Verteilte Systeme: Übung 4

Max Wisniewski, Alexander Steen

Tutor: Philipp Schmidt

Aufstellen des Graphen

Um zu garantieren, dass der Graph zusammenhängend ist, fügen wir initial bei der Erschaffung einen großen Kreis in den Graphen: Dabei verbinden wir Knoten i mit Knoten j wenn $i + 1 mod n = j$, also ist jeder Knoten mit dem Knoten "eins weiter (mod n)" verbunden. Weiterhin fügen wir eine Kante hinzu, falls die Zufallszahl *ran.nextDouble*() dies zulässt.

```
List<GHSWork> network; // Liste von GHS-Knoten

for(int i=0; i<n; ++i){
    for(int j=0; j<n; ++j){
        if(i==j) continue; // Keine Selbstkanten
        if(ran.nextDouble()<p || (i+1 % n)==j){
            GHSWork a,b;
            a = network.get(i);
            b = network.get(j);
            int weight = ran.nextInt(5*n);
            BidirectionalPipe pipe = new BidirectionalPipe();
            a.addNeighbor(pipe.gehtLeft(), weight);
            b.addNeighbor(pipe.gehtRight(), weight);
        }
    }
}
```

Der Algorithmus

Unsere Implementierung ist ziemlich lang geworden, da wir uns ziemlich viele Zustände merken müssen und bei verschiedenen Nachrichten verschieden reagieren müssen. Aus diesem Grund beschränken wir uns in dieser Abgabe auf die Grundideen der Implementierung.
Die Implementierung von GHSNode, in unserem Falle GHSWork, besitzt genau wie die synchronen Algorithmen sende-, Empfangs- und Rechenphasen. Der Unterschied aber ist, dass nun durch einen fehlenden Sequenzer[1] nicht auf ACKS gewartet werden muss und jeder Knoten ununterbrochen und ohne Wartezeit nacheinander diese Phasen durchläuft. Um unsere Zustände zu verwalten, haben wir eine Reihe von Int-

Konstanten statisch der Klasse hinzugefügt, da diese nur als mnemoische Kennung dienen sollen, ist der genaue Wert völlig irrelevant. Das erste, was wir bei Start des Algorithmuses tun, ist die Sortierung der

Adjazenzlisten. Da die meißten Datentypen zu umständlich zu bedienen waren, haben wir uns für 2 Listen eintschieden, die nur über die Indizes verbunden sind. Da wir auf dieser Struktur die Sortierung selber schreiben mussten, haben wir uns, da es die Laufzeit insgesammt gesehen asymptotisch nicht verschelchter, Selectionsort verwendet. Dies macht das handeln mit der Indizeverlinkung wenigstens noch recht angenehm. Nun besteht die Sende- wie auch die Rechenphase im Grunde genommen aus einem großen switch-Block über

[1] In unserem Falle haben wir genau genommen immer noch eine Sequencer-Klasse, allerdings kann diese nur noch start und stop auf das Netzwerk ausüben und erfüllt darum nicht mehr die Aufgabe eines Sequencers.

den Zustand des Knotens. Dort haben wir ein Konstrukt der Form:

```
switch (this.state) {
case INIT:
case MWOESEARCH:
case MWOEREPORT:
case KMWOEREPORT:
case CONNECT:
case ABSORB:
case CHANGEROOT:
case READY:
case END:
}
```

Dabei wollen wir zuerst die Intention der einzelnen Zustände kurz erklären:

init Der Zustand init ist dafür da, die aktuelle Komponenten-ID und die aktuelle Stufe von der Wurzel in alle Knoten der Komponente zu transportieren. Das machen wir, in dem wir die Nachricht an alle unsere Kinder des bisherigen Teil-MSTs schicken.

mwoesearch Ist ein Knoten in diesem Zustand, so ist es aktuell das Ziel die MWOE der eigenen Komponente zu finden. Dies wird durch das Senden von Test-Nachrichten an adjazente Knoten (aufsteigend nach ihrem Gewicht) realisiert. Dabei muss man also auch evtl. auf minimale Kanten von allen eigenen Unterbäumen warten.

mwoereport In diesem Zustand wird die Information über die minimale ausgehende Kante eines Knotens an den Vaterknoten geschickt. Dabei merkt sich jeder Knoten den Weg zum nächsten Knoten, der auf dem Weg zu potenziellen MWOE liegt.

kmwoereport In diesem Zustand hat die Wurzel entschieden, welche Kante die MWOE ist und schickt diese Information anhand der gespeicherten Rückweg-Kanten zurück (also über die Kante, über die die MWOE gemeldet wurde). Sind wir der Knoten, die die MWOE meldete, können wir in CONNECT übergehen und den Wunsch äußern, uns verbinden zu wollen.

