap,global(80)` für eine Auswahl des Graphen `genrmf1` mit der FIFO-Strategie bei Verwendung von Gap-Relabeling und einer Global-Relabel-Operation nach allen 80 Standard-Relabel-Operationen.

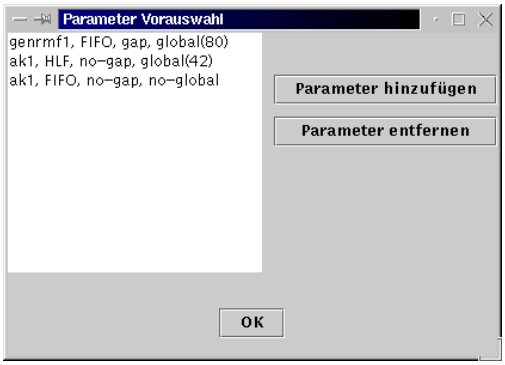


Abbildung 3.21 *Fenster für die Parameterwahl in der Statistikeinheit*

- **Anzeigedefinition** — Das letzte Fenster, das durch das Statistik-Menü geöffnet werden kann, dient zur Festlegung der Anzeige der Ergebnisse (siehe Abbildung 3.22). Hier können bis zu vier verschiedene Funktionen festgelegtwerden, die dann in einem Darstellungsfenster gezeichnet werden.

  Da nicht immer eindeutig bestimmt ist, ob auf der Abszisse die Knotenanzahl oder die Parameterwahl dargestellt werden soll, muß dies durch den Benutzer ausgewählt werden. In Fällen, in denen die Knotenanzahl verwendet wird und zu einer der auftretenden Knotenanzahlen mehrere Meßwerte als zur gleichen Funktion gehörend, angegeben sind (in solchen Fällen handelt es sich um keine Funktion mehr, diese Situation kann aber so ausgewählt werden), wird nur einer der Werte angezeigt.

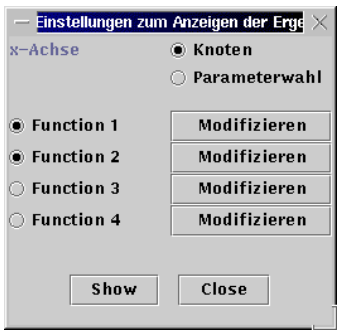


Abbildung 3.22 *Fenster für die Anzeigedefinition in der Statistikeinheit*

Die vier Funktionen selbst können durch Anklicken des jeweiligen Knopfes "Modifizieren" festgelegt werden. Dies geschieht in einem weiteren Fenster (siehe Abbildung 3.23). In diesem Fenster können der Funktionsname, die Farbe, in der die Funktion gezeichnet werden soll, sowie die zu einer Funktion gehörenden Parameterauswahlen, die die Werte auf der Absizze bestimmen, festgelegt werden. Eine Auswahl von mehreren Werten aus der dargestellten Liste ist durch das Anklicken mit der Maus und dem gleichzeitigen Drücken der Tasten SHIFT bzw. CTRL möglich. Als Funktionswert stehen für eine Funktion die neun weiter oben aufgezählten Meßwerte zur Verfügung.

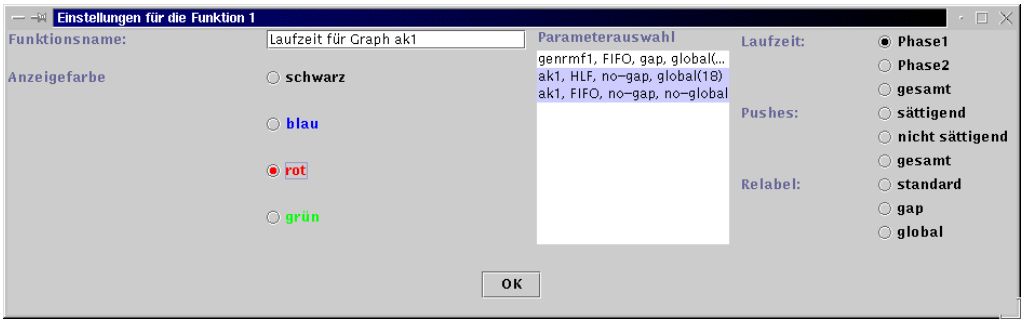


Abbildung 3.23 *Fenster für die Funktionsdefinition in der Statistikeinheit*

Das Drücken der Taste "Show" im Hauptfenster der Definitionsfestlegungen (siehe Abbildung 3.22) bewirkt das Anzeigen der ausgewählten Funktionen. Hierfür wird ein Fenster verwendet, in dem die Ergebnisse in einem Koordinatensystem dargestellt werden (siehe Abbildung 3.24). Durch den Menüpunkt "Starten der Berechnungen" können die Berechnungen der definierten Testläufe manuell gestartet werden. Beim Anzeigen der Funktionen wird zur Sicherheit überprüft, ob es Testläufe gibt, die noch nicht berechnet sind. Wenn dies zutrifft, werden die entsprechenden Testläufe automatisch durchgeführt.

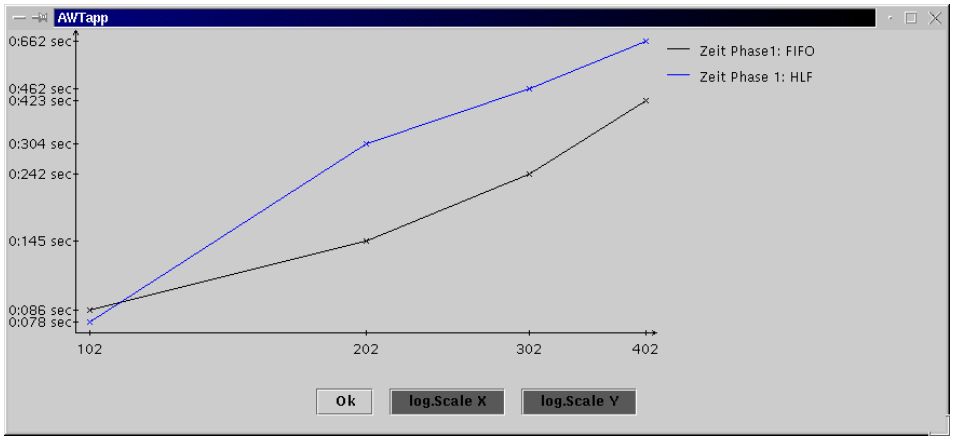


Abbildung 3.24 *Fenster zum Zeichen der Funktionen in der Statistikeinheit*

Auf der Abszisse sind je nach Wahl die Knotenanzahlen der verwendeten Graphen oder die Nummern der Parameterwahlen dargestellt. Durch zwei Knöpfe kann für beide Achsen zwischen linearer und logarithmischer Darstellung gewechselt werden.

Kapitel 4

Analyse

Wie bereits angekündigt, werden in diesem Kapitel die im Theorie-Teil (Kapitel 2) vorgestellten Verbesserungen und Strategien analysiert und bewertet. Diese Untersuchungen werden mittels der Graphfamilien, die durch die implementierten Generatoren (siehe Teilabschnitt 3.3.2) erzeugt werden, durchgeführt. Eine Verwendung dieser Generatoren liegt nahe, da sie auch in dem ebenfalls bereits erwähnten *First DIMACS Implementation Challenge* zum Vergleich von verschiedenen Flußalgorithmen verwendet wurden und somit ein repräsentatives Ergebnis erwarten lassen. Ferner können die Ergebnisse von früheren Tests zur Einordnung des Push-Relabel-Algorithmus in die Menge der Flußalgorithmen verwendet werden.
Das Kapitel ist zweigeteilt. Im ersten Teil werden die Methoden HLF und FIFO verglichen. Diese Analyse wird anhand der Untersuchung der Eignung und Besonderheiten der einzelnen Graphtypen für den Push-Relabel-Algorithmus durchgeführt. Nach jeweils einer kurzen theoretischen Überlegung werden die zwei verschiedenen Auswahltechniken für die aktiven Knoten in mehreren Testläufen angewandt. Anhand der Ergebnisse sollen die theoretischen Überlegungen überprüft werden.
Der zweite Teil beschäftigt sich mit den vorgestellten Strategien und der Entwicklung einer Heuristik für ihren Einsatz. Auch für diesen Zweck werden die generierten Graphfamilien verwendet.
Zum Schluß werden die gewonnenen Ergebnisse zusammengefaßt. Sie werden ebenfalls mit denen verglichen, die in der Arbeit von Cherkassky und Goldberg [CG94] angegeben sind.

4.1 Analyse der generierten Graphen

Die Untersuchungen werden für alle vier Graphtypen getrennt durchgeführt. Die Testläufe werden, bis auf die Auswahl der aktiven Knoten, alle mit der gleichen Parameterwahl durchgeführt. D.h. es wird sowohl Global- als auch Gap-Relabeling verwendet. Das Intervall für das Global-Relabeling wird durch die Knotenanzahl der jeweiligen Graphen bestimmt. Diese Parametereinstellung wurde anhand der Ergebnisse der oben angesprochenen Arbeit von Cherkassky und Goldberg [CG94] vorgenommen. Es wurden die gleichen Parametereinstellungen verwendet, um Testreihen zu erhalten, die mit dieser Arbeit vergleichbar sind. Ferner scheinen diese Einstellungen günstig zu sein, da sie in der Arbeit von Cherkassky und Goldberg die besten Ergebnisse lieferten.
Bei den Laufzeitmessungen wurden für jede Parameterwahl immer mehrere Testläufe

durchgeführt. Da die Testläufe nicht in einem Einbenutzer-System durchgeführt wurden, kam es bei den Testläufen, je nach aktueller Last, zu unterschiedlichen Ergebnissen. Ferner wird durch das Arbeiten mit Java und dem damit verbundenen Einsatz der *Java Virtual Machine* an gewissen Stellen eine sogenannte *garbage collection* durchgeführt, die die Laufzeit teilweise erhöht. Die Testläufe für eine Graphfamilie wurden daher direkt hintereinander vorgenommen und jeweils der kleinste Meßwert aufgezeichnet. Dies ermöglicht eine annähernd gleiche Last auf dem Rechner und verringert zusätzlich den Einfluß der JVM. Durch dieses Verfahren lieferten die einzelnen Testreihen sinnvolle Werte, d.h. innerhalb einer Testreihe können die Werte ohne Probleme miteinander verglichen werden. Die Vergleiche von mehreren Testreihen für verschiedene Graphen können jedoch mit nicht vernachlässigbaren Fehlern behaftet sein.

4.1.1 Graphfamilie Ak

Der Ak Generator wurde von Cherkassky und Goldberg entwickelt, um Graphen zu erhalten, die "schwere" Flußprobleme für den Push-Relabel-Algorithmus darstellen.

Theoretische Überlegungen

Zuallererst soll die Aussage "schwere" Flußprobleme erläutert werden. Der Graph kann wie bereits beschrieben in zwei Teilgraphen unterteilt werden. Jeder dieser Teilgraphen ist so angelegt, daß relativ viele Push- und Relabel-Operationen ausgeführt werden müssen. Bei der Initialisierung der Entfernungsfunktion erhalten in beiden Teilgraphen die Knoten die in Abbildung 4.1 dargestellten Werte.

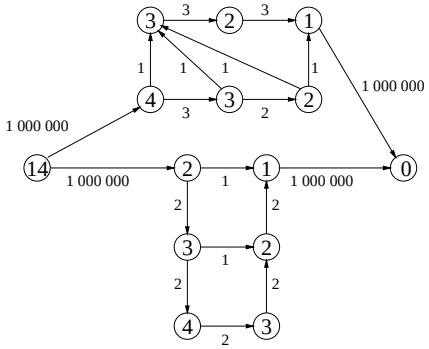


Abbildung 4.1 *Entfernungswerte nach der Initialisierung im Graphen Ak*

In der ersten Phase bewirkt diese Initialisierung, daß auf (fast) jeden Knoten in der unteren bzw. linken Reihe eine Relabel-Operation durchfgeührt werden muß. Diese Operation findet vor jedem zweiten Push dieser Knoten statt. Die vielen Push-Operationen treten auf, da alle Push-Operationen von der unteren in die obere bzw. von der linken in die rechte Reihe immer nur eine Flußeinheit verschieben können.
Auch die Strategien des Gap- bzw. Global-Relabeling bewirken keine Verbesserung, da die Entfernungsfunktion immer den exakten Wert für einen kürzesten Pfad zur Senke angibt und keine Lücken bei den Entfernungswerten auftreten.
Bei einem Vergleich der beiden Methoden zur Auswahl der aktiven Knoten ist zu erkennen, daß es wegen der Struktur des unteren Teilgraphen sinnvoller ist, die HLF-Variante zu verwenden.

Im oberen Teilgraph ist dies noch nicht zu erkennen. Der Teilgraph ist so aufgebaut, daß bei beiden Varianten die Flußeinheiten, die im linken oberen Knoten ankommen immer weitergeschoben werden, bevor weitere Flußeinheiten von anderen Knoten eintreffen. Da immer nur eine Flußeinheit ankommt, bedeutet dies, daß für jede Flußeinheit immer ein eigener Push durchgeführt wird. Wesentlich für diesen Effekt ist die Kante zwischen den beiden letzten Knoten der oberen und unteren Reihe. Sie bewirkt die sich ungünstig auswirkende Initialisierung der Entfernungswerte, so daß auch beim Einsatz von HLF immer wieder der Knoten links oben zum Zuge kommt. Dieser Effekt tritt, wenn auch vermindert, für die anderen Knoten der oberen Reihe auf (umso schwächer, je weiter rechts der Knoten steht). Im unteren Teilgraph bringt die HLF-Variante deutliche Vorteile. Mit ihr ist es möglich, zuerst die Knoten der linken Reihe (von oben nach unten) und erst dann die Knoten der rechten Reihe (von unten nach oben) abzuarbeiten. Bei der FIFO Methode hingegen werden in der Abarbeitungsreihenfolge immer wieder Knoten der rechten Reihe verwendet, bevor alle Flußeinheiten, die über sie fließen, angekommen sind. Es sind also unnötig viele Push-Operationen erforderlich.

