

Graphenalgorithmen

Seminar Graphen im Reengineering

Stefan Göller
stefan.goeller@gmx.net

Abstract

Dieses Dokument definiert zunächst den Begriff des „Graphen“. Danach werden diverse Graphenalgorithmen vorgestellt, wobei einerseits die Idee, andererseits die Implementierung des Algorithmus beschrieben wird und auf die Laufzeit eingegangen wird. Es wird grundlegendes Suchen wie Tiefen- und Breitensuche, die Berechnung des minimalen Spannbaums und auf das Problem des kürzesten Weges (ssst,ssat,apsp) eingegangen. Desweiteren wird die Flußoptimierung in Netzwerken, die Planarität und der Schnitt von Graphen behandelt, wobei letzteres auf einer speziellen Darstellung beruht. Als Anwendung im Compilerbau werden Schleifen in Flußgraphen untersucht. Als Abschluß werden Graphenprobleme zusammengefaßt, welche als hartnäckig bezüglich ihrer Laufzeit klassifiziert werden.

1. Allgemeines über Graphen

1.1. Ungerichteter Graph

$G = (V, E)$ heißt (endlicher,) ungerichteter Graph, falls

1. V eine endliche Menge ist. (Knotenmenge)
2. $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$ (Kantenmenge)

Bemerkung: Hierbei sind zunächst keine Schlingen erlaubt, desweiteren gibt es keine Multikanten.

1.2. Gerichteter Graph

$G = (V, E)$ heißt (endlicher,) gerichteter Graph, falls

1. V eine endliche Menge ist.
2. $E \subseteq V \times V$

1.3. Gewichteter Graph

$G = (V, E, \delta)$ heißt gewichteter Graph mit Kantengewichten aus \mathbb{R}^+ , falls G ein (ungerichteter/gerichteter) Graph ist und δ eine totale Funktion ist: $\delta : E \rightarrow \mathbb{R}^+$

1.4. Markierter Graph

$G = (V, E, \varphi, \psi)$ heißt markierter Graph mit Knotenmarkierungsmenge A und Kantenmarkierungsmenge B , falls G ein (ungerichteter/gerichteter) Graph ist und

1. $\varphi : V \rightarrow A$ total ist
2. $\psi : E \rightarrow B$ total ist

2. Darstellungen von Graphen

Es gibt unterschiedliche interne Darstellungen eines Graphen, nämlich die Darstellung als:

1. **Inzidenzliste:** Sowohl Knoten als auch die Kanten werden in separaten Listen gespeichert. Jeder Kanten-eintrag beinhaltet eine Referenz zum Quellknoten und Zielknoten.
2. **Adjazenzliste:** Auch hier werden die Knoten in einer Liste gespeichert. Die Referenzen zu allen Nachbarn jedes einzelnen Knotens sind alle in einer Liste gespeichert.
3. **Adjazenzmatrix** Für $G = (V, E)$ und $n = |V|$ beschreibt eine $n \times n$ -Matrix den Graphen. Wenn die Menge der Knoten nun als $V = \{1, \dots, |V|\}$ umbenannt werden, ist $A_{i,j} = 1$ genau dann wenn $(i, j) \in E$ und 0 sonst. Bei gewichteten Graphen wird bei einer bestehenden Kante statt der 1 eben die Gewichtung $\delta(i, j)$ eingetragen.

Falls der Graph ungerichtet sein sollte, gibt es für alle 3 Darstellungsarten eben die Möglichkeiten, entweder jede Kante $\{u, v\}$ einfach einzutragen, so daß eine Kante von u nach v oder von v nach u führt, oder daß es sowohl eine Kante von u nach v als auch eine Kante von v nach u gibt.

Bei der Analyse der Laufzeit wird jedoch höchstens zwischen der Darstellung mit Adjazenzlisten und -matrizen verglichen.

3. Suchen

3.1. Tiefensuche

geg: $G = (V, E)$

ges: Systematisches Besuchen aller Knoten, wobei in die Tiefe gesucht werden soll.

Vorgehen: Zunächst setzen wir alle beteiligten Knoten auf nicht besucht.

```
for all  $v \in V$  {
    besucht[v] := false;
}
i := 0;

for all  $v \in V$  {
    if besucht[v] = false {
        i := i + 1;
        i-te Zusammenhangskomp.
        Dfs(v);
    }
}
```

Dies ist nun die eigentliche Tiefensuchfunktion, welche vom oben angegebenen Hauptprogramm aufgerufen wird.

```
void Dfs( $v \in V$ ) {
    if (! besucht[v]) {
        besucht[v] := true;
        Knoten  $v$  bearbeiten;
        for all  $w \in V : (v, w) \in E$  {
            Dfs(w);
        }
    }
}
```

Aufwand: $\mathcal{O}(|V| + |E|)$ bei Adjazenzlisten und $\mathcal{O}(|V|^2)$ bei Adjazenzmatrizen.

Abwandlungen: Abwandlungen der Tiefensuche, welche ebenfalls allesamt linearen Aufwand besitzen sind folgende Probleme:

- Auffinden aller Zyklen eines Graphen