Ausarbeitung des Seminarvortrags

„Greedy Algorithmen – Algorithmen von Prim und Kruskal“

John Freytag, 24.06.2008

Inhaltsverzeichnis

Einführung in das Thema

In dieser Ausarbeitung zum Informatik-Seminar-Vortrag „Thema 11: Greedy – Algorithmen von Prim und Kruskal“ sollen primär zwei Algorithmen, die die „Greedy“-Methodik anwenden, erklärt und an Beispielen gezeigt werden.

Greedy – Algorithmen

„Greedy Algorithms“ (dt. „gierige Algorithmen“) sind Lösungsverfahren für Optimierungsprobleme. Obwohl die Greedy-Methodik nur auf solche angewendet werden kann wird das Verfahren häufig zu allg. Algorithmen-Designs wie bspw. „Divide and Conquer“ hinzugezählt[1]. Zusätzlich wird Verfahren häufig auch für Approximations-Algorithmen verwendet, deren Anwendungsfall mit einer nur annähernd optimalen Lösung zufrieden sind.

Von einem Greedy-Algorithmus spricht man, wenn der Algorithmus aus einer Folge von Einzelschritten besteht, die folgende Voraussetzungen erfüllen[2]:

Die Schritte sind

- *gültig*, Erfüllen also die Bedingungen des Problems
- *ein lokales Optimum*, jeder einzelne Schritt ist also von allen gültigen Optionen zum Zeitpunkt des Schrittes die beste (lokale) Wahl
- *unumkehrlich,* d.h. Schritte können, einmal gemacht, nicht zurückgenommen oder verändert werden

In jedem Schritt wird also stets die für diesen Moment bestmögliche Wahl getroffen. Aufgrund dieser Tatsache erklärt sich auch der Name des Verfahrens - „gierig“ wird das beste genommen, was zu kriegen ist.

Das Verfahren soll nun anhand eines einfachen Beispieles verdeutlich werden.

Das „Kassierer-Problem“[3]

Bei dem „Kassierer-Problem“ handelt es sich um eine Aufgabe, die nahezu jeder Kassierer auf der Welt intuitiv nach dem Greedy-Verfahren löst:

1 Vgl. Anany Levitin - „The Design and Analysis of Algorithms“, S. 307
2 Ebd., S. 308
3 Ebd., S. 307

Beim Herausgeben des Wechselgeldes sollen dem Kunden, auf Basis der verfügbaren Geld-Einheiten, so wenige Einheiten (der Einfachheit halber im folgenden „Münzen“) wie möglich zurückgegeben werden. Dies ist einfach zu lösen, wenn man dem Kunden schrittweise immer die größtmögliche Münze auszahlt, deren Wert den noch auszuzahlenden Restbetrag nicht überschreitet.

Nehmen wir an, dem Kassierer stünden eine beliebige Menge von Münzen mit folgenden Geldwerten (in Cent) zur Verfügung:

200 (2€), 100 (1€), 50, 20, 10 und 1

Er würde einen Wechselgeldebetrag von 364 Cent (3,64€) nun also durch folgende Auswahl von Münzen herausgeben:

- 200 Restbetrag: 164
- 100 Restbetrag: 64
- 50 Restbetrag: 14
- 10 Restbetrag: 4
- viermal 1 Restbetrag: 0

Bestimmen eines minimalen Spannbaums

Im Folgenden sollen zwei Algorithmen vorgestellt werden (die Algorithmen von Prim und Kruskal), welche dazu dienen, zu einem gegebenen, verbundenen, gewichteten und ungerichteten Graphen den minimalen Spannbaum zu finden.