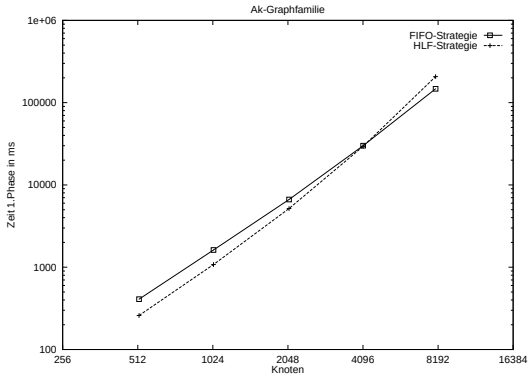
Versuchsergebnisse

Die Ergebnisse der Überlegungen lassen sich auch in den Testläufen bestätigen. Beim Einsatz der HLF-Variante werden nur ungefähr halb soviele nichtsättigende Push-Operationen durchgeführt (siehe Tabelle 4.2).

Methode / Parameter r	10	20	30	40
HLF	76	251	526	901
FIFO	131	461	991	1721

Abbildung 4.2 *Anzahl der nichtsätt. Push-Operationen für Ak-Graphen*

Dieses Ergebnis spiegelt sich ebenfalls in den Werten der Laufzeitmessungen wieder, die in Abbildung 4.3 dargestellt sind.



Knoten	Kanten	FIFO	HLF
518	775	0:409	0:259
1030	1543	1:619	1:078
2070	3103	6:649	5:166
4102	6151	29:962	29:420
8006	12007	147:306	207:416

Abbildung 4.3 *Laufzeitmessung für die Graphklasse Ak*

Der relative Unterschied der einzelnen Zeitmessungen ist jedoch deutlich geringer als der Faktor 2, der bei den nichtsättigenden Push-Operationen aufgetreten ist. Dies hat

unterschiedliche Gründe. Einen wesentlichen Einfluß auf die Laufzeit besitzt das Relabeling. Da bei beiden Varianten diese Operation gleich häufig auftritt, entsteht hierbei kein Unterschied. Ferner ist die Verwaltung der aktiven Knoten in einer Entfernungsliste aufwendiger als bei einem Einsatz einer einfachen Warteschlange. Gerade bei den größeren Graphen wirkt sich dies aus und führt zu einer Verschiebung der Meßergebnisse.
Die Ergebnisse der Testreihen lassen, außer bei sehr großen Graphen, einen deutlichen Vorteil für die HLF-Variante erkennen. Die beiden Strategien Gap- und Global-Relabeling bringen keine Verbesserungen. Durch den erhöhten Verwaltungsaufwand wird sogar eine geringfügige Verschlechterung der Laufzeiten erzielt (siehe Abschnitt 4.2).

4.1.2 Graphfamilie Ac

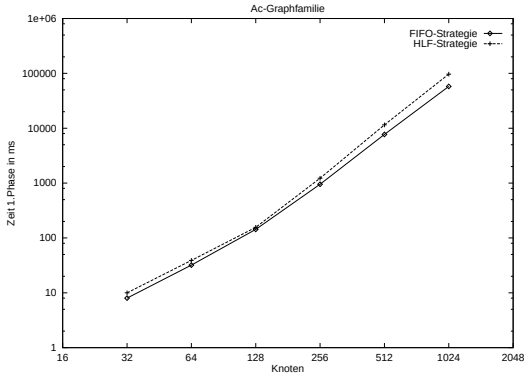
Dieser Teilabschnitt behandelt die von einem Generator von Setubal (eine C-Version eines Generators von Waissi) erzeugten Graphen. Die Struktur eines Graphen mit n Knoten ist vorgegeben, nur die Kantenkapazitäten werden zufällig gewählt.

Theoretische Überlegungen

Für diese Graphen ist es nicht so einfach eine Aussage zu treffen, um eine Präferenz für eine spezielle Auswahlmethode der aktiven Knoten zu bestimmen. Dies ist durch die zufällige Wahl der Kantenkapazitäten und mehr noch durch die wirkliche Arbeitsweise des Algorithmus gegeben. Da nach dem Initialisieren der Entfernungsfunktion alle Knoten, bis auf Quelle und Senke, den Wert 1 zugewiesen bekommen, kann anhand dieses Wertes keine Unterscheidung der Knoten vorgenommen werden. Die Auswahl des ersten Knotens hängt also vom Zufall, oder genauer gesagt von der exakten Vorgehensweise des Algorithmus, ab. Bei der FIFO-Methode trifft diese Aussage auch auf spätere Punkte des Algorithmus zu. Bei der HLF-Methode wird der Zufall etwas eingeschränkt, ist jedoch dennoch vorhanden. Häufig ist die Auswahl durch die Reihenfolge der Speicherung der Kanten und Knoten des Graphen gegeben.
Da es sich bei der Graphfamilie um azyklische Graphen handelt, könnte man jedem Knoten einen Wert zuordnen, so daß für jede Kante der Ausgangsknoten einen echt kleineren Wert als der Zielknoten besitzt (topoligische Sortierung). Eine gute Heuristik wäre es, wenn man zuerst die Knoten mit einer kleineren Beschriftung abarbeitet. Diese Knoten könnten dann jeweils Flußeinheiten direkt in die Senke verschieben oder an andere Knoten weiterleiten. Idealerweise würde in diesem Fall ein Knoten, so er denn nicht seinen gesamten Überfluß direkt in die Senke schieben kann, den verbleibenden Rest an einen Knoten weitergeben, der dies durchführen könnte. Hierdurch könnte mit großer Wahrscheinlichkeit ein unnötiges Verschieben der Flußeinheiten verhindert werden. Die ebenfalls im Theorieteil beschriebene, jedoch nicht implementierte Wave-Methode (siehe Teilabschnitt 2.3.1) würde diese Heuristik unterstützen.
Allgemein setzt die Heuristik ein gezieltes Nutzen der topologiscen Struktur des Eingabegraphen voraus. Bei den implementierten Varianten des Algorithmus (FIFO und HLF) ist dieses Wissen jedoch nicht verwendet. Somit ist leider nicht möglich die Heuristik mit dem realisierten System zu untersuchen. Es bleibt also zu testen, welche der beiden Techniken sich für den gegebenen Algorithmus, auf den bereitgestellten Graphen als die bessere erweist.

Versuchsergebnisse

Die Testläufe liefern die in Abbildung 4.4 dargestellten Ergebnisse.



Knoten	Kanten	FIFO	HLF
32	496	0:008	0:010
64	2016	0:032	0:039
128	8128	0:143	0:156
256	32640	0:951	1:227
512	130816	7:698	11:506
1024	523776	57:835	96:838

Abbildung 4.4 *Laufzeitmessung für die Graphklasse Ac*

Wie man aus der Graphik erkennen kann, erhält man für die FIFO-Strategie deutlich bessere Ergebnisse (speziell für große Graphen). Da keine theoretische Überlegung für dieses Ergebnis vorliegt, muß der Ablauf des Algorithmus genauer untersucht werden. Bei einem schrittweisen Vergleich der jeweiligen Abläufe kann folgende Tatsache feststellen:
Werden bei der HLF-Variante von einem Knoten mit größerem Entfernungswert Flußeinheiten an einen anderen Knoten gesendet, kommt der andere Knoten relativ bald zum Zug, da er ebenfalls einen großen Entfernungswert besitzen muß (Voraussetzung für den Push). Wenn dieser den neuen Überfluß jedoch nicht mehr abbauen kann, schiebt er ihn, nach einem Relabel, (in den meisten Fällen) wieder zum ursprünglichen Knoten zurück. Dieser kommt wegen der großen Entfernung ebenfalls bald wieder an die Reihe und verschiebt die Flußeinheiten erneut. Wurden beim anfänglichen Verschieben die Flußeinheiten nicht nur an einen, sondern an mehrere solcher Knoten gesendet, werden die Flußeinheiten häufig zurückgeschoben. Nach jedem Zurückschieben werden die entsprechenden Flußeinheiten (fast) sofort an einen anderen Knoten verteilt.
Beim Verwenden der FIFO-Methode könnten zuerst alle zurückgeschobenen Flußeinheiten gesammelt und dann gemeinsam weitergeleitet werden. In Abbildung 4.5 ist dieser Effekt dargestellt. Auf der linken Seite der Abbildung ist die HLF- und auf der rechten Seite die FIFO-Variante vorgestellt. Die dick umrandeten Knoten stellen die Knoten mit Überfluß dar. Die Entfernungswerte sind in den Knoten, die Größe des Überflusses ist in separaten Kästchen angetragen. Auf die Kantenkapazitäten und weiter nicht verwendete Kanten wurde aus Übersichtlichkeitsgründen verzichtet. Ebenso wurden die Relabel-Schritte nicht dargestellt.
Wie zu sehen ist, finden bei der HLF-Methode mehr Schritte zum Verteilen der zurückgeflossenen Einheiten statt. Bei der FIFO-Methode wird diese Verteilung erst ganz am Schluß und in einem Schritt durchgeführt. Dieser Effekt zeichnet sich auch deutlich in der Anzahl der Push-Operationen ab. In Abbildung 4.6 werden, exemplarisch für einen Graphen, die Anzahlen der Push- und der Relabel-Operationen tabellarisch gegenübergestellt. Für andere Graphen wurden analoge Ergebnisse erzielt.

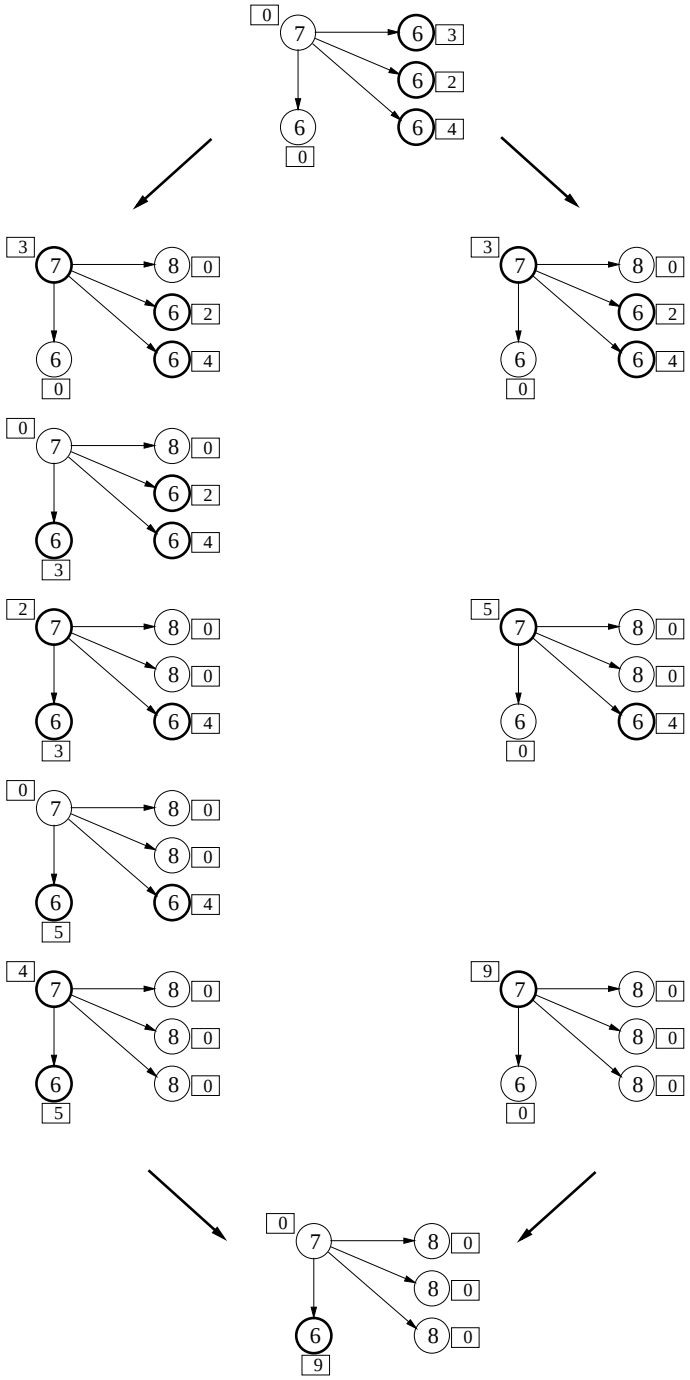


Abbildung 4.5 *Darstellung des Ablaufs für Ac-Graphen*

Methode	sätt. Push	nicht-sätt. Push	gesamt	Relabel
FIFO	320	314	634	87
HLF	432	414	846	91

Abbildung 4.6 *Anzahl der Push- und Relabel-Operationen für Ac-Graphen*

Diese Ergebnisse erklären die gemessenen Werte, die besagen, daß die FIFO-Methode schneller als die HLF Methode ist. Ebenfalls berücksichtigt werden muß wieder der größere Aufwand für die Verwaltung bei der HLF-Methode, dies wirkt sich zusätzlich auf die Laufzeitunterschiede aus.