connect In dieser Phase haben wir entweder schon eine Connect-Anfrage der Gegenseite erhalten oder schicken als erster Knoten diese Anfrage. Dann warten wir evtl. auf die Connect-Antwort der Gegenseite und beginnen dann je nach Stufe in den Absorb-Zustand zu wechseln oder von der Gegenseite absorbiert zu werden. Implizit wird hier bei gleicher Stufe ein Merge ausgelöst.

absorb In diesem Zustand warten wir, bis die andere Komponente absorbiert wurde und wir eine Ready-Nachricht bekommen. Ist dies der Fall, leiten wir diese Nachricht an unseren Vaterknoten weiter.

changeroot Ist ein Knoten in diesem Zustand, so liegt er auf dem Weg zwischen absorbierter/gemergter Wurzel. So muss die Kante umgedreht werden, da wir nun eine neue Wurzel besitzen (bauen also die neue Komponente als Teil unseres Teil-MST über die MWOE weiter auf).

ready In diesem Zustand warten wir darauf, dass die Wurzel der Komponente eine neue Runde startet.

end In diesem Zustand befindet sich die Wurzel, wenn keine MWOE mehr gefunden werden konnte, sie also von allen Kindern keine MWOE mehr geliefert bekommt. Hier werden alle Knoten die Ende-Nachricht bekommen und der Algorithmus beendet sich.

Suchen der MWOE:

In Abbildung 1 ist schematisch das Vorgehen gezeigt, wie wir im Status MWOESearch die MWOE der eigenen Komponente suchen: Mit Testnachrichten werden alle Knoten ihre adjazenten Knoten in aufsteigender Gewichtreihenfolge testen und schließlich eine minimale Kante zu einem Knoten außerhalb der eigenen Komponente gefunden haben (oder natürlich keine) und dies dem Vaterknoten mitteilen. Dieser wartet, dass alle seine Kinderknoten eine Antwort geliefert haben und leitet dann das Minimum dieser Kanten wiederum an seinen Vaterknoten. Dabei speichert er sich, von welcher Kante die vorgeschlagene MWOE kam in der Variable *MWOEFrom*. Dies wird nun rekursiv solange gemacht, bis die Wurzel die Reportnachrichten bekommt und eine Kante als MWOE auswählt. Entlang der gespeicherten Rückreferenz wird dann die Nachricht, dass genau diese Kante als MWOE ausgewählt wurde, zurückgeschickt, bis diese Nachricht den Knoten erreicht, die diese MWOE fand (rechts in der Abbilding). Dann schickt dieser Knoten über die MWOE eine connect-Anfrage.

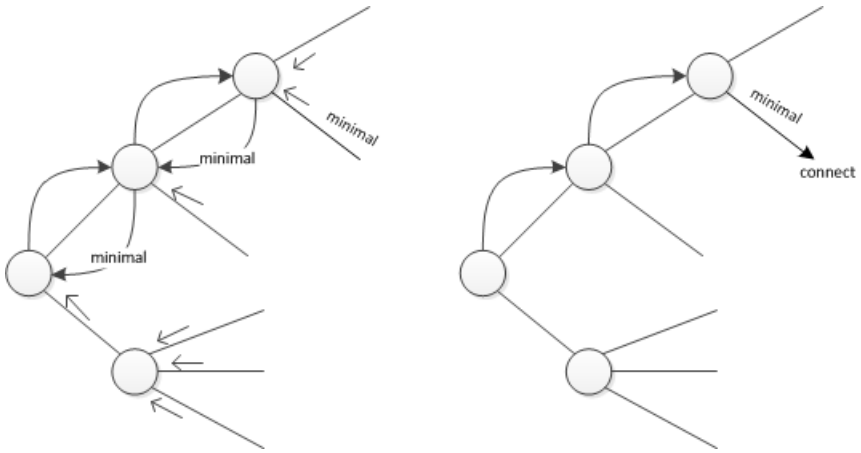

Figure 1: Suche und Bericht eines MWOE (links),Anfrage von demselben nach außen (rechts)

An dieser Stelle hat unsere Implementierung noch ein paar Macken. Wir finden die MWOE zwar für Komponenten der größe 1, aber in der Implementierung dieses gezeigten Schemas hatten wir noch einige Probleme, die hauptsächlich in der Unübersichtlichkeit des Codes fußten.