Bevor wir uns jedoch der Betrachtung dieser beiden Algorithmen widmen gilt es zunächst, die verwendeten Begriffe zu definieren:

- Ein verbundener, ungerichteter und gewichteter Graph ist eine Menge von Knoten (V, aus dem Englischen für „vertex“), die untereinander durch ungerichtete, gewichtete Kanten (E, „edges“) verbunden sind. Der Graph gilt dann als „verbunden“, wenn jeder Knoten von jedem anderen Knoten aus, dem Graphen über die Kanten folgend, erreichbar ist. Kanten sind dann ungerichtet, wenn sie in allen Richtungen „passierbar“ sind, also man bspw. vom Knoten „B“ auch „A“ erreicht, wenn man „B“ über „A“ erreichen kann.

- Als Kantengewicht soll in diesen Beispielen die Länge der Kante dienen, da dies besonders anschaulich ist. Das Gewicht einer Kante beschreibt die „Kosten“, diese Kante zu passieren, kann also bspw. mit der Länge von Straßen verglichen werden, würde man ein Straßennetz durch einen Graphen abbilden.

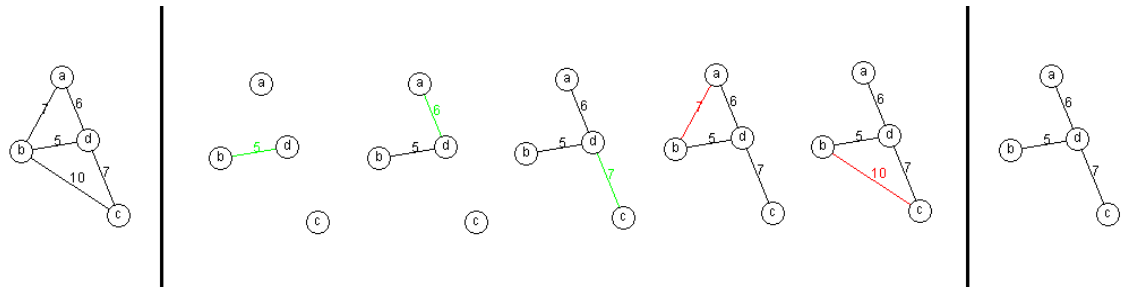
- Ein „Baum“ ist, in der Graphentheorie, ein Graph, welcher keine Zyklen (auch „Kreise“ genannt) enthält. Zyklen liegen dann vor, wenn man einen Knoten ohne einen Richtungswechsel mehrfach erreichen kann. Bspw. also wenn „A“ mit „B“ verbunden ist, „B“ mit „C“ und „C“ wieder mit „A“.
- Ein Spannbaum ist Baum, der als Subgraph (Teilgraph) zu einem gegebenen Graphen alle Knoten des Originalgraphens enthält. Er enthält also nur eine Teilmenge der Kanten des Original-Graphens und alle Knoten, so dass der neu entstehende Graph keine Zyklen enthält.
- Ein Spannbaum ist dann minimal, wenn die Summe aller Kantengewichte minimal ist. Minimale Spannbäume werden bspw. zur Planung von Straßen- oder Kommunikationsnetzen benutzt, um ein Netzwerk zu bestimmen, dass alle gewünschten Punkte (Knoten) miteinander verbindet und dabei möglichst wenig Materialkosten (oder Weglänge) benötigt.

Der Algorithmus von Kruskal

Der Algorithmus von Kruskal folgt einer sehr einfachen Idee, deren algorithmische Umsetzung allerdings einige Schwierigkeiten mit sich bringt:

- Alle Kanten des Originalgraphens werden nach ihrem Gewicht in aufsteigender Reihenfolge sortiert
- Schrittweise werden dann alle Kanten in der sortierten Reihenfolge betrachtet:
  - Würde eine Kante im neuen Graphen zu einem Zyklus führen, so wird sie verworfen
  - Andernfalls wird sie für den Spannbaum übernommen

Das soll an einem kurzen Beispiel verdeutlicht werden:



Ganz links findet sich der Ausgangsgraph, ganz rechts der daraus entstandene minimale Spannbaum. Dazwischen lassen sich die einzelnen Schritte des Algorithmuses nachhvollziehen:

Farblich markierte Kanten werden im aktuellen Schritt (v.l.n.r.) betrachtet. Grüne werden übernommen, rote verworfen. Dass in diesem Beispiel nur die beiden letzten Kanten verworfen werden liegt an der geringen Größe des verwendeten Graphens, bei größeren Beispielen würde der Algorithmus also auch auf Kanten, die zu Zyklen führen würden, treffen, bevor er alle anderen Kanten betrachtet hat.

