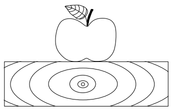
Dynamic-Trees und Anwendungen in Flußalgorithmen

Klaus Holzapfel



Technische Universität München — Institut für Informatik

Seminarvortrag im Rahmen des Kurses 2
"Bäume – Algorithmik und Kombinatorik"
Ferien-Akademie der FAU Erlangen und der TU München
Sarntal / Südtirol — 19. Sept. bis 1. Okt. 1999

Dynamic-Trees und Anwendungen in Flußalgorithmen

Klaus Holzapfel
(e-mail: `Klaus.Holzapfel@in.tum.de`)

Kursleiter: Prof. Dr. Ernst W. Mayr
: Prof. Dr. Volker Strehl
Betreuer: Dr. Thomas Erlebach

Zusammenfassung

Dieser Seminarvortrag stellt die Datenstruktur der "Dynamic Trees", ein sich dynamisch ändernder Wald von Bäumen, vor. Es werden verschiedene Möglichkeiten zur Implementierung erläutert, wobei jeweils die Komplexität und Einfachheit besprochen wird. Diese Implementierungen stützen sich auf die Datentypen "Biased Binary Trees" bzw. "Splay Trees". Beide Implementierungen haben gemeinsam, daß ein Dynamic Tree durch eine Sammlung von knotendisjunkten Pfaden dargestellt wird.

Abschließend wird die Anwendung der Dynamic Trees in einer Variante des Push-Relabel-Algorithmus zur Bestimmung des maximalen Flusses in einem Graphen erläutert.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Übersicht

Die Datenstruktur der Dynamic Trees ermöglicht es, eine Sammlung von knotendisjunkten Bäumen zu erzeugen. Diese Bäume können miteinander verbunden oder aufgetrennt werden. Den einzelnen Kanten in dem Wald werden Gewichte zugeordnet. Diese Gewichte können über Methoden verändert werden. Insgesamt gibt es acht Operationen (im folgenden als Baum-Operationen bezeichnet), die eine Manipulation der Bäume und der Gewichte ermöglichen.

Die Datenstruktur der Dynamic Trees kann so implementiert werden, daß alle Operationen in einer Laufzeit von $O(\log n)$ durchführbar sind. Diese Laufzeit kann bei einfach strukturierten Implementierungen (z.B. durch Verwendung von Splay Trees (siehe Abschnitt 3.3)) als amortisierte Kosten erreicht werden. Bei etwas komplizierteren Implementierungen (z.B. bei globally Biased Binary Trees (siehe Abschnitt 3.2)) kann eine worst-case Laufzeit von $O(\log n)$ erzielt werden.

Durch den Einsatz der Dynamic Trees können für einige Problemstellungen neue schnellere Algorithmen gefunden werden. Im folgenden werden einige Anwendungsbeispiele erwähnt. Die Beispiele werden, bis auf eines, nicht näher beschrieben, sondern nur entsprechende Referenzen gegeben. Eine Kurzbeschreibung der Lösung mittels Dynamic Trees findet sich in der Arbeit von Sleator und Tarjan [ST83]. Als Beispiele sind zu erwähnen.

- Berechnung von *nearest common ancestors* [AHU76]. Bei dieser Problemstellung wird der nächste gemeinsame Nachbar von zwei Knoten gesucht. In der genannten Referenz kann das Problem in $O((m+n)\log n)$ Zeit und $O(n \log n)$ Platz gelöst werden. Mittels Dynamic Trees kann das Problem in $O(\log n)$ Zeit pro Operation und $O(n)$ Platz gelöst werden.

- Lösen von verschiedenen Problemen für Netzwerke, wie z.B. maximale Flüsse, blockierende Flüsse und azyklische Flüsse. Im Abschnitt 4 wird das Problem des maximalen Flusses genauer beschrieben.

- Berechnen von *constrained minimum spanning trees* [GT84]. Bei diesem Problem soll zu einem gegebenen zusammenhängenden ungerichteten Graphen mit gewichteten Kanten, wobei eine Kante entweder die Farbe schwarz oder weiß besitzen kann, ein Spannbaum mit minimalem Gewicht und genau k schwarzen Kanten (k ist ein gewählter ganzzahliger Wert) berechnet werden. Im beschriebenen Algorithmus wird $n-1$ mal zu einem gegebenen Spannbaum eine Kante hinzugefügt und dann ein entstehender Zyklus durch Entfernen der schwersten Kante auf diesem Zyklus beseitigt. Diese Operation kann mit einem Dynamic Tree effizient gelöst werden ($O(\log n)$) und das ergibt eine Gesamtlaufzeit von $O(n \log n)$.

- Simplex Algorithmus für *minimum-cost flows* [Chv83]. Auch in diesem Algorithmus wird ein sogenanntes swapping von Kanten durchgeführt. Durch Dynamic Trees kann diese Operation nunmehr in $O(\log n)$ Zeit, anstelle wie bisher $O(n)$ Zeit, durchgeführt werden.

In den folgenden Abschnitten der Arbeit werden zuerst die Operationen auf den Dynamic Trees beschrieben (Abschnitt 2). Danach wird die Methode zur Darstellung von Bäumen durch Mengen von knotendisjunkten Pfaden erklärt. Diese Methode wird sowohl bei der Implementierung der Dynamic Trees durch Biased Binary Trees als auch durch Splay Trees verwendet. Diese beiden Implementierungsarten werden in Abschnitt 3 erläutert. Abschließend wird die Verwendung der Dynamic Trees in der Push-Relabel-Methode zur Berechnung eines maximalen Flusses in Graphendargestellt (Abschnitt 4).

2 Arbeiten mit Dynamic Trees

Wie bereits in der Einleitung erwähnt, wird bei den Dynamic Trees ein Wald von knotendisjunkten Bäumen verwendet. Den Kanten sind Gewichte zugeordnet. Bereits an dieser Stelle sei darauf hingewiesen, daß die Gewichte auf zweierlei Arten gespeichert werden können. Einerseits direkt in einer Kante, falls für diese explizit eine Implementierung existiert. Andererseits können die Gewichte auch in einem Knoten gespeichert werden. Die Kosten der Kante (v, w), wobei w der Knoten ist, der näher an der Wurzel liegt, werden im Knoten v gespeichert. Somit ist eine eineindeutige Zuordnung der Gewichte zu den Kanten möglich. Ebenso kann man auch den Knoten Kosten zuordnen. Die Kosten eines Knotens n entsprechen dann dem Gewicht der Kante, die den Knoten mit seinem Vaterknoten $p(n)$ verknüpft.

2.1 Manipulations-Operationen der Dynamic Trees

Zur Konstruktion und Destruktion der Bäume stehen zwei Operationen zur Verfügung. Dies sind:

- *link*(v, w, x) — Durch diese Operation wird ein Baum als Teilbaum in einen anderen Baum eingehängt. Die Operation wird auf zwei Knoten v und w ausgeführt. Als Ergebnis wird der Baum mit der Wurzel v als Kind des Knotens w eingehängt. Der neuen Kante (v, w) wird das Gewicht x zugeordnet. Beim Aufruf der Operation wird vorausgesetz, daß sich v und w in verschiedenen Bäumen befinden und v die Wurzel des zu v gehörenden Baumes ist. Diese Operation wird in Abbildung 2.1 veranschaulicht.

- *cut*(v) — Diese Operation bricht einen Baum in zwei Teilbäume auf. Die Operation hat einen Knoten als Parameter. Die Operation wird auf dem Baum ausgeführt, der den Knoten enthält. Das Ergebnis eines Aufrufs ist das Aufteilen des Baumes in den Unterbaum, der den Knoten v als Wurzel hat, und den Teilbaum, der übrig bleibt, wenn dieser Unterbaum entfernt wird. Als Rückgabewert erhält man das Gewicht der Kante die entfernt wurde. Es wird als Vorbedingung angenommen, daß es sich bei v um keinen Wurzelknoten handelt. Die Wirkung der Operation wird in Abbildung 2.2 gezeigt.

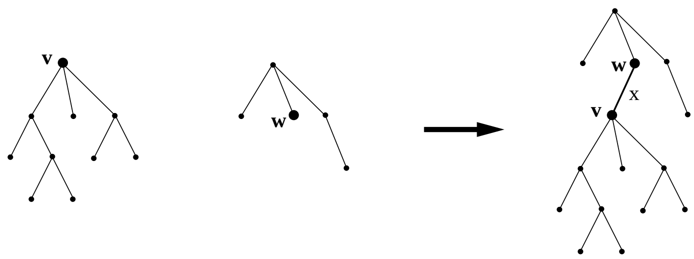


Abbildung 2.1 *Operation link*(v, w, x)

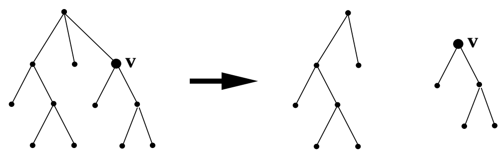


Abbildung 2.2 *Operation cut*(v)

Bei verschiedenen Anwendungen ist es ebenfalls nötig, einen vorhandenen Baum so zu reorganisieren. Die Datenstruktur der Dynamic Trees kennt dafür folgende Operation:

- *evert*(v) — Durch diese Operation wird ein Knoten aus dem Baum zur neuen Wurzel bestimmt. Der bisherige Vaterknoten wird als neues Kind hinzugefügt. Als Parameter erhält die Operation den Knoten, der zur neuen Wurzel werden soll. Die Operation wird auf dem Baum ausgeführt, der den übergebenen Knoten enthält. Als Resultat wird der Baum des Knotens v umstrukturiert. Bildlich kann man sich dies so vorstellen, daß der Baum an der neuen Wurzel aufgehängt wird und sich alle Äste entsprechend einpendeln. Das Ergebnis dieser Operation kann man in Abbildung 2.3 sehen.

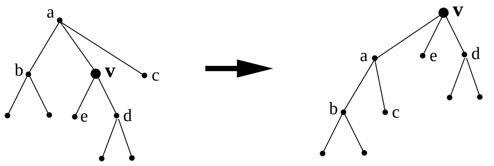


Abbildung 2.3 *Operation evert*(v)

Die Gewichte der Kanten können ebenfalls über eine Operation verändert werden. Diese Operation ist so konzipiert, daß nicht einzelnen Gewichte, sondern immer die Gewichte von eine Menge von Kanten verändert werden. Diese Vorgehensweise mag anfänglich etwas unnatürlich erscheinen, ermöglicht aber bei der Anwendung von Dynamic Trees, z.B. in den in der Einleitung angesprochenen Problemfamilien, eine effiziente Programmierung. Die Manipulationsoperation heißt:

- *update*(v, x) — Diese Operation bewirkt, daß alle Gewichte der Kanten die auf einem Pfad liegen um den gleichen Wert erniedrigt werden. Die Operation erhält zwei Parameter. Erstens den Knoten v, der zur Bestimmung des Pfades dient, und zweitens den Wert x um den die Gewichte verringert werden sollen. Die Operation bewirkt, daß auf dem Pfad, den man erhält, wenn man in dem Baum, der den Knoten v enthält, von v zur Wurzel geht, alle Kantengewichte um den Wert x erhöht werden. Die Wirkung der Operation ist in Abbildung 2.4 verdeutlicht. In der Abbildung sind aus Übersichtlichkeitgründen nicht alle Kantengewichte angegeben.

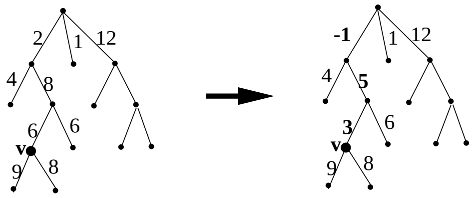


Abbildung 2.4 *Operation update*$(v, -3)$

2.2 Selektor-Operationen der Dynamic Trees

Nachdem vorangehend alle Operationen beschrieben sind, die zur Manipulation der Bäume und der Gewichte dienen, folgt nun eine Liste der Selektoren:

- *parent*(v) — Diese Operation liefert zu dem Knoten v den Vaterknoten im zugehörigen Baum. Falls der Knoten selbst die Wurzel im Baum ist, wird ein Sonderwert (*null*) zurückgeliefert.
- *root*(v) — Diese Operation liefert zu dem übergebenen Knoten v den Wurzelknoten des Baumes in dem sich der Knoten befindet.
- *cost*(v) — Diese Operation liefert das Gewicht der Kante vom Knoten v zu dessen Vaterknoten. Auch in diesem Fall wird gefordert, daß v einen Vaterknoten besitzt.
- *mincost*(v) — Diese Operation arbeitet auf dem gleichen Pfad, den auch die Operation *update*(v, x) verwendet würde. Zu einem übergebenen Knoten v wird der Knoten auf dem Weg zur Wurzel des zugehörigen Baumes zurückgeliefert, so daß die Kante des Rückgabeknotens zu dessen Vater minimales Gewicht unter allen Kanten auf dem Pfad hat. Falls es mehrere solche Knoten gibt, wird der zurückgeliefert, der am nächsten zur Wurzel liegt.

