

**Beschleunigung kürzeste Wege Algorithmen in der Praxis**  
Seminar: Verkehr und Logistik 2009  
Alan Hollesen Basse

# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>3</b>
1.1 Vorwort	3
1.2 Problemstellung	3
1.3 Zielsetzung	3
1.4 Gliederung	3
<b>2. Die reduce heap Technik</b>	<b>4</b>
2.1 Einleitung	4
2.2 Algorithmus von Ramalingam und Reps ohne <i>reduce heap</i>	4
2.2.1 Grundidee	4
2.2.2 Datenstruktur	5
2.2.3 Funktionsaufruf	6
2.2.4 Algorithmus	7
2.3 Algorithmus von Ramalingam und Reps mit <i>reduce heap</i>	8
2.3.1 Funktionsaufruf	8
2.3.2 Algorithmus	9
2.4 Unterschiede	10
2.5 Weitere Algorithmen	11
<b>3. Fazit</b>	<b>12</b>
3.1 Vergleich	12
3.2 Graphische Darstellung	12
3.3 Komplexitätsklasse	13
<b>4. Quellenverzeichnis</b>	<b>14</b>

# 1. Einleitung

## 1.1 Vorwort

In diesem Artikel geht es um das Präsentieren einer Technik zur Beschleunigung einiger kürzester Wege Algorithmen. Diese „*reduce heap*“ Technik basiert auf der Veröffentlichung „Speeding up dynamic shortest path algorithms“ von Luciana Buriol, Mauricio Resende und Mikkel Thorup aus dem Jahre 2008 [1].

## 1.2 Problemstellung

Unser Problem liegt in der Findung eines kürzesten Weges zwischen zwei Punkten. Die Aufgabe besteht also darin, die beste Route zu finden, und das auch unter Alternativen.

Generell haben wir den Vorteil, dass solch ein Problem mathematisch berechenbar ist. Hierzu gibt es unter anderem den Algorithmus von Dijkstra (1959). Dieser berechnet in erster Linie aber nur das statische Problem.

Betrachtet man nun die Realität, so stellt man fest, dass unser Routingproblem in der Praxis nur selten statisch ist. Wir erweitern also das statische Problem zu einem dynamischen Problem: Fährt man gerade von A nach B, so kann sich auf dem schnellsten Wege Staus bilden oder Straßen sind unbefahrbar, z.B. durch Bauarbeiten, Schnee oder ähnliches. Während unserer Navigation werden also neue Berechnungen durchgeführt werden müssen, um die neue (ggf. gleiche), kürzeste Route zu finden.

Hierzu gibt es auch bereits Ansätze und Lösungen. Und wegen der Praxisnähe zur Navigation und Routenberechnung wird auch weiterhin geforscht. Es ergeben sich zwei Themengebiete:

- die Entwicklung neuer Methoden und Algorithmen
- das Verbessern/Erweitern bereits bestehende Algorithmen

Mit dem letzteren beschäftigt sich die Veröffentlichung „Speeding up dynamic shortest path algorithms“ von Buriol, Resende und Thorup.

## 1.3 Zielsetzung

Ziel dieses Artikels ist es, die *reduce heap* Technik zu vorzustellen. Ihre Funktionsweise werde ich anhand des Algorithmus von Ramalingam und Reps [2] erläutert. Des Weiteren gibt es einen kurzen Blick auf andere Algorithmen.

Als Ausblick gab es Resultate, wo eine Beschleunigung der Berechnung um den Faktor 5 erreicht wurde, nur durch die Verwendung der *reduce heap* Technik.

## 1.4 Gliederung

Im Folgenden werde ich zuerst im Abschnitt 2.2. den Algorithmus von Ramalingam und Reps ohne die *reduce heap* Technik vorstellen. Anschließend folgt im Abschnitt 2.3. die Vorstellung des Ramalingam und Reps mit der *reduce heap* Technik. Danach werde ich kurz in 2.4. die Unterschiede eingehen, bevor ich im Abschnitt 3. zu einem Fazit kommen werde, um die Sache abzurunden.

## 2. Die reduce heap Technik

### 2.1 Einleitung

Die *reduce heap* Technik basiert auf die Grundidee anderer Algorithmen. Einer dieser ist der Algorithmus von Ramalingam und Reps.