Das Union-Find-Problem

Für einen Menschen ist es klar ersichtlich, ob und wann eine neu hinzuzufügende Kante zu einem Zyklus führen würde, dies aber effizient allgorithmisch zu implementieren ist die Hauptanforderung bei der Implementierung des Algorithmuses von Kruskal. Eine häufig verwendete Methode soll hier vorgestellt werden: Die Union-Find – Struktur.

Idee dieser Herangehensweise ist folgende:

- Zu Anfang des Algorithmuses wird aus jedem Knoten des Original-Graphens ein Subbaum erstellt, der keine Kanten enthält und als einzigen Knoten eben diesen Knoten beinhaltet.
- Beim Betrachten einer Kante wird nun geprüft, ob die beiden Knoten, die diese Kante verbindet, in verschiedenen Subbäumen sind
    - ist dies der Fall kann es durch Hinzufügen der Kante zum Spannbaum also nicht zu einem Zyklus kommen, da zwischen den beiden Subbäumen der beiden betrachteten Knoten (noch) keine Verbindung besteht. Also wird die Kante übernommen und somit beide Subbäume zu einem gemeinsamen vereinigt.
    - Andernfalls wird die Kante verworfen

Diese Methodik glieder den Algorithmus von Kruskal in zwei Teilprobleme:

- *Find*: Welcher Sub-Baum enthält den aktuell betrachteten Knoten?
- Union: Wie soll das Zusammenlegen zweier Sub-Bäume (effizient) realisiert werden?

Zur effizienten Implementierung werden hierbei die jeweiligen Sub-Bäume nicht durch eine gesonderte Datenstruktur, sondern durch s.g. „kanonische Knoten“ repräsentiert. Das bedeutet, dass ein Knoten des Subbaumes bestimmt wird, der diesen Subbaum repräsentiert. Ergo sind alle Knoten, die mit diesem kanonischen Knoten verbunden sind, im gleichen Subbaum.

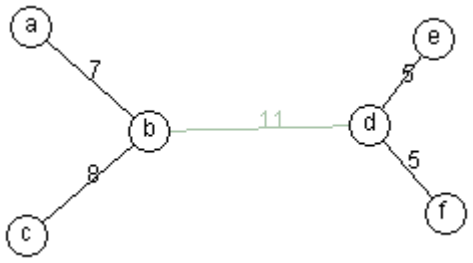
Alle Subbäume werden dabei durch eine (Daten-)Baumstruktur gespeichert, deren Struktur nur indirekt etwas mit dem zugrunde liegendem Graphen zu tun hat:

Wurzelelement eines solchen Baumes ist der kanonische Knoten, alle anderen Knoten des Subbaumes sind über eine Vater-Referenz mit an diesen Baum angehängt. Wenn ein Knoten in dieser Speicherart nun Vater eines anderen Knoten ist, so bedeutet das nicht zwingend, dass es im tatsächlichen Graphen auch eine Kante gibt, die diese beiden Knoten direkt miteinander verbindet, sondern nur, dass beide Knoten im gleichen Subbaum sind. Diesen Unterschied zwischen der abstrakten Baum-Speicherung von Subbäumen und tatsächlichen Graphen zu verstehen ist wichtig für das weitere Verständnis des Algorithmuses, da die gemeinsame Terminologie leicht zu Verwirrung führen kann (in beiden Fällen ist von „Bäumen“ die Rede).