2.3 Zusammenfassung

Diese acht Operationen können zum Arbeiten auf dem Wald der knotendisjunkten Bäume verwendet werden. Bei Analyse von amortisierten Kosten wird im Rest dieser Arbeit immer davon ausgegangen, daß am Anfang einer Reihe von Operationen alle Bäume aus genau einem Knoten bestehen. Dies wird nicht mehr immer ausdrücklich erwähnt, sollte dem Leser jedoch immer bewußt sein.

Bevor nun verschiedene Möglichkeiten der Implementierung betrachtet werden, ist in Abbildung 2.5 noch eine Gruppierung aller Baum-Operationen dargestellt.

	Manipulatoren	Selektoren
Struktur	link cut evert	parent root
Gewicht	update	cost mincost

Abbildung 2.5 *Gruppierung der Baum-Operationen*

3 Datenstrukturen

Nachdem im vorangegangenen Abschnitt die Operationen auf der Datenstruktur der Dynamic Trees besprochen wurden, sollen nun Möglichkeiten der Implementierung dargestellt werden.
Die einfachste Art der Implementierung ist sicherlich die, in jedem Knoten das Gewicht der Kante zu seinem Vaterknoten und eine Referenz auf diesen abzuspeichern. In dieser Implementierung könnten die Operationen *cost*, *parent*, *link* und *cut* in $O(1)$ Zeitkomplexität durchgeführt werden. Die Komplexität der anderen Operationen *update*, *mincost*, *root* und *evert* wäre proportional zu der Länge des Pfades, der vom übergebenen Knoten zur Wurzel des zugehörigen Baumes führt. Im worst-case würde damit die Komplexität $O(n)$ erreicht werden.
Im Teilabschnitt 3.1 wird eine andere Möglichkeit vorgestellt, wie ein Baum repräsentiert werden kann, nämlich durch eine Sammlung von knotendisjunkten Pfaden. Für diese Art der Darstellung werden zwei Implementierungen besprochen, die eine bessere Zeitkomplexität aufweisen.
Die erste, eine sehr komplizierte Version (Teilabschnitt 3.2), ermöglicht es, eine worst-case Komplexität von $O(\log n)$ zu erreichen. Bei dieser Implementierung handelt es sich jedoch um eine sehr aufwendige Vorgehensweise, die in der Praxis daher als wenig praktikabel erscheint.
Die zweite, eine relativ einfache Art der Implementierung (Teilabschnitt 3.3), erreicht die Komplexität von $O(\log n)$ nur bei Betrachtung der amortisierten Kosten. Trotz der etwas ungünstigeren Komplexität besticht die zweite Art der Implementierung durch die einfache Möglichkeit, Dynamic Trees zu verwirklichen.

Bei allen Komplexitätsangaben wird immer davon ausgegangen, daß für die Anzahl n der Knoten, die in den Dynamic Trees gespeichert werden, gilt: $n \geq 2$.

3.1 Speicherung von Bäumen als Menge von Pfaden

In diesem Teilabschnitt wird vorgestellt, wie ein Baum durch eine Sammlung von knotendisjunkten Pfaden dargestellt werden kann. Bei den Pfaden handelt es sich um die Pfade, die sich ergeben, wenn man von einem Knoten im Baum zur Wurzel des Baumes geht. An Knoten, an denen sich Pfade vereinen, also inneren Knoten, wird nur einem Pfad erlaubt, sich fortzusetzen. Die anderen Pfade werden beendet und erhalten an Ihrem letzten Knoten eine Referenz auf den inneren Knoten (Feld *dparent*) und speichern dort ebenfalls das Gewicht der aufgetrennten Kante (Feld *dcost*). Im folgenden werden die aufgetrennten Kanten als gestrichelte und alle anderen als durchgezogene Kanten bezeichnet.
Die Pfade werden als gerichtet angesehen, d.h. sie haben einen Anfang und ein Ende. Ein Pfad soll immer an dem sich am tiefsten im Baum befindenden Knoten des Pfades starten und an dem Knoten des Pfades enden, der am höchsten im Baum liegt. Bildlich bedeutet dies, daß wenn in einer Abbildung eines Baumes die Vaterknoten immer oberhalb der Söhne gezeichnet sind, die Pfade von unten nach oben verlaufen.
Das Schema der Umwandlung eines Baumes in knotendisjunkte Pfade ist in Abbildung 3.1 veranschaulicht. In der Abbildung sind die zu den Pfanden gehörenden Kanten als durchgezogene Linien gezeichnet, und die Referenzen durch gestrichelte Linien dargestellt.

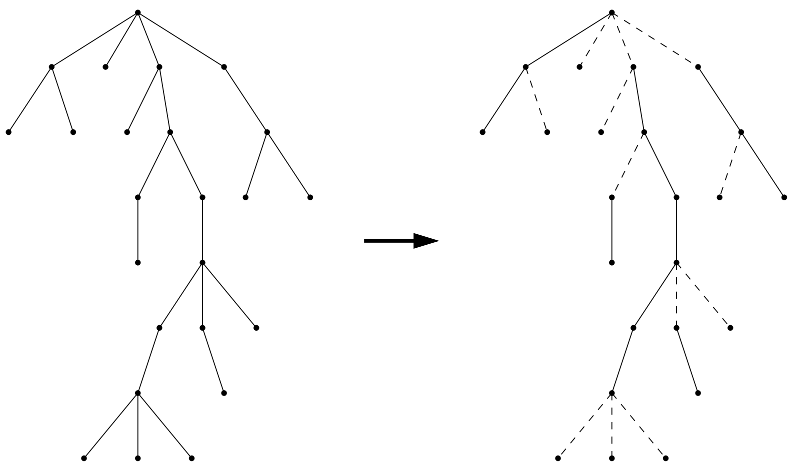


Abbildung 3.1 *Darstellung eines Baumes durch eine Menge von Pfaden*

Mit dieser Definition der Pfade ist es also möglich, einen Baum durch verschiedene Mengen von Pfaden zu repräsentieren. D.h. eine Menge von Pfaden kann so umorganisiert werden, daß sie immer noch den gleichen Baum darstellt. Diese Eigenschaft wird in den später dargestellten Implementierungen ausgenutzt werden.

3.1.1 Operationen auf Pfaden

Bevor nun die einzelnen Implementierungen dargestellt werden, ist es wichtig, sich mit den Operationen auf den Pfaden (Pfad-Operationen) vertraut zu machen. Mit diesen Operationen werden später die Baum-Operationen realisiert. Zur Manipulation der Pfade bzw. der Gewichte der Kanten stehen folgende Operationen zur Verfügung:

- $concatenate(\textbf{path } p, \textbf{path } q, \textbf{real } x)$ — Diese Operation verbindet die beiden Pfade p und q durch eine Kante mit dem Gewicht x, die das Ende des Pfades p mit dem Anfang des Pfades q verbindet.
- $reverse(\textbf{path } p)$ — Mit dieser Operation wird der Pfad umgedreht, d.h. die Knoten in umgedrehter Reihenfolge angeordnet.
- $split(\textbf{node } v)$ — Diese Operation wird verwendet, um den Pfad, der den Knoten v enthält, so aufzubrechen, daß die Kanten vor und nach v entfernt werden. Als Rückgabe erhält man ein Quadrupel $[p, q, x, y]$, bestehend aus den beiden Pfaden, die durch diese Operation entstehen, also dem Pfad p vom Anfang des ursprünglichen Pfades bis zum Knoten vor v und dem Pfad q vom Knoten nach v bis zum letzten Knoten des ursprünglichen Pfades. Ferner werden noch die entsprechenden Gewichte der entfernten Kanten, x und y, zurückgeliefert. Sollte einer der Pfade nicht existieren, da es sich bei v um den Anfangs- bzw. Endknoten eines Pfades handelt, so wird als entsprechender Pfad (p bzw. q) ein spezieller Wert ($null$) und für das Gewicht (x bzw. y) der Wert *undefiniert* zurückgeliefert.
- $pupdate(\textbf{path } q, \textbf{real } x)$ — Diese Operation bildet die Möglichkeit, die Gewichte in einem Pfad zu verändern. Der Wert x wird zu allen Gewichten der Kanten in einem Pfad addiert.

Neben diesen Operationen stehen auch einige Selektor-Funktionen zur Verfügung:

- $path(\textbf{node } v)$ — Durch diese Operation wird der Pfad zurückgeliefert, in dem sich der Knoten v befindet.
- $head(\textbf{path } p)$ — Hierdurch ist es möglich, den Anfangsknoten eines Pfades zu bestimmen.
- $tail(\textbf{path } p)$ — Analog zu $head(p)$ dient diese Operation zur Bestimmung des Endknotens eines Pfades.
- $before(\textbf{node } v)$ — Mittels dieser Operation kann der Knoten in einem Pfad bestimmt werden, der direkt vor der Knoten v liegt. Falls es sich bei v bereits um den Anfangsknoten handelt, wird wieder der Spezialwert ($null$) zurückgeliefert.
- $after(\textbf{node } v)$ — Analog zu $before(v)$ kann durch diese Operation der Knoten, der im Pfad direkt hinter dem Knoten v liegt, bestimmt werden. Auch in diesem Fall wird gegebenenfalls der Spezialwert zurückgeliefert.
- $pcost(\textbf{node } v)$ — Durch diese Operation kann das Gewicht der Kante vom Knoten v zum Knoten $after(v)$ ermittelt werden. Es wird vorausgesetzt, daß es sich bei v nicht um das Ende des Pfades $path(v)$ handelt.
- $pmincost(\textbf{path } p)$ — Diese Operation ermöglicht das Auffinden desjenigen Knotens v, der am nächsten zum Ende des Pfades $tail(p)$ liegt und unter den Knoten des Pfades minimalen Wert $pcost$ besitzt. (d.h. $pcost(v) = \min\{pcost(w) | w \textit{ liegt auf } p\}$) Vorbedingung zu dieser Operation ist, daß der Pfad aus mehr als einem Knoten besteht, d.h. mindestens eine Kante besitzt.

Bei der Realisierung von Baum-Operationen durch Pfad-Operationen werden verschiedene Arbeitsschritte häufig wiederholt. Aus diesem Grund werden hierfür zwei neue Operationen eingeführt die auf Pfad-Operationen zurückgreifen und dazu dienen, gestrichelte in durchgezogene Kanten zu verwandeln (und umgekehrt). Hierbei darf die Bedingung, daß maximal eine durchgezogene Kante in einem Knoten ankommen darf, nicht verletzt werden. Die Operationen arbeiten wie folgt:

- *splice*(**path** p) — Diese Operation dient dazu, den Pfad p zu verlängern, indem die gestrichelte Kante am Ende des Pfades $tail(p)$ zu einer durchgezogenen Kante umgeformt wird. Dies hat zur Folge, daß eine durchgezogene Kante, die in dem nun erreichbaren Knoten angekommen ist, gestrichelt werden muß. Diese Operation setzt voraus, daß es sich bei $tail(p)$ nicht um die Wurzel des Baumes handelt ($tail(p) \neq root(tail(p))$). Es wird der erweiterte Pfad zurückgeliefert.
- *expose*(**node** v) — Durch diese Operation wird in einem Baum der Weg von dem Knoten v bis zur Wurzel $root(v)$ hervorgehoben, indem alle Kanten auf diesem Weg durchgezogen werden, d.h. es wird ein neuer Pfad gebildet, der in v startet und in $root(v)$ endet. Alle zu diesem Pfad inzidenten Kanten werden gestrichelt. Die Operation liefert als Rückgabewert den neu gebildeten Pfad.

Diese beiden Operationen werden in den Abbildungen 3.2 und 3.3 verdeutlicht.

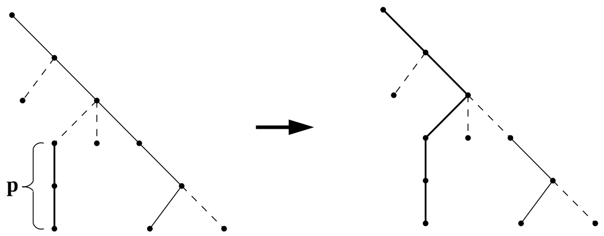


Abbildung 3.2 *Operation splice*(p)

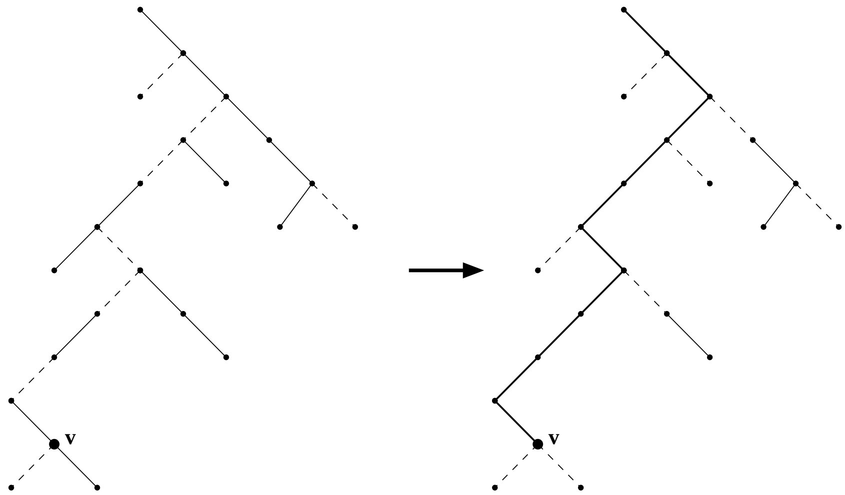


Abbildung 3.3 *Operation expose*(v)

Wie bei den Bäumen werden auch dieses Mal alle Operationen auf den Pfaden und den zugehörigen Bäumen in einer Tabelle zusammengefaßt (Tabelle 3.4).