4.1.3 Graphfamilie Genrmf

Bei dem Generator Genrmf ist es möglich, Einfluß auf die Struktur des Graphen zu nehmen. Durch die Parameter a und b kann das Aussehen des Graphen bestimmt werden. In den Testläufen wurden zwei Typen von Graphklassen verwendet: Sogenannte lange Graphen, d.h. die Pfade von der Quelle zur Senke sind relativ lang, und breite Graphen, d.h. die Pfade sind relativ kurz und ferner sind manchen Entfernungswerten viele Knoten zugeordnet. Bildlich kann man sich das wie folgt vorstellen. Man trägt alle Knoten eines Graphen so in ein Gitter ein, daß Knoten mit gleichem initialen Entfernungswert auf einer gemeinsamen vertikalen Linie liegen, und Linien, die kleinere Entfernungen repräsentieren, links von denen liegen, die für größere Werte stehen. In dem Bild befindt sich also die Quelle ganz links und die Senke ganz rechts. Erhält man einen eher langgezogenen Schlauch, so spricht man von einem langen, bei einem eher schmalen hohen Gebilde von einem breiten Graphen.
Beim Generieren von Graphen mit 2^x Knoten wurden für lange Graphen (*GenrmfLong*) bzw. breite Graphen (*GenrmfWide*) folgende Parameter verwendet:

GenrmfLong	$a = 2^{x/4}$	$b = 2^{x/2}$	$c_1 = 1$ $c_2 = 100$
GenrmfWide	$a = 2^{2x/5}$	$b = 2^{x/5}$	$c_1 = 1$ $c_2 = 100$

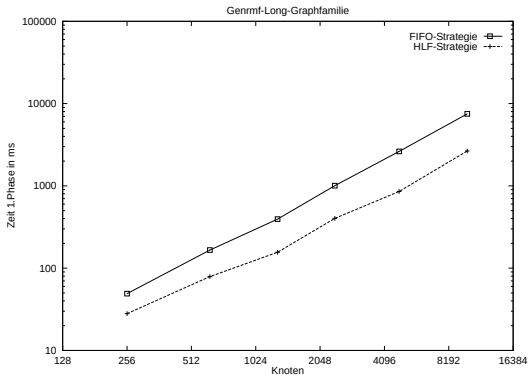
Theoretische Überlegungen

Während in den vorangegangenen Überlegungen immer sehr spezielle Fälle von Graphen betrachtet wurden, stellen die Graphen der Genrmf-Familie allgemeinere Graphen dar. Es gibt recht kompakte Gebiete (die einzelnen Gitter) und auch relativ dünne Teilgraphen (Übergänge zwischen den Gittern). Nun soll betrachtet werden, welche Vorteile die einzelnen Methoden zur Auswahl der aktiven Knoten mit sich bringen.
Der Einsatz von HLF bewirkt, daß vorzugsweise die Knoten bearbeitet werden, von denen man annimmt, daß sie die Senke nur über einen langen Pfad erreichen können. Es wird also versucht, die Flußeinheiten möglichst parallel von der Senke zur Quelle zu verschieben. D.h. man will erreichen, daß möglichst wenig Knoten, die nahe an der Senke liegen, frühzeitig bearbeitet werden. Hierdurch verhindert man, daß diese Knoten mehrmals hintereinander Flußeinheiten zum gleichen Knoten (in Richtung Senke) verschieben.
Wie schon in der Analyse des Ablaufs für Graphen der Ac-Familie gesehen, kann dies jedoch auch genau den entgegengesetzten Effekt erzielen. Dieser Fall tritt auf, wenn man in eine Sackgasse kommt. Durch das Abarbeiten von vielen Knoten mit höherem Entfernungswert bemerkt man diese Tatsache erst sehr spät, besonders dann, wenn es sich um sehr lange Sackgassen handelt. Hier wäre folglich der Einsatz der FIFO-Strategie vorteilhaft. Da bei der Genrmf-Familie die einzelnen Gitter jedoch so aufgebaut sind, daß Sackgassen nur zwischen den Gittern auftreten. Eine Sackgasse bedeuten also, daß zwei Gitter nicht mehr miteinander verbunden sind. Diese Tatsache wird sowohl durch die Verwendung von Global- als auch Gap-Relabeling schnell erkannt. Eine Verlangsamung durch Sackgassen tritt beim Einsatz dieser Strategien also kaum auf.

Bleibt noch der genauere Einfluß der verwendeten Auswahlmethode auf das Global- bzw. Gap-Relabeling zu betrachten. Der Einsatz von HLF und dem damit verbundenen Abarbeiten von Knoten, die weit von der Senke entfernt sind, bewirkt, daß diese Knoten schneller einen größeren Entfernungswert zugeordnet bekommen, als dies bei der FIFO-Strategie der Fall ist. Treten also bereits an dieser Stelle Lücken bei den Entfernungswerten auf, so können diese schneller entdeckt werden. Dies wird ebenfalls dadurch verstärkt, daß bei den Knoten, die näher an der Senke liegen, noch keine oder nur kaum Relabel-Operationen durchgeführt wurden. Hierdurch treten große Entfernungswerte fast nur bei von der Quelle weit entfernt liegenden Knoten auf. Somit fallen dort auftretende Lücken verstärkt ins Gewicht und können besser detektiert werden. Gerade bei den Graphen der Familie *GenrmfLong*, bei denen es viele Gitter gibt, wird sich dies bemerkbar machen. Sind die Übergänge zu dem jeweils nächsten Gitter erschöpft, wird dies bemerkt und die isolierten Knoten werden nicht mehr bearbeitet.
Dieser Effekt bringt jedoch auch Nachteile mit sich. Durch das verstärkte Arbeiten auf weit entfernten Knoten, was natürlich auch etliche Relabel-Operationen nach sich zieht, werden nahe an der Senke kaum Flußeinheiten verschoben. Tritt in diesem Stadium bereits ein Global-Relabeling ein, so werden sich die Entfernungswerte der Knoten nahe an der Senke kaum ändern. D.h. es kann für sie keine neue Information gewonnen werden. Beim Einsatz der FIFO-Methode hingegen würde auch nahe der Quelle eine Änderung der Entfernungswerte auftreten und somit mehr Information bei einem Global Relabeling gewonnen werden, was ein verbessertes Arbeiten des Algorithmus nach sich zieht. Gerade bei breiten Graphen, also für den Typ *GenrmfWide*, würden schnell Knoten nahe der Senke erreicht werden. Dies würde also einen Vorteil für diesen Graphtyp beim Einsatz der FIFO-Methode bedeuten. Eine genaue Auswirkung muß jedoch durch Testläufe bestimmt werden, da bei breiten Graphen die Knoten mit größerem Entfernungswert nicht so weit von der Senke entfernt sind wie bei langen Graphen, was den Effekt wieder verringert.
Faßt man die Überlegungen zusammen, kann man Argumente für den Einsatz von beiden Auswahlmethoden finden. Da zu erwarten ist, daß einige Lücken bei den Entfernungswerten auftreten, kann vermutet werden, daß der Einsatz der HLF-Methode allgemein bessere Ergebnisse liefert als die FIFO-Methode. Es bleibt jedoch abzuwarten, wie dieser Vorteil durch den vermehrten Verwaltungsaufwand abgeschwächt wird. Im Vergleich der beiden Graphtypen sollte für die Graphen das Typs *GenrmfLong*, relativ gesehen zu denen des Typs *GenrmfWide*, der Einsatz der Methode HLF gewinnbringender sein.

Versuchsergebnisse

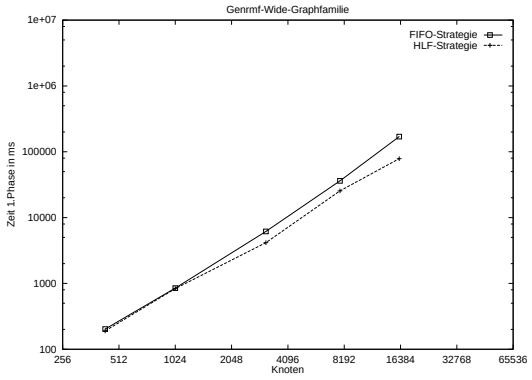
Die Resultate der Testläufe werden für die beiden Graphtypen getrennt präsentiert. Die Ergebnisse für den Typ *GenrmfLong* sind in Abbildung 4.7 und für den Typ *GenrmfWide* in Abbildung 4.8 dargestellt.
In beiden Graphiken kann man erkennen, daß sich die obigen Überlegungen bestätigt haben. Besonders deutlich ist zu erkennen, daß für die Graphen des Typs *GenrmfLong* der Einsatz der HLF-Methode eine wesentliche Verbesserung bewirkt. Bereits bei Graphen mit geringer Knotenzahl ist ein deutlich beschleunigter Ablauf zu erkennen.



Knoten	Kanten	FIFO	HLF
256	1008	0:049	0:028
625	2600	0:166	0:079
1296	5580	0:395	0:156
2401	10584	1:007	0:403
4800	21536	2:626	0:856
10000	45900	7:510	2:654

Abbildung 4.7 *Laufzeitmessung für die Graphklasse GenrmfLong*

Ferner ist zu bemerken, daß sich bei breiten Graphen (*GenrmfWide*) der Einsatz der HLF-Methode erst für größere Graphen bemerkbar macht. Bei dieser Methode treten, im Vergleich zur FIFO-Methdoe, weniger Push- jedoch mehr Relabel-Operationen auf. Erst bei den großen Graphen macht sich dies durch Laufzeitunterschiede bemerkbar.



Knoten	Kanten	FIFO	HLF
432	1872	0:203	0:190
1024	4608	0:850	0:835
3125	14500	6:157	4:156
7776	36720	36:148	25:503
16128	76992	169:924	78:640

Abbildung 4.8 *Laufzeitmessung für die Graphklasse GenrmfWide*

4.1.4 Graphfamilie Washington

Für diesen Generator gibt es zwei verschiedene Aufrufmöglichkeiten. Die erste (Typ *Random Level*) erzeugt bekanntlich Graphen, bei denen die Knoten in Gitterform angeordnet sind. Für diesen Typ werden wieder lange (*WashingtonRLGLong*) und breite Graphen (*WashingtonRLGWide*) erzeugt. Die zweite Art, Graphen zu generieren (Typ *Basic Line*), erzeugt ihrerseits Graphen, bei denen die Knoten in einer Linie angeordnet sind und zufällige Kanten (in Richtung Senke) besitzen (*WashingtonLine*). Es ist also per se nicht möglich, von langen oder breiten Graphen zu sprechen. Als Parameter zum Erzeugen von Graphen mit 2^x Knoten werden folgende Werte verwendet:

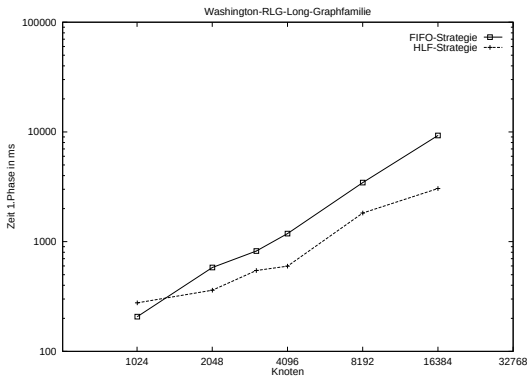
WashingtonRLGLong	$m = 64$	$n = 2^{x-6}$	$r = 10^4$
WashingtonRLGWide	$m = 2^{x-6}$	$n = 64$	$r = 10^4$
WashingtonLine	$m = 2^{x-2}$	$n = 4$	$r = 2^{(x/2)-2} = \sqrt{n \cdot m}/4$

Theoretische Überlegungen

Für diese Graphen gelten im wesentlichen die Überlegungen, die bereits für die Graphen des Generators Genrmf besprochen wurden. Es ist also zu vermuten, daß für alle drei Typen die HLF-Methode schneller als die FIFO-Methode ist. Dieser Effekt wird sich bei Graphen des Typs *WashingtonRLGLong* stärker bemerkbar machen als bei denen vom Typ *WashingtonRLGWide*.
Die Struktur der Graphen des Typs *WashingtonLine* ist wesentlich vom Zufall abhängig. Die zufällige Wahl der Kanten und ihrer Gewichte wird bei den Meßergebnissen zu Schwankungen führen. Es bleibt jedoch zu hoffen, daß sich eine eindeutige Tendenz zugunsten einer Strategie herausbildet.

Versuchsergebnisse

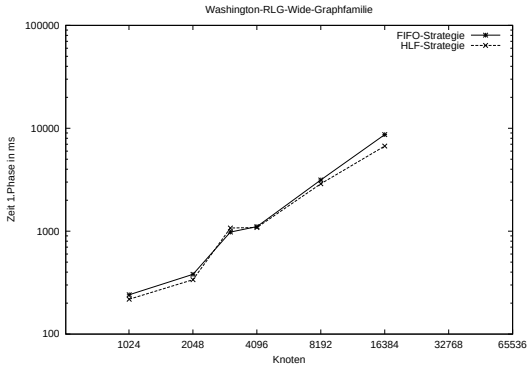
Die Ergebnisse der Messungen sind in den Abbildungen 4.9 (*WashingtonRLGLong*), 4.10 (*WashingtonRLGWide*) und 4.11 (*WashingtonLine*) wiedergegeben.