Verschmelzen durch CONNECT-Anfrage:

Wenn wir nun die CONNECT Nachrichten ausgetauscht haben (und zwar beidseitig). Wird weiter gearbeitet. Wir haben in unserer Implementierung, damit es nicht noch unüberschaubarer wird, absorb nur als einen Untertypus von Merge aufgefasst. Wir beschreiben daher nun erst einmal Merge. Wie wir in der Fugure 2, dem oberen Teil, sehen können, haben wir die Situation, dass zwei Komponenten Connect ausgeführt haben und beide ihre Kanten noch auf die Wurzel ihrer eigenen Komponenten gerichtet haben.

Nun wird die größere der beiden Komponenten gewählt (die größe kID), das dies die neue neue Wurzel stellen soll. Dazu wird ein ChangeRoot in die Komponente kId=5 geschickt, die alle Kanten von der MWOE bis zur alten Wurzel umdreht.

Durch diesen Trick haben wir zum Unserem MST die neue Kante hinzugenommen und trotzdem noch ein strenge Hirarchie erhalten, die wir im Algorithmus brauchen.

Wir haben das ChangeRoot in einem seperaten Test einmal Laufen lassen und es ging gut. Wir haben bisher also nur Probleme mit dem finden der MWOE.

Die Implementierung von ChangeRoot ist an sich nicht schwer. Wir schicken von dem Knoten der an der MWOE liegt (in kID = 5 in unserem Beispiel) die Nachricht CHANGEROOT an unseren alten Parent. Danach setzen wir unseren Parent auf die MWOE.

Alle die nun CHANGEROOT erhalten, setzen ihr Parent auf das Kind, von dem sie die Nachricht haben und senden ihrerseits die Nachricht an den alten Parent weiter.

Erhält die alte Wurzel die Nachricht (die ja keinen Parent hat), setzt diese ihren Parent ein letztes mal auf den Sender und schickt nun ein READY entlang dieses Weges.

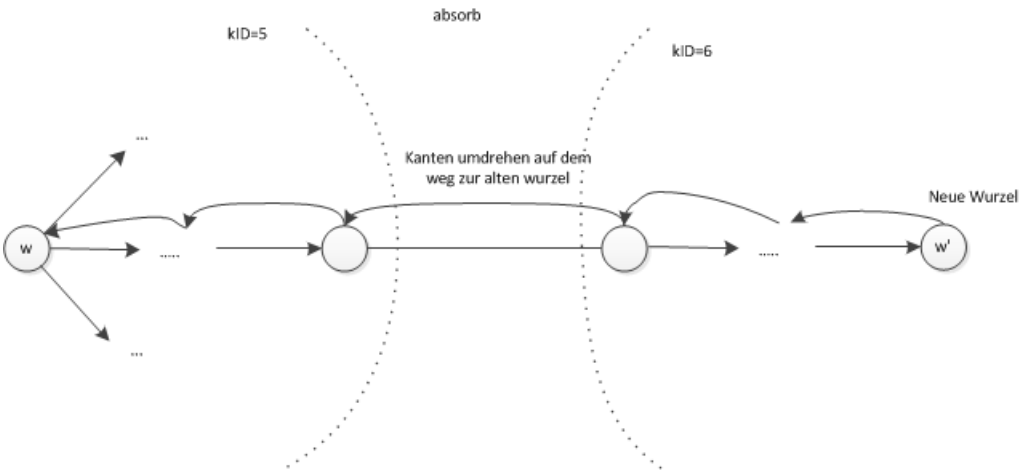

Figure 2: Merge und ChangeRoot Mechanismus für das Zusammenfassen von Komponenten

Dieses READY wird einmal komplett bis an die neue Wurzel geleitet und diese erhöht nun die Stufe und sendet ein neues INIT an alle Knoten. Damit sind wir in die nächste Phase eingetreten.

Ein Wurzel weiß nun, dass der Algorithmus fertig ist, wenn es von all seinen Kinder die Nachricht NOMWOE erhalten hat.

Diese Nachricht sagt an, dass sich unter einem Knoten keine OutgoingEdge mehr befunden hat. Setzt sich dies bis zur Wurzel fort, wissen wir, dass wenn der Graph zusammenhängend war, die Wurzel nun mit jedem Knoten verbunden sein muss.

Darauf sendet die Wurzel ein END an alle Kinder (rekursiv) und darauf beenden sich alle. (an dieser Stelle könnte sich ein anderer Algorithmus anschließen, der auf MSTs arbeitet).

Wir haben in dieser Abgabe versucht die Idee des Algorithmuses zu erklären, wie wir ihn Implementiert haben. Der Code an sich ist in den einzelnen Cases und Methoden kommentiert, aber wenn wir alle "wichtigen" Stellen hier rein kopiert hätten, wären es mehrere Seiten geworden und man hätte weniger Verstanden als bei einer Erläuterung der Implementierung.