	Manipulatoren	Selektoren
Pfad	concatenate split reverse	head tail before after path
Baum	expose splice	
Gewicht	pupdate	pcost pmincost

Abbildung 3.4 *Gruppierung der Pfad-Operationen*

3.1.2 Realisierung der Baum-Operationen

In diesem Unterabschnitt, soll nun Beschrieben werden, wie die einzelnen Baum-Operationen durch die Pfad-Operationen realisiert werden können. Hierfür sollen zunächst die Operationen *splice* und *expose* durch die einzelnen Pfadoperationen realisiert werden. Der zugehörige Code findet sich in den Abbildungen 3.5 und 3.6.

```
FUNCTION splice (Path p)
   Node v; Path q,r; real x,y;
   v := dparent(tail(p));
   [q,r,x,y] := split(v);
   IF(q != NULL)
     dparent(tail(q)) := v;
     dcost(tail(q))   := x;
   FI
   p := concatenate(p,path(v),dcost(tail(p)));
   IF(r != NULL)
     p := concatenate(p,r,y);
   FI
   RETURN p;
END splice
```

Abbildung 3.5 *Implementierung der splice-Operation*

```
FUNCTION expose (Node v)
   Path p,q,r; real x,y;
   [q,r,x,y] := split(v);
   IF(q != NULL)
     dparent(tail(q)) := v;
     dcost(tail(q))   := x;
   FI
   IF( r = NULL)
     p := path (v);
   ELSE
     p := concatenate(path(v),r,y);
   IF
   WHILE ( dparent(tail(p)) != NULL )  DO
     p := splice(p);
   OD
   RETURN p;
END expose
```

Abbildung 3.6 *Implementierung der expose-Operation*

Mit diesen Methoden ist es nun möglich die acht Baum-Operationen umzusetzen. In Abbildung 3.7 ist eine Auflistung dieser Implementierung gegeben.

```
FUNCTION parent (Node v)
  IF v = tail(path(v))
    RETURN dparent(v);
  ELSE
    RETURN after(v);
  FI
END parent

FUNCTION root (Node v)
    RETURN tail(expose(v));
END root
```

```
FUNCTION cost (Node v)
  IF v = tail(path(v))
    RETURN dcost(v);
  ELSE
    RETURN pcost(v);
  FI
END cost

FUNCTION mincost (Node v)
  RETURN pmincost(expose(v));
END mincost

PROCEDURE update (Node v, real x)
  pupdate(expose(v),x);
END update

PROCEDURE link (Node v, Node w , real x)
  concatenate( path(v), expose(w), x);
END link

FUNCTION cut (Node v)
 Path, p,q; real x,y;
 expose(v);
 [p,q,x,y] := split(v);
 dparent(v) := null;
 RETURN y;
END cut

PROCEDURE evert (Node v)
  reverse(expose(v));
  dparent(v) := null;
END evert
```

Abbildung 3.7 *Implementierung der Baum-Operationen*

3.1.3 Komplexitätsanalysen

Bereits an dieser Stelle soll eine kurze Komplexitätsanalyse der Baum-Operationen gemacht werden. Betrachtet man die einzelnen Funktionen, so trifft man auf den Aufruf der *expose*-Operation und der darin enthaltenen Schleife. In der Originalarbeit von Sleator und Tarjan [ST83] wird der Beweis zu folgendem Theorem geführt.

Theorem 3.1 *Sei n die Anzahl der vorhandenen Knoten, von denen jeder die Wurzel eines Baumes der Größe* 1 *ist. Bei einer Hintereinanderausführung von m Baum-Operationen auf diesem Wald, werden maximal $m(3\lfloor\log_2 n\rfloor + 1)$* splice-*Operationen ausgeführt.*

Aus diesem Theorem (3.1) und der angegebenen Implementierung leitet sich sofort das folgende Theorem ab:

Theorem 3.2 *In einer Folge von m Operationen auf Dynamic Trees werden $O(m \log n)$ Pfad-Operationen ausgeführt. (Hierbei sind die* splice- *und* expose-*Operationen bereits in die einfachen Pfad-Operationen aufgebrochen).*

Würde man nun die Pfade durch effiziente Baumdatenstrukturen wie z.B. höhen- oder gewichts-balanzierte Bäumen realisieren, könnte man für die Pfad-Operationen eine Zeitkomplexität von $O(\log n)$ erreichen, was zusammen mit dem Theorem 3.2 in einem weiteren Theorem formuliert werden kann.

Theorem 3.3 *Bei einer Darstellung von durchgezogenen Pfaden mittels balanzierten Binärbäumen kann eine Folge von m Operationen auf Dynamic Trees mit einer Zeitkomplexität von $O(m(\log n)^2)$ durchgeführt werden.*

Durch weitere Überlegungen und Modifizierungen, die in den folgenden Teilabschnitten behandelt werden, kann nun diese obere Schranke weiter gedrückt werden.

3.2 Biased Binary Trees

Die erste Implementierung, die vorgestellt wird, wurde 1982 von Sleator und Tarjan in dem Paper "A Data Structure for Dynamic Trees" [ST83] vorgestellt. Grundlegend für diese Implementierung sind die von W. Bent untersuchten *Biased Binary Trees*, die in dem Paper "Biased Search Trees" von Bent, Sleator und Tarjan [BST85] besprochen wurden. Die Datenstruktur ermöglicht es, in einem Baum mit positiv gewichteten Knoten auf einen Knoten v mit $O(\log(W/wt(v)))$ zuzugreifen ($wt(v)$ ist das Gewicht von v; W ist die Summe der Gewichte aller Knoten des Baumes). Mit dieser Implementierung ist es möglich, eine $O(\log n)$ worst-case Laufzeit pro Baum-Operation zu erreichen.

In diesem Abschnitt soll nur eine Übersicht über die Implementierung gegeben werden. Für eine exakte Beschreibung der Vorgehensweise sei auf die Originalarbeiten verwiesen. Es wird zuerst eine einfachere Art der Implementierung beschrieben, bei der eine amortisierte logarithmische Laufzeit erreicht werden kann. Später wird dann eine Erweiterung dargestellt, die auch für jede einzelne Operation logarithmische Komplexität aufweisen kann.

Aufbauend auf die Darstellung von Bäumen durch eine Sammlung von knotendisjunkten Pfaden wird nun eine effiziente Datenstruktur zur Verwaltung der Pfade erläutert. Die Aufteilung des Baumes in einzelne Pfade wird, wie bereits weiter oben beschrieben, durch die einzelnen Operationen bestimmt. D.h. die Tatsache, ob eine Kante durchgezogen oder gestrichelt ist, ist davon abhängig, wann welche Operation ausgeführt wurde. Die einzelnen Operationen können also nach Belieben die Struktur verändern, solange sie die Bedingung über den Baumaufbau nicht verletzten. Aus diesem Grund wird die Implementierung in der Originalarbeit auch *naive partitioning* genannt.
Die Grundidee der Implementierung besteht darin, die Pfade durch vollständige Binärbäume darzustellen. Die Knoten auf dem Pfad werden durch die Blätter und die Kanten durch die internen Knoten der Binärbäume dargestellt. Diese Vorgehensweise ist in Abbildung 3.8 verdeutlicht.

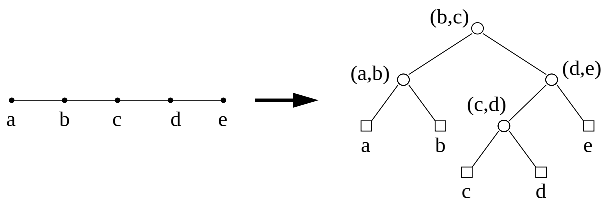


Abbildung 3.8 *Umwandlung eines Pfades in einen Binärbaum*

Um eine leichtere Handhabe der Pfade zu ermöglichen, wird in jedem externen Knoten ein Verweis auf den zugehörigen inneren Knoten gespeichert. Die inneren Knoten erhalten neben einer Referenz auf den Vaterknoten *bparent* noch fünf weitere Referenzen. Jeweils eine auf den linken bzw. rechten Sohn (*bleft* bzw. *bright*), und je eine auf das Blatt (Knoten), das am Anfang bzw. Ende des Pfades steht, der sich ergibt, wenn man den Unterbaum, der durch den inneren Knoten aufgespannt wird, betrachtet (*bhead* bzw. *btail*). Desweiteren hat jeder interne Knoten ein Feld *reversed* (Boole'scher Wert). Dieses Feld kann verwendet werden, um die Pfad-Operation *reverse* einfacher zu implementieren. Wenn das Feld *reversed* den Wert *true* enthält, sind die Bedeutung von "links" und "rechts" zu vertauschen (d.h $bhead(v)$ zeigt auf das Ende, etc.).
Um nun die Pfad-Operationen, die in Teilabschnitt 3.1.1 beschrieben sind, zu realisieren, müssen noch die Gewichte der Kanten gespeichert werden. Um eine effiziente Implementierung zu erreichen, werden hierfür die Gewichte nicht direkt in den Kanten gespeichert. Statt dessen werden in jedem inneren Knoten zwei Werte *netcost* und *netmin* gespeichert. Wenn jeder Kante v das Gewicht $grosscost(v)$ zugeordnet ist, und $grossmin(v)$ das Minimum aller Nachfolger von v (v eingeschlossen) ist, dann definieren sich *netcost* und *netmin* wie folgt:

$$\begin{aligned} netcost(v) &= grosscost(v) - grossmin(v) & \\ netmin(v) &= grossmin(v) & \text{, falls } v \text{ Wurzelknoten} \\ &= grossmin(v) - grossmin(bparent(v)) & \text{, sonst} \end{aligned}$$

So definiert ist der Wert von $netcost(v)$ für alle internen Knoten nicht negativ. Der Wert von $netmin(v)$ kann nur beim Wurzelknoten negative Werte annehmen. Die Werte von $grossmin(v)$ lassen sich durch Summation der Werte $netmin(w)$ aller innerer Knoten auf dem Pfad von v zur Wurzel des Binärbaums berechnen. Der Wert für $grosscost(v)$ ergibt sich aus der Summe von $netcost(v)$ und $grossmin(v)$. Um die

beiden Werte zu berechnen, muß maximal Zeit proportional zur Tiefe des jeweiligen Knotens aufgewendet werden.
In Abbildung 3.9 sind sowohl die Struktur des inneren Knotens sowie ein Beispiel zur Darstellung eines gewichteten Pfades gegeben.