Um auf diese Baumstrukturen nun effizient zugreifen zu können, werden alle Subbäume des Algorithmuses durch ein *einzelnes Array* abgelegt, dass zu jedem Knoten im Graphen dessen Vater-Knoten in der abstrakten Repräsentation eines Subbaumes beinhaltet. Ist die Vater-Referenz eines Knotens dabei der Knoten selbst, so gilt er als Wurzelelement und ist somit der kanonische Knoten seines Subbaumes.

Zur Verdeutlichung folgendes Beispiel:

Nehmen wir an, der Algorithmus wäre nach einigen Schritten in folgender Situation:

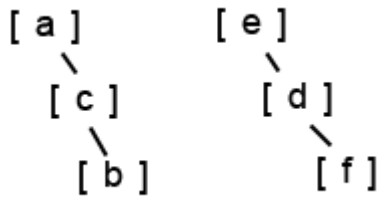


Der Algorithmus betrachtet hier die mittlere Kante mit dem Gewicht „11“, muss also entscheiden, ob Knoten „b“ und „d“ im gleichen Subbaum sind. Da dies offensichtlich nicht der Fall ist, haben

wir also zwei Subbäume: Einen aus den Knoten „a“, „b“ und „c“, sowie einen aus den Knoten „d“, „e“ und „f“. In unserer Array-Struktur sähe das beispielsweise so aus (tabellarisch dargestellt - obere Spalte die jeweiligen Indizes, untere die Verweise auf Vaterknoten, dargestellt durch deren Bezeichner):

a	**b**	**c**	**d**	**e**	**f**
a	c	a	e	e	d

Dieses Array würde damit folgende Baumstruktur abbilden:



Hier ist zu sehen, dass diese Struktur nicht dem tatsächlichen Graphen entsprechen muss: Zwischen "a" und "c" gibt es keine Kante. Beide Knoten sind aber im gleichen Subbaum, weshalb dieser von der abgebildeten Struktur gültig dargestellt wird.

Find

Möchte man nun also bestimmen, in welchem Subbaum ein Knoten ist, so kann dies einfach durch Prüfen der mit diesen Knoten verknüpften Vaterelemente passieren: Man "hangelt" sich so lange von Vaterelement zu Vaterelement, bis man einen Eintrag findet, dessen Vaterreferenz der aktuell betrachtete Knoten selbst ist. Somit hat man die Wurzel der den Subbaum repräsentierenden Baumstruktur gefunden und kennt den kanonischen Knoten des Subbaums - sind beide so ermittelte kanonische Knoten gleich, befinden sich die durch die im aktullen Schritt des Algorithmuses betrachtete Kante verbunden Knoten im gleichen Subbaum.

Damit wäre das "Find"-Problem auf durchschnittlich log n Operationen reduziert (wenn n die Anzahl der Knoten in einem Teilbaum ist), die Gesamtlaufzeitcharakteristik dieses Teilproblems ist also, für den gesamten Algorithmus (v sei die Anzahl aller Knoten im Graphen):

O (v log n)

Union

Mit der geschaffenen Datenstruktur wird die Umsetzung des Union-Problems trivial und in konstanter Zeit lösbar:

Möchte man zwei Subbäume auch in der Datenstruktur zu einem gemeinsamen Subbaum zusammenfassen, so setzt man einfach die Vater-Referenz des Wurzelelements eines Subbaums auf das Wurzelelement des anderen.

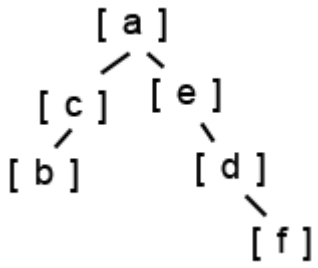
Aus dem im vorherigen Beispiel verwendeten Array

a	**b**	**c**	**d**	**e**	**f**
a	c	a	e	e	d

würde dann also

a	**b**	**c**	**d**	**e**	**f**
a	c	a	e	**a**	d

Was dieser Baumstruktur entspricht:



Man kann also leicht sehen, dass durch einfaches Setzen eines Array-Wertes beide ehemaligen Subbäume nun das gleiche Wurzelelement haben, also nun zum gleichem Subbaum mit dem kanonischen Knoten "a" gehören.