Knoten	Kanten	FIFO	HLF
1026	3040	0:207	0:277
2050	6112	0:581	0:361
3074	9184	0:821	0:546
4098	12256	1:183	0:598
8194	24512	3:451	1:822
16386	49088	9:280	3:049

Abbildung 4.9 *Laufzeitmessung für die Graphklasse WashingtonRLGLong*

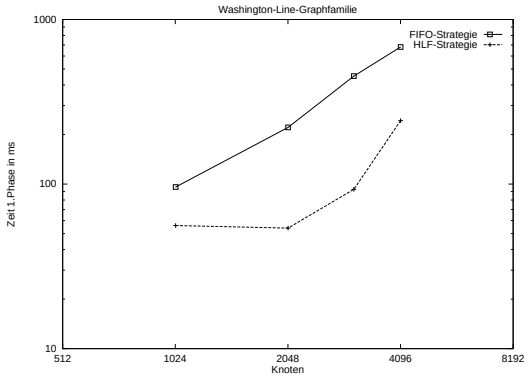
Für den Typ *Random Level* beweist sich die obige Annahme, daß die HLF-Methode besser ist als die FIFO-Methode. Auch das bessere Abschneiden dieser Methode auf langen Graphen ist deutlich zu erkennen. Aus gleichen Grund wie beim *Genrmf*-Generator ist die HLF-Methode für breite Graphen erst ab einer gewissen Größe schneller.



Knoten	Kanten	FIFO	HLF
1026	3040	0:241	0:218
2050	6080	0:381	0:338
3074	9120	0:980	1:173
4098	12160	1:107	1:085
8194	24448	3:147	2:888
16386	48896	8:868	6:720

Abbildung 4.10 *Laufzeitmessung für die Graphklasse WashingtonRLGWide*

Wie zu Beginn des Kapitels erwähnt, ist es nicht immer möglich, die Ergebnisse von verschiedenen Meßreihen qualitativ zu vergleichen. Es fällt jedoch auf, daß bei beiden Reihen für den Typ *Random Level* die Zeiten für die FIFO-Methode für Graphen mit gleicher Knotenzahl annähernd identisch sind. Bei den Ergebnissen der HLF-Methode können jedoch deutliche Unterschiede erkannt werden.



Knoten	Kanten	FIFO	HLF
1026	8068	0:096	0:056
2050	24300	0:221	0:054
3002	41615	0:453	0:093
4098	65025	0:681	0:243

Abbildung 4.11 *Laufzeitmessung für die Graphklasse WashingtonLine*

Die Ergebnisse für den Graphtyp *WashingtonLine* sind recht eindeutig, auch in diesen Fällen ist die HLF-Variante des Algorithmus deutlich zu bevorzugen. Man kann jedoch die Schwankungen der einzelnen Meßergebnisse erkennen.

4.2 Vergleich der Strategien

In diesem Kapitel soll nun untersucht werden, wie welche der beiden Strategien Gap- und Global-Relabeling am besten eingesetzt werden. Da ein Erfolg des Einsatzes dieser Strategien von den verwendeten Graphen abhängt, werden einzelnen Graphfamilien wieder getrennt untersucht.
Es werden mehrere Testreihen durchgeführt, wobei es genügt, sich auf einige kleine Graphen der jeweiligen Graphfamilie zu beschränken. Dies ist ausreichend, da andere hier nicht aufgeführte Tests gezeigt haben, daß die Werte für eine gute Parameterwahl bei großen und kleinen Graphen des gleichen Graphtyps nur geringen Schwankungen unterworfen sind.
Für jede der beiden Strategien zur Auswahl der aktiven Knoten (FIFO und HLF) wurden seperate Testläufe durchgeführt. Da es sich bei den zwei zu bestimmenden Parametern um einen Boole'schen Parameter (Gap-Relabeling) und einen ganzzahligen Parameter (Intervall für das Global-Relabeling) handelt, sind zwei Meßreihen denkbar. Erstens die Laufzeitmessung für mehrere Intervallgrößen beim Einsatz von Gap-Relabeling und zweitens die Messungen ohne Verwendung des Gap-Relabelings. In den folgenden Abschnitten wird nur die erste Variante anhand von Meßergebnissen präsentiert. Für die zweite Meßreihe hat sich bei stichpunktartigen Untersuchungen herausgestellt, daß die Intervallgrößen, die beim Einsatz von Gap-Relabeling am besten sind, auch dann die besten Ergebnisse liefern, wenn diese Strategie nicht verwendet wird.
Die getesteten Werte für die Intervallgröße werden jeweils so gewählt, daß sich der ideale Wert in dem untersuchten Bereich befindet. Wenn der Einsatz des Global Relabeling sinnvoll ist, dann bildet sich um einen idealen Wert ein Minimum. Es ist offensichtlich, daß für die Extremwerte, d.h. das Ausführen eines Global Relabelings nach jedem

Schritt bzw. nach keinem einzigen Schritt, wegen des hohen Aufwands bzw. der vielen unnötigen Schritte, längere Laufzeiten gemessen werden. Beim Test für Werte zwischen diesen beiden Grenzen werden geringere Laufzeiten gemessen, da man sich einem lokalen Optimum annähert. Im allgemeinen wird es keinen scharfen Wert geben, der eine ideale Laufzeit des Algorithmus liefert, da es im wesentlichen auf die Anzahl der durchgeführten Global-Relabel-Operationen ankommt. Die Anzahl der Global-Relabeling-Operationen ist durch den Wert $\lfloor \frac{Anzahl\ \ der\ \ Standard-Relabel-Operationen}{Intervallgr\ddot{o}\ss e} \rfloor$ bestimmt und somit gegenüber geringen Schwankungen der Intervallgröße unempfindlich. Durch Schwankungen in der Intervallgröße wird nur der genaue Zeitpunkt der Global-Relabel-Operationen festgelegt. Es genügt daher die getesteten Werte entsprechend zu streuen. In den Messungen werden Werte verwendet, die dem 2^i-fachen der Knotenzahl n ($i \in \mathbb{Z}$) entsprechen. Es hat sich gezeigt, daß sich günstige Werte zwischen den Grenzen $n/16$ und $4 \cdot n$ bewegen. Vereinzelt liegt das Minimum sehr nahe oder direkt bei Intervallgrößen, für die kein Global-Relabeling mehr durchgeführt wird (d.h. es gibt nicht genug Standard-Relabel-Operationen). In diesen Fällen wird der Testlauf auf dieser Seite der Testreihe abgebrochen. Um dennoch einen Vergleich zu besitzen, wird immer ein Testlauf durchgeführt, bei dem ganz auf die Heuristik des Global-Relabelings verzichtet wird.
In jeder Zeile der Tabellen, in denen im folgenden die Ergebnisse angegeben werden, ist die kürzeste Laufzeit fett gedruckt. Allgemein sei an dieser Stelle nochmals auf die Schwankungen hingewiesen, die bei der Laufzeitmessung möglicherweise auftreten. Es kann also sein, daß an einer Stelle die Laufzeit nicht ideal in das Gesamtumfeld paßt, sondern um einen kleinen Fehler abweicht.

4.2.1 Graphfamilie Ak

Die Graphen der Ak-Familie sind bewußt so aufgebaut, daß keine der beiden Strategien anwendbar ist. Die Entfernungsfunktion gibt stets die Länge des kürzesten Pfades zur Senke an, d.h. ein Global-Relabeling liefert keine neue Information. Weiter treten auch keine Löcher in der Menge der Entfernungswerte auf, was den Einsatz des Gap-Relabelings zunichten macht. Diese Aussage wird auch durch die folgenden Meßergebnisse bestätigt.

FIFO:

Knoten	no-global	$n/8$	$n/4$	$n/2$
518	**0:280**	0:311	0:283	0:282
1030	**1:165**	1:804	1:173	1:209
2070	**5:452**	6:618	6:240	5:716

HLF:

Knoten	no-global	$n/8$	$n/4$	$n/2$
518	**0:182**	0:210	0:201	0:194
1030	**0:825**	0:934	0:856	0:845
2070	**4:027**	4:256	4:237	4:155

Abbildung 4.12 *Messungen zur Parameterwahl für Ak*

Durch das Weglassen des Gap-Relabeling unter Beibehaltung des Global-Relabelings kann zwar die Laufzeit etwas gedrückt werden, man kommt jedoch nicht unter die Zeit, die erzielt wird, wenn man keine der beiden Heuristiken anwendet.

4.2.2 Graphfamilie Ac

Bei dieser Graphfamilie kann man nun einen optimalen Wert für den Abstand zwischen zwei Global-Relabel-Operationen feststellen. Nachfolgende Tabellen zeigen wieder die gemessenen Werte.

FIFO:

Knoten	no-global	$n/4$	$n/2$	n	$2 \cdot n$	$4 \cdot n$	$8 \cdot n$
128	0:143	0:148	0:140	0:141	**0:124**	0:139	0:148
256	1:021	0:925	0:851	**0:816**	0:860	0:883	0:912
512	5:765	5:995	5:615	**5:524**	5:563	5:852	6:021

HLF:

Knoten	no-global	$n/4$	$n/2$	n	$2 \cdot n$	$4 \cdot n$	$8 \cdot n$
128	0:153	0:140	0:126	0:115	**0:113**	0:138	0:165
256	1:208	1:152	0:996	0:980	0:995	**0:958**	1:052
512	6:237	6:734	5:715	**4:648**	5:091	5:003	5:327

Abbildung 4.13 *Messungen zur Parameterwahl für Ac*

Bei einer gute Parameterwahl verhalten sich die FIFO- und die HLF-Varianten sehr ähnlich. Das Global-Relabeling ist nach ca. n Standard-Relabel-Operationen durchzuführen. Weitere Tests ergaben, daß das Weglassen des Gap-Relabelings längere Laufzeiten bewirkt.

4.2.3 Graphfamilie Genrmf

Für die beiden Typen dieser Graphfamilie ergeben sich verschiedene Resultate.

GenrmfLong

FIFO:

Knoten	no-global	$n/8$	$n/4$	$n/2$	n	$2 \cdot n$
1296	0:855	0:458	**0:417**	0:452	0:516	0:567
2401	2:793	1:143	**1:008**	1:175	1:319	1:463
4800	10:313	3:421	**3:390**	3:437	4:042	4:805

HLF:

Knoten	no-global	$n/4$	$n/2$	n	$2 \cdot n$	$4 \cdot n$
1296	0:149	0:178	0:150	**0:131**	0:149	1:530
2401	0:546	0:498	**0:496**	0:498	0:583	0:575
4800	1:217	1:121	1:148	0:975	**0:949**	1:155

Abbildung 4.14 *Messungen zur Parameterwahl für GenrmfLong*

Es ist zu erkennen, daß beim Einsatz der FIFO-Methode ein häufigeres global-Relabeling sinnvoller ist als bei der HLF-Methode (FIFO: $n/4$, HLF: n). Eine genauere Untersuchung zum Einsatz des Gap-Relabelings ergibt, daß dessen Weglassen bei der FIFO-Methode bessere Ergebnisse liefert, bei der HLF-Methode der Einsatz aber gewinnbringender ist.

GenrmfWide

In diesem Fall ergaben die Testläufe die nachfolgend dargestellten Werte.

FIFO:

Knoten	no-global	$n/16$	$n/8$	$n/4$	$n/2$	n
432	0:371	0:234	**0:204**	0:208	0:216	0:233
1024	2:945	0:868	**0:763**	0:835	0:928	1:144
3125	33:873	8:258	**7:215**	7:556	7:519	13:654

HLF:

Knoten	no-global	$n/16$	$n/8$	$n/4$	$n/2$	n
432	0:306	0:281	0:213	0:199	**0:192**	0:207
1024	1:834	1:014	0:980	0:859	**0:840**	0:897
3125	19:279	7:837	7:377	5:702	**4:621**	5:954

Abbildung 4.15 *Messungen zur Parameterwahl für GenrmfWide*

Auch hier liegen die idealen Werte für die beiden Methoden bei unterschiedlichen Werten (FIFO: $n/8$, HLF: $n/2$). In beiden Fällen erweist sich das Nicht-Ausführen des Gap-Relabelings als sinnvoll. Die Beschleunigung ist vor allem bei der FIFO-Methode sehr groß und bewirkt, daß diese dann schneller wird als die HLF-Methode.

4.2.4 Graphfamilie Washington

Auch bei dieser Graphfamilie werden die einzelnen Typen wieder getrennt dargestellt.

WashingtonRLGLong

FIFO:

Knoten	no-global	$n/16$	$n/8$	$n/4$	$n/2$	n	$2 \cdot n$
2050	0:602	0:604	0:525	**0:520**	0:530	0:548	0:568
4098	1:197	1:166	1:112	**1:021**	1:033	1:042	1:135
8194	7:335	4:648	3:885	**3:636**	4:190	4:533	5:501

HLF:

Knoten	no-global	$n/16$	$n/8$	$n/4$	$n/2$	n	$2 \cdot n$
2050	0:280	0:483	0:386	0:334	0:295	0:294	**0:280**
4098	1:032	1:182	1:054	**1:000**	1:010	1:075	1:053
8194	2:096	3:338	2:505	2:214	2:439	**1:678**	1:961

Abbildung 4.16 *Messungen zur Parameterwahl für die WashingtonRLGLong*

Wieder ergeben sich unterschiedliche Werte für die Abstände zwischen zwei Global-Relabel-Operationen (FIFO: $n/4$, HLF: n). Bei der HLF-Methode ist es wichtig, das Gap-Relabeling einzusetzen, um kurze Laufzeiten zu erhalten, bei der FIFO-Methode hingegen ist das Weglassen sinnvoller. Die Verbesserungen führen jedoch nicht dazu, daß die FIFO-Methode besser als die HLF-Methode wird.