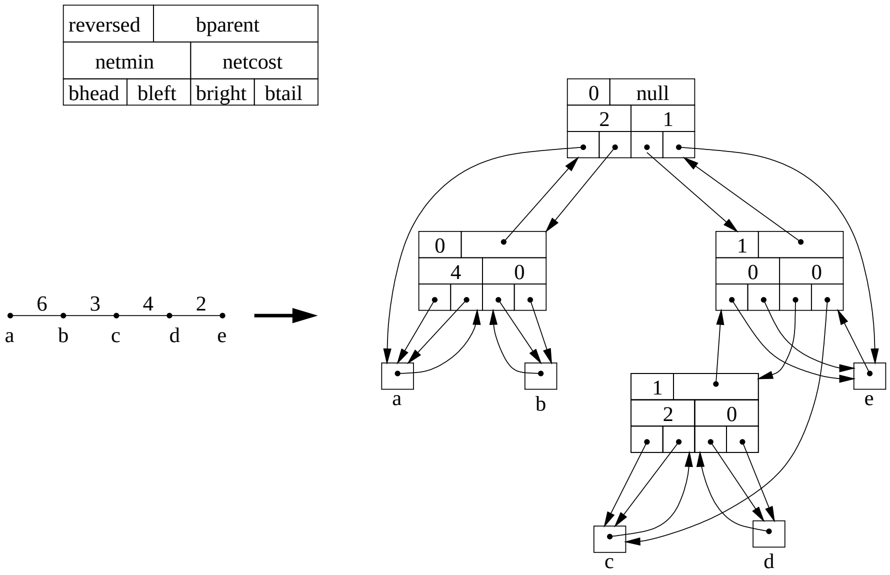


Abbildung 3.9 *Struktur und Beispiel für die Repräsentation eines Pfades durch einen Binärbaum*

Die in dem Abschnitt 3.1.3 angegebene obere Schranke von $O(m(\log n)^2)$ für balanzierte Bäume soll nun um den Faktor $O(\log n)$ gedrückt werden. An dieser Stelle kommen, wie bereits erwähnt, *locally biased binary trees* [BST85] zur Anwendung. Um diese Datenstruktur verwenden zu können, muß jedem Knoten in dem Binärbaum ein positives Gewicht zugeordnet werden. Um die Gewichtsfunktion $wt(v)$ festzulegen, wird zuerst für jeden Knoten v eine Größe $size(v)$ eingeführt. Der Wert von $size(v)$ ist die Anzahl der Knoten in dem durch v aufgespannten Unterbaum. Die Funktion $wt(v)$ definiert sich nun wie folgt:

$$\begin{aligned} wt(v) &= \text{size(v)} && \text{, falls keine durchgezogene Kante in } v \text{ ankommt} \\ &= \text{size(v) - size(w)} && \text{, falls die durchgezogene Kante } (w,v) \text{ in } v \text{ ankommt} \end{aligned}$$

Mittels einiger Komplexitätsbetrachtungen über die Implementierung der Baum-Operationen durch Pfad-Operationen unter Zuhilfenahme von Eigenschaften der *locally biased binary trees* und der Definition der Gewichtsfunktion $wt(v)$ kann nun gezeigt werden, daß folgendes Theorem gilt:

Theorem 3.4 *Bei der Verwendung von naive partitioning und einer Darstellung von durchgezogenen Pfaden durch* locally biased trees *kann eine Folge von m Operationen auf Dynamic Trees in Zeit $O(m \log n)$ durchgeführt werden.*

In anderen Worten bedeutet dies, daß bei amortisierter Kostenrechnung jede Baum-Operation in $O(\log n)$ durchgeführt werden kann. Um nun auch eine worst-case Komplexität von $O(\log n)$ zu erhalten, muß die vorgestellte Implementierung noch weiter modifiziert werden.
In der bisherigen Implementierung hing die Struktur der erzeugten Bäume von der Reihenfolge der durchgeführten Baum-Operationen ab. In der neuen Implementierung wird nun die Aufteilung in Pfade nicht mehr durch die Operationen, sondern durch den Aufbau des Dynamic Trees Baumes bestimmt. Ausschlaggebend ist nun die Größe der einzelnen Teilbäume der Dynamic Trees, weshalb diese Aufteilung auch *partitioning by size* genannt wird.
In der erweiterten Implementierung ist eine Kante (v, w) in den Dynamic Trees genau dann durchgezogen, wenn für die Größen der Teilbäume von v und w gilt: $2size(v) > size(w)$. Bei einer Implementierung der Baum-Operationen durch die *expose*-Operation kann es vorkommen, daß eine durchgezogen Kante zu einer gestrichelten wird, und umgekehrt. Daher muß nach jeder Baum-Operation überprüft werden, ob ein solcher Fall aufgetreten ist, und ggf. die Folgen beseitigt werden.
Desweiteren wird an Stelle der *locally biased binary trees* die Datenstruktur der *globally biased binary trees* [BST85] verwendet. Diese Datenstruktur kann noch bessere worst-case Komplexitäten für die Operationen *split* und *concatenate* aufweisen.

Mit diesen Änderungen kann jetzt auch eine Obere Schranke für die *expose*-Operation angegeben werden, die im folgendem Theorem formuliert ist.

Theorem 3.5 *Bei der Verwendung von* partitioning by size *kann die* expose-*Operation in* $O(\log n)$ *Zeit durchgeführt werden*

Mittels dieses Theorems (3.5) und weiteren Betrachtungen über die noch weiter modifizierten Pfad-Operationen [ST83] kann folgendes abschließendes Theorem bewiesen werden:

Theorem 3.6 *Bei der Verwendung von* partitioning by size *und einer Darstellung von durchgezogenen Pfaden durch* globally biased trees *kann jede Operation auf den Dynamic Trees in einer Zeit von* $O(\log n)$ *durchgeführt werden.*

3.3 Splay Trees

In diesem Abschnitt soll nun eine zweite Implementierungsmöglichkeit der Dynamic Trees vorgestellt werden. Sie basiert auf dem Prinzip von selbstorganisierenden Binären Suchbäumen, sogenannten Splay Trees [ST85]. Im folgenden soll nun zuerst eine kurze Einführung in das Prinzip der Splay Trees gegeben werden. Dann wird dieses Wissen angewendet, um die Dynamic Trees zum implementieren.

3.3.1 Grundlagen der Splay Trees

Ähnlich wie bei der move-to-front Taktik bei linearen Listen, die bis auf einen konstanten Faktor eine optimale Taktik bei Operationen auf linearen Listen bildet, werden auch bei Splay Trees referenzierte Elemente im Baum verschoben.
Bei den Splay Trees handelt es sich um geordnete Binärbäume, d.h. jeder Knoten hat maximal zwei Kinder, wobei jedes Kind im linken Unterbaum ein kleineres und jedes Kind im rechten Unterbaum ein größeres Gewicht besitzt. Anstelle der Sortiertheit der Gewichte kann man auch von einer symmetrischen Ordnung sprechen. Diese Ordnung muß bei allen Operationen aufrecht erhalten bleiben. Diese Tatsache ist besonders bei der Implementierung der Dynamic Trees durch Splay Trees, die im folgenden Teilabschnitt besprochen wird, wichtig.
Grundlegend bei der Realisierung der Splay Trees ist eine Operation, die *splay* genannt wird. Diese Operation wird auf einen Knoten v im Baum ausgeführt und bewirkt, daß dieser Knoten entlang dem Pfad von v bis zur Wurzel bis an die Wurzel verschoben wird. Die Verschiebung wird mittels Vertauschungen durchgeführt und verändert somit die Struktur, jedoch nicht die Sortiertheit des Baumes.
Die einfachste Möglichkeit besteht darin, jeweils den Knoten mit seinem Vaterknoten zu vertauschen (unter geeigneter Vertauschung der Kinder, zur Beibehaltung der Ordnung), und dieses Verfahren solange fortzusetzen, bis der Knoten an der Wurzel angelangt ist. Die benötigte Zeit für ein solches Vorgehen wäre proportional zur Tiefe und würde im schlimmsten Fall proportional zur Anzahl der Knoten im Baum sein. Auch mit amortisierter Rechnung läßt sich dieser Malus nicht beheben.
Durch Einführung einer etwas komplizierteren Vertauschungsregel kann jedoch eine Verbesserung erreicht werden. Es wird nun nicht nur der Vaterknoten, sondern, falls vorhanden, auch der Großvater betrachtet und dann eine Verschiebung des jeweiligen Knotens um zwei Positionen in Richtung Wurzel vorgenommen. Durch diese Vorgehensweise wird nicht nur der Knoten zur Wurzel verschoben, sondern ebenfalls die Baumstruktur in der Art modifiziert, daß entartete Teilbäume ausgeglichener werden. Dies bewirkt, daß bei mehreren solchen Operationen die Pfadlängen ausgeglichener werden und sich damit die Komplexität verbessert. Anschaulich kann man sich dieses Prinzip so vorstellen, daß ähnlich wie bei höhen-balanzierten Bäumen versucht wird, eine Balanzierung zu erreichen. Dieser Schritt wird jedoch nicht bei jeder Modifizierung der Baumstruktur vorgenommen, sondern nur dann, wenn ein Zugriff auf Elemente stattfindet. D.h. die Balanzierung findet verzögert und nur teilweise statt. Dennoch kann durch diesen Schritt, wie später zu sehen ist, eine amortisierte Komplexität von $O(\log n)$ für alle Operationen erreicht werden.
Das Auftreten von amortisierten Komplexitäten bedeutet zwar, daß einzelne Operationen bedeutend länger benötigen können, wirkt sich aber in den Fällen, in denen nur die Zeit für eine Folge von Operationen ausschlaggebend ist, nicht sonderlich aus. Mehr noch, in einzelnen Fällen kann es vorkommen, daß eine Folge von Operationen sogar günstiger ist, obwohl bei einer anderen Implementierung bessere worst-case Komplexitäten erreicht werden. Dies ist dann häufig durch den verminderten Organisationsaufwand zu begründen. Nun aber zu der bereits erwähnten Vertauschungsregel. Bei einer Operation *splay*(v) können drei Fälle unterschieden werden:

1. (*zig*) — Der Vaterknoten von v ist bereits die Wurzel. Dann findet eine einfache Rotation statt.
2. (*zig-zig*) — Bei v und bei dem Vaterknoten von v handelt es sich jeweils um linke oder rechte Kinder. In diesem Fall wird eine entsprechende Rotationsregel angewandt, und dann rekursiv an der neuen Position von v mit den Rotationen fortgesetzt.
3. (*zig-zag*) — Der Knoten v ist ein linkes Kind und der Vaterknoten von v ist ein rechts Kind oder umgekehrt. Auch hier wird nach einer entsprechenden Rotation rekursiv an der neuen Position von v fortgefahren.

Die jeweiligen Rotationen in den einzelnen Fällen sind der Abbildung 3.10 zu entnehmen. Die Rotation findet dabei für den Knoten x statt ($v = x$).

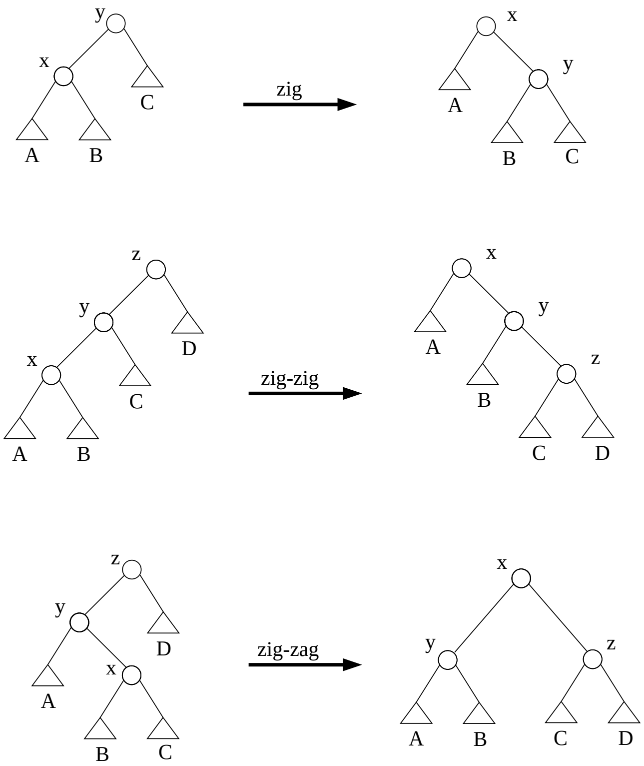


Abbildung 3.10 *Rotationen in Splay Trees*

Werden nun für die einzelnen Knoten wieder beliebige aber feste positive Gewichte $wt(v)$ vergeben, so kann man zwei zusätzliche Werte festlegen. Einerseits die Größe $s(v)$ als die Summe der Gewichte aller Elemente in dem durch v aufgespannten Unterbaum, und den Rang $r(v)$ als $\log_2(s(v))$. Das Potential eines Baumes wird nun durch die Summe der Ränge aller zu dem Baum gehörenden Knoten definiert. Mit diesen Festlegungen kann man nun folgendes Theorem zeigen.

Theorem 3.7 (Access Lemma) *In einem Baum mit der Wurzel t beträgt die amortisierte Zeit einer* splay*-Operation um den Knoten v maximal*

$$3(r(t) - r(v)) + 1 = 3\log\left(\frac{s(t)}{s(v)}\right) + 1 = O\left(\log\left(\frac{s(t)}{s(v)}\right)\right)$$

Der Beweis wird über die drei einzelnen Fälle der Splay-Operation gezeigt, es zeigt sich dabei, daß ein *zig-zig* -Schritt der teuerste Teil der *splay*-Operation ist. Für eine genaue Ausführung sei auf die Originalarbeit verwiesen.
Es können noch weitere Theoreme für eine Folge von m Zugriffs-Operationen gegeben werden. Auch diese werden hier nur aufgelistet, der Beweis jedoch nicht geführt, hierfür sei wieder auf die Originalarbeit verwiesen [ST85].

Theorem 3.8 (Balance Theorem) *Die Gesamt-Zugriffszeit ist höchstens*

$$O((m+n)\log(n+m))$$

Dieses Theorem bedeutet, daß bei einer genügend langen Folge von Zugriffen ($m > n$) ein Griff genauso effizient implementiert werden kann wie ein Zugriff auf einen balanzierten Baum. In beiden Fällen ist die Zugriffszeit logarithmisch.