### 2.2 Algorithmus von Ramalingam und Reps ohne reduce heap

Im Jahre 1996 veröffentlichten Ganesan Ramalingam und Thomas Reps in Madison, Wisconsin, ihre Forschungsergebnisse zur Verallgemeinerung von einem „single source shortest path problem“. Sie präsentierten dabei die Verbesserung eines dynamischen Algorithmus zum Lösen von Routenproblemen zwischen zwei Punkten [2, Seite 3].

Wir wollen von A nach B. Man fährt von A aus nun so lange eine Kante ab, bis man an eine Kreuzung ankommt. An dieser Stelle hat man einen neuen Knoten erreicht, nennen wir ihn  $A_1$ . Jetzt haben wir an sich ein neues Problem: wir suchen den Weg von  $A_1$  nach B und fahren wieder eine Kante ab. Die neue Situation lautet  $A_2$  nach B. Fährt man  $n$  Kanten ab, so landet man im Knoten  $A_n$ , es entsteht das Problem:  $A_n$  nach B.

Wie man sieht, ist unser Anfangsknoten nach dem Ablaufen einer Kante ein neuer Knoten, unser Zielknoten aber bleibt derselbe. Diese Erkenntnis wollen wir ausnutzen. Man nennt dies ein „single destination problem“ [1, Seite 1].

Gegeben ist ein Ausgangsgraph, z.B. eine Stadtkarte. Dazu existiert ein Subgraph oder auch Teilgraph. Dieser Teilgraph, weiter auch als Zielgraph gekennzeichnet, beinhaltet von jedem Punkt aus den kürzesten Weg zu unseren Zielknoten im Graphen. Dies ist nur möglich nach unserer Vorüberlegung über das „single destination problem“.

Es gibt nun die Möglichkeit, dass zwei verschiedene Routen von einem Knoten aus gleich lang sind. In diesem Fall werden beide Wege mit aufgenommen. Deswegen handelt es sich um einen Zielgraphen und nicht um einen Zielbaum (was auch ein möglicher Lösungsansatz darstellt).

#### 2.2.1 Grundidee

Der Grundgedanke von Ramalingam und Reps ist eine dynamische Lösung, welche weiter weiterhin den Algorithmus nach Dijkstra verwendet. Jedoch soll zuvor die Menge der relevanten Knoten und Kanten eingeschränkt werden. Durch die Reduzierung dieser Menge, also Speicher, erreicht man auch eine Reduzierung der Laufzeit ihres Algorithmus bei ausreichender Vorarbeit.

Noch zu sagen ist, dass der Algorithmus zwischen Inkrementierung und Dekrementierung unterscheidet. Tatsächlich veröffentlichten Ramalingam und Reps zuerst nur die Version zur Inkrementierung und erst kurze Zeit später den Algorithmus für Dekrementierungen.

## 2.2.2 Datenstruktur

Ich werde im Folgenden den Algorithmus anhand von Pseudocode erklären, weswegen man eine kurze Einführung in die Datenstruktur braucht.

Der Ausgangsgraph wird durch drei Arrays beschrieben:

In dem ersten Array, hier **fw – forward** genannt, befinden sich Knotenpaare. Hierzu gibt es unter anderem die Funktionen head und tail, wobei tail den Anfangsknoten einer Kante beschreibt und head den Zielknoten. Das fw-Array ist sortiert nach tail-Knoten. Der Index dieses Arrays identifiziert die Kante.

Der Name forward deutet darauf hin, dass auch ein reverse-Array existiert. Dies ist nur für andere Algorithmen notwendig, die auch in der Veröffentlichung erklärt werden. Für uns ist das allerdings nicht relevant.

Das zweite Array **p – point** beschreibt, ab welcher Position ein Knoten k im fw-Array beschrieben ist. D.h. wo befindet sich im fw-Array die Kante, die als tail den Knoten k hat. Als Index werden die Knotennamen verwendet.

Das letzte Array **w – weight** beinhaltet die Kantenlänge bzw. das Gewicht einer Kante. Als Index dient die Kantenbezeichnung, welche in fw festgelegt ist.

Der Zielgraph wird ebenfalls durch drei beschrieben:

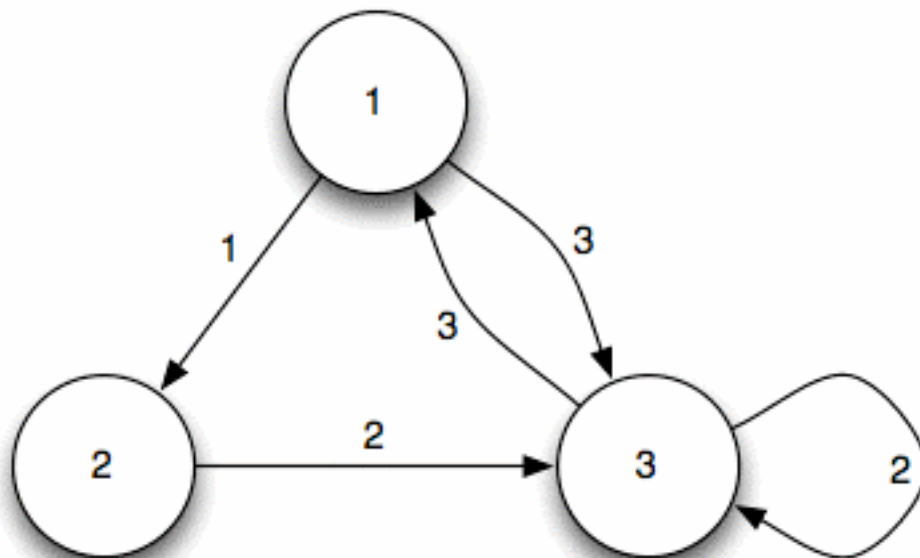
Das Array **d – distance** speichert die Gesamtstrecke von dem Knoten k zum Zielknoten. D.h. die Struktur des Zielgraphen kann erst aufgebaut werden. Wenn das Ziel feststeht. Als Index dienen die vorhandenen Knoten.

Das Array **gSP – graph shortest path** ist gefüllt mit Booleanwerten, bzw. die Werte 0/1. Index hierfür sind die Kanten. Gehört eine Kante zum Zielgraphen, also ist ein Teil eines kürzesten Weges, so steht an dessen Position im Array eine 1.

Das Array **δ – delta** beschreibt, wie viele Kanten aus einem Knoten k rausgeht, wobei  $\delta_k$  Element vom Zielgraphen.

Hier ein kleines Beispiel, wie die Datenstruktur aussehen würde. Hierbei ist zu beachten, dass Knoten 3 das Ziel darstellt.

Der Graph, bestehend aus drei Knoten (1 – 3) und fünf Kanten (a – e):



Die dazugehörige Datenstruktur (Indizes grau unter den Werten):

<b>fw</b>	1 → 2 a	1 → 3 b	2 → 3 c	3 → 1 d	3 → 3 e
<b>p</b>	a 1	c 2	d 3		
<b>w</b>	1 a	3 b	2 c	3 d	2 e
<b>d</b>	3 1	2 2	0 3		
<b>gSP</b>	1 a	1 b	1 c	0 d	0 e
<b>δ</b>	2 1	1 2	0 3		

### 2.2.3 Funktionsaufruf

*procedure RR+ (a, w, d, δ, gSP)*

Der Algorithmus braucht einige Parameter, hier ist die Syntax zu sehen. Das **a** stellt die Kante dar, welche dynamisch verändert wurde.

Der Algorithmus wird nur dann aufgerufen, wenn es eine Gewichtsveränderung stattgefunden hat, in unserem Fall eine Inkrementierung einer Kante. Dies muss also in irgendeiner Weise signalisiert werden. Hierfür eignet sich z.B. ein Beobachter, welcher aber für den Algorithmus selbst keine Rolle spielt. Um Platz im Hauptspeicher zu sparen, werden alle Parameter per „call by reference“ aufgerufen, so dass sie nicht noch mal kopiert werden müssen.

Im Laufe des Algorithmus füllen wir unser heap **H** mit Knoten auf. Am Ende wird der Algorithmus von Dijkstra mit dem heap **H** aufgerufen. Der Vorteil eines heaps liegt darin, dass man in konstanter Zeit das kleinste Element raussuchen kann.

Vorgeschlagen wird ein fibonacci-heap, damit Funktionen wie insertIntoHeap oder findAndDeleteMin in konstanter bzw. logarithmischer Zeit ablaufen. Wie aber genau unser heap arbeitet, soll uns aber an dieser Stelle nicht weiter interessieren.

## 2.2.4 Algorithmus

```
procedure  $RR+(a, w, d, \delta, gSP)$ 
1   if  $gSP_a = 0$  return;
2    $gSP_a = 0$ ;
3    $\delta_u = \delta_u - 1$ ;
4   if  $\delta_u > 0$  then return;
5    $Q = \{u\}$ ;
6   for  $u \in Q$  do
7        $d_u = \infty$ ;
8       for  $e = (s, u) \in IN(u)$  do
9           if  $gSP_e = 1$  then
10               $gSP_e = 0$ ;
11               $\delta_s = \delta_s - 1$ ;
12              if  $\delta_s = 0$  then  $Q = Q \cup \{s\}$ ;
13          end if
14      end for
15  end for
16  for  $u \in Q$  do
17      for  $e = (u, v) \in OUT(u)$  do
18          if  $d_u > d_v + w_e$  then  $d_u = d_v + w_e$ ;
19      end for
20      if  $d_u \neq \infty$  then InsertIntoHeap( $H, u, d_u$ );
21  end for

... // dijkstra( $H$ );
```

Der Algorithmus fängt damit an, zu gucken, ob die Kante  $a$  Element vom Zielgraphen ist (Zeile 1). Hierzu wird einfach das  $gSP$  an der Stelle  $a$  ausgelesen und verglichen. Steht an der Position eine 0, so kann man aufhören, weil eine Inkrementierung auf eine Kante keine Auswirkungen hat, falls er nicht zu den kürzesten Wegen gehört. Die Zeilen 2 bis 4 überprüfen anschließend, ob eine alternative Route existiert, die zuvor genau so gut war. Hierzu wird die Kante aus dem Zielgraphen raus genommen und die Anzahl der ausgehenden Kanten für den betroffenen Knoten um 1 verringert. Steht nun in  $\delta$  an der Position  $tail(a) = u$  ein Wert größer 0, so besitzt der Zielgraph weitere optimale Kanten. Somit gäbe es eine Alternativroute, die genau so gut war wie die Route über der Kante  $a$  zuvor. In diesem fall terminiert der Algorithmus wieder.

Andernfalls fängt das eigentliche Vorbereiten an. Es muss nun aussondiert werden, welche Knoten relevant für unser Problem sind, nämlich alle Knoten, die über der Kante  $a$  routen.

Hierfür wird  $Q$  mit  $tail(a) = u$  initialisiert (Zeile 5).  $Q$  stellt eine dynamische Menge dar, welche mit Knoten aufgefüllt wird. In den Zeilen 7 bis 15 wird zuerst die relevanten Knoten aus  $gSP$  identifiziert und der Menge  $Q$  hinzugefügt. In einer Iteration über die dynamische Menge  $Q$ , überprüft man für jede Kante, die in den Knoten (der Laufvariable) reingehen, ob diese Element des Zielgraphen ist. Ist das der Fall, so

wird zunächst die Kante  $a$  imaginär raus genommen. Dafür bekommt die Distanz zum Zielknoten den Wert  $\infty$  (Zeile 7).

Nun wird geprüft, ob es für jene Kante  $e$  eine Alternativroute existiert. Falls es keine gibt, so ist der Knoten in  $Q$  aufzunehmen (Zeile 10 – 12). Hierdurch ergeben sich die weiteren Knoten, worüber weiter iteriert wird. Es funktioniert also durch den Umkehrschluss: Gibt es keine Alternativroute von einem Knoten  $u$  aus, so muss dieser Knoten für die Berechnung aufgenommen werden.

Die Zeilen 16 bis 21 prüfen, ob es außerhalb von  $Q$  eine neue Route gibt. Ist das der Fall, so werden die Distanzen aktualisiert.

In  $Q$  befinden sich jetzt alle relevanten Knoten vor der Kante  $a$ . In der Iteration über  $Q$  überprüfen wir nun für jede Kante  $e$ , die aus der Laufvariable (Knoten  $u$ ) ausgehen, ob ein anderer, kürzerer Weg existiert (Zeile 18). Die Distanz setzt sich hierbei zusammen aus der Distanz des Knoten  $tail(e) = v$  und das Gewicht der Kante  $e$ . Zuletzt werden die Knoten aus  $Q$  in unser heap  $H$  aufgenommen, sofern die Kante noch existiert (Zeile 20).

Anschließend führen wir den Algorithmus von Dijkstra auf unser heap  $H$  aus und die Datenstruktur implizit angepasst. Wir erhalten so den aktualisierten Zielgraph und dazugehörend die kürzeste Route, die wir gesucht haben.

Zusammenfassend hatten Ramalingam und Reps die Idee den Algorithmus von Dijkstra zu beschleunigen, indem sie die Menge der zu betrachtenden Knoten einschränken, bevor der Algorithmus ausgeführt wird. Dies erreichen sie auf Kosten von Iterationen über Knoten vor der inkrementierten Kante  $a$ .

Diese Vorberechnungen basieren auf einige Vergleiche, welche beschleunigt werden durch eine etwas umfangreichere Datenstruktur.

Die Komplexitätsklasse bleibt dieselbe, also  $O(m + n \log(n))$  wobei  $n$  und  $m$  eine Teilmenge des Graphen darstellt.

## 2.3 Algorithmus von Ramalingam und Reps mit reduce heap

Kommen wir nun zur eigentlichen Erneuerung von der Veröffentlichung. Grundidee ist, dass man die Menge noch weiter einschränkt, bevor man den Algorithmus ausführt.

### 2.3.1 Funktionsaufruf

*procedure rhRR+ (a, w, d,  $\delta$ , gSP, inc)*

Es kommt lediglich ein weiterer Parameter hinzu: **inc** beinhaltet die Größe der Inkrementierung.

### 2.3.2 Algorithmus

```
procedure rhRR + (a, w, d,  $\bar{\delta}$ , gSP, inc)
1  if  $gSP_a = 0$  return;
2   $gSP_a = 0$ ;
3   $\bar{\delta}_u = \bar{\delta}_u - 1$ ;
4  if  $\bar{\delta}_u > 0$  then return;
5   $Q = \{u\}$ ;
6  for  $u \in Q$  do
7       $d_u = d_u + inc$ ;
8      for  $e = (s, u) \in IN(u)$  do
9          if  $gSP_e = 1$  then
10              $gSP_e = 0$ ;
11              $\bar{\delta}_s = \bar{\delta}_s - 1$ ;
12             if  $\bar{\delta}_s = 0$  then  $Q = Q \cup \{s\}$ ;
13         end if
14     end for
15 end for
16 if  $inc > 1$  then
17      $dist = dk$ ;
18     for  $e = (k, v) \in OUT(k)$  do
19         if  $dk > dv + w_e$  then  $dk = dv + w_e$ ;
20     end for
21      $diff = dist - dk$ ;
22     for  $u \in Q \setminus k$  do
23          $d_u = d_u - diff$ ;
24          $flag = 0$ ;
25         for  $e = (u, v) \in OUT(u)$  do
26             if  $d_u > dv + w_e$  then
27                  $d_u = dv + w_e$ ;
28                  $flag = 1$ ;
29             end if
30         end for
31         if  $flag = 1$  then InsertIntoHeap(H, u,  $d_u$ );
32     end for

... // dijkstra(H);

... end if
a  for  $u \in Q$  do
b      for  $e = (u, v) \in OUT(u)$  do
c          if  $d_u = w_e + dv$  then
d               $gSP_e = 1$ ;
e               $\bar{\delta}_u = \bar{\delta}_u + 1$ ;
f          end if
g      end for
h  end for
```

Vergleicht man den Anfang der Algorithmen, so stellt man fest, dass es lediglich eine kleine Veränderung gab: In Zeile 7 wird die Distanz von  $u$  wirklich berechnet, anstatt die Kante auf unendlich zu setzen.

Ansonsten ändert sich bis Zeile 15 nichts. Das bedeutet, wir filtern wieder die relevanten Knoten, welche über die Kante  $a$  routen.

Ab Zeile 16 überstreckt sich über den Rest der Prozedur eine if-Schleife. Hat die Inkrementierung die Größe 1, so brauchen wir gar kein heap. Hierbei gehen wir davon aus, dass wir nur ganzzahlige Kantengewichte haben. Ist dies nicht der Fall, so ist an dieser Stelle die kleinstmögliche Größe des Gewichts einzutragen. Der Gedanke, der hierhinter steckt, ist, dass wir zuvor schon sicherstellen, dass keine gleichgute Alternativroute existiert. Somit muss eine Route die um das Minimale erweitert wurde, weiter eine beste Route sein, auch wenn es jetzt alternativ auch andere gleichgute Routen existieren. Es muss am Ende nur noch die Datenstruktur angepasst werden (ab Zeile a). Hierzu wird jeweils einmal über die Knoten und dazugehörigen Kanten iteriert.

Gehen wir jetzt davon aus, dass die Größe der Inkrementierung größer als 1 ist: Wir führen eine neue Variable  $dist$  ein, die die Distanz von den direkten  $tail(a) = k$  erhält (Zeile 17). An sich passiert dasselbe wie vorher, wir betrachten allerdings den ersten Knoten gesondert (Zeile 18 – 20).

Die Variable  $diff$  bildet nun die Differenz zwischen  $dist$  und  $dk$ , wobei das  $dist$  der alte Wert von  $dk$  ist (Zeile 21). Anschließend iteriert man über die restlichen Knoten vor  $tail(a) = u$ . Erst zieht man die Differenz von der Gesamtdifferenz ab, um den aktualisierten Stand zu erhalten. Das  $flag$  dient als Boolean, welches auf  $true$  gesetzt wird, wenn der Knoten im heap aufgenommen werden soll (Zeile 31).

Wie auch zuvor, aktualisieren wir nun die Gesamtdistanzen, nur sorgen wir genau an dieser Stelle dafür, dass nicht alle Knoten im heap aufgenommen werden, sondern wirklich nur die signifikanten Knoten.

Anschließend wird der Algorithmus nach Dijkstra ausgeführt und die Datenstruktur angepasst.

## 2.4 Unterschiede

Vergleicht man die beiden Algorithmen, so stellt man fest, dass es keinen großen Unterschied gibt. Die Idee steckt darin, dass man nur die Knoten aufgenommen werden ins heap  $H$ , welche eine tatsächliche Veränderung vorweisen. Das Aufnehmen wird schließlich durch das  $flag$  gesteuert.

Zudem haben wir die bereits erklärte, kleine Verbesserung in der Zeile 16.

## 2.5 Weitere Algorithmen

Erläutert wurde die Technik anhand des Algorithmus von Ramalingam und Reps, und zwar nur für die Inkrementierung einer Kante. Sie ist übertragbar auf eine Dekrementierung einer Kante. Hierzu drehen wir logisch gesehen nur die Bedingungen um, z.B. können wir sofort aufhören, falls sich eine Kante, über die wir bereits routen, sich weiter verbessert. So bleiben wir beim selben Weg, welcher nur noch kürzer wurde.

Die *reduce heap* Technik ist allerdings auch auf weitere Algorithmen übertragbar. In der Veröffentlichung präsentieren die Autoren noch drei weitere Algorithmen:

- Ramalingam & Reps – Tree
- King & Thorup – Tree
- Demestrecu – Tree (nur für Inkrementierung)

### 3. Fazit

#### 3.1 Vergleich

Durch die Technik erreichen wir eine Verbesserung durch eine gewisse Vorarbeit. Die Vorarbeit kostet natürlich ein bisschen Zeit, weswegen der Einsatz der *reduce heap* Technik erst bei einer gewissen Größe von Knoten und Kanten rechnet. Betrachtet wir kurz noch die Mengen  $Q$ , welche die relevanten Knoten beinhalten:

$$Q_{RR} \supseteq Q_{rhRR}$$

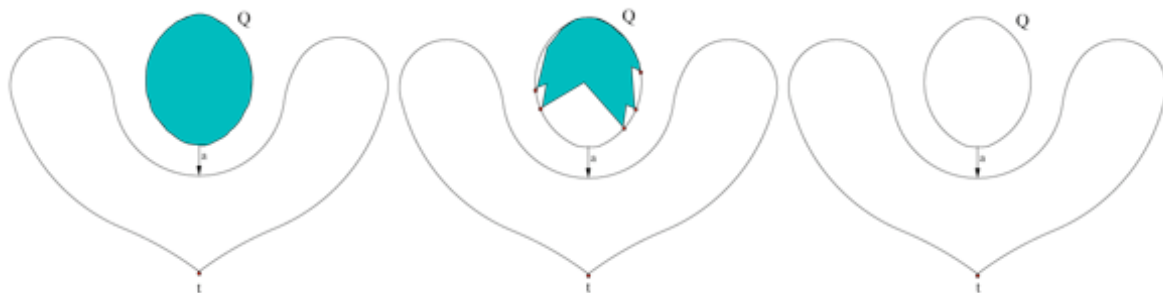
Im worst case tritt der Fall ein, dass die Mengen gleich sind. Jedoch ist dies nur der Fall, falls die inkrementierte Kante direkt vor unserem Zielknoten liegt. In jedem anderen Fall bildet die Menge  $Q$  mit der *reduce heap* Technik eine echte Teilmenge gegenüber dem  $Q$  aus Originalalgorithmus. D.h. eine Verbesserung ist nicht garantiert, eine Verschlechterung ist wiederum aber ausgeschlossen.

Es ist aber möglich, dass die *reduce heap* Technik zu einer langsameren Berechnung führt, als die ohne der Technik. Denn gehen wir vom worst case aus, so haben wir dasselbe Ergebnis bei mehr Vorberechnung. Diese Tatsache relativiert sich jedoch recht schnell, weswegen man nicht sagen kann, dass die Technik sich nicht rentiert.

#### 3.2 Graphische Darstellung

Hier ist der Vergleich auch graphisch zu sehen. Türkis eingefärbt erkennt man die Menge, welche in das heap  $H$  aufgenommen wurden.

Die rechte Abbildung zeigt ein Beispiel mit dem Originalalgorithmus von Ramalingam und Reps. Zum Vergleich erkennt man die Struktur nach der *reduce heap* Technik in der Mitte. Die linke Abbildung stellt das Beispiel  $inc = 1$ , es wird kein heap benötigt.



### 3.3 Komplexitätsklasse

Da wir den Dijkstra Algorithmus weiter verwenden, haben wir immer noch dieselbe Komplexitätsklasse:  $O(m + n \log(n))$ . Durch die Reduzierung von  $m$  und  $n$  kann man es auch auffassen als  $O(k [m + n \log(n)])$ , bei  $0 < k < 1$

Dies führte bei den Tests von den Autoren zu einer weiteren Beschleunigung von bis zum Faktor 5, bzw. ein  $k$  mit dem Wert 0,2.

Letztendlich hat jeder Algorithmus Vor und Nachteile. Der am meist geeignete Algorithmus hängt davon ab, in welchem Einsatzgebiet man sich befindet. Eine Vorüberlegung über das gegebene Problem ist deswegen notwendig.

Entscheidend sind Fragen wie:

Handelt es sich um ein dynamisches Problem oder reichen statische Algorithmen?

In welcher Größenordnung bezogen auf die Anzahl der Knoten und Kanten befinden wir uns?

Was für dynamische Möglichkeiten gibt es? Handelt es sich dabei nur um Inkrementierungen, dürfen Kanten neu hinzukommen?

Meist wird das Ergebnis lauten: Es gibt keinen besten Algorithmus. Bei dynamischen Problem gilt allerdings: Jede Wahl eines der genannten Algorithmen (siehe 2.5) scheint besser zu sein als der reine Algorithmus nach Dijkstra. Wir befinden uns zwar immer noch in derselben Komplexitätsklasse... Doch eine Beschleunigung von bis zu 149.371,17\* ist meiner Meinung nach ein Argument...

\* (hierbei ist zu beachten, dass diese Zahl eine veröffentlichte Zahl ist, die von den Autoren und nicht von mir stammt)

#### **4. Quellenverzeichnis**

[1]

*Speeding up dynamic shortest path algorithms*

Luciana S. Buriol, Mauricio G. C. Resende, Mikkel Thorup

2008, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasilien

2008, INFORMS Journal of Computing, Vol 20, Issue 2

[2]

*An incremental algorithm for a generalization of the shortest path problem*

Ganesan Ramalingam, Thomas Reps

1996, University of Wisconsin, Madison, USA

1996, Journal of Algorithms, Vol 21, Issue 2

#### **Bildverzeichnis:**

Seite 5: eigenes Bild

Seite 12: [1] Seite 17, Figure 10.