

1. Einführung

1.1. Algorithmentheorie

Was ist überhaupt ein Algorithmus? Der Name geht zurück auf den arabischen Mathematiker Abu Ja'far Mohammed Ben Musa al-Khwarizmi, dessen Familienname verfremdet wurde und nun das Wort stellt. Al-Khwarizmi lebte 780-850 n.Chr. Er brachte 2 Standardwerke heraus: der Titel des ersten lautet sinngemäß „Al-Khwarizmi über die Hindu-Art des Rechnens“, vom zweiten Titel „Hisab al-jabr w'al-muqabala“ leitet sich das Wort „Algebra“ ab.¹ Vermutlich geht der Ursprung der Verfremdung auf die lateinische Übersetzung seines 1. Buches zurück: „Algoritmi de numero Indorum“.

Der wahrscheinlich erste Algorithmus wurde von Euklid formuliert, und ist folgerichtig als euklidischer Algorithmus bekannt: Er behandelt die Bestimmung des größten gemeinsamen Teilers durch Division mit Rest.

Allgemein betrachtet ist ein Algorithmus eine systematische Prozedur, die in einer endlichen Anzahl von Schritten die Antwort auf eine Frage bzw. die Lösung für ein Problem liefert.² Normalerweise bedeutet das, daß ein Algorithmus ähnlich einer mathematischen Funktionsvorschrift gewisse Rechenschritte mit einem gegebenen Anfangswert ausführt und daraufhin ein eindeutiges Ergebnis ausgibt. Dies kann eine Zahl, aber auch Text oder eine graphische Darstellung sein. Das Ergebnis muß bei gleichen Grundvoraussetzungen reproduzierbar sein.

In der Informatik sind Algorithmen ein Standardwerkzeug um komplexere Aufgabenstellungen zu bewältigen, beispielsweise das Komprimieren einer Datei, Sortieren eines Datenbestandes oder das Durchsuchen eines solchen. Aufgrund der Definition eines Algorithmus enthält jedes Computerprogramm mindestens einen (sich selbst), jedoch wird es im Normalfall viele Subalgorithmen enthalten, die die Behandlung häufig wiederkehrender Aufgaben vereinfachen.

1.2. Laufzeitberechnung

Um die Effizienz eines Algorithmus zu bestimmen, benutzt man normalerweise seine Laufzeit. Je schneller ein Algorithmus arbeitet, desto höher ist seine Effizienz. Natürlich müssen bei einer Programmentwicklung auch andere Faktoren wie Speicherkapazität etc. berücksichtigt werden, die Laufzeit bietet jedoch einen guten Anhaltspunkt zur Beurteilung der Nützlichkeit eines Algorithmus. So gut wie immer gibt es nämlich für ein Problem verschiedene Lösungsansätze (=Algorithmen), und als Programmierer hat man somit die Qual der Wahl. Um die Rechenzeit eines Algorithmus zu beschreiben, hat man verschiedene Funktionsgruppen eingeführt: Dies sind die Theta-, Omega- und Omikronfunktionsgruppen. Dabei wird zwischen den großen und den kleinen (griechischen) Buchstaben unterschieden. Die Funktionsgruppen bezeichnen nicht exakt die Rechenzeit des Algorithmus, sondern genauer das Wachstum der Laufzeit desselben. Diese Betrachtungsweise konzentriert sich auf sehr große Eingabewert (im folgenden als n bezeichnet) und läßt geringere Beträge für n außer Acht. Für diese Beträge nähert sich die Kurve der Wachstumsfunktion im Idealfall, jedoch durchaus nicht immer, beliebig nah an die tatsächliche Zeitfunktion an.

¹ FAQ von comp.theory

² <http://www.britannica.com>

Algorithmische Techniken zur Berechnung von Minimum Spanning Trees

Dies sind die verschiedenen Funktionsgruppen³:

- Die Omega-Funktionsgruppe Ω :

Formell bezeichnet $\Omega(g(n))$ für eine gegebene Funktion $g(n)$ die Funktionsgruppe oder das Funktionsset $\Omega(g(n)) = \{ f(n) : \text{es existieren positive Konstanten } c \text{ und } n_0, \text{ so daß } 0 \leq c g(n) \leq f(n) \text{ für alle } n \geq n_0 \}$. $f(n)$ ist hierbei die tatsächliche Zeitfunktion. Etwas verständlicher: Für große Werte (ab n_0) bietet die Omega-Funktionsgruppe eine untere Grenze für die Laufzeit, also die maximal erreichbare Effizienz eines Algorithmus. Beispiele zur Einordnung⁴:

n^2-1 ist nicht Element von $\Omega(n^3)$, denn n^2-1 wächst langsamer als n^3

n^3-1 ist Element von $\Omega(n^3)$, denn n^3-1 wächst genauso schnell wie n^3

n^4-1 ist Element von $\Omega(n^3)$, denn n^4-1 wächst schneller als n^3

Das kleine griechische Omega ω bezeichnet dabei ausschließlich die Funktionen, die schneller als $\omega(g(n))$ wachsen. Demnach ist n^3-1 nicht Element von $\omega(n^3)$, wohl aber n^4-1 .

- Die Omikron-Funktionsgruppe O :

Parallel zur Omega-Funktionsgruppe bietet auch die Omikron-Funktionsgruppe eine Grenze, hier jedoch nach oben: Sie beschreibt das „worst-case-scenario“, d.h. die denkbar schlechteste Effizienz die ein Algorithmus erreichen kann. Formell:

$O(g(n)) = \{ f(n) : \text{es existieren positive Konstanten } c \text{ und } n_0, \text{ so daß } 0 \leq f(n) \leq c g(n) \text{ für alle } n \geq n_0 \}$. Die Einordnung der Beispiele ist demnach wie folgt:

n^2-1 ist Element von $O(n^3)$, denn n^2-1 wächst langsamer als n^3

n^3-1 ist Element von $O(n^3)$, denn n^3-1 wächst genauso schnell wie n^3

n^4-1 ist nicht Element von $O(n^3)$, denn n^4-1 wächst schneller als n^3

Der griechische Kleinbuchstabe o bezeichnet wiederum Funktionen, die ausschließlich langsamer als $o(g(n))$ wachsen, also muß man auch hier n^3-1 für $o(n^3)$ ausschließen.