Möchte man nun aber vermeiden, dass sich durch unglückliches Zusammenfügen von diesen Baumstrukturen zu Listen degenerierte Bäume entwickeln (ein Fall der bspw. auftreten kann, wenn man stets "rechten" Baum an den "linken" hängt oder ähnliche beliebige Kriterien verwendet), reicht es, die Bäume nach Größe (Gesamtanzahl der Knoten) oder Tiefe (Anzahl der Knoten auf dem längsten Weg im Baum) zu beurteilen und jeweils immer den kleineren Baum an den größeren zu hängen. Man kann leicht sehen, dass dieses Verfahren zu ausreichend balancierten Bäumen führt, die das Laufzeitverhalten der Find-Operation garantieren. Da das Thema der Optimierung von Baumstrukturen nur sekundär mit dem Algorithmus von Kruskal zu tun hat, soll es hier nicht weiter

behandelt werden.

Es folgt eine Beispielimplemtierung in Pseudo-Code, die das Union-Find-Problem weiter erhellen soll. In diesem Beispiel werden die Bäume nach Größe beurteilt, die Größe der Bäume wird dabei in einem gesonderten Array gespeichert, dass zu jedem kanonischen Knoten die Größe des durch ihn repräsentierten Subbaumes enthält.

Zwei Arrays, deren Größe der Anzahl an Knoten im Graphen entspricht:

Vertex[] parentMap ← Weist einem Knoten (Vertex) ein Vaterelement zu

int[] size ← Hält zu jedem kanonischen Knoten die Größe des durch ihn repräsentierten Subbaumes. Initial sind alle Werte 1.

```
Vertex find ( Vertex v ) {
        Vertex currentVertex = v;
        while (parentMap[currentVertex] != currentVeretx) {
                currentVertex = parentMap[currentVertex];
        }
        return currentVertex; //der gefundene kanonische Knoten des Subbaumes, der v enthält
}

// a und b sind hier kanonische Knoten, repräsentieren also einen Sub-Baum
void union ( Vertex a, Vertex b) {
        if ( size[a] < size [b]) {
                parentMap[a] = b;  //b ist jetzt Wurzelelement des neuen Subbaumes
                size[b] = size[b] + size[a];
        } else {
                parentMap[b] = a;  //a ist jetzt Wurzelelement des neuen Subbaumes
                size[a] = size[a] + size[b];
        }
}
```

Laufzeitanalyse

Das Laufzeitverhalten von Kruskals Algorithmus setzt sich aus den Charakteristika seiner Teilprobleme zusammen:

Größen:

- e: Anzahl der Kanten des Originalgraphen (Edges)
- v: Anzahl der Knoten (Vertices)
- n: Anzahl der Knoten in einem Teilbaum

Teilprobleme:

- Sortieren der Kanten: O(e log e)
- Finden der Teilbäume: O(v log n)
- Union-Operationen: O(v)

Ergibt zusammen:

O ((e + v) log v)

Da in einem verbundenen Graphen in der Regel die Zahl der Kanten die der Knoten übersteigt, kann der Ausdruck vereinfacht werden:

O (e log v)

Der Algorithmus von Prim

Der Algorithmus von Prim wurde eigentlich 1930 vom Tschechen Vojtěch Jarník entwickelt und nur 1957 von Robert C. Prim, sowie 1959 von Edsger Dijsktra wieder entdeckt.[4]