Theorem 3.9 (Static Optimal Theorem) *Wenn auf jedes Element eines Splay Trees mindestens einmal zugegriffen wird, und $q(i)$ die Zugriffshäufigkeit auf das Element i ist, so beträgt die Gesamt-Zugriffszeit maximal*

$$O\left(m + \sum_{i=1}^{n} q(i)\log(\frac{m}{q(i)})\right)$$

Ein Standard Theorem aus der Informationstheorie besagt, daß die Gesamtzugriffszeit von m Zugriffen auf einen fixen Baum $\Omega(m + \sum_{i=1}^{n} q(i)\log(m/q(i)))$ ist [Abr83]. Hieraus und aus dem obigem Theorem (3.9) folgt also, daß ein Splay Tree für eine feste Zugriffsfolge genauso effizient implementiert werden kann wie ein fixer Suchbaum. Dies beinhaltet auch den optimalen Suchbaum.
Als Betrachtungen der *splay*-Operation soll dies genügen. Es soll nun eine Liste der Operationen gegeben werden, mit denen auf einen Splay Tree zugegriffen werden kann.

- *access*(i, t) — Diese Operation liefert eine Referenz auf das Element i zurück, wenn es sich im Baum t befindet, anderenfalls wird die *null*-Referenz zurückgeliefert.
- *insert*(i, t) — Mit dieser Operation kann ein Element i in den Baum t eingefügt werden, wenn es sich noch nicht in dem Baum befindet.
- *delete*(i, t) — Entfernen des Elements i aus dem Baum t. Dabei wird vorausgesetzt, daß sich i in dem Baum befindet.
- *join*(t_1, t_2) — Es werden die beiden Bäume t_1 und t_2 zusammengefügt. Dabei wird vorausgesetzt, daß alle Elemente in dem Baum t_1 kleiner als die im Baum t_2 sind. Die beiden Bäume werden durch die Operation zerstört und der resultierende Baum zurückgeliefert.
- *split*(i, t) — Mit dieser Operation wird der Baum t in zwei Bäume zerlegt, dabei wird der Baum t zerstört. Es werden die Bäume t_1 und t_2 gebildet, wobei t_1 alle die Elemente x aus t enthält, für die gilt $x \leq i$, und t_2 die restlichen.

Die Operationen werden wie folgt realisiert. Um ein *access*(i, t) durchzuführen, wird im Baum t von der Wurzel beginnend nach dem Element i gesucht (Hierbei wird in den Baum abgestiegen, bis das Element oder eine *null*-Referenz gefunden wird). Abschließend wird auf das Element i eine *splay*-Operation angewendet und dann eine Referenz auf i zurückgeliefert. Befindet sich das Element i nicht in dem Baum, wird auf das letzte Element bevor die *null*-Referenz getroffen wird eine *splay*-Operation ausgeführt.
Bei der *split*-Operation wird zuerst ein *access* auf das Element i ausgeführt. Wird i gefunden, so befindet sich i oder falls i nicht im Baum ist, das größte Element, das kleiner als i ist, an der Wurzel. Es muß also nur noch der rechte Unterbaum von der neuen Wurzel entfernt werden (dies ist der neue Baum t_2), und zurück bleibt der Baum t_1 (siehe Abbildung 3.11).
Die *join*-Operation wird durch einen Splay auf das größte Element in t_1 realisiert. Dadurch wandert das Element an die Wurzel von t_1 und besitzt wegen der Sortiertheit von t_1 keinen rechten Nachbarn. Es kann also der Baum t_2 als neuer rechter nachbar eingehängt werden, da ja alle Elemente in t_2 größer als die neue Wurzel sind (siehe Abbildung 3.11).
Die Operationen *insert* und *delete* werden meist durch die Operationen *join* und *split* realisiert. Um *insert* zu realisieren, wird zuerst ein *split* ausgeführt und die beiden neu entstehenden Bäume als linke und rechte Kinder von dem Knoten i verwendet. Bei einem *delete* wird zuerst ein *access* auf i ausgeführt, i entfernt und dann eine *join* Operation auf die beiden ehemaligen Kinder von i ausgeführt. Diese Vorgehensweisen sind in Abbildung 3.12 dargestellt.

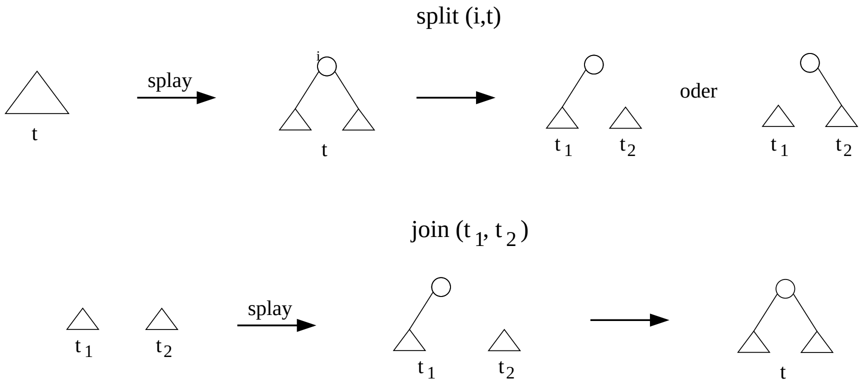


Abbildung 3.11 *Splay Tree Operationen split*(i,t) *und join*(t_1,t_2)

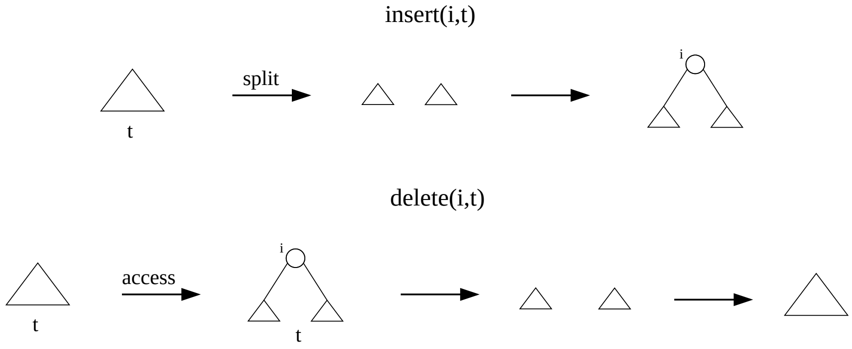


Abbildung 3.12 *Splay Tree Operationen insert*(i,t) *und delete*(t_1,t_2)

Nachdem nun die einzelnen Operationen auf die Splay Trees bekannt sind, soll nun die Komplexität für die Operationen betrachtet werden. In der Abbildung 3.13 sind die amortisierten Kosten für die einzelnen Splay Tree Operationen aufgeführt. Der Wert W stehe dabei für die Summe der Gewichte aller Knoten in den Bäumen, die an der Operation beteiligt sind. Die Werte $i-$ bzw. $i+$ bedeuten der nächst kleinere bzw. größere Wert als i, der in dem jeweiligen Baum vorkommt. Zur Bestimmung der amortisierten Kosten wurde dabei das Theorem 3.7 über die amortisieren Kosten einer *splay*-Operation und die Abschätzung, daß $W \geq w(x)$ (für alle Knoten aus dem Bäumen, aus deren Knoten W berechnet wurde) herangezogen.

$$
\begin{array}{lll}
access(i,t) & 3\log\left(\frac{W}{w(i)}\right)+1 & \text{,falls } i \text{ in } t \text{ liegt} \\
 & 3\log\left(\frac{W}{min\{w(i-),w(i+)\}}\right)+1 & \text{,falls } i \text{ nicht in } t \text{ liegt} \\
 & & \\
join(t_1,t_2) & 3\log\left(\frac{W}{w(i)}\right)+O(1) & \text{,mit } i \text{ ist das max. Element in } t_1 \\
 & & \\
split(i,t) & 3\log\left(\frac{W}{w(i)}\right)+O(1) & \text{,falls } i \text{ in } t \text{ liegt} \\
 & 3\log\left(\frac{W}{min\{w(i-),w(i+)\}}\right)+O(1) & \text{,falls } i \text{ nicht in } t \text{ liegt} \\
 & & \\
insert(i,t) & 3\log\left(\frac{W-w(i)}{min\{w(i-),w(i+)\}}\right)+\log\left(\frac{W}{w(i)}\right)+O(1) & \\
 & & \\
delete(i,t) & 3\log\left(\frac{W}{w(i)}\right)+3\log\left(\frac{W-w(i)}{w(i-)}\right)+O(1) &
\end{array}
$$

Abbildung 3.13 *Komplexität der Splay Tree Operationen*

Wenn für alle Gewichte der Knoten der Wert auf $\frac{1}{n}$ gesetzt wird, so ergibt sich für W der Wert 1 und damit können die Argumente aller Logarithmen durch n abgeschätzt werden. Somit ergibt sich für alle Operationen eine logarithmische Zeitkomplexität. Abschließend kann dies in folgendem Theorem formuliert werden:

Theorem 3.10 (Balance Theorem mit Ergänzung) *Bei einer Folge von m beliebigen Operationen auf einen Wald von anfänglich leeren Splay Trees wird insgesamt $O(m + \sum_{j=1}^{m} \log n_j)$ Zeit benötigt, wobei n_j die Anzahl der Elemente in den Bäumen sind, die an der $j$ten Operation beteiligt sind.*

3.3.2 Splay Trees in Dynamic Trees

In diesem Teilabschnitt soll nun das Prinzip der Splay Trees verwendet werden, um Dynamic Trees zu implementieren. Als erstes stellt sich die Frage, wie ein Dynamic Tree durch einen Splay Tree repräsentiert werden kann. Hierfür wird für jeden Splay Tree einer der Bäume betrachtet, in denen bereits die knotendisjunkten Pfade eingetragen sind (vgl. Teilabschnitt 3.1). Jeder Pfad wird durch einen Splay Tree dargestellt. Somit hat jeder Knoten einen linken und einen rechten Sohn, d.h. man hat wieder Binärbäume. Dabei sind alle Pfade so realisiert, daß die Reihenfolge der Knoten sich aus der symmetrischen Ordnung der Binärbäume ergibt. Es entspricht also jeder Splay Tree einem durchgezogenen Pfad. Alle gestrichelten Kanten werden wie folgt umgesetzt. Sei (v, w) eine gestrichelte Kante im Dynamic Tree, dann erhält der Wurzelknoten des Splay Trees in dem sich v befindet eine Referenz auf w. v wird damit zu einem sogenannten mittleren Kind von w (siehe Abbildung 3.14, mittlere Kinder sind durch gestrichelte Kanten eingezeichnet).

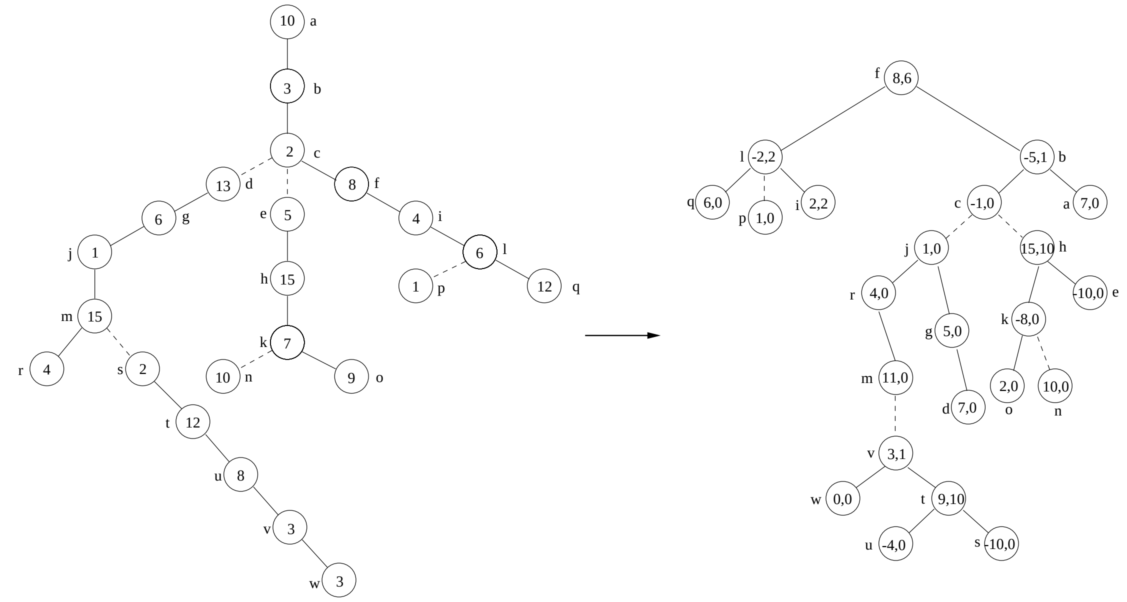


Abbildung 3.14 *Repräsentation eines Dynamic Trees durch einen Splay Tree*

Bei allen Rotationen werden die mittleren Kinder von einem Knoten nicht berücksichtigt, d.h. die Referenzen beiben bestehen und die mittleren Kinder werden einfach mitgenommen. Eine einfache Rotation ist in Abbildung 3.15 dargestellt. Die Existenz der mittleren Kinder wirkt sich also nicht auf die Splay Tree Operationen aus, weshalb die Überlegungen, die bisher zu den Splay Trees gemacht wurden, weiterhin ihre Gültigkeit behalten. Um Mißverständnissen vorzubeugen wird im folgenden von erweiterten Splay Trees gesprochen, falls Splay Trees mit mittleren Kindern gemeint sind.