- Und schließlich die Theta-Funktionsgruppe Θ :

Die formelle Beschreibung gibt schon einmal einen recht guten Einblick in die Funktionsweise dieser Gruppe:

$\Theta(g(n)) = \{ f(n) : \text{es existieren positive Konstanten } c_1, c_2 \text{ und } n_0,$

so daß $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ für alle $n \geq n_0 \}$. Übersetzt heißt das, daß die

$f(n)$ -Funktion zwischen die beiden $c_1 (g(n))$ und $c_2 (g(n))$ eingebettet wird, so daß man eine annähernde Voraussage über den Wert treffen kann, der sich letztendlich für $f(n)$ ergeben wird, vorausgesetzt c_1 und c_2 wurden geschickt gewählt. Es zeigt sich eine nicht weiter verwundernde Einordnung für die Beispiele:

n^2-1 ist nicht Element von $\Theta(n^3)$, denn n^2-1 wächst langsamer als n^3

n^3-1 ist Element von $\Theta(n^3)$, denn n^3-1 wächst genauso schnell wie n^3

n^4-1 ist nicht Element von $\Theta(n^3)$, denn n^4-1 wächst schneller als n^3

Da $f(n) = \Theta(g(n))$ eine „stärkere“ Schreibweise als $f(n) = O(g(n))$ oder $f(n) = \Omega(g(n))$ ist, impliziert $f(n) = \Theta(g(n))$ sowohl $f(n) = O(g(n))$ als auch $f(n) = \Omega(g(n))$. Natürlich gibt es hierfür auch Beweise, jedoch würden diese den Rahmen dieser Arbeit sprengen.

Ein $\theta(g(n))$ gibt es übrigens nicht, es wäre auch ziemlich unsinnig.

³ Alle aus: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest: "Introduction to Algorithms"

⁴ Teilweise entnommen von: <http://www.tutorialpage.de>

2. Der Minimum Spanning Tree

2.1. Problematik

Im Berufsalltag treten für verschiedene Personengruppen eine Vielfalt Problemstellungen auf:

- Ein Platinenhersteller will möglichst kostensparend eine Gruppe von relevanten Knotenpunkten elektrisch äquivalent machen, indem er sie alle mit einem elektrischen Leiter verbindet.
- Eine große Firma will ihre vielen Zweigstellen möglichst eng zusammenarbeiten lassen. Zu diesem Zweck will sie die unterschiedlichen Zweigstellen sowie die Hauptniederlassung der Firma mit Glasfaserleitungen verbinden. Da Glasfaser sehr teuer ist, hat der Vorstand entschieden, daß nur die jeweils kürzesten Verbindungen zum Verlegen der Leitungen in Frage kommen.
- Ein Bauer möchte seine Kühe möglichst schnell von einer Weide zur anderen schaffen – er hat 5 dieser Weiden – damit sie so wenig Gewichtsverlust wie möglich auf dem Weg erleiden, schließlich bekommt er für magere Kühe weniger Geld.

So unterschiedlich diese Probleme auch aussehen mögen, sie haben alle dieselbe Grundlage: Ein System von Knotenpunkten, normalerweise deutlich über 3, soll mit möglichst wenig Aufwand verbunden werden. Für diese Aufgabe existieren natürlich Algorithmen, die hier vorgestellt werden sollen. Dafür müssen jedoch einige Grundbegriffe geklärt werden⁵:

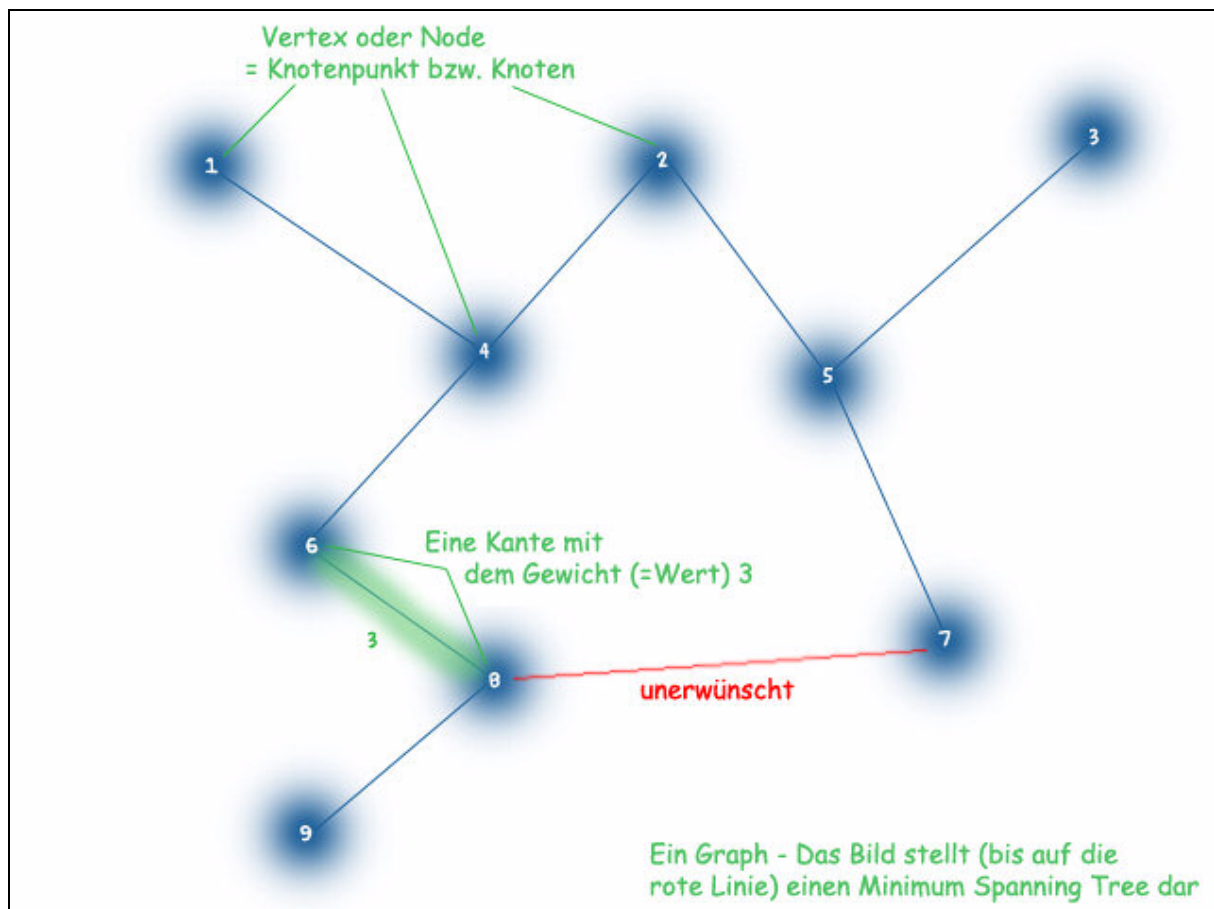


Bild 2.1.1 - Ein Minimum Spanning Tree

⁵ Englische Bezeichnungen aus „Introduction to Algorithms“, deutsche meist mit (Fach-) Wörterbuch übersetzt