Dieser Algorithmus vermeidet durch seinen Aufbau bereits das Bilden von Zyklen im berechneten Spannbaum, eine gesonderte Union-Find – Problemstellung wie beim Algorithmus von Kruskal ist also nicht nötig. Dennoch wird in der Praxis häufiger der Algorithmus von Kruskal eingesetzt, obwohl das Laufzeitverhalten beider Algorithmen gleich ist. Vermutlich liegt das an der geringeren Speicherkomplexität des Algorithmuses von Kruskal sowie daran, dass der Algorithmus von Prim zusätzlich eine effiziente Implementierung einer Vorrang-Warteschlange („Priority-Queue“)

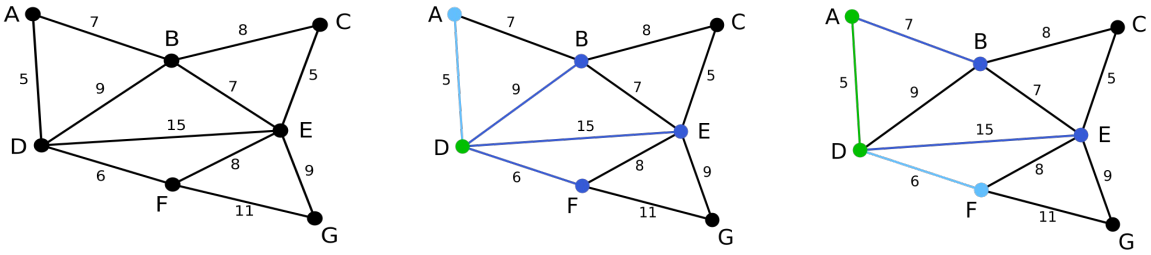
4 Vgl. http://de.wikipedia.org/wiki/Algorithmus_von_Prim

benötigt.

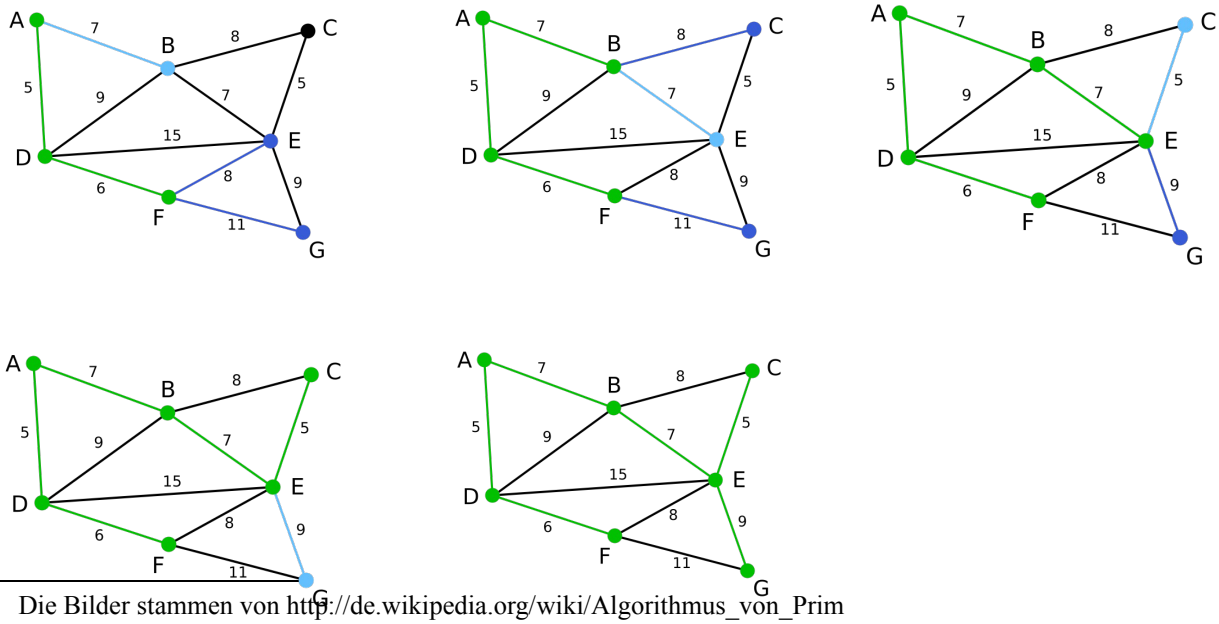
Der Idee des Algorithmuses ist es, mit einem beliebigen Knoten zu starten und ihn schrittweise zum minimalen Spannbaum wachsen zu lassen:

- Starte mit einem beliebigen Knoten des Graphens als Start-Baum T
- Solange in T noch nicht alle Knoten enthalten sind:
  - Wähle Kante e mit minimalen Gewicht, die T mit einem Knoten v verbindet, der noch nicht in T ist.
  - Füge e und v zu T hinzu

Dies soll durch folgendes Beispiel verdeutlicht werden[5]:



Schrittweise wird hier der Spannbaum um einen Knoten und die ihn verbindende Kante erweitert. Dabei wird immer der Knoten ausgewählt, der zum bisherigen Gesamtbaum T mit der minimalen Kante verbunden ist.



5 Die Bilder stammen von http://de.wikipedia.org/wiki/Algorithmus_von_Prim

Blaue Kanten stellen in diesem Beispiel solche dar, die erreichbare Knoten mit dem Baum T verbinden. Ist ein Knoten vom Baum aus über mehrere Kanten erreichbar, ist nur die minimale Kante blau markiert.

Effiziente Implementierung

Für den Algorithmus bietet es sich an, den Graphen als Adjazenzmatrix (Angrenzungsmartix) zu speichern, die zu jedem Knotenpaar speichert, ob diese durch eine Kante miteinander verbunden sind. So können zu jedem Knoten alle an ihn angrenzenden Knoten effizient abgerufen werden.

Um nun die minimale Kante zu bestimmen, die einen Knoten, der noch nicht in T ist, mit T selbst verbindet, werden alle noch nicht in T aufgenommenen Knoten in einer Priority-Queue sortiert. Das Sortiergewicht ist dabei das Gewicht der minimalen Kante, mit dem sich der Knoten zu T verbinden lässt, oder *unendlich **(inital alle)***, wenn keine solche Verbindung besteht. Zusätzlich wird die verbindende Kante gespeichert.

Wird nun ein Knoten zum Baum T hinzugefügt, werden alle angrenzenden Knoten v in der Priority-Queue angepasst:

Ist ihr Sortiergewicht größer als das Gewicht w der Kante e zu v, dann wird das Sortiergewicht auf w gesetzt und e als neue Kante am Eintrag gespeichert. Dadurch werden immer nur solche Knoten mit Informationen aktualisiert, die auch für T in Frage kommen.

Laufzeitanalyse

Auch hier setzt sich das Laufzeitverhalten aus den einzelnen Charakteristika zusammen, es gelten die gleichen Größen wie beim Algorithmus von Kruskal:

- Alle Operationen auf der Priority-Queue (Entfernung des Kopfes, Einfügen, Ändern der Priorität) haben das Laufzeitverhalten O(log V), wenn die Queue als Fibbonacci-Heap implementiert wird.
- Es werden V – 1 Löschungen aus der Queue durchgeführt
- sowie E Überprüfungen und mögliche Änderungen der Prioritäten

ergibt: $(V - 1 + E)\ O\ (\log V) = \mathbf{O\ (E \log V)}$

Quellenangaben:

Primäre Quelle: Thomas Ottmann, Peter Widmayer - „Algorithmen und Datenstrukturen“
(4. Aufl., erschienen im „Spektrum Akademischer Verlag“, 2002)

Nebenquellen: Anany Levitin - „The Design and Analysis of Algorithms“
(Second Edition, erschienen im „Pearson – Addison Wesley“ - Verlag, 2007)

http://de.wikipedia.org/wiki/Algorithmus_von_Prim