WashingtonRLGWide

FIFO:

Knoten	no-global	$n/16$	$n/8$	$n/4$	$n/2$	n	$2 \cdot n$
2050	0:456	0:438	0:376	0:378	**0:351**	0:353	0:454
4098	1:734	1:280	1:070	**1:013**	1:045	1:264	1:275
8194	7:467	3:633	3:200	**3:179**	3:246	3:792	4:564

HLF:

Knoten	no-global	$n/16$	$n/8$	$n/4$	$n/2$	n	$2 \cdot n$
2050	0:419	0:494	0:362	**0:304**	0:368	0:314	0:415
4098	2:611	1:652	1:386	**1:296**	1:455	1:376	2:085
8194	9:464	3:599	2:870	2:925	**2:633**	3:306	4:143

Abbildung 4.17 *Messungen zur Parameterwahl für die WashingtonRLGWide*

Aus den Testläufen für die HLF-Methode ist nicht ganz klar zu erkennen, wie der optimale Wert zu bestimmen ist. Es liegt jedoch nahe, daß er sich um den Wer $n/4$ oder etwas darüber befindet. Für die FIFO Variante ergibt sich ein ähnlicher, aber etwas kleinerer Wert. Wie bereits bei den Graphtypen *WashingtonRLGWide* bringt die Verwendung des Gap-Relabelings nur für die HLF-Variante eine bessere Laufzeit.

WashingtonLine

FIFO:

Knoten	no-global	$n/16$	$n/8$	$n/4$
518	**0:084**	0:146	0:118	0:092
1030	**0:180**	0:370	0:297	0:221
2070	**0:501**	1:298	0:918	0:616

HLF:

Knoten	no-global	$n/16$	$n/8$	$n/4$
518	**0:047**	0:096	0:071	0:058
1030	**0:101**	0:179	0:139	—
2070	**0:290**	0:597	0:396	—

Abbildung 4.18 *Messungen zur Parameterwahl für WashingtonLine*

Die qualitativen Ergebnisse sind denen der Ak-Graphen recht ähnlich. D.h. die schnellsten Laufzeiten werden erzielt, wenn kein Global-Relabeling verwendet wird. Läßt man ferner noch das Gap-Relabeling beiseite, so kann man noch bessere Ergebnisse erzielen.

4.3 Zusammenfassung

In diesem Abschnitt sollen nun die gewonnenen Ergebnisse zusammengefaßt werden. Im ersten Teil der Analyse wurden die beiden Methoden FIFO und HLF zur Auswahl der aktiven Knoten während des Ablaufs des Algorithmus untersucht. Mit den verwendeten Parametern stellte sich heraus, daß die HLF-Methode im allgemeinen bessere Ergebnisse liefert als die FIFO-Methode.
Nach einer genaueren Untersuchung der beiden verwendeten Heuristiken im zweiten Teil muß diese Aussage jedoch differenzierter betrachtet werden. Dafür gibt es zwei wesentliche Gründe:

1. Es wurde festgestellt, daß eine durchschnittliche Intervallgröße von $n/4$ für die FIFO-Methode optimal ist. Für die HLF-Methode schwankt dieser Wert stärker, ist jedoch durchwegs größer und liegt ungefähr zwischen $n/2$ und n. Da im ersten Teil ein Wert der Größe n verwendet wurde, ist die HLF-Methode bevorzugt worden.

2. In den meisten Fällen konnte durch das Weglassen des Gap-Relabelings bei der FIFO-Methode ein besseres Ergebnis erzielt werden, als bei dessen Verwendung. Bei der HLF-Methode war dies seltener der Fall, und wenn, dann meist in den Sonderfällen in denen auch ein Einsatz des Global-Relabeling nicht sinnvoll war. Es wurde also auch in diesem Punkt die FIFO-Methode etwas benachteiligt.

Diese beiden Punkte müßen jedoch geeignet interpretiert werden. Die meisten der verwendeten Graphen hängen in ihrer Form bzw. in den Kantenkapazitäten von Zufallsgrößen ab. Dies trifft also auch indirekt für die Intervallgröße des Global-Relabelings zu. Die ermittelten Werte können sich folglich von Fall zu Fall etwas verschieben. Ferner wird durch den Einsatz von Java im Vergleich zu anderen Programmiersprachen, in denen Zeigerarithmetik für eine effiziente Programmierung verwendet werden können, die Verwaltung der Entfernungslisten aufwendiger. Das Gap-Realbeling wurde deswegen häufig eher als "Bremse" denn als gewinnbringende Kraft empfunden.
Zieht man die Ergebnisse aus der Arbeit von Cherkassky und Goldberg [CG94] ebenfalls mit in den Vergleich ein, so ist dort für alle Graphtypen ein deutlicher Vorteil der HLF-Methode und des Gap-Relabelings zu erkennen. Wie bereits erwähnt, wurden die gleichen Parametereinstellungen verwendet wie im ersten Teil der Ausarbeitung. Auch diese Ergebnisse bestätigen die hier in der Arbeit gewonnenen Resultate. Die quantitatifen Unterschiede, d.h. die etwas geringeren Laufzeitunterschiede der aktuellen Arbeit, sind durch die unterschiedlichen Programmiersprachen bedingt. Aufgrund der, durch den Einsatz im Internet, notwendigen Festlegung auf die Sprache Java können einzelne Programmteile nicht so effizient gelöst werden, wie dies z.B. in C, das in der anderen Arbeit eingesetzt wurde, möglich ist. Trotz dieser Einschränkung konnten die Parameterfestlegungen sehr gut bestimmt werden. Ferner ist nach den Ergebnissen in dieser Diplomarbeit zu vermuten, daß auch in der Arbeit von Cherkassky und Goldberg die Laufzeitunterschiede bei einer ausgewogeneren Parameterwahl (also Verwendung des Gap-Relabelings bevorzugt bei der HLF Variante und geringeren Intervallgrößen bei der FIFO-Methode) nicht so deutlich ausfallen würden.

Kapitel 5

Zusammenfassung und Ausblicke

In diesem Kapitel soll nochmals die gesamte Arbeit betrachtet werden. Eine besondere Bedeutung kommt hierbei der Bewertung des implementierten Systems sowie der gewonnenen Ergebnisse für die Parametereinstellung zu. Ebenfalls wird ein Ausblick auf ergänzende Fragestellungen gegeben.

5.1 Zusammenfassung

In Kapitel 3 wurde ein System vorgestellt, das in seiner jetzigen Form eine gute Möglichkeit darstellt, um Algorithmen im Internet zu präsentieren. Gerade durch die Kombination von Tutorial und Grapheditor ist es dem Benutzer gestattet, individuell die Theorie und Erläuterungen zum Algorithmus zu erarbeiten und dann ohne viel Aufwand die gewonnenen Erkenntnisse anhand von einigen Testbeispielen nachzuvollziehen. Auch die Möglichkeit zum Verwenden von eigenen Beispielen, wie dies bei den Graphalgorithmen ja bereits umgesetzt ist, stellt eine wichtige Voraussetzung für die Akzeptanz eines Systems dar.
Die klare Strukturierung des Systems ermöglicht es dem Verwalter, schnell und einfach Änderungen vorzunehmen. Durch den Einsatz von HTML-Bäumen im Tutorial ist es nicht notwendig, daß der Verwalter eigens eine neue Beschreibungsmethode lernen muß, um neue Tutorials zu erzeugen. Ferner ist hierdurch völlige Freiheit zur Gruppierung und Organisation der Tutorials gegeben. Diese können also optimal auf das jeweilige Umfeld angepaßt werden.
Die gegebene Struktur der Visualisierungskomponenten wurde durch die Hierarchie der Editorfenster aufgegriffen. Somit ist es möglich, für einzelne Datenstrukturen eigene auf die Bedürfnisse abgestimmte Fenster zu erzeugen. Werden die vorhandenen Konventionen zur Ableitung der einzelnen Klassen eingehalten, können diese neuen Fenster und ggf. auch Datenstrukturen nahtlos in das System integriert werden. Dies gilt auch für neue Algorithmen. Es sind nur sehr wenige Einschränkungen vorhanden, die ein Entwickler beachten muß, wenn er einen Algorithmus umsetzen möchte. Gerade im Bereich der Graphalgorithmen stehen etliche Visualisierungsmöglichkeiten zur Verfügung, die die Umsetzung erheblich vereinfachen.

Bei der Analyse des Algorithmus (vergleiche Kapitel 4) wurden die Verbesserungen mittels FIFO- und HLF-Methode sowie einige Strategien miteinander verglichen. Die Ergebnisse bestätigten im wesentlichen die Resultate von Untersuchungen aus früheren Arbeiten über den Algorithmus.

Bei einer genauen Betrachtung des idealen Abstands zwischen zwei Global-Relabeling-Operationen konnte festgestellt werden, daß dieser bei der Verwendung der HLF-Methode (ungefähr Anzahl der Knoten n) größer ist als bei der FIFO-Methode (ca. $n/4$). Die Bewertung des Einsatzes der Strategie Gap-Relabeling stellte sich als schwieriger heraus. In der in Netzumgebungen eingesetzten Programmiersprache Java konnte die Verwaltung der Distanzwerte nicht so effizient gestaltet werden, daß eine Verbesserung für alle Abläufe nachgewiesen werden konnte. Es stellte sich jedoch heraus, daß die Verwendung dieser Strategie bei der HLF-Variante in einigen Fällen eine wichtige Voraussetzung für die schnellen Laufzeiten ist. Abschließend kann die Aussage getroffen werden, daß der Einsatz der HLF-Methode bei Verwendung der Strategien Gap-Relabeling und Global-Relabeling (Intervallgröße ungefähr Anzahl der Knoten) eine sehr gute Wahl für den Algorithmus darstellt.

5.2 Future Work

In der Diplomarbeit ist ein stabiles und umfassendes System geplant und entwickelt worden. Natürlich können im Rahmen einer Diplomarbeit nicht alle Fragen gelöst und alle Wünsche an eine Implementierung erfüllt werden. Deshalb sollen zum Abschluß noch kurz einige fortführende Gedanken und Möglichkeiten zur Verbesserung des Benutzungskomforts aufgelistet werden.

- Eine interessante Untersuchung besteht im Einfluß des Gap-Relabelings im Zusammenspiel mit Global-Relabeling auf die Laufzeit des Algorithmus. Um die hier nicht zu umgehenden Einschränkungen des Netzbetriebs auszugrenzen, wäre es denkbar, die gegebenen Algorithmen in anderen Sprachen, wie z.B. C, zu implementieren, um dann dort umfangreiche Test durchzuführen. Es ist denkbar, sich hierbei nicht nur die DIMACS-Graphen zu verwenden, sondern noch weitere Graphfamilien zu entwickeln.

- Im derzeitigen Stand des Client-Server-Systems ist nur implementiert, daß Algorithmen auf dem Client laufen. Das System könnte daraufhin erweitert werden, daß Algorithmen ohne Visualisierung auch auf einem leistungsfähigen Server durchgeführt werden. Bei der Planung der Algorithmen-Verwaltung im Controller auf der Clientseite wurde dies bereits berücksichtigt. In einem solchen Umfeld ist es dann jedoch sinnvoll, den Zugang auf den Server insoweit zu begrenzen, daß die Rechenleistung nicht beliebigen Netzbenutzern zur Verfügung steht, sondern diese an ausgewählte Personen vergeben werden kann.

- Um eine noch bessere Verwendbarkeit des Systems zu ermöglichen, ist es nützlich, die Menge der vorhandenen Datenstrukturen zu ergänzen. Zu denken ist hierbei an Fibonacci-Heaps, Splay-Trees oder ähnliche Datenstrukturen. Ferner könnten die vorhandenen Datenstrukturen noch um einige individuelle Methoden erweitert werden. So ist es denkbar, für Graphen ein automatisches Layout und Planaritätstests anzubieten, oder für Arrays automatisches Sortieren zur Verfügung zu stellen.

Diese Punkte sollen nur eine Anregung für Ergänzungen des Systems geben. Das Ergebnis wäre ein dann noch komfortableres System, das so auch ideal zur Vertiefung von Vorlesungen oder Praktika eingesetzt werden könnte.

Anhang

Anhang A

Beispiel für einen Graph-Algorithmus

In diesem Anhang findet sich der Code für eine mit den Basisklassen implementierte Breitensuche. Die Klasse hat eine Warteschlange zur Verwaltung der noch abzuarbeitenden Knoten (sihe unten, Programmzeile 6). Diese Schlange wird ebenfalls im Editorfenster angezeigt(Z. 38). Nach dem Start des Algorithmus wird der Benutzer nach einem Knoten gefragt, von dem die Breitensuche starten soll (Z. 53-55). Im Laufe des Algorithmus (Z. 56 -97) wird den Knoten/Kanten eine entsprechende Farbe/Label zugewiesen, die den aktuellen Status repräsentiert. Die Bedeutung der Farben wird in einer Legende des Graphen angezeigt (Z.39-45).
In Abbildung A.1 ist ein Screenshot dargestellt der während des Laufs des Algorithmus aufgenommen wurde. In ihm ist die oben beschriebene Darstellung des Graphen während des Algorithmus exemplarisch abgebildet.