2.2.1. Grundbegriffe der Graphenrechnung

Auf dem Bild sind insgesamt 9 **Punkte**, oder präziser, **Knoten** (vertices / nodes – die zwei Begriffe meinen das gleiche) zu sehen. Jeder von ihnen ist mit mindestens einem anderen Knoten verbunden. Die Verbindungsstrecken, mathematisch korrekt **Kanten** (edges), haben einen bestimmten Wert, der z.B. für Materialaufwand oder Länge stehen kann; er wird als **Gewicht** (weight) der Kante bezeichnet. Wenn jeder Knoten von jedem anderen Knoten direkt erreichbar ist, so spricht man von einem **zusammenhängenden Graphen** (connected graph); wenn nur $|V|-1$ (siehe unten) Kanten existieren, und ihr addiertes Gesamtgewicht minimal für diese Konstellation ist, spricht man von einem **minimalen (auf)spannenden Baum** (minimum spanning tree, MST). Da diese Begriffe im englischen geprägt wurden, gibt es teilweise keine anerkannten deutschen Übersetzungen, oder sie variieren sehr stark. Die Übersetzung von minimum spanning tree klingt beispielsweise so gestelzt, daß sie der Mühe nicht wert ist; ich werde weiterhin den englischen Begriff benutzen⁶. Ein **Baum** ist hierbei kein Synonym für einen Graphen, sondern steht erst einmal für **freier Baum**; dies ist ein **zusammenhängender, kreisloser, nicht gerichteter Graph** (connected, acyclic, undirected graph), d.h. er darf keine Kreise enthalten, und Richtungsangaben bei den einzelnen Kanten sind unzulässig. Ein **Wald** (forest) ist eine Ansammlung von Bäumen. Ein **aufspannender Baum** (spanning tree) enthält $|V|-1$ Kanten und verbindet alle Knoten miteinander. Für den **MST** werden fast ausschließlich Bäume benötigt, aus dem einfachen Grund, daß dort außer den relevanten Kanten keine weiteren (die den Graphen zyklisch machen würden) enthalten sein dürfen. Durch die Originalnamen ergeben sich folgende gebräuchliche Abkürzungen: **V** ist die Menge aller Knoten, **E** die aller Kanten. **G** bezeichnet einen Graphen, **T** einen Baum. Man schreibt: $G = (V, E)$, mit einer Gewichtsfunktion $w: E \rightarrow \mathbf{R}$. Kanten werden durch Anfangs- und Endknoten dargestellt, in der Form **Kante** (u, v) , wobei u und v die beiden Knoten sind und sich beliebig vertauschen lassen, da der Graph ja nicht gerichtet sein soll.

2.2.2. Wie läßt man einen Minimum Spanning Tree wachsen ?

Es gibt 2 gängige Algorithmen zur Lösung dieser Aufgabe, nebst einer Vielzahl anderer, die aber kompliziert oder unpraktisch sind und daher nicht weiter erwähnt werden sollen.

Beide bauen auf einer simplen Allgemeinform auf:

Man nimmt einen Graph der Form wie gerade dargestellt, mit $G = (V, E)$ und einer Gewichtsfunktion w , die E auf R abbildet. Außerdem wird das Set **A** neu eingeführt, das zu jedem Zeitpunkt der Ausführung des Algorithmus ein Unterset eines MST T sein soll. Nun bildet man den MST, indem man solange eine Kante (u, v) zu A hinzufügt, die die Bedingung für A nicht verletzt, bis alle Knoten miteinander verbunden sind. Die Bedingung in gebräuchlicher Mengenschreibweise: $A \cup \{(u, v)\}$ soll immer noch ein Unterset eines MST sein. Eine solche Kante, die problemlos hinzugefügt werden kann, ohne die Bedingung zu verletzen, nennt man eine für A **sichere Kante** (safe edge), da sie sich sicher (im Sinne von unbedenklich) hinzufügen läßt. Hier der entsprechende Pseudocode:

ALLGEMEINER MST(G, w)

- 1 $A \leftarrow \emptyset$
- 2 Solange A kein MST ist
- 3 Finde eine Kante (u, v) die für A sicher ist
- 4 $A \leftarrow A \cup \{(u, v)\}$
- 5 Gib A als Ausgabewert zurück

⁶ Eine präzise Definition dieses Begriffs folgt übrigens im Appendix.

Algorithmische Techniken zur Berechnung von Minimum Spanning Trees

In der ersten Zeile wird A die leere Menge zugewiesen. Damit wird bereits die Bedingung erfüllt! In der zweiten Zeile steht die Bedingung für Zeilen 3 und 4, die so lange ausgeführt werden, bis A nicht mehr nur Unterset eines MST, sondern einem solchen gleichwertig geworden ist. Zeile 5 gibt dann den MST aus.

Bei der Ausführung des Algorithmus wird gefordert, daß es eine Kante (u, v) gibt, die für A sicher ist. Dies wird durch die Bedingung für A sichergestellt: Wenn es einen MST T gibt, so daß $A \subseteq T$, und wenn es eine Kante (u, v) gibt, so daß $(u, v) \in T$ und $(u, v) \notin A$, dann muß (u, v) für A sicher sein. Gäbe es keine Kante wie gefordert, hätte dies in Zeile 2 festgestellt werden müssen und der Algorithmus hätte A ausgegeben und beendet.

Die einzige Schwierigkeit ist nun die Implementation der 3. Zeile.

Um diese geschickt hinzubekommen, stellt man sich vor, daß der Graph mit einem **Schnitt** (cut) geteilt wird, und zwar in die Mengen S und $V-S$. Beispielsweise könnte in Bild 2.1.1 unter 4 und 5 eine Linie gezogen werden, die diesen Schnitt symbolisiert. Jede Kante, die nun einen Knoten auf der oberen Seite mit einem auf der unteren Seite verbindet, **überquert** (crosses) diesen Schnitt. Im Beispiel sind dies $(4, 6)$ und $(5, 7)$. Man sagt, daß ein Schnitt A **respektiert** (respects), wenn keine Kante in A den Schnitt überquert. Wichtig werden jetzt die **leichten Kanten** (light edges), die von allen Kanten, die den Schnitt überqueren, das *minimale Gewicht* haben. Davon können durchaus mehrere auftreten, dies stört nicht weiter. Nach der Grundlagenklärung ist es nun möglich, ein Hilfstheorem zu formulieren, daß eine gute Implementation von Zeile 3 ermöglicht:

1. Theorem:

$G = (V, E)$ soll ein verbundener, nicht gerichteter Graph sein, dessen Gewichtsfunktion w auf E definiert ist. A soll ein Unterset von E sein, das in einem MST für G vorkommt. Der Schnitt $(S, V-S)$ soll ein beliebiger Schnitt von G sein, der A respektiert, und die Kante (u, v) soll eine leichte Kante sein, die $(S, V-S)$ überquert. Dann ist die Kante (u, v) eine für A sichere Kante.