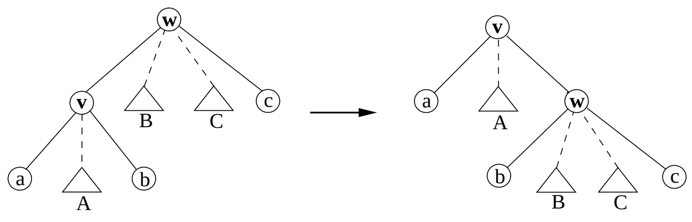


Abbildung 3.15 *Rotation in einem erweiterten Splay Tree*

Die Gewichte der Dynamic Trees werden in den Knoten $cost(v)$ abgespeichert. D.h. im Knoten v wird das Gewicht der Kante von v zu seinem Vaterknoten $p(v)$ abgespeichert. Zur leichteren Handhabe wird das Gewicht wie bereits früher nicht direkt gespeichert, sondern in Differenzform. Desweiteren wird auch noch ein Wert gespeichert der die Berechnung des minimalen Wertes in einem Pfad ermöglicht. Auch dieser Wert wird in Differenzform gespeichert. Die beiden Felder $\Delta cost$ und Δmin werden also wie folgt belegt.

$$\begin{aligned} \Delta cost(x) &= cost(x) && \text{, falls } x \text{ der Wurzelknoten eines Splay Trees ist} \\ &= cost(x) - cost(p(x)) && \text{, sonst} \\ \Delta min(x) &= cost(x) - mincost(x) \end{aligned}$$

Bei Rotationen, wie sie bei *splay*-Operationen auftreten, müssen die Werte $\Delta cost$ und Δmin entsprechend angepaßt werden. Dies kann in $O(1)$ Zeit durchgeführt werden.
Exemplarisch wird dies für eine einfache Rotation (*zig*) demonstriert. Dazu soll um die Kante zwischen den beiden Knoten v und $w = p(v)$ rotiert werden. Seien a und b die Kinder von v vor der Rotation, und b und c die Kinder von w nach der Rotation, dann sind die Werte wie folgt zu modifizieren (die gestrichenen Funktionen geben dabei die neuberechnete und die ungestrichenen die alten Werte an; alle nicht spezifizierten Werte bleiben unverändert):

$$\begin{aligned} \Delta cost'(v) &= \Delta cost(v) + \Delta cost(w) \\ \Delta cost'(w) &= -\Delta cost(v) \\ \Delta cost'(b) &= \Delta cost(v) + \Delta cost(b) \\ \Delta min'(w) &= \max\{0, \Delta min(b) - \Delta cost'(b), \Delta min(c) - \Delta cost(c)\} \\ \Delta min'(v) &= \max\{0, \Delta min'(w) - \Delta cost'(w), \Delta min(a) - \Delta cost(a)\} \end{aligned}$$

Später wird es noch vorkommen, daß mittlere Kinder v zum linken Kind seines Vaterknotens w, und dadurch ein eventuell vorhandenes linkes Kind u zu einem mittleren wird. Auch in diesem Fall können die Werte wie folgt angepaßt werden:

$$\begin{aligned} \Delta cost'(v) &= \Delta cost(v) - \Delta cost(w) \\ \Delta cost'(u) &= \Delta cost(u) + \Delta cost(w) \\ \Delta min'(w) &= \max\{0, \Delta min(v) - \Delta cost'(v), \Delta min(right(w)) - \Delta cost(right(w))\} \end{aligned}$$

Zur Berechnung von $cost(x)$ sind also die Werte $\Delta cost(x)$ auf dem Weg von x zur Wurzel in dem zugehörigen Splay Tree aufzuaddieren. In Abbildung 3.14 ist eine Umwandlung eines Dynamic Trees in einen erweiterten Splay Tree dargestellt. Die Gewichte der Knoten des Dynamic Trees sind in den Knoten des Dynamic Trees, die beiden Werte $\Delta cost$ und Δmin sind, in dieser Reihenfolge, in den Knoten des Splay Trees eingetragen.
Um $mincost(x)$ zu berechnen muß die Differenz $cost(x) - \Delta min(x)$ gebildet werden. Wenn der Knoten x an der Wurzel des Splay Trees steht, wie dies z.B. nach der Ausführung einer *Splay*-Operation auf den Knoten der Fall ist, kann die Berechnung in $O(1)$ erfolgen.
Eine *splay*-Operation in dem erweiterten Splay Tree kann durch drei hintereinander ausgeführte Operationen erreicht werden. Somit kann erreicht werden, daß auch ein Knoten aus einem der Teilbäume der mittleren Kinder an die Wurzel des erweiterten Splay Trees verschoben wird. Hierbei werden Kanten in dem zugehörigen Dynamic Tree von durchgehend nach gestrichelt und umgekehrt überführt. Die drei Phasen bestehen je aus einem Durchlaufen des Pfades von dem Knoten v, auf dem die Operation ausgeführt wird, bis zur Wurzel. In der ersten Phase wird in den einzelnen Teilbäumen jeweils eine *splay*-Operation auf den Knoten ausgeführt. Somit wird erreicht, daß der Pfad von v zum Wurzelknoten im erweiterten Splay Tree nur aus gestrichelten Kanten besteht. In der zweiten Phase wird der jeweilige Knoten jeweils zum linken Sohn des Vaterknotens gemacht. Der ehemals linke Sohn des Vaterknotens wird dann ein mittleres Kind. Im zugehörigen Dynamic Tree bedeutet dies, daß die Kante vom Ende des Pfades, der durch den ehemaligen linken Sohn bestimmt war, gestrichelt wird, und der des ehemals mittleren Kindes mit einer durchgehenden Kante mit dem Vaterknoten verbunden wird. Dieses Vorgehen wurde in Abschnitt 3.1.1 als *splice*-Operation eingeführt. Durch die zweite Phase wird somit wird erreicht, daß der Pfad von v zum Wurzelknoten nur aus durchgezogenen Kanten besteht und die Knoten immer die linken Kinder des zugehörigen Vaterknotens sind. Jetzt kann durch eine einfache Folge von *zig-zig* und evtl. einer *zig*-Operation der Knoten in dem zum Wurzelknoten gehörenden Splay Tree an die Wurzel verschoben werden. Eine in die drei Phasen aufgeschlüsselte erweiterte *splay*-Operation ist in Abbildung 3.16 dargestellt.
Für die Komplexitätsbetrachtung dieser erweiterten Splay-Operation wird das für Splay Trees definierte Potential so abgeändert, daß nun die Summe der doppelten Ränge der Knoten des Baumes verwendet wird (linke, rechte und mittlere Kinder). Mit dieser Überlegung kann man nun analog zu dem Beweis über die normale *splay*-Operation zeigen, daß die Kosten der drei Phasen zusammen amortisiert maximal $12 \log n + 2 = O(\log n)$ sind.

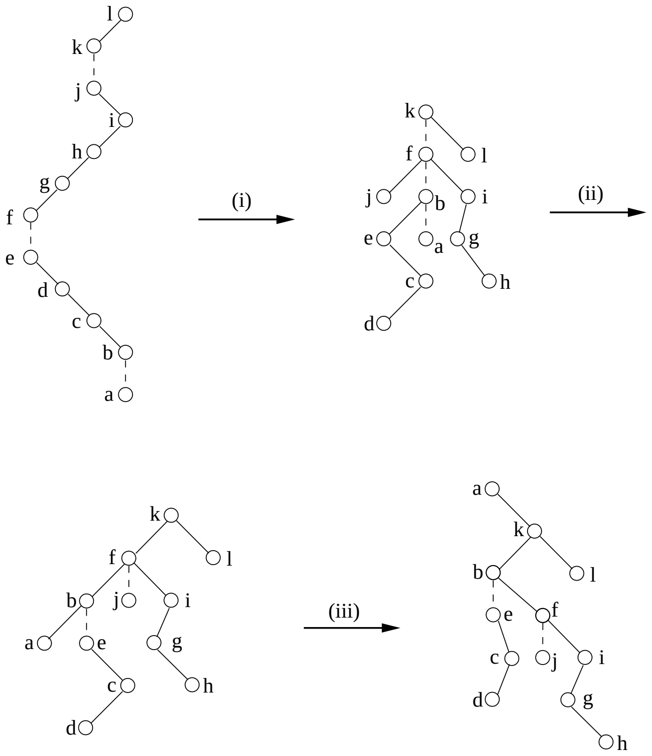


Abbildung 3.16 *Erweiterte splay-Operation*

Alle Dynamic Tree Operationen ohne die die *parent-* und die *evert*-Operation, diese werden gesondert behandelt, können durch ein $O(1)$-maliges Ausführen der erweiterten *splay*-Operation erreicht werden. Die Umsetzung dieser Dynamic Tree Operationen wird wie folgend realisiert:

- $cost(v)$ — Eine *splay*-Operation auf v und Rückgabe des Wertes $\Delta cost$ des neuen Wurzelknotens v
- $root(v)$ — $splay(v)$ verschiebt v an die Wurzel, dann wird entlang der rechten Kinder bis zum letzten nicht *null*-Knoten w gegangen, nach einem $splay(w)$ Rückgabe des Knoten w
- $mincost(v)$ — nach einer *splay*-Operation auf v wird in dem durchgezogenen Pfad von v der letzte Knoten w mit minimalen Kosten gesucht. Dieser Knoten kann in einem Durchlauf mittels $\Delta cost$ und Δmin bestimmt werden. Nach einem $splay(w)$ wird w zurückgegeben.
- $update(v, x)$ — nach einem $splay(v)$ wird der Wert x zu $\Delta cost(v)$ addiert und falls v ein linkes Kind w besitzt von $\Delta cost(w)$ subtrahiert
- $link(v, w)$ – nach den beiden Operationen $splay(v)$ und $splay(w)$ wird v zu einem neuen mittleren Kind von w gemacht (setze $p(v) = w$)
- $cut(v)$ — nach einem $splay(v)$ wird, falls das rechte Kind w existiert, der Wert $\Delta cost(v)$ zu $\Delta cost(w)$ addiert und dann das rechte Kind abgekoppelt ($p(right(v)) = null$ und $right(v) = null$).

Auf Grund der Definition der erweiterten Splay Trees ist der Vaterknoten von v der nächste Knoten in symmetric order in dem durchgezogen Pfad von v falls dieser existiert. Falls v keinen solchen Nachfolger hat, ist es der Vaterknoten der Wurzel des durchgezogenen Teilbaums in dem sich v befindet. D.h. der Vater kann in gleicher Komplexität wie eine erweiterte Splay Operation gefunden werden und anschließend durch ein *splay* an die Wurzel verschoben und anschließend zurückgegeben werden. Die *parent*-Operation hat also die gleiche Komplexität wie eine *splay*-Operation.
Um die *evert*-Operation ausführen zu können, ist eine Modifikation der Baumstruktur notwendig. Hierzu muß wieder ein *reversed*-Bit eingeführt werden, das angibt, daß die Bedeutung der linken und rechten Kinder vertauscht wird. Dieses Bit wird in Differenzform gespeichert. Die Operation $evert(v)$ kann nun wie folgt umgesetzt werden. Durch ein $splay(v)$ wird ein durchgehender Pfad bis zur Wurzel im Dynamic Tree erzeugt, auf dem auch der Knoten v liegt. Da v der Wurzelknoten ist, liegen alle Knoten die auf

diesem Pfad vor v kommen im linken Unterbaum von v. Wird das linke Kind also zu einem mittleren Kind gemacht, so startet der Pfad nun bei v und endet an der Wurzel des Dynamic Trees. Durch Invertieren des *reversed*-Bits von v wird dieser Pfad umgedreht, d.h. v zum neuen Wurzelknoten des Dynamic Trees gemacht. Im Splay Tree bedeutet dies, daß das rechte Kind von v zum linken wird und umgekehrt. In Abbildung 3.17 wird ausgehend von einem Dynamic Tree T einmal direkt eine *evert*-Operation ausgeführt (daraus resultiert T') und andererseits T in einen Splay Tree umgewandelt, in diesem Splay Tree die oben beschriebenen Schritte durchgeführt (das Invertieren des *reversed* Bits wird durch das Vertauschen der Kinder realisiert) und dann der entstehende Splay Tree in einen Dynamic Tree umgewandelt. Dieser Dynamic Tree hat die gleiche Struktur wie T'. Da bei der ganzen Operation im wesentlichen nur eine *splay*-Operation durchgeführt wurde, kann also auch diese Operation amortisiert in $O(\log n)$ realisiert werden.
Mit den vorangegangenen Aussagen kann nun also folgendes abschließendendes Theorem formuliert werden:

Theorem 3.11 *Durch die Verwendung von erweiterten Splay Trees kann eine Folge von m Dynamic Tree Operationen, auf einen anfänglich nur aus ein-elementigen Bäumen bestehenden Wald mit n Knoten, in $O(m \log n)$ Zeit ausgeführt werden.*

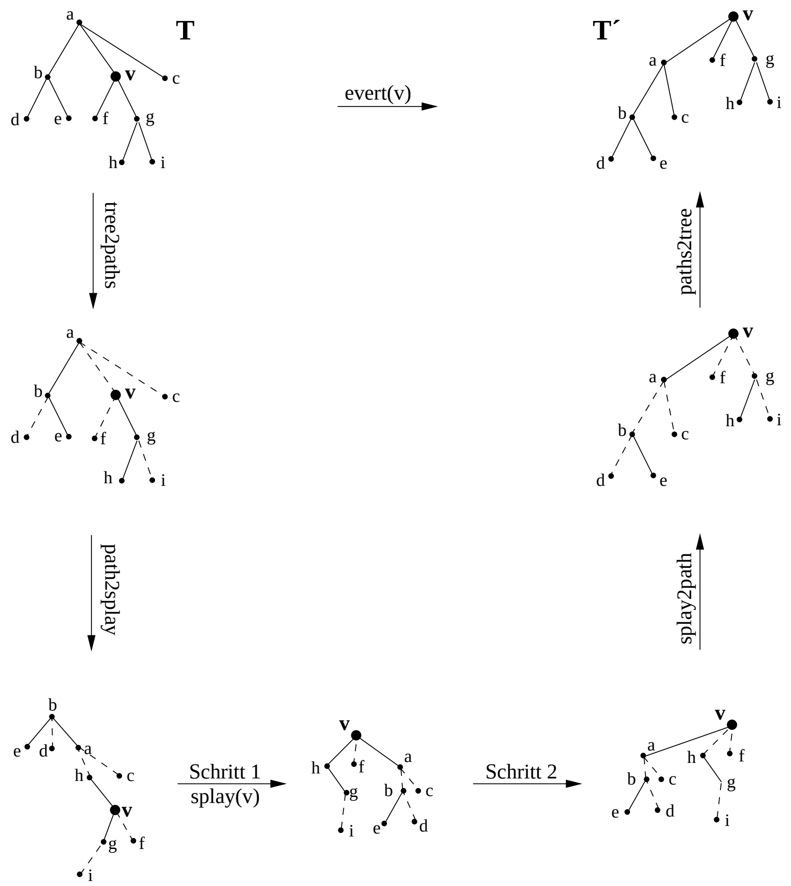


Abbildung 3.17 *evert(v) in einem Splay Tree*

4 Anwendungsbeispiel: Maximales Fluß Problem

Nachdem in den beiden vorangegangenen Abschnitten die Theorie und Implementierungsmöglichkeiten besprochen wurden, soll nun der Einsatz der Dynamic Trees im Push-Relabel-Algorithmus dargestellt werden.

4.1 Problemstellung und Definitionen

Bei dem Maximalen Fluß Problem [Erl98] soll zu einem gegebenen Netzwerk ein maximaler Fluß von der Quelle zur Senke (zwei ausgezeichnete Knoten) gefunden werden. Zu Beginn sollen einige Begriffe definiert werden.

Definition 4.1 *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 positives ganzzahliges Gewicht $c(e)$ zuordnet:* $E \to \mathbb{N}_0$
- s — *die Quelle:* $s \in V$
- t — *die Senke:* $t \in V$

In Abbildung 4.1 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, $n - 1 \leq m \leq n^2$.

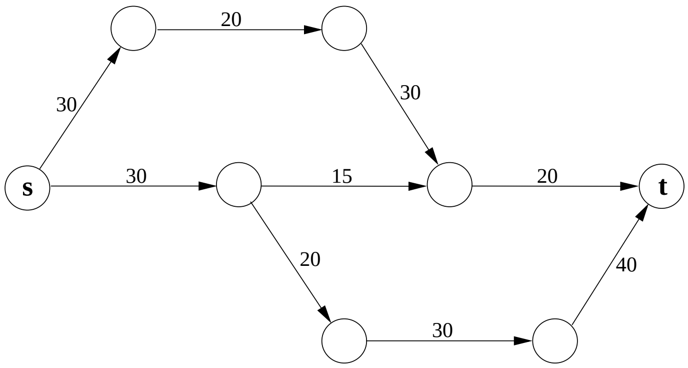


Abbildung 4.1 *Beispiel für ein Netzwerk*

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 $\searrow (v) := E \cap \{w \times v | w \in V\}$, muß gleich der Summe der Flüsse über die Kanten sein, die in v starten $\nearrow (v) := E \cap \{v \times w | w \in V\}$. Als Definition formuliert heißt dies:

Definition 4.2 *Ein 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 4.3 *Der Wert eines gültigen Flusses f_G in einem 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 wesentlichen 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 4.4 *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 Problem des maximalen Flusses in einem Netzwerk genügt es, einen der maximalen Flüsse anzugeben, falls es mehrere solche gibt. In Abbildung 4.2 ist ein maximaler Fluß angegeben. An den Kanten sind zwei Werte angegeben, der erste Wert entspricht den Flußeinheiten und der zweite Wert der Kantenkapazität.

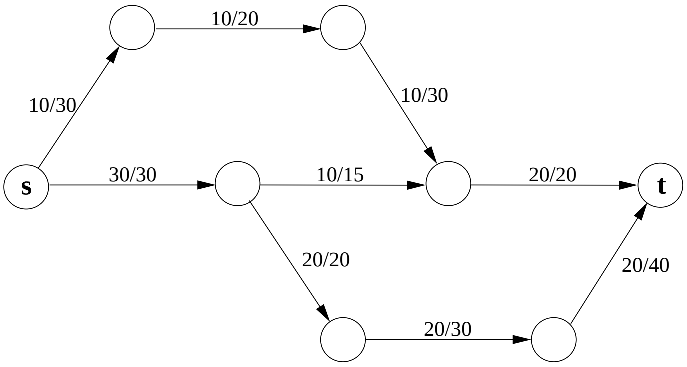


Abbildung 4.2 *Beispiel für einen Maximalen Fluß*

Zur Lösung des maximalen Fluß Problems wurden viele Algorithmen gefunden [May98]. In Abbildung 4.3 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 Daten der Abbildung entnommen sind.

Datum	Entdecker	Referenz	Komplexität
1970	Dinic	[Din70]	$O(n^2 m)$
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]	$O(m^{3/2} \log(n^2/m) logU)$
			$O(n^{2/3} m \log(n^2/m) logU)$

Abbildung 4.3 *Übersicht der verschiedenen Algorithmen für das maximale Fluß Problem*

Der Algorithmus von Goldberg und Tarjan wird im Unterabschnitt 4.2 genauer beschrieben. In Unterabschnitt 4.3 wird dann eine Optimierung dieses Algorithmus vorgestellt, in der die Dynamic Trees verwendet werden.

4.2 Push-Relabel-Algorithmus

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.

4.2.1 Definitionen

Ein Präfluß $\tilde{f}$ ist eine Funktion, die den Kanten eines Netzwerks, wie es in Teilabschnitt 4.1 definiert ist, einen positiven 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 4.2) 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 4.5 *Ein Präfluß $\tilde{f}$ zu einem Netzwerk $G = (V, E, c, s, t)$ ist eine Funktion, für die gilt:*

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

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

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

Aus Definition 4.5 und 4.6 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 $e(t) = -e(v)$.
Diese Eigenschaft wird in dem Algorithmus von Goldberg und Tarjan genutzt. Der Algorithmus arbeitet in zwei Phasen. In der ersten Phase wird von der Quelle die größtmögliche Menge an Flußeinheiten ausgeschüttet und möglichst weit in Richtung der Senke verschoben. Hierdurch kommen in der Senke maximal viele Flußeinheiten an (d.h. in der Senke ist der Überfluß gleich dem maximalen Fluß). In der zweiten Phase werden nun alle überflüssigen Flußeinheiten, die sich noch im Netz befinden, wieder in die Quelle zurückgeschoben. Somit wird erreicht, daß der Überfluß in allen Knoten bis auf Quelle und Senke den Wert 0 annimmt. Der Präfluß wird also zu einem Fluß. Da am Ende der ersten Phase der Überfluß in der Senke maximal war und in der zweiten Phase nicht verändert wurde, handelt es sich bei dem entstehenden Fluß um einen maximalen Fluß. Korrekterweise müßte an dieser Stelle noch die offensichtliche Tatsache gezeigt werden, daß bei einen Präfluß mit maximalem Überfluß in der Senke, dieser Wert dem Wert des maximalen Flusses entspricht. Es wird jedoch auf diesen Beweis verzichtet.
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 der 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 gebildet 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 4.4 darstellt. Formal läßt sich ein Residuennetzwerk wie folgt definieren.

Definition 4.7 *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 $map: \ V \rightarrow V'$*
- 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 4.4). 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.

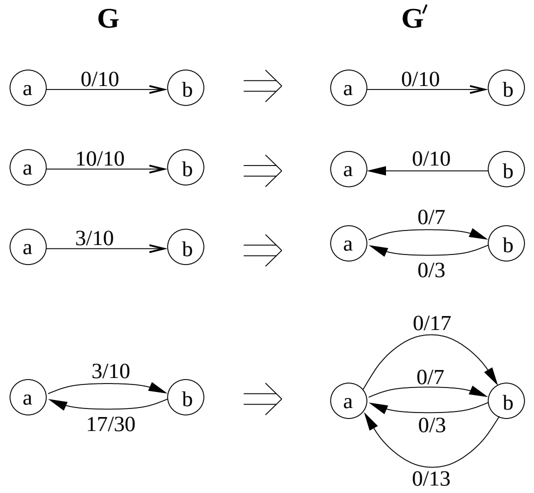


Abbildung 4.4 *Regeln zur Bildung des Residuennetzwerkes*

Die angesprochene Entfernungsfunktion d (engl. distance) ist eine Funktion, die jedem Knoten in einem Graphen ein Wert aus $\mathbb{N}_0$ zuweist. Die gültigen Werte hängen vom jeweiligen Fluß und den damit verbundenen Residuennetzwerk ab (siehe Definition 4.8).

Definition 4.8 *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.*

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

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

Definition 4.10 *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 durch f ein Überfluß hervorgerufen wird, d.h. $e(v) > 0$.*

4.2.2 Algorithmus

Mit diesen Definitionen kann nun der Algorithmus von Goldberg und Tarjan beschrieben werden. Wie bereits weiter oben angedeutet, wird zuerst ein gültiger Präfluß $\tilde{f}$ im Graphen erzeugt.
Dieser Präfluß soll die Eigenschaft besitzen, daß eine maximale Menge an Flußeinheiten in das Netz gegeben wird. Hierfür werden alle Kanten e, die die Quelle verlassen, mit maximal vielen Flußeinheiten versorgt ($\tilde{f}(e) = c(e)$), alle anderen Kanten $\bar{e}$ bekommen keine Flußeinheiten ($\tilde{f}(\bar{e}) = 0$). Die Entfernungsfunktion wird so initialisiert, daß der Wert der Länge eines kürzesten gerichteten Pfades in dem durch $\tilde{f}$ induzierten Residuennetzwerk, von dem jeweiligen Knoten bis zur Senke entspricht. Der Wert für die Senke wird auf 0 und der Wert für die Quelle wird auf die Anzahl der Knoten gesetzt ($d(t) = 0$ und $d(s) = n$).
Durch diesen 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) und 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. 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 ein Fluß und nach obiger Überlegung auch ein maximaler Fluß.
Die *Push-Relabel-Operation* auf einem Knoten v arbeitet in zwei Schritten:

1. Solange in dem Knoten v noch 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. Dieser Schritt wird solange wiederholt, bis entweder der Überfluß abgebaut ist oder keine zulässige ausgehende Kante mehr existiert.
   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 sich nach dem ersten Schritt noch Überfluß im Knoten v befindet, 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.

Bei dem Durchlaufen der Schleife im ersten Schritt der *Push-Relabel-Operation* muß das Residuennetzwerk immer auf dem aktuellen Stand gehalten werden.

4.2.3 Komplexitätsbetrachtungen

Für die Komplexitätsanalyse werden nur die wesentlichen Ideen und Theorem angegeben. Die Beweise sind der Originalarbeit [GT88] zu entnehmen. 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 *Pushes* 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 folgende Theoreme beweisen:

Theorem 4.1 *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.*

Theorem 4.2 *Die Anzahl der insgesamt ausgeführten sättigenden Push-Operationen ist maximal* $2nm = O(nm)$

Theorem 4.3 *Die Anzahl der insgesamt ausgeführten nicht-sättigenden Push-Operationen ist maximal* $4n^2m = O(n^2m)$

Mit diesen Theoremen kann man also eine obere Schranke von $O(n^2m)$ nachweisen. Wenn man nun noch weitere Modifizierungen vornimmt, wie Anlegen einer Liste der ausgehenden Kanten eines Knotens und Merken der aktuellen Kante für spätere Pushes und Verwalten der aktiven Knoten in einer FIFO-Warteschlange, kann man folgende Verbesserung erzielen.