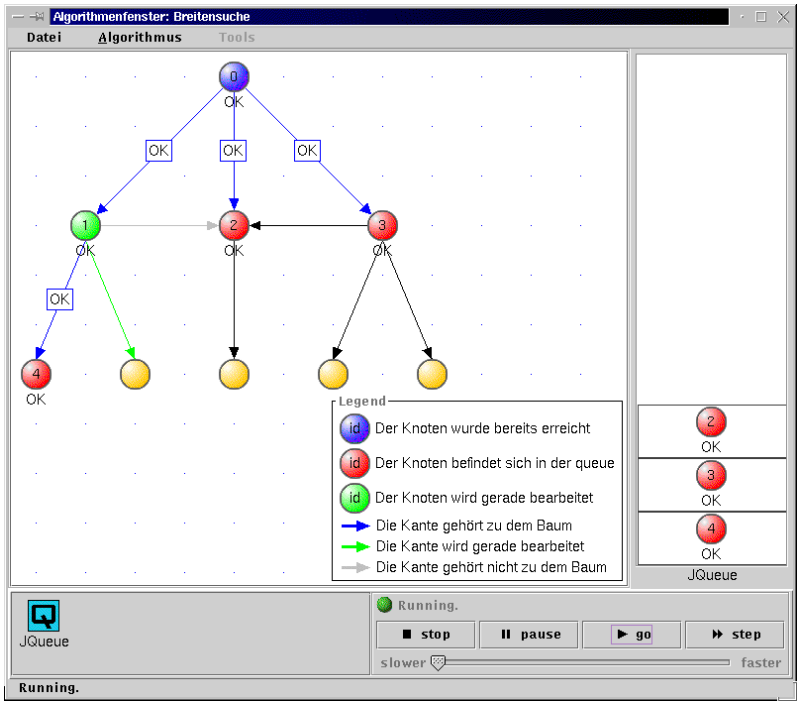


Abbildung A.1 *Screenshot des Algorithmus BFS*

Im weiter unten dargestellten Code wird der Aufruf des Algorithmus in der `main`-Methode wie folgt umgesetzt (Codezeilen 102 - 108). Zuerst wird eine Instanz des Algorithmus und ein Editorfenster erzeugt. Dann wird dem Editorfenster der Algorithmus zugeordnet, und das Fenster angezeigt. Mit diesen Aufrufen erscheint das Editorfenster am Bildschirm und der Benutzer kann einen Graphen laden/zeichnen und den Algorithmus starten.
Der nun folgende Code stellt eine lauffähige Implementierung dar. Neben den in dieser Ausarbeitung beschriebenen Methoden der Basisklassen werden noch weiter zusätzliche Methoden verwendet, für deren Beschreibung auf die Diplomarbeit von Ilia Dub [Dub98] und die Dokumentation mittels JavaDoc verwiesen wird.

```
(1)  public class BFS extends JGraphAlgorithm
(2)  {
(3)    ///////////////
(4)    // Variablen //
(5)    ///////////////
(6)    private JQueue queue   = null;
(7)    private int    counter = 0;
(8)
(9)    ////////////////////
(10)   // Konstruktoren //
(11)   ////////////////////
(12)   public BFS()
(13)   {
(14)     super();
(15)     queue = new JQueue();
(16)   }
(17)
(18)   public BFS(EditorWin editor)
(19)   {
(20)     super(editor);
(21)     queue = new JQueue();
(22)   }
(23)
(24)   /////////////////
(25)   // Algorithmus //
(26)   /////////////////
(27)   public void run() throws Throwable
(28)   {
(29)     // Klassen-Variablen initialisieren
(30)     counter         = 0;
(31)     queue.RemoveAllElements();
(32)     // Methoden-Variablen anlegen und initialisieren
(33)     Vector outEdges = null;
(34)     JEdge  e        = null;
(35)     JNode  n        = null;
(36)     JNode  n2       = null;
```

```
(37)      // Editorfenster initialisieren
(38)      addDataStruct(queue);
(39)      graph.addNodeLegend(Color.blue, "Der Knoten wurde bereits erreicht");
(40)      graph.addNodeLegend(Color.red, "Der Knoten befindet sich in der queue");
(41)      graph.addNodeLegend(Color.green, "Der Knoten wird gerade bearbeitet");
(42)      graph.addEdgeLegend(Color.blue, "Die Kante gehoert zum Baum");
(43)      graph.addEdgeLegend(Color.green, "Die Kante wird gerade bearbeitet");
(44)      graph.addEdgeLegend(Color.lightGray, "Die Kante gehoert nicht zum Baum");
(45)      graph.setLegendVisible(true);
(46)      queue.removeAllElements();
(47)      graph.resetDefaults();
(48)      graph.resetUserProperties();
(49)      graph.setNodePropertiesVisible(true);
(50)      graph.setEdgePropertiesVisible(true);
(51)
(52)      // Startknoten anfordern und in die Queue aufnehmen
(53)      this.setStatusBarText("Please choose a start node.");
(54)      this.setStatusBarTextColor(Color.red);
(55)      n = graph.askUserForNode();
(56)      this.setStatusBarText("Running.");
(57)      this.setStatusBarTextColor(JGraphEditorWin.DEFAULT_STATUSBAR_TEXTCOLOR);
(58)      n.setID(Integer.toString(counter++));
(59)      n.setBackground(Color.red);
(60)      n.putUserProperty("state", "OK");
(61)      queue.append(n);
(62)
(63)      // Abarbeiten der Queue
(64)      while (!queue.isEmpty())
(65)      {
(66)        step();
(67)        n = (JNode)queue.pop();
(68)        if (graph.isDirected())
(69)            outEdges = n.getOutEdges();
(70)        else
(71)            outEdges = n.getAdjEdges();
(72)        n.setBackground(Color.green);
(73)        for (int i = 0; i < outEdges.size(); i++)
(74)        {
(75)          step();
(76)          e = (JEdge)outEdges.elementAt(i);
(77)          if (e.getUserProperty("state") != null)
(78)              continue;
(79)          e.setBorderColor(Color.green);
(80)          n2 = e.getOppositeNode(n);
(81)          step();
(82)
(83)          if (n2.getUserProperty("state") != null)
(84)          {
(85)            e.setBorderColor(Color.lightGray);
(86)            continue;
(87)          }
```

```
(88)            n2.putUserProperty("state", "OK");
(89)            e.setBorderColor(Color.blue);
(90)            e.putUserProperty("state", "OK");
(91)            queue.append(n2);
(92)            n2.setID(Integer.toString(counter++));
(93)            n2.setBackground(Color.red);
(94)          }
(95)          step();
(96)          n.setBackground(Color.blue);
(97)        }
(98)      }
(99)
(100)     ////////////////////
(101)     // Main-Methode //
(102)     ////////////////////
(103)     public static void main(String[] args)
(104)     {
(105)       BFS algo = new BFS();
(106)       JGraphEditorWin ew = new JGraphEditorWin();
(107)       ew.setNewAlgorithm(0,"JGraph","Breitensuche",algo);
(108)       ew.setVisible(true);
(109)     }
(110)   }
```

Anhang B

Algorithmen für das verteilte System

In diesem Anhang sollen vertiefende Aspekte zur Implementierung von Algorithmen für das verteilte System und den Gebrauch des Repository gegeben werden.

B.1 Parameterfenster

Wie bereits in Teilabschnitt 3.2.4 beschrieben, besteht die Möglichkeit zur Implementierung eines speziellen Parameterfensters für jeden Algorithmus. Vor dem Instantiieren eines Algorithmus im Controller des Client wird versucht, dieses Fenster zu erzeugen.

B.1.1 Allgemeines

Das Fenster muß von der Basisklasse `edu.tum.dal.ParameterWindow` abgeleitet sein. Dieses Fenster stellt eine Grundlage für jedes Parameterfenster dar. In ihm ist bereits die Möglichkeit zur Auswahl einer Datenstruktur als Parameter gegeben. Für die Auswahl stehen drei Optionen zur Verfügung.

- Es kann eine Datenstruktur vom Server angefordert werden. Hierfür werden die auf dem Server als Parameter gespeicherten Datenstrukturen verwendet. Zur Identifikation des Datentyps besitzt das Parameterfenster einen String, in dem dieser gespeichert wird.
- Die Datenstruktur kann von der lokalen Platte geladen, oder
- aus einem bereits geöffneten Editorfenster bezogen werden. In diesem Fall wird aus dem Editorfenster, das markiert ist, die zur Zeit angezeigte Datenstruktur angefordert. Falls die Datentypen nicht übereinstimmen, wird der Ladevorgang abgebrochen.

Alle weiteren Auswahlmöglichkeiten eines Parameterfensters sind durch den Programmierer selbst zu implementieren. Elemente, die im Parameterfenster anzuzeigen sind, können mittels eines `Panel` in dem Parameterfenster, das durch die Basisklasse erzeugt wird, eingefügt werden (Feld `Center` des `BorderLayout`). Die Felder `North` und `South` sind bereits durch die Elemente zur Auswahl der Datenstruktur und der Knöpfe des Fensters belegt. Das Fenster der Basisklasse ist in Abbildung B.1 dargestellt.

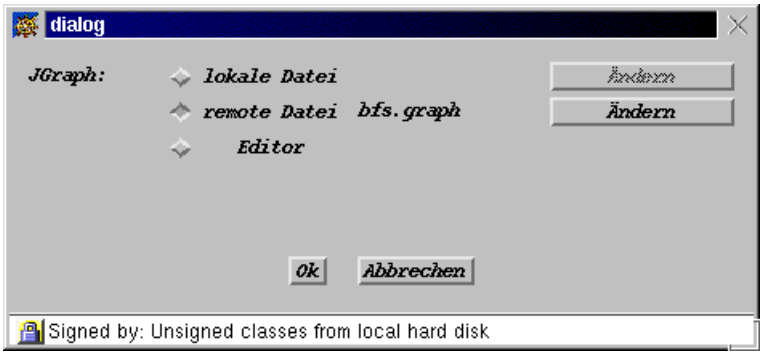


Abbildung B.1 *Parameterfenster der Basisklasse* `ParameterWindow`

B.1.2 Implementierungsaspekte

In diesem Teilabschnitt werden nun spezielle Informationen zum Konstruktor und zu den einzelnen Methoden, die teilweise überschrieben werden müssen, gegeben.

Konstruktor

Damit das Parameterfenster in den Feldern `North` und `South` die richtigen Einträge bekommt, muß, wie bereits bei den Algorithmen, im Konstruktor eines abgeleiteten Parameterfensters der Konstruktor der Basisklasse aufgerufen werden. Dies ist in Abbildung B.2 dargestellt.

```
public BFSParameterWindow(Frame                    parent,
                          InterfaceControllerClient controller,
                          String                   algoName,
                          String                   dataType,
                          Integer                  id,
                          Integer                  exampleNr,
                          String                   examplePath)
{
 super (parent,controller,algoName,dataType,id,expampleNr,examplePath);
 ...
}
```

Abbildung B.2 *Rahmen für einen Konstruktor eines abgeleiteten Parameterfensters*

Die Parameter, die der Konstruktor übergeben bekommt, haben folgende Bedeutung:

- *parent* — Dieser `Frame` wird dem Parameterfenster, das als `Dialog` implementiert ist, als übergeordnetes Fenster zugewiesen.
- *controller* — Es handelt sich um eine Referenz auf den Controller des Client, damit das Parameterfenster Methoden in diesem aufrufen kann.
- *algoName* — Bezeichner des Algorithmus, zu dem das Parameterfenster, aus Sicht des Controllers, gehört. Er kann dazu verwendet werden, um eventuelle Fehler frühzeitig zu erkennen.

- *dataType* — Analog zu `algoName` wird der Datentyp, dem der Algorithmus zugeordnet ist, übergeben.
- *id* — Dieser Identifikator bezeichnet die Nummer des Parameterfensters, unter dem dieses beim Controller gespeichert ist. Dieser Identifikator ist bei Methodenaufrufen im Controller immer anzugeben, damit der Controller weiß, an wen er eine Antwort senden muß.
- *exapmleNr* — Soll das Parameterfenster mit einer vordefinierten Beispielbelegung, die auf dem Server abgelegt ist, initialisiert werden, muß die Nummer dieses Beispiels dem Parameterfester bekanntgegeben werden.
- *examplePath* — Analog zu `exampleNr` handelt es sich hier um einen absoluten Dateinamen einer Informationsdatei, die auf der lokalen Platte gespeichert ist und eine Beispielbelegung enthält.

An der Stelle des in der Abbildung B.2 dargestellten Rahmens für den Konstruktor kann der Implementierer wieder eigenen Code, z.B. zum Initialisieren von weiteren Eingabefeldern, einfügen.
Die durch das Parameterfenster eingestellten Parameter werden in einer eigens hierfür vorhandenen Parameterstruktur (Klasse `edu.tum.dal.Parameter`) gespeichert. Diese Parameterstruktur wird dann auch dem Algorithmus übergeben, der diese dann auswertet. Diese Struktur besteht im wesentlichen aus einem Feld von Referenzen auf Objekte. Diese Objekte enthalten dann die einzelnen Parameter. Da es sich um Referenztypen handelt, müssen eventuelle Basistypen in Java (z.B. `int`, `float`) durch ihre Referenztypen (z.B. `Integer`, `Float`) dargestellt werden.
Die Hauptdatenstruktur wird immer als erste Referenz, also an dem Index 0 des Feldes, gespeichert. Die Anordnung der anderen Parameter ist dem Entwickler freigestellt. Die exakte Handhabung der Parameterstruktur ist der zugehörigen JavaDoc-Dokumentation zu entnehmen.

Methoden

Die Basisklasse stellt einige Methoden bereit, die die Minimalauswahl, also die Bestimmung einer Datenstruktur als Parameter, gewährleisten. Diese werden nun kurz beschrieben. Wenn ein Entwickler das Parameterfenster erweitert, sollte er diese Methoden überschreiben und jeweils zu Beginn der neuen Methoden die jeweilige Methode der Basisklasse aufrufen bzw. die nötigen Ausführungen in die neue Methode übernehmen. Die folgenden Beschreibungen stellen nur das wesentliche Konzept dar. Für Details sei auf den Sourcecode verwiesen.