Beweis:

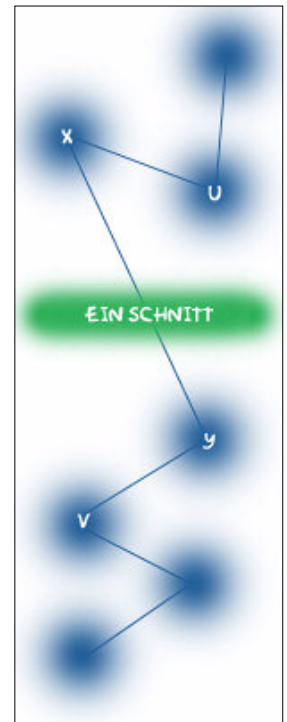
T soll ein MST sein, der A , aber nicht (u, v) enthält. Man konstruiert nun einen anderen MST T^* , der A und (u, v) enthält. Dafür entfernt man eine andere Kante (x, y) . Man läßt u und v auf unterschiedlichen Seiten des Schnitts $(S, V-S)$ sein. Wenn man die beiden Knoten verbindet, überquert man damit den Schnitt. Da T ein MST sein soll, muß eine andere Kante ebenfalls den Schnitt überqueren, sonst hätte man zwei getrennte Graphen vorliegen. Man läßt die Eckpunkte dieser Kante x und y sein. Da der Schnitt laut Definition A respektiert, ist (x, y) nicht in A enthalten. Wenn man nun (x, y) entfernt, so bricht T in die erwähnten getrennten Graphen auf. Diese werden nun wieder durch die *leichte Kante* (u, v) verbunden. Das verbindet die zwei Graphen zum neuen MST $T^* = T - \{(x, y)\} \cup (u, v)$.

Da leichte Kanten von allen Kanten, die einen Schnitt überqueren, das minimale Gewicht haben, ist $w(u, v) \leq w(x, y)$. Dies bedeutet für $w(T^*)$ folgendes: $w(T^*) = w(T) - w(x, y) + w(u, v) \leq w(T)$.

Da T aber auch ein MST sein soll, ist $w(T^*) \geq w(T)$. Daraus ergibt sich letztlich: $w(T^*) = w(T)$. (x, y) war also ebenfalls eine leichte Kante, und das war ja im Theorem gefordert. Es bleibt noch zu zeigen, daß (u, v) wirklich eine für A sichere Kante ist:

Bekannt ist, daß $A \subseteq T^*$, da $A \subseteq T$ und $(x, y) \notin A$. Also ist auch $A \cup \{(u, v)\} \subseteq T^*$.

Da aber T^* bewiesenermaßen ein MST ist, ist (u, v) eine für A sichere Kante. q.e.d.



Folgerungssatz des 1. Theorem:

$G = (V, E)$ soll ein verbundener, nicht gerichteter Graph sein, dessen Gewichtsfunktion w auf E definiert ist. A soll ein Unterset von E sein, das in einem MST für G vorkommt. C soll ein verbundenes Unterset (ein Baum) im Wald $G_A = (V, A)$ sein. Wenn die Kante (u, v) eine leichte Kante ist, die C mit einem anderen Komponenten von G_A verbindet, so ist sie eine für A sichere Kante.

Beweis:

Der Schnitt $(C, V-C)$ respektiert A . (u, v) ist also eine leichte Kante für diesen Schnitt.

3. Die Algorithmen von Kruskal und Prim

Diese beiden Algorithmen verwenden eigene Implementierungen für den ALLGEMEINEN MST Algorithmus. Kruskals Algorithmus verwendet dabei ein Set von Bäumen, das sich immer weiter vereinigt, Prim's Algorithmus hat nur einen Baum der stetig wächst. Über die Personen an sich lassen sich leider so gut wie keine Informationen finden, Joseph B. Kruskal soll bei Bell Labs gearbeitet haben, R. C. Prim läßt überhaupt keine Informationen über sich an die Öffentlichkeit dringen (oder es ist niemand interessiert genug um sich die Mühe zu machen). Wichtig ist aber prinzipiell nur eines: Der Algorithmus an sich. Hier ist der erste:

3.1. Kruskals Algorithmus

Der von den beiden einfacher zu verstehende Algorithmus ist mit Sicherheit Kruskals Algorithmus. Er basiert direkt auf dem ALLGEMEINEN MST aus 2.2.2. Durch die Benutzung von **disjunkten** (d.h. **getrennten**) **Datensets** (disjoint data sets) läßt sich vor allem eine Prozedur gut einfügen: FIND-SET, die feststellt ob zwei Knoten bereits miteinander verbunden sind. Dies ist erst einmal der komplette Algorithmus:

3.1.1. Syntax

MST-KRUSKAL(G, w)

- 1 $A \leftarrow \emptyset$
- 2 Für jeden Knoten $v \in V[G]$:
- 3 Führe MAKE-SET aus
- 4 Sortiere die Kanten von E nach Gewicht w in aufsteigender Reihenfolge
- 5 Für jede Kante $(u, v) \in E$, in aufsteigender Reihenfolge:
- 6 Wenn FIND-SET(u) \neq FIND-SET(v):
- 7 Dann $A \leftarrow A \cup \{(u, v)\}$
- 8 UNION(u, v)
- 9 Gib A als Ausgabewert zurück

Das bedarf natürlich einiger Erklärung. Die erste Zeile weist A die leere Menge zu, damit die in 2.2.2. bereits angesprochene Bedingung erfüllt ist. 2-3 initiieren dann die einzelnen Sets, in denen die Knoten gespeichert werden. Diese Sets bilden die Grundlage für das Funktionieren des Algorithmus; ihre Eigenschaften stellen sich wie folgt dar:

Ein Set besteht aus einer beliebigen Abfolge von Elementen, die die einzelnen Knoten darstellen. In diesem speziellen Fall symbolisiert ein Set eine Verbindung von mehreren Knoten, die miteinander direkt oder indirekt verbunden sind – das Set $\{1; 3; 6\}$ würde also

Algorithmische Techniken zur Berechnung von Minimum Spanning Trees

ausdrücken, daß die Knoten 1, 3 und 6 miteinander verbunden sind. Dazu reichen bereits zwei Kanten; eine Verbindung zwischen 1 und 3 sowie eine Verbindung zwischen 3 und 6 wäre also für dieses Set denkbar. Natürlich könnte auch zusätzlich eine direkte Verbindung zwischen 1 und 6 bestehen, d.h. eine dritte Kante - darüber gibt diese Schreibweise keine Auskunft. Um derartige Information zu erhalten, muß man also eine andere Methode anwenden. Zu Hilfe kommt dabei die schon erwähnte, nützliche Prozedur FIND-SET; sie wählt aus einem beliebigen Set einfach einen **Repräsentanten** (representative), der für gleiche Sets immer gleich ausfällt. Die Bedeutung hiervon wird sofort klar, wenn man sich Zeile 6 anschauen: Hiermit kann festgestellt werden, ob zwei Sets gleich sind, indem einfach deren Repräsentanten miteinander verglichen werden. Aber der Reihe nach:

MAKE-SET aus Zeile 3 erstellt v Sets, wobei jedes Set S_i den Knoten v_i enthält, und zwar ausschließlich. Wenn man also einen Knoten kennt, kann man ihn sofort einem Set zuordnen. Zeilen 4-6 bieten den Schlüssel zur Funktionsweise von Kruskals Algorithmus: In 4 werden die Kanten nach Gewicht sortiert, so daß sie in Zeile 5 in der Reihenfolge von leicht nach schwer abgearbeitet werden. Wenn eine Kante nun hinzugefügt wird, ist es immer die *leichtestmögliche*, d.h. eine leichte Kante. Damit dabei aber keine redundanten Kanten hinzugefügt werden, ist noch die Abfrage in Zeile 6 notwendig: Sie sorgt dafür daß keine **Kreise** (cycles) entstehen. Diese liegen vor, wenn man von einem Knoten aus zu dem gleichen Knoten zurückkehren kann, ohne auf dem Weg eine Kante zweimal zu überqueren. Die Abfrage ist wichtig, da das Ziel des Algorithmus ja darin besteht, alle Knoten mit einem möglichst geringen Gesamtgewicht der Kanten zu verbinden. Wenn zwei Knoten nun aber schon indirekt oder direkt miteinander verbunden sind, und eine andere direkte oder indirekte Verbindung zwischen ihnen hinzugefügt würde, hätte dies keinerlei positiven Effekt; die Knoten waren bereits verbunden. Andererseits würde sich aber das Gesamtgewicht der Kanten erhöhen. Um diesen Effekt zu vermeiden überprüft man also auf Kreise.

FIND-SET(u) liefert den Repräsentanten für ein beliebiges Set S_x , welches u enthält. Dies kann während der gesamten Ausführungszeit immer nur exakt ein Set ein, dafür sorgen MAKE-SET und später UNION. Wenn man also den Repräsentanten von $u \in S_x$ mit dem Repräsentanten von $v \in S_{x2}$ vergleichen, dann können diese immer nur gleich sein, wenn auch die Sets dieselben sind; das gebietet die klare Definition der Sets durch MAKE-SET und wurde weiter oben schon angesprochen. Die Wahl des Repräsentanten ist übrigens relativ unwichtig; man kann sich beispielsweise immer für die niedrigste Zahl des Sets o.ä. entscheiden. Zu diesem Zeitpunkt weiß man also, ob die beiden Knoten u und v im selben Set, respektive verbunden sind. Sind sie es nicht, so wird nun dafür gesorgt: Zeile 7 fügt A die Kante (u, v) hinzu, und in Zeile 8 vereinigt die UNION Prozedur beide Sets miteinander. Die Prozedur an sich stellt sich etwa wie folgt dar:

UNION

- 1 $S_x \leftarrow S_x \cup S_{x2}$
- 2 $S_{x2} \leftarrow \emptyset$

Auf diese Art und Weise kann ein Knoten u immer nur in einem einzigen Set enthalten sein, was zur eindeutigen Identifikation erforderlich ist (siehe Zeile 2). Zu Anfang waren alle Knoten in den ihnen eigenen Sets gespeichert, ein Knoten pro Set. Am Ende der Ausführung gibt es nur noch ein Set mit Inhalt, das dann aber auch alle Knoten enthält (In diesem Beispiel übrigens immer Set 1, da dieses den Knotenpunkt mit dem niedrigsten Wert enthält; früher oder später werden alle Sets zu diesem Set hinzugefügt.).

3.1.2. Laufzeit

Zur Laufzeitbetrachtung des Algorithmus ist es am sinnvollsten, sich die Implementation anzuschauen, die am schnellsten von allen Möglichkeiten läuft. Unterschiede ergeben sich im vorliegenden Fall vor allem durch verschiedene Modelle zur Darstellung der getrennten Datensets. Die beste Implementationsmöglichkeit bieten hier **disjunkte Baumsets** (disjoint forest sets) mit speziellen Optimierungsalgorithmen, die hier kurz vorgestellt werden sollen: Einzelne Sets werden von Bäumen repräsentiert, die diesmal aber gerichtet sind; eigentlich wäre es erst mit dieser Änderung erlaubt, das Attribut *frei* (siehe 2.2.1.) wegzulassen. Zugunsten bequemerer Schreibweise habe ich aber schon zuvor darauf verzichtet.