Theorem 4.4 *Bei Verwendung einer FIFO-Warteschlange zur Verwaltung der aktiven Knoten beträgt die Anzahl der insgesamt ausgeführten nicht-sättigenden Push-Operationen maximal* $4n^3 = O(n^3)$

Mit diesem Theorem und den Aussagen über die Anzahl der *Relabel*- und der sättigenden *Push*-Operationen (Theoreme 4.1 und 4.2) und den Überlegungen zur Initialisierung kann nun folgendes abschließendes Theorem formuliert werden.

Theorem 4.5 *Bei Verwendung einer FIFO-Warteschlange zur Verwaltung der aktiven Knoten beträgt die Zeitkomplexität für den Push-Relabel-Algorithmus* $O(n^3)$.

Andere bekannte Methoden für die Auswahl der aktiven Knoten, nämlich die Verwendung des Knotens mit der jeweils größten Distanz oder eine Wellen-Methode, erreichen ebenfalls eine obere Schranke von $O(n^3)$.

4.3 Verwendung der Dynamic Trees

Nachdem nun der Algorithmus von Goldberg und Tarjan beschrieben wurde, soll nun der Einsatz von Dynamic Trees erläutert werden. Die Idee besteht darin, sich die Kanten, über die ein nichtsättigender Push durchgeführt wurde, zu merken und später wiederzuverwenden. Wenn man sich also zu jedem Knoten den Knoten merkt, zu dem man einen Push ausführen kann, und dies für mehrere Knoten durchführt, kann man sich einen Baum aufbauen, in dem Pfade gespeichert werden, über die ein Push laufen kann. Die Dynamic Trees liefern eine solche Datenstruktur. Im Algorithmus wird nun ein Push nicht mehr nur über eine Kante, sondern wenn möglich gleich über mehrere Kanten ausgeführt.
Bei dem nun beschriebenen Algorithmus wird noch eine Operation $size(v)$ benötigt, die die Größe des zum Knoten v gehörenden Dynamic Trees liefert. Diese Operation kann so implementiert werden, daß sie in $O(\log n)$ Zeit durchzuführen ist.

4.3.1 Algorithmus

Wesentlicher Bestandteil des Algorithmus ist die Prozedur $send(v)$ die für einen Knoten v versucht möglichst viele Flußeinheiten entlang des Pfades zur Wurzel des zugehörigen Dynamic Trees zu verschieben. Dies entspricht einem Verschieben von Flußeinheiten in Richtung der Senke des Netzwerks. Für alle Kanten (v, w), die bei diesem Vorgehen gesättigt werden ($cost(v) = 0$) wird der Dynamic Tree an dieser Stelle aufgebrochen. Der Code für die Prozedur ist in Abbildung 4.5 zu sehen.

```
PROCEDURE send (Node v)
  int delta;
  WHILE root(v) != v DO
    delta := min( e(v), cost(mincost(v)));
    update(v,-delta);
    WHILE cost(mincost(v)) = 0 DO
      u := findmin(v);
      cut(u);
      update(u, INFINIT );
    OD
  OD
END send
```

Abbildung 4.5 *Algorithmen-Operation* $send(v)$

Der Algorithmus selbst läuft nach dem gleichen Schema ab wie in Teilabschnitt 4.2 beschrieben. D.h. es werden ebenfalls die aktiven Knoten in einer Queue verwaltet und dann der Reihe nach immer die *push/relabel*-Operation aufgerufen. Nur wird jetzt diese Operation so lange aufgerufen, bis sich der Überfluß abgebaut hat oder die Distanz erhöht hat (in der bisherigen Operation wurde dies bereits durchgeführt, ist jetzt aus Übersichtlichkeitgründen aus der Operation herausgenommen worden). Der Code für die neue Prozedur kann der Abbildung 4.6 entnommen werden. Hierbei handelt es sich bei dem Wert k um eine später in der Komplexitätsbetrachtung noch festzulegende Konstante und bei dem Wert $r(e)$ um die Restkapazität einer Kante e, d.h. der Anzahl an Flußeinheiten, die noch über die Kante geschickt werden könne($r(e) = c(e) - f(e)$). Desweiteren wird ein Wert `INFINIT` zugelassen. Dieser Wert wird den Wurzeln der Dynamic Trees zugewiesen, um sicherzustellen, daß nicht der Wert des Wurzelknotens bei der Operation $findmin$ zurückgeliefert wird.

```
     PROCEDURE push/relabel (Node v)
       CO  e=(v,w) is current edge of v OC
       IF d(v) = d(w) +1 AND r(e) > 0
(1)      IF size(v) + size(w) <= k
 (1a)      link(v,w,r(e));
           send(v);
         ELSE
 (1b)      push(v,w,min(e(v),r(e)));      CO Push over edge e OC
           send(w);
         FI
       ELSE
(2)      IF e != last edge of v THEN
 (2a)      set current edge of v to next edge in the list
         ELSE
 (2b)      set current edge of v to first edge in the list
           FOR all children u of v OD
             cut(u)
             update(u, INFINIT);
           OD
           relabel(v)
         FI
       FI
     END push/relabel
```

Abbildung 4.6 *Algorithmen-Operation push/relabel*(v)

Nun zu einer kurzen Beschreibung der Prozedur. Die Prozedur wird auf einem Knoten ausgeführt, der Überfluß besitzt. Bei diesem Knoten handelt es sich um einen Knoten, der in der Wurzel eines Dynamic Trees steht. Man kann zwei mögliche Abläufe unterscheiden.

1. Fall `(1)` — in diesem Fall handelt es sich bei der aktuellen Kante des Knotens um eine zulässige Kante mit ausreichender Restkapazität, d.h. es kann ein Push über diese Kante, zum Knoten w, durchgeführt werden. An dieser Stelle wird jetzt unterschieden, wie groß die einzelnen Dynamic Trees von v und w sind. Dies ist für die spätere Komplexitätsbetrachtung nötig, soll an dieser Stelle jedoch nicht ausschlaggebend sein. Wichtig ist, daß in beiden Fällen möglichst viele Flußeinheiten von v nach w geschickt werden, und dann diese Flußeinheiten von w aus weiter im Dynamic Tree von w verteilt werden (Dies geschieht einmal, indem v als Unterbaum von w eingehängt wird, gefolgt von einem *send*(v) und das andere mal durch einen Push von v nach w gefolgt von einem *send*(w)). Drei kurze Bemerkungen sind an dieser Stelle noch wichtig. Erstens wird durch die Fallunterscheidung am Anfang von `(1)` sichergestellt, daß keiner der Dynamic Trees mehr als k Knoten enthält. Zweitens kann es nie vorkommen, daß zwei Knoten eines Dynamic Trees durch eine *link*-Operation nochmals miteinander in Bezug gebracht werden. Ein Link findet nur dann statt, wenn $d(v) = d(w) + 1$. Befände sich w bereits in dem Baum von v, müßte jedoch, aus gleichem Grund für den Aufbau dieses Baumes, gelten, daß $d(w) > d(v)$, was einen Widerspruch bedeutet. Drittens wird durch die Operation sichergestellt, daß nach Ablauf der Operation ein Überfluß nur in den Wurzelknoten der Dynamic Trees auftreten kann. Die einzige Zeitpunkt während der Operation an dem dies nicht zutrifft ist während des Falls `(1)`. Die Operation *send* ist jedoch so aufgebaut, daß auf alle Knoten die Überfluß besitzen und ihn nicht weitersenden können durch einen *cut* zu einem Wurzelknoten gemacht werden.

2. Fall `(2)` — Dieser Fall tritt ein, wenn über die aktuelle Kante kein Push ausgeführt werden kann. Handelt es sich bei der aktuellen Kante nicht um die letzte Kante in der Kantenliste des Knotens (Fall `(2a)`), wird die aktuelle Kante weitergeschaltet und die Prozedur beendet (es kann jetzt für die "neue" aktuelle Kante die Prozedur erneut aufgerufen werden). Ist jedoch die letzte Kante bereits

erreicht (Fall `(2b)`), muß ein Relabel durchgeführt werden. Da für einen Knoten jedoch ein Dynamic Tree existiert, muß erst diese Datenstruktur entsprechend abgeändert werden. Ein Relabel des Knotens v bedeutet, daß nicht mehr sichergestellt ist, daß die Kinder von v automatisch Flußeinheiten nach v schieben können. Um dies zu unterbinden, müssen die Kinder u von v getrennt werden (Operationen $cut(u)$ und $update(u, \texttt{INFINIT})$). Erst dann kann ein Relabel ausgeführt werden. Jetzt ist es wieder möglich, die Operation erneut auf v auszuführen.

Abschließend bleibt noch zu bemerken, daß durch die Verwendung der Dynamic Trees nun die Information über die Restkapazität einer Kante sowohl im Residuennetzwerk als auch im Dynamic Tree gespeichert wird. Es ist also immer darauf zu achten, daß diese Informationen konsistent gehalten werden. D.h. bei einer *cut* Operation müssen die Restkapazitäten im Residuennetzwerk immer aktualisiert werden.

4.3.2 Komplexitätsanalyse

Abschließend soll nun gezeigt werden, daß der Einsatz von Dynamic Trees zu einer Verbesserung der Komplexität des *Push-Relabel*-Algorithmus führt. Konnte mit dem bisherigen Algorithmus eine Komplexität von $O(n^3)$ erzielt werden, ist nun eine obere Schranke von $O(nm\log(n^2/m))$ möglich. In dem Originalpapier [GT88] wird dieser Beweis in zwei Schritten ausgeführt. Die Beweise werden im folgenden nicht ausgeführt, sondern nur die Beweisideen dargestellt. Als erstes wird folgende Aussage gezeigt:

Theorem 4.6 *Der Push-Relabel-Algorithmus mit Einsatz von Dynamic Trees benötigt $O(nm\log k)$ Zeit plus $O(\log k)$ Zeit pro Hinzufügen eines Knotens zur Liste der aktiven Knoten.*

Der Beweis basiert auf folgenden Gedanken. Durch die Begrenzung der Größe der Dynamic Trees auf k benötige jede Dynamic Tree Operation $O(\log k)$ Zeit (bei amortisierter Rechnung, welche hier zulässig ist). Jeder Durchlauf der Push-Relabel Operation benötigt $O(1)$ Dynamic Tree Operationen plus die Zeit für die *cut*-Operationen, die verursacht werden (diese wird getrennt berechnet). Also benötigt bei getrennter Berechnung eine Operation $O(\log k)$ Zeit. Für alle Relabels wird $O(nm)$ Zeit benötigt. Die restliche Anzahl der Durchläufe ist Proportional zur Anzahl der Hinzufügungen von aktiven Knoten an die Liste. Insgesamt werden maximal $O(nm)$ *cut*-Operationen ausgeführt. Faßt man diese Überlegungen zusammen kommt man auf die im Theorem gegebene Komplexität.
Bleibt nur noch die Überlegung wie häufig Knoten an die Liste der aktiven Knoten angefügt werden. Es kann folgendes Theorem gezeigt werden.

Theorem 4.7 *Beim Push-Relabel-Algorithmus mit Einsatz von Dynamic Trees wird höchstens $O(nm + n^3/k)$ mal ein Knoten an die FIFO-Liste angehängt.*

Für den Beweis wird die Anzahl der Relabels und der Erhöhung des Überflusses eines Knotens von 0 auf einen positiven Wert betrachtet. So kann gezeigt werden, daß maximal $2n^2 + 20nm + 8n^3/k$ oft ein Knoten an die Liste angehängt wird.
Aus den beiden vorangegangenen Theoremen kann man nun schließlich eine obere Schranke für die Gesamtlaufzeit des Algorithmus angeben.

Theorem 4.8 *Der Push-Relabel-Algorithmus mit Einsatz von 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. Man erhält dann den bereits früher beschriebenen Algorithmus.

Literatur

[Abr83] N. Abramson. *Information Theory and Coding*. McGraw-Hill, New York, 1983.

[AHU76] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. On finding lowest common ancestors in trees. *SIAM J. Comput.*, 5(1):115–132, March 1976.

[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.

[BST85] Samuel W. Bent, Daniel D. Sleator, and Robert E. Tarjan. Biased search trees. *SIAM Journal on Computing*, 14(3):545–568, 1985.

[Chv83] Vašek Chvátal. *Linear Programming*. W. H. Freeman and Company, New York, 1983.

[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.

[Erl98] T. Erlebach. Praktikum Algorithmen-Entwurf. Praktikumsunterlagen, Lehrstuhl für Effiziente Algorithmen, Institut für Informatik, Technische Universität München, 1998. Teil 7.

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

[GT84] H.N. Gabow and R.E. Tarjan. Efficient algorithms for a family of matroid intersection problems. *J. Algorithms*, 5(1):80–131, 1984.

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

[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.

[ST85] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.