- `receiveDataStructure(JAVLComponent dataStructure, boolean success)`
  Diese Methode wird vom Controller aufgerufen, wenn eine Datenstruktur, die zu einem früheren Zeitpunkt vom Parameterfenster angefordert wurde, gesendet wird. Normalerweise wird es sich bei dieser Datenstruktur um die Hauptdatenstruktur des Parameterfensters handeln. Die Methode ist in der Basisklasse entsprechend implementiert, d.h. die Datenstruktur wird als erster Parameter in der Parameterstruktur abgelegt. Ein Entwickler muß diese Methode also überschreiben, wenn er zusätzlich andere Datenstrukturen vom Controller empfangen möchte.

- `receiveExample(Object[] array, boolean success)`
  Wenn das Parameterfenster mit einer vordefinierten Parameterstruktur initialisiert wird, wird diese Methode aufgerufen. In dem Methodenparameter `array` befindet sich eine Reihung von Referenzen auf Objekttyen, die vom zugehörigen ExampleLoader (siehe Teilabschnitt B.2) eingelesen wurden. Eine Auswertung der übergebenen Referenzen ist entsprechend zu implementieren.
  Die Basisklasse erwartet als ersten Wert einen `Integer`-Referenztyp, der angibt, von wo die Haupt-Datenstruktur geladen werden soll (1: lokale Datei, 2: remote Datenstruktur, 3: Editorfenster). In dem Fall, daß sich diese Datenstruktur in einer lokalen Datei befindet, enthält die Reihung an den Indizes 1 und 2 den Pfad- und den Dateinamen. Wenn eine remote Datenstruktur vorgesehen ist, findet sich an dem Index 1 der Bezeichner der entsprechenden Datenstruktur auf dem Server.

- `receiveParameterNames(String[] names)`
  Wenn Datenstrukturen vom Server geladen werden sollen, muß der Benutzer aus einer Liste der vorhandenen Datenstrukturen auswählen können. Diese Liste kann vom Controller mit der Methode `loadRemoteDataStructure(...)` (siehe JavaDoc) angefordert werden. Bei der Anforderung können auch gewisse Eigenschaften von den Datenstrukturen verlangt werden. Es können jedoch nur solche Eigenschaften benutzt werden, die auch bei der Erzeugung des Repository auf dem Server untersucht wurden (siehe Abschnitt B.3). Wenn der Controller die Antwort vom Server bekommt, wird die Liste der Namen mittels der hier beschriebenen Methode übergeben.

B.2 ExampleLoader

In diesem Abschnitt soll die bereits vielfach erwähnte Klasse zum Laden von vordefinierten Parameterbelegungen beschrieben werden. Alle vom Entwickler erzeugten ExampleLoader müssen das Interface `edu.tum.dal.ExampleLoader` implementieren. Dieses Interface schreibt vor, daß zwei Methoden vorhanden sein müssen. Diese Methoden werden, neben dem Konstruktor mit der leeren Parameterliste, vom System aufgerufen. Es handelt sich um folgende Methoden:

- `boolean readFromFile(String fileName)`
  Nach der Erzeugung des ExampleLoader wird diese Methode verwendet, um eine Datei, die die Parameterbelegung enthält, zu lesen. Der Aufbau dieser Datei ist dem Entwickler gänzlich freigestellt. Einzige Konvention ist, daß die Methode einen Boole'schen Rückgabewert besitzt, der angibt, ob das Einlesen der Belegung erfolgreich verlief.

- `Object[] getExample()`
  Diese Methode wird dazu verwendet, die eingelesenen Parameter abzufragen. Sie werden in einer Reihung von Referenztypen als Rückgabewert der Methode geliefert. Diese Reihung wird dann an das zugehörige Parameterfenster übergeben, d.h. die Reihenfolge der Referenzen kann wieder frei vom Benutzer gewählt werden, entscheidend ist nur, daß sie dem entsprechenden Parameterfenster, das ja auch vom Entwickler implementiert wird, bekannt ist.
  Wenn der Entwickler im Parameterfenster die Methode `receiveExample` der Ba-

sisklasse aufruft, ist die Belegung der ersten Elemente der Reihung eingeschränkt (siehe Beschreibung dieser Methode in Abschnitt B.1).

B.3 Repository

Der letzte Abschnitt dieses Anhangs beschäftigt sich mit dem sogenannten Repository. Ein Repository wird verwendet, um die auf dem Server gespeicherten Datenstrukturen zu verwalten, genauer gesagt, um einen effizienten Zugriff auf die entsprechenden Bezeichner zu ermöglichen. Wie in Teilabschnitt 3.2.4 bereits beschrieben, wird hierzu, durch eine Instanz der Klasse `edu.tum.dal.Repository`, eine entsprechende Datei generiert.
Zur Laufzeit findet die Verwaltung jedoch nicht in einer solchen Insatnz, sondern in einer der Klasse `edu.tum.dal.TypeList` statt.

B.3.1 Klasse `Repository`

Die Informationsdatei kann durch einen geeigneten Aufruf der `main`-Methode dieser Klasse erzeugt werden. Als Parameter wird das Verzeichnis übergeben, in dem sich die Unterverzeichnisse für die entsprechenden Datenstrukturen befinden. In dieses Verzeichnis wird auch die Informationsdatei abgelegt.
Während der Abarbeitung der `main`-Methode werden alle Unterverzeichnisse gelesen und die enthaltenen Dateien in eine entsprechende interne Struktur (`TypeList`) aufgenommen. Wenn die einzelnen Datenstrukturen nach verschiedenen Kriterien sortiert werden sollen, muß die Klasse `Repository` entsprechend erweitert werden. Dies geschieht in der Methode `handleFile`, vergleiche hierzu die derzeitige Sonderbehandlung `handleJGraph` für die Klasse `JGraph`. Sie überprüft, ob es sich bei der jeweiligen Instanz um einen gerichteten oder ungerichteten Graphen handelt. Etwaige Sortierkriterien werden in der Informationsdatei gespeichert und können zur Laufzeit des Systems für die Auswahl von Instanzen herangezogen werden.

B.3.2 Klasse `TypeList`

Diese Datenstruktur wird eingesetzt, um die Informationen der vorhandenen Instanzen zu speichern. Sie wird sowohl bei der Erzeugung des Repository, als auch zur Laufzeit des Systems verwendet. Die wichtigsten Methoden, die während des Systems aufgerufen werden, sind die folgenden. Eine Beschreibung der restlichen Methoden ist der Dokumentation des Codes (JavaDoc) zu entnehmen.

- `void readFromFile(String fileName)`
  Diese Methode liest eine zuvor durch die Klasse `Repository` erzeugte Informationsdatei ein und ermöglicht ab dann den Zugriff auf darin enthaltene Daten. Der absolute Dateiname der Informationsdatei wird in dem Methodenparameter `fileName` übergeben.

- `String[] getFittingNames(String type, String[] propertyList)`
  Wenn von einem Parameterfenster die Bezeichner von Datenstrukturen, die evtl. zusätzlich noch verschiedene Eigenschaften besitzen, angefordert werden, wird diese Methode aufgerufen. Die Parameter `type` und `propertyList` geben den gewünschten Datentyp sowie eine gewünschte Liste von Eigenschaften an. Es wird hierbei gefordert, daß die zurückgelieferten Datentypen alle Eigenschaften erfüllen.

Soll keine Einschränkung auf den Datentypen erfolgen, wird an Stelle der Referenz auf die Liste eine `null`-Referenz übergeben.
Die Bezeichner aller Instanzen, die der Anfrage entsprechen, werden in einer Reihung als Rückgabewert der Methode bereitgestellt. Wenn die Menge der gefundenen Instanzen leer ist, wird hierfür die `null`-Referenz verwendet.

Anhang C

Code der 1. Phase des Algorithmus

In diesem Kapitel ist die Methode zur Abarbeitung der ersten Phase wiedergegeben. Am Ende des *listings* befindet sich eine kurze Beschreibung des Ablaufs.

```
(1)  void  phase1()
(2)  {
(3)    Node v,w,opposite;
(4)    Edge e;
(5)
(6)    // initialize the counter for global relabeling
(7)    int stepCount = globalRelabelingPeriod;
(8)
(9)    while((v=getActiveNode())!= null)
(10)   {
(11)     if(show)
(12)       {
(13)         nodesJ[v.id].setBackground(Color.red);
(14)         step();
(15)       }
(16)
(17)     if(v.actual < v.actualNr)
(18)       e = v.actualEdges[v.actual];
(19)     else
(20)       e = null;
(21)     while(v.excess != 0 && e != null)
(22)       {
(23)         if(e.target.dist == v.dist-1 && e.cap > 0)
(24)           {// push possible using edge e
(25)             if(show)
(26)               {
(27)                 edgesJ[e.id].setBackground(Color.red);
(28)                 step();
(29)               }
(30)             //push
(31)             int f =0;
```

```
(32)              if(v.excess>=e.cap)
(33)                {
(34)                  satPushesCount++;
(35)                  f= e.cap;
(36)                }
(37)              else
(38)                {
(39)                  nonSatPushesCount++;
(40)                  f=v.excess;
(41)                }
(42)              opposite = e.target;
(43)              v.excess -= f;
(44)              if(opposite.id  != sRes.id)
(45)                opposite.excess += f;
(46)              else
(47)                opposite.excess -= f;
(48)              e.cap    -= f;
(49)              if(e.source.id == edges[e.id].source.id)
(50)                {// kante verl"auft parallel => flow erh"ohen
(51)                  edges[e.id].flow += f;
(52)                  anti[e.id].cap   += f;
(53)                }
(54)              else
(55)                {// kante verl"auft antiparallel => flow erniedrigen
(56)                  edges[e.id].flow -= f;
(57)                  parallel[e.id].cap += f;
(58)                }
(59)
(60)              if(show)
(61)                {
(62)                  edgesJ[e.id].setCaption(edges[e.id].flow
(63)                                          +"/"+edges[e.id].cap);
(64)                  nodesJ[e.source.id].setCaption(resg[e.source.id].excess
(65)                                          +"/"+resg[e.source.id].dist);
(66)                  nodesJ[opposite.id].setCaption(resg[opposite.id].excess
(67)                                          +"/"+resg[opposite.id].dist);
(68)                  step();
(69)                }
(70)
(71)              if(! active[opposite.id] && opposite.id != sRes.id
(72)                                       && opposite.id != tRes.id)
(73)                {
(74)                  if(show)
(75)                    nodesJ[opposite.id].setBackground(Color.blue);
(76)                  addActiveNode(opposite);
(77)                }
(78)            }
(79)          else if(show)
(80)            {// mark edge as not useful
(81)              edgesJ[e.id].setBackground(Color.green);
(82)              step();
(83)            }
```

```
(84)          if(show)
(85)            {// reset Edge color
(86)              edgesJ[e.id].setBackground(Color.white);
(87)              step();
(88)            }
(89)          // if necessary adjust edge referenze
(90)          if (v.excess != 0)
(91)            {
(92)              v.actual++;
(93)              if(v.actual < v.actualNr)
(94)                e = v.actualEdges[v.actual];
(95)              else
(96)                e = null;
(97)            }
(98)        }
(99)
(100)     //test if relabel is necessary
(101)     if(v.excess > 0)
(102)       {
(103)         if(useGlobalRelabeling)
(104)           stepCount--;
(105)         relabeltime -= (new Date()).getTime();
(106)         relabelCount++;
(107)         if(useGapRelabeling)
(108)           removeElementFromGapList(v,v.dist);
(109)         int oldDist = v.dist;
(110)         int minDist = 3*nodeNr;
(111)
(112)         if(selectActiveNode == 1)
(113)           {
(114)             // HLF
(115)             v.actual = 0;
(116)             e = v.first;
(117)             while(e != null)
(118)               {
(119)                 if(e.target.dist< minDist && e.cap > 0)
(120)                   {
(121)                     v.actualEdges[0] = e;
(122)                     v.actual = 1;
(123)                     minDist = e.target.dist;
(124)                   }
(125)                 else if (e.target.dist == minDist && e.cap > 0)
(126)                   {
(127)                     v.actualEdges[v.actual] = e;
(128)                     v.actual++;
(129)                   }
(130)                 e = e.next;
(131)               }
(132)             v.actualNr = v.actual;
(133)           }
```

```
(134)           else
(135)             {
(136)               // sonst
(137)               e = v.first;
(138)               while(e != null)
(139)                 {
(140)                   if(e.target.dist< minDist && e.cap > 0)
(141)                     {
(142)                       minDist = e.target.dist;
(143)                     }
(144)                   e = e.next;
(145)                 }
(146)             }
(147)
(148)           v.dist = minDist+1;
(149)           if(useGapRelabeling)
(150)             addElementToGapList(v,v.dist);
(151)           v.actual = 0;
(152)           v.current = v.first;
(153)           if(v.dist < nodeNr)
(154)             {
(155)               addActiveNode(v);
(156)               if(show)
(157)                 {
(158)                   nodesJ[v.id].setCaption(resg[v.id].excess
(159)                                           +"/"+resg[v.id].dist);
(160)                   nodesJ[v.id].setBackground(Color.blue);
(161)                   step();
(162)                 }
(163)             }
(164)           else if(show)
(165)             {
(166)               nodesJ[v.id].setCaption(resg[v.id].excess
(167)                                       +"/"+resg[v.id].dist);
(168)               nodesJ[v.id].setBackground(Color.orange);
(169)               step();
(170)             }
(171)
(172)           relabeltime += (new Date()).getTime();
(173)
(174)           // if demanded test for gap
(175)           if(useGapRelabeling)
(176)             {
(177)               testGap(oldDist);
(178)
(179)               if(show)
(180)                 step();
(181)             }
(182)         }
(183)       else if(show)
(184)         {
(185)           nodesJ[v.id].setBackground(Color.orange);
(186)           step();
(187)         }
```

```
(188)       if(useGlobalRelabeling)
(189)         {
(190)           if(stepCount == 0 && activeNodesIsEmpty() == false)
(191)             {// do global relabeling
(192)               stepCount = globalRelabelingPeriod;
(193)               globalRelabelingCount++;
(194)               if(editor != null)
(195)                 {
(196)                   this.setStatusBarText("global relabeling running");
(197)                   this.setStatusBarTextColor(Color.red);
(198)                 }
(199)               if(show)
(200)                 step();
(201)               bfs();
(202)               if(editor != null)
(203)                 {
(204)                   this.setStatusBarText("global relabeling finished");
(205)                   this.setStatusBarTextColor(Color.red);
(206)                 }
(207)               if(show)
(208)                 step();
(209)               if(editor != null)
(210)                 {
(211)               this.setStatusBarText("Running.");
(212)               this.setStatusBarTextColor
(213)                      (JGraphEditorWin.DEFAULT_STATUSBAR_TEXTCOLOR);
(214)                 }
(215)             }
(216)         }
(217)     }// end while((v=getActiveNode())!= null)
(218)   }// end phase1()
```

Die Methode wird nach der Initialisierung des Präflußes aufgerufen. Der wesentliche Bestandteil der Methode ist die `while`-Schleife zur Abarbeitung der aktiven Knoten (Zeile 9 - 217). In jedem Durchlauf werden die nachfolgenden Schritte durchgeführt.

1. Für den aktuellen Knoten wird solange die folgende Schleife (Z. 21 - 98) für die verbleibenden Kanten der Kantenliste (beginnend bei der aktuellen Kante) ausgeführt, bis entweder die letzte Kante der Kantenliste abgearbeitet wurde oder im Knoten kein Überfluß mehr vorhanden ist (Z. 21).

   (a) Falls die aktuelle Kante zulässig (Z. 23) ist, werden möglichst viele Flußeinheien über die Kante geschoben. Hierbei muß das Original- und das Residuennetz akutalisiert werden (Z. 25 - 69). Eventuell muß noch der gegenüberliegende Knoten in die Menge der aktiven Knoten aufgenommen (Z. 71 - 77) werden.

   (b) Wenn der Knoten immer noch Überfluß besitz, wird die aktuelle Kante durch die ihm nachfolgende in der Kantenliste ersetzt (Z. 90 - 97).

2. Konnte der Überfluß nicht vollständig abgebaut werden, wird eine Relabel-Operation auf den Knoten durchgeführt (Z. 101 - 182). Diese läuft wie folgt ab:

   (a) Nach einer kurzen Initialisierung (Z. 103 - 110), wird der neue Entfernungswert bestimmt. Da bei der HLF-Methode an dieser Stelle auch die Kantenliste

optimiert werden kann, wird je nach verwendeter Methode ein anderer Code abgearbeitet (HLF: Z. 113 - 133, andere: Z. 135 - 146).

(b) Der neue Wert wird dem Knoten zugewiesen, ggf. die Distanzverwaltung aktualisiert, und die aktuelle Kante des Knotens neu gesetzt (Z. 148 - 152).

(c) Falls der Entfernungswert nicht zu groß geworden ist, wird der Knoten wieder in die Menge der aktiven Knoten aufgenommen (Z. 154 - 163) anderenfalls wird er zurückgesetzt und nicht mehr bearbeitet (Z. 165 -170).

(d) Falls die Strategie des Gap-Relabelings verwendet wird, muß nun überprüft werden, ob eine Lücke entsanden ist. Wenn dem so ist, dann werden die entsprechenden Knoten aus der Menge der aktiven Knoten entfernt (Z. 175 - 181).

3. Zum Ende eines jeden Durchlaufs muß, falls dies eingestellt ist, getestet werden (Z. 188 - 190), ob ein Global-Relabeling durchzuführen ist (Z. 191 -215). Der Zähler hierfür wird während der Relabel-Operationen angepaßt (Z. 149-150).

Anhang D

Code der 2. Phase des Algorithmus

Auch für die zweite Phase wird die verwendete Methode angegeben. Wieder befindet sich anschließend eine kurze Erklärung zum Ablauf.

```
(1)  void phase2()
(2)   {
(3)     int WHITE = 0;
(4)     int GREY  = 1;
(5)     int BLACK = 2;
(6)     int[]  rank    = new int[nodeNr];
(7)     Node[] nl_prev = new Node[nodeNr];
(8)     Node[] nl_next = new Node[nodeNr];
(9)     Node n=null;
(10)    Node m= null;
(11)    Node r= null;
(12)    Node tos= null;
(13)    Node bos= null;
(14)    Node restart= null;
(15)    Edge a= null;
(16)    int delta = 0;
(17)
(18)    for(int i=0;i<nodeNr;i++)
(19)      {
(20)        rank[i]=WHITE;
(21)        nl_prev[i] = null;
(22)        nl_next[i] = null;
(23)        nodes[i].current = nodes[i].first;
(24)        resg[i].current = resg[i].first;
(25)      }
```

```
(26)      for(int i = 0;i<nodeNr; i++)
(27)        {
(28)          n = nodes[i];
(29)
(30)          if(rank[n.id] == WHITE && n.id != tRes.id)
(31)            {
(32)              r = n;
(33)              rank[n.id] = GREY;
(34)              while(true)
(35)                {
(36)                  for(;n.current != null;n.current = n.current.next)
(37)                    {
(38)                      a = n.current;
(39)                      if(a.flow>0 && a.target.id != sRes.id
(40)                                  &&a.target.id != tRes.id)
(41)                        {
(42)                          m = a.target;
(43)                          if(rank[m.id] == WHITE)
(44)                            {
(45)                              rank[m.id] = GREY;
(46)                              nl_prev[m.id] = n;
(47)                              n = m;
(48)                              break;
(49)                            }
(50)                          else
(51)                            if(rank[m.id] == GREY)// cycle
(52)                              {
(53)                                delta = a.flow;
(54)                                while(true)
(55)                                  {
(56)                                    a = m.current;
(57)                                    delta = Math.min( delta, a.flow);
(58)                                    if( m.id == n.id)
(59)                                      break;
(60)                                    else
(61)                                      m = m.current.target;
(62)                                  }
(63)                                m = n;
(64)                                while(true)
(65)                                  {
(66)                                    a = m.current;
(67)                                    a.flow -= delta;
(68)                                    parallel[a.id].cap += delta;
(69)                                    anti[a.id].cap     -= delta;
(70)                                    m = a.target;
(71)
(72)                                    if(m.id == n.id)
(73)                                      break;
(74)                                  }
```

```
(75)                                restart = n;
(76)                                for(m = n.current.target; m.id != n.id;
(77)                                    m = a.target)
(78)                                  {
(79)                                    a = m.current;
(80)                                    if(rank[m.id] == WHITE || a.flow ==0)
(81)                                       {
(82)                                         rank[m.current.target.id]=WHITE;
(83)                                         if(rank[m.id] != WHITE)
(84)                                           restart = m;
(85)                                       }
(86)                                  }
(87)
(88)                                if(restart.id != n.id)
(89)                                  {
(90)                                    n = restart;
(91)                                    n.current=n.current.next;
(92)                                    break;
(93)                                  }
(94)                              }
(95)                         }
(96)                    }
(97)
(98)                  if(n.current == null)
(99)                    {
(100)                     rank[n.id] = BLACK;
(101)                     if( n.id != sRes.id && n.id != tRes.id)
(102)                       {
(103)                         if(bos == null)
(104)                           {
(105)                             bos = resg[n.id];
(106)                             nl_next[n.id] = null;
(107)                             tos = resg[n.id];
(108)                           }
(109)                         else
(110)                           {
(111)                             nl_next[tos.id] = resg[n.id];
(112)                             nl_next[n.id]   = null;
(113)                             tos = resg[n.id];
(114)                           }
(115)                       }
(116)
(117)                     if(n.id != r.id)
(118)                       {
(119)                         n = nl_prev[n.id];
(120)                         n.current = n.current.next;
(121)                       }
(122)                     else
(123)                       break;
(124)                   }
(125)               }// end while(true)
(126)           }
(127)       }
```

```
(128)     // return excess
(129)     // note that sink is not on the stack
(130)
(131)     if(bos != null)
(132)       {
(133)         n = bos;
(134)         while(true)
(135)           {
(136)             a = n.first;
(137)             while( n.excess > 0)
(138)               {
(139)                 if( a == anti[a.id] &&a.cap > 0)
(140)                   {
(141)                     delta = Math.min(n.excess,a.cap);
(142)                     a.cap -= delta;
(143)                     edges[a.id].flow -= delta;
(144)                     parallel[a.id].cap += delta;
(145)                     n.excess -= delta;
(146)                     if(a.target.id != sRes.id)
(147)                       a.target.excess += delta;
(148)                     else
(149)                       a.target.excess -= delta;
(150)                   }
(151)                 a = a.next;
(152)               }
(153)
(154)             n = nl_next[n.id];
(155)             if(n == null)
(156)               break;
(157)           }//end while(true)
(158)       }
(159)   }
```

Die zweite Phase läßt sich in zwei Teile aufspalten. Der erste Teil entfernt Flußeinheiten auf etwaigen Zyklen und legt die Knoten in toplogischer Sortierung auf einen Stack ab (Z. 26 -127). Im zweiten Teil wird dann unter Verwendung der topologischen Sortierung der Überfluß der einzelnen Knoten in die Quelle zurückgeschoben (Z. 132 -157).
Der erste Schritt wird im wesentlichen für jeden Knoten die folgende Schleife durchlaufen:

1. Für den Knoten wird, falls er nicht schon vorzeitig auf den Stack gelegt wird (in diesem Fall ist für ihn der Schleifendurchlauf bereits beendet) eine Tiefensuche gestartet (Z. 32 - 125).

2. Wird während der Tiefensuche ein Zyklus entdeckt (Z. 51), wird der zirkulierende Fluß bestimmt (Z. 53 - 62) und von allen Kanten auf dem Zyklus entfernt (Z. 63 - 74). Hierdurch wird der Fluß, der über eine der Zyklenkanten fließt ganz entfernt. Es wird der Zielknoten dieser Kante bestimmt und zum aktuellen Knoten in der Tiefensuche gemacht (Z. 75 - 93). Der letzte aktuelle Knoten wird gespeichert (Z. 46) um später als Wiederaufsatzpunkt zu dienen (Z. 117 - 123).

3. wird in der Tiefensuche ein Blatt erreicht (Z. 98), so wird das Blatt auf den Stack gelegt (Z. 100 - 115) und ggf. ein Wiederausatzpunkt verwendet, anderenfalls wird die Schleife für den nächsten Knoten durchlaufen.

Der zweite Schritt verwendet die im ersten Schritt aufgebauten topologische Sortierung. Die Knoten werden in der Reihenfolge, in der sie auf den Stack geschoben wurdenabgearbeitet. Hierbei wird der im Knoten vorhandene Übefluß jeweils über die Kanten zurückgeschoben, von denen Einheiten in den Knoten fließen. Die Wahl der Kanten ist beliebig, da die topologische Sortierung sicherstellt, daß die Knoten auf der anderen Seite der Kante erst noch bearbeitet werden. Es kann also nicht vorkommen, daß Fluß zu einem Knoten geschoben wird, der schon bearbeitet wurde. Auf diese Weise werden die Einheiten langsam Richtung Quelle zurückgeschoben, bis sie letztlich in ihr ankommen.

Abbildungsverzeichnis

Literaturverzeichnis

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[AO87] Ravindra K. Ahuja and James B. Orlin. A fast and simple algorithm for the maximum flow problem. Preprint OR 165-87, Operations Research Center, M.I.T., June 1987.

[CG94] Boris V. Cherkassky and Andrew V. Goldberg. On implementing push-relabel method for the maximum flow problem, September 1994.

[Din70] E.A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Sov. Math., Dokl.*, 11:1277–1280, 1970.

[Dub98] Ilia Dub. Algorithmenvisualisierung in Java. Diplomarbeit, Institut für Informatik, Technische Universität München, 1998.

[GG88] D. Goldfarb and M.D. Grigoriadis. A Computational Comparison of the Dinic and Network Simplex Method for Maximum Flow. *Annals of Oper.Res*, 13:83–123, 1988.

[GR98] Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition Barrier. *J. ACM*, 45(5):783–797, 1998.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[GT90] Andrew V. Goldberg and Robert E. Tarjan. Finding minimum-cost circulations by successive approximations. *Math. Oper. Res.*, 15(3):430–466, 1990.

[JM93] D.S. Johnson and C.C. McGeoch. Network Flows and Matching: 1rst DIMACS Implementation Challenge. Technical report, AMS, 1993.

[Kar74] A.V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Sov. Math., Dokl.*, 15:434–437, 1974.

[May98] E.W. Mayr. Effiziente Algorithmen und Datenstrukturen II. Vorlesungsmitschrift, Lehrstuhl für Effiziente Algorithmen, Institut für Informatik, Technische Universität München, 1998. Kap. 6.

[Sle80] D.D. Sleator. An $O(nm \log n)$ algorithm for maximum network flow. Technical Report STAN-CS-80-831, Computer Science Dept., Stanford University, 1980.

[ST83] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, June 1983.