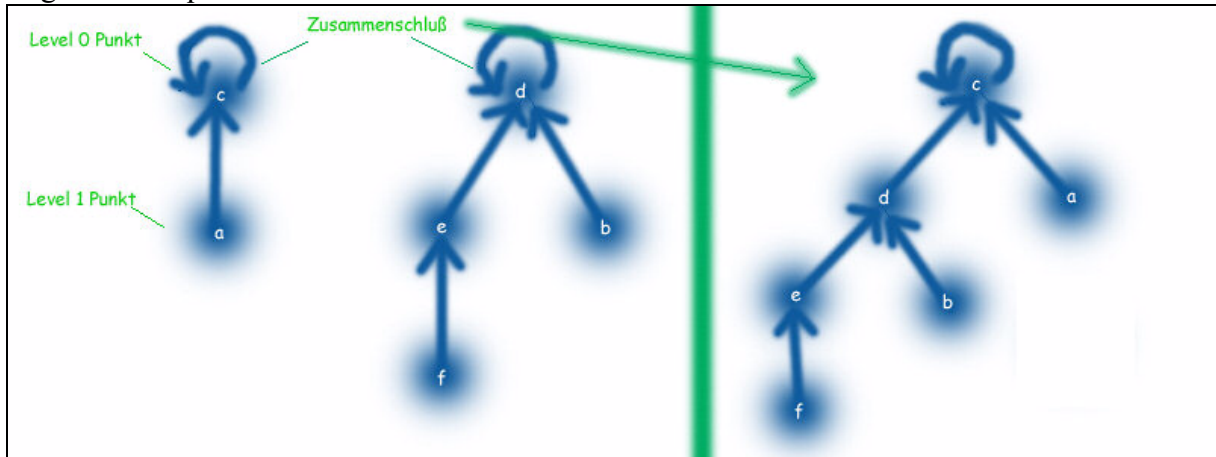


Bild 3.1.2.1. – Zwei Baumsets schließen sich zusammen

Wie man auf dem Bild erkennen kann, zeigen die unteren Knoten jeweils auf die ihnen höhergestellten Knoten. Diese heißen regulär **Eltern** (parents) der unteren Knoten, die entsprechend **Kinder** (children) genannt werden. Von den Knoten, die sich ganz oben befinden, sagt man, daß sie sich auf Level 0 befinden. Diejenigen, die darunter liegen, liegen auf Level 1, die darunter liegen auf Level 2, etc. Die Knoten auf Level 0 sind jeweils **Wurzeln** (roots) des Baums. Diese zeigen auf sich selbst, eine sogenannte **Schleife** (loop). Übertragen auf den MST-KRUSKAL (und damit die konkrete Anwendung) bilden einzelne Bäume die Sets, in denen die Knoten gespeichert werden, die Wurzeln bilden den Repräsentanten. Werden zwei Sets miteinander verbunden, zeigt die Wurzel des einen auf die des anderen, und ein neuer Baum entsteht (siehe Bild). Will man herausfinden, wie der Repräsentant eines Knotens ist, folgt man den Pfeilen bis zur Schleife, bei der man die Wurzel gefunden hat. Dieser Aufbau bietet Ansatz für zwei Optimierungen, auf die oben bereits hingewiesen wurde: **Vereinigung nach Rang** (union by rank) sowie **Pfadkompression** (path compression). Diese Namen klingen gefährlicher, als sich die Prozeduren wirklich darstellen: Vereinigung nach Rang bedeutet nichts anderes, als daß jeweils der Baum, der weniger Knoten beinhaltet, an den anderen angehängt wird, so daß man im Endeffekt weniger Repräsentanten updaten muß, was ja für jeden Knoten einzeln geschieht. Dafür speichert man für jeden Knoten in einer zusätzlichen Variable die Höhe bzw. den **Rang** (rank) des Baumes, der sich unter ihr befindet. Im Falle einer UNION-Operation muß dann nur noch die Wurzel mit dem geringeren Rang auf die mit dem größeren Rang zeigen. Auch Pfadkompression ist eine mit geringem Aufwand zu implementierende Funktion mit hoher Effektivität: Statt bei jeder Suche nach einer Wurzel eines Baums den kompletten Baum zu durchlaufen, benutzt man einen schönen rekursiven Algorithmus, der den **Zeiger** (pointer) eines Knotens direkt auf die Wurzel statt auf den jeweiligen höhergestellten Knoten stellt. Schleifen werden entfernt. Zum besseren Verständnis des Algorithmus sollte man außerdem noch wissen, daß dieser Zeiger eines Knotens x auf einen anderen Knoten mit $p[x]$ bezeichnet wird. Eltern eines Knotens x werden dagegen mit $\pi[x]$ bezeichnet ! Nächste Seite: der sehr kurze Algorithmus.

Algorithmische Techniken zur Berechnung von Minimum Spanning Trees

FIND-SET(x)

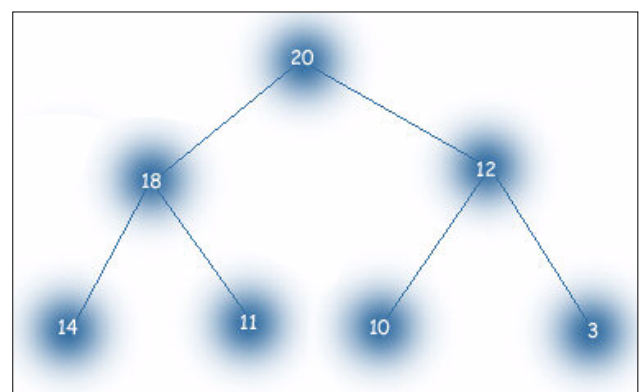
- 1 Wenn $x \neq p[x]$:
- 2 Dann $p[x] \leftarrow \text{FIND-SET}(p[x])$
- 3 Gib $p[x]$ aus

Dies ist eine sogenannte **Zwei-Durchläufe Methode** (two-pass method): Zuerst wird jedem Zeiger von x , der ungleich x selber ist (ist nur bei der Schleife der Wurzel gegeben), der Zeiger seines Elternteils zugewiesen, und zwar von unten nach oben ($p[x]$ erhöht ja x bei jedem Durchlauf um eins). Wenn man oben angelangt ist, ist $x = p[x]$ (durch die Schleife). Dann kehrt sich der Rekursionsablauf um, dem Zeiger des ersten Knotens x_{n-1} unter der Wurzel wird der Wert des Zeigers der Wurzel zugewiesen (Zeile 2). Dann wird dem Zeiger des Knotens x_{n-2} unter dem vorletzten Knoten der Wert dieses Zeigers zugewiesen, der aber gerade schon den Wert des Zeigers der Wurzel erhalten hat. Das geht so weiter bis x_1 , der natürlich auch den Wert des Wurzelzeigers zugeteilt bekommt. In diesem Beispiel durchläuft der Algorithmus einen Pfad von x_1 bis x_n , wie er es auch im regulären Fall macht. Zusätzlich läuft er sogar noch zurück. Man sollte also intuitiv sogar eine schlechtere Laufzeit erwarten! Dies tritt im ersten Anwendungsfall auch ein, jedoch wird dieser Nachteil durch den späteren Algorithmusverlauf mehr als wettgemacht: da die Zeiger jetzt alle direkt auf die Wurzel zeigen, muß nicht bei jedem Aufruf von FIND-SET die gesamte Baumlänge durchlaufen werden, sondern es kann zumeist direkt die Wurzel ausgegeben werden. Durch den Einsatz dieser beiden Techniken läßt sich die Laufzeit von MST-KRUSKAL auf $O(E \lg E)$ drücken. Da es V Knoten gibt, dauert die Ausführung von Zeilen 2 und 3 (die Initialisierung) ziemlich genau $O(V)$ Zeiteinheiten. Der Sortieralgorithmus in Zeile 4 dauert bei geschickter Implementation $O(E \lg E)$ Einheiten. Insgesamt werden $O(E)$ Operationen an den getrennten Baumsets durchgeführt, die alles in allem $O(E \alpha(E, V))$ Zeiteinheiten in Anspruch nehmen.

α ist dabei die sogenannte Inversion einer äußerst komplizierten Funktion (der Ackermann'schen Funktion, um präzise zu sein), die man für $\alpha(E, V)$ auch als $O(\lg E)$ schreiben kann: $\alpha(E, V) \approx O(\lg E)$. Diese wurde beim Verbessern der getrennten Baumset-Datenstruktur erzeugt, aber nicht näher beschrieben. Dies soll auch jetzt nicht geschehen, es wäre zu viel Stoff für die Zielsetzung dieser Arbeit. Man erhält letztendlich für die Laufzeit von MST-KRUSKAL $O(E \lg E)$, wie schon gesagt. Wie ich gleich zeigen werde, ist die Laufzeit für Prim's Algorithmus $O(E + V \lg V)$. Da die Laufzeit bei Kruskal einzig von der Anzahl der Kanten abhängig ist, nimmt man ihn vorzugsweise für sogenannte **spärliche** (sparse) Graphen, bei denen $|E| = \Theta(V)$, was normalerweise der Fall bei über 100 Knoten ist. Andererseits ist der Einfluß von E auf Prim nicht ganz so hoch, was für **dichte** (dense) Graphen wünschenswert ist, für die $E = \Theta(V^2)$, was normalerweise bei $V < 100$ der Fall ist.⁷

3.2. Prim's Algorithmus

Dieser Algorithmus ist etwas schwieriger zu verstehen, vornehmlich da er eine weitere neue Datenstruktur zur Verwaltung der Sets einsetzt: **Prioritätswarteschlangen** (priority queues). Diese werden normalerweise in Form von **Haufen** (heaps) implementiert, wobei man den Begriff „Haufen“ nicht mißverstehen sollte: Obwohl seine Ursprungsbedeutung diejenige



⁷ <http://www.cs.sunysb.edu/~algorithm/>

Algorithmische Techniken zur Berechnung von Minimum Spanning Trees

ist, die in diesem Abschnitt erklärt wird, wird er auch gern mit ungeordneten Daten assoziiert. Das ist hier *nicht* der Fall. Wenn man sich einen solchen Haufen anschaut, versteht man schnell, wie er zu diesem Namen kam: er hat die Form einer Pyramide, die spitz nach oben zuläuft, wie es auch „normale“ Haufen tun. Auf dem Bild ist ein **Binärhaufen** (binary heap) zu sehen, der seinen Namen aus seiner Struktur erhält: Von jedem Knoten oberhalb des vorletzten Levels gehen exakt 2 Unterknoten aus. Im Beispiel ist der einzige Knoten oberhalb dieses Levels Knoten 20. Einen Level darunter können die Knoten 0-2 Kinder haben, auf dem Bild haben 18 und 12 jeweils 2 Kinder. Diese liegen auf der untersten Ebene und dürfen daher keine Kinder haben. Der Endknoten eines jeden **Pfades** (path) von der Wurzel hinab zur (für diesen Pfad) tiefstmöglichen Ebene heißt **Blatt** (leaf). In diesem Fall sind die Blätter 14, 11, 10 und 3. Außerdem ist für einen **geordneten** (ordered) Binärbaum gefordert, daß auf diesem Pfad die Zahlen immer kleiner werden. Für eine Prioritätswarteschlange gibt es nun eine Operation, die für Prim's Algorithmus besonders relevant wird: Dies ist die Funktion $\text{EXTRACT-MIN}(S)$. Sie extrahiert aus einem Set S den Wert mit dem kleinsten möglichen **Schlüssel** (key). Dies ist ein beliebiger Wert, der einem Element von S zugewiesen werden kann, üblicherweise der „Schlüsselwert“ für eine Anwendung, im Fall des MST also das Gewicht der einzelnen Kanten. EXTRACT-MIN löscht das entsprechende Element aus der Warteschlange und gibt es dann als Ausgabewert zurück.

Nun läßt sich der Algorithmus angemessen erklären. Hier also gleich der entsprechende Code:

3.2.1. Syntax

$\text{MST-PRIM}(G, w, r)$

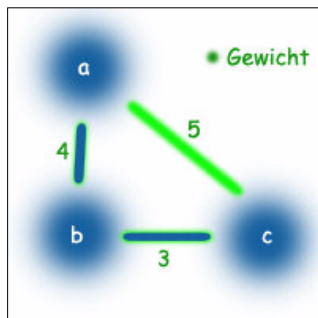
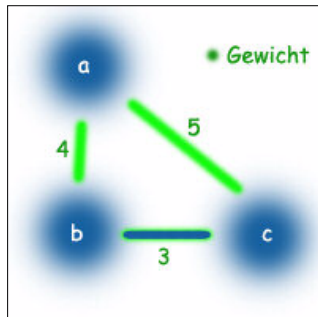
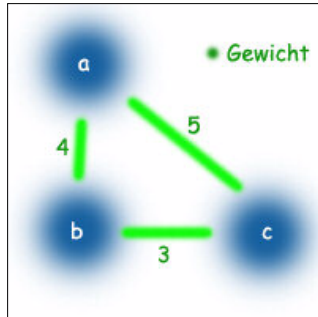
```
1   $Q \leftarrow V[G]$ 
2  Für jedes  $u \in Q$ :
3       $key[u] \leftarrow \infty$ 
4   $key[r] \leftarrow 0$ 
5   $\pi[r] \leftarrow \text{NULL}$ 
6  Während  $Q \neq \emptyset$ :
7       $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
8      Für jedes  $v \in \text{Adj}[u]$ :
9          Wenn  $v \in Q$  und  $w(u, v) < key[v]$ :
10             Dann  $\pi[v] \leftarrow u$ 
11              $key[v] \leftarrow w(u, v)$ 
```

Eine durchaus interessante Struktur. Hier die sicher nötige Analyse:

Zeile 1 erstellt die gerade besprochene Prioritätswarteschlange Q . Der Buchstabe ergibt sich natürlich aus dem englischen Namen „Queue“. Der Warteschlange werden zuerst einmal alle Knoten des Graphen zugeteilt. Dann erhält der Schlüsselwert dieser Knoten nach Konvention den Wert „Unendlich“, da die Größe des Werts nicht meßbar ist – noch existieren für den Algorithmus überhaupt keine Kanten zwischen den Knoten, die ein Gewicht haben könnten. Dann wird in Zeile 4 dem Knoten r , der auch ein Übergabeparameter ist, ein Schlüsselwert von 0 zugeteilt. r ist der Startknoten des Algorithmus. Er ist nun der Knoten mit dem niedrigsten Schlüsselwert und wird somit von der Anweisung in Zeile 7 als erstes an u zugeteilt. Zuvor bekommt er jedoch noch in Zeile 5 das Attribut einer Wurzel $\pi[r] \leftarrow \text{NULL}$, d.h. er hat keine Eltern. NULL ist gleichermaßen der Informatikerersatz für die leere Menge. Jetzt kommt der interessante Teil: solange Q ungleich 0 ist, d.h. sich noch Werte in der Prioritätswarteschlange befinden, wird in Zeile 7 das Element mit dem minimalen Schlüsselwert extrahiert. Dies ist ein Knoten, der noch nicht verarbeitet wurde; er ist noch *nicht* Teil des zu bildenden MSTs. Durch die Minimalität des extrahierten Werts ist nun aber

Algorithmische Techniken zur Berechnung von Minimum Spanning Trees

gegeben, daß die Verbindung von u zum MST ebenfalls minimal groß ist, da dies ein und derselbe Wert ist; $key[u]$ wird in Zeile 11 als Gewicht der Kante (u, v) definiert ! Die Zeilen 8-10 sorgen dafür, daß diese Definition außerdem immer aktuell bleibt; aktualisiert werden dort alle Knoten die mit u verbunden sind und sich zusätzlich noch in der Prioritätswarteschlange befinden. Dabei sorgt die Abfrage in Zeile 8 dafür, daß die Knoten mit u verbunden sind: $Adj(u)$ ist die sogenannte **Adjazenzliste** (adjacency list) von u , in der alle Knoten, die direkt mit u verbunden sind, in beliebiger Reihenfolge gespeichert werden. Ein einfaches Beispiel



soll die Funktionsweise von Prim's Algorithmus verdeutlichen: Gegeben ist der links abgebildete Graph G mit 3 möglichen Kanten sowie bereits eingetragenen Gewichten dieser Kanten. Der Startpunkt r soll in diesem Fall der Knoten b sein. Der Algorithmus startet mit den bekannten Zuweisungen und entfernt danach b aus der Liste Q . Diese besteht nun nur noch aus $\{a; c\}$. Dann werden für a und c , die sich in der Adjazenzliste von b befinden, alle Werte aktualisiert: a erhält als Schlüssel 4, c erhält als Schlüssel 3. Beide erhalten als Elternteil b . Dann läuft der Anweisungsblock ab Zeile 6 zum zweiten Mal durch: aus Q wird der Wert mit dem niedrigsten Schlüsselwert extrahiert, in diesem Fall c mit $key[c] = 3$. $\pi[c]$ ist immer noch b , eine geeignete Implementation des Algorithmus würde jetzt ihren MST updaten und die Strecke (b, c) hinzufügen. Da hier nur die elementare Funktionsweise der Baumfindung und nicht die der Entstehung demonstriert werden soll, findet sich keine entsprechende Anweisung im Pseudocode. Nach der Extraktion von c ist Q jetzt nur noch bestehend aus $\{a\}$. Für dieses a wird jetzt der Versuch unternommen, den Schlüssel- oder Elternteilwert zu aktualisieren; da b für a aber immer noch die günstigste Verbindung darstellt, ändert sich nichts. So geht die Schleife (ab Zeile 6) in die letzte Runde: a wird extrahiert und könnte mit $\pi[a] = b$ jetzt zum MST hinzugefügt werden. Danach ist Q leer und der Algorithmus terminiert. Wegen der Einfachheit dieses Beispiels wird leider nicht sehr deutlich, was der wesentliche Unterschied von Prim's Algorithmus gegenüber Kruskal's Algorithmus ist: Prim fügt immer nur Knoten zum MST hinzu, die direkt mit diesem verbunden werden können (eigentlich ist mit MST sein Unter-

set A gemeint, aber es ist im Prinzip dasselbe). Zum Glück läßt sich das aber auch mit relativ wenig Denkarbeit von selbst erschließen: Da immer nur die Adjazenzliste von Knoten aktualisiert wird, die danach zu A gehören, d.h. immer nur Knoten einen Wert $< \infty$ erhalten, die eine *Verbindung* zu A haben, werden auch immer nur diese aus Q extrahiert werden. So baut sich der Baum auf sukzessive Art und Weise auf.

3.2.2. Laufzeit

Auch Prim's Algorithmus ist davon am abhängigsten, wie man die Knotenverwaltung implementiert: Die Prioritätswarteschlange, über einen Binärhaufen implementiert, braucht $O(V)$ Zeiteinheiten, um die Initialisierung in den Zeilen 1-4 auszuführen. Die Schleife wird $|V|$ -mal ausgeführt. EXTRACT-MIN braucht dabei jedesmal $O(\lg V)$ Einheiten, daher ergibt sich für die Gesamtzeit an Aufrufen von EXTRACT-MIN der Wert $O(V \lg V)$. Die zweite Schleife in den Zeilen 8-11 wird $O(E)$ mal ausgeführt, da die Summe der Länge aller Adjazenzlisten $2|E|$ ist und konstante Vorzeichen in der Funktionsgruppenschreibweise nicht beachtet werden. Zeile 9 läßt sich in konstanter Zeit realisieren, wenn man für jeden Knoten ein Bit aufrecht erhält, das die Mitgliedschaft in Q behandelt. Vergleichsoperatoren wie $<$ oder $>$ lassen sich

von Natur aus in konstanter Zeit implementieren. Für Zeile 11 benötigt die entsprechende Operation jedoch $O(\lg V)$ Zeiteinheiten. Die Gesamtzeit zur Ausführung des Algorithmus beträgt also $O(V \lg V + E \lg V)$, was für genügend große n gegen $O(E \lg V)$ strebt, was asymptotisch gesehen die gleiche Zeit ist, die auch Kruskals Algorithmus benötigt. Wie ich jedoch schon in 3.1.2. angemerkt habe, läßt sich dieser Zeitbedarf durch die Verwendung einer *äußerst* komplexen Datenstruktur namens **Fibonacci-Haufen** (Fibonacci heap) reduzieren, so daß man bei genügend freier Zeit für die Implementation auch eine Laufzeit von $O(E + V \lg V)$ erreichen kann.

4. Gierige Algorithmen

Beide Algorithmen, speziell Prim's Algorithmus, werden unter die Obergruppe der **gierigen Algorithmen** (greedy algorithms) eingeordnet. Diese Gruppe besteht aus Algorithmen, die nicht den Gesamtzusammenhang einer Problemstellung betrachten, sondern immer die für ihre jeweilige Situation beste Entscheidung treffen. Prim's Algorithmus fällt eindeutig in diese Kategorie, da er beim stetigen Voranschreiten immer nur die Kanten zu A hinzufügt, die in seiner jeweiligen Situation den kleinsten Betrag zur Gesamtsumme aller Kantengewichte beiträgt. Kruskals Algorithmus wird im allgemeinen ebenfalls zu dieser Gruppe gezählt, da auch er ein ähnliches Prinzip verfolgt, obwohl er strenggenommen die Kriterien zur Einordnung auch in andere Gruppen erfüllen würde. Dennoch gelten beide MST-Algorithmen als *das* klassische Beispiel für gierige Algorithmen.

5. Appendix

5.1.1. Quellennachweis: Bücher

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest: "Introduction to Algorithms", MIT Press 2000, Auflage 24.

5.1.2. Quellennachweis: Links

- <news:comp.theory>
- <http://www.britannica.com>
- <http://www.tutorialpage.de>
- <http://www.cs.sunysb.edu/~algorith/files/minimum-spanning-tree.shtml>
- <http://www.ics.uci.edu/~eppstein/gina/mst.html>
- <http://brahms.fmi.uni-passau.de/~blochg/mst/>
- <http://hissa.nist.gov/dads/HTML/minspantree.html>
- <http://www.math.uni-hamburg.de/home/diestel/books/graphentheorie/download.html>

5.1.3 Definition: Minimum Spanning Tree

Ein azyklisches Subset T eines Graphen $G = (V, E)$, wobei V die Anzahl aller Knoten in G und E die aller Kanten in G ist, und für den für jede Kante $(u, v) \in E$ eine Gewichtsfunktion $w(u, v)$ existiert, und für das das Gesamtgewicht $w(T) = \sum_{(u,v) \in T} w(u,v)$ minimiert ist, heißt MST.

Dies steht für „minimum-weight spanning tree“, oder abgekürzt „minimum spanning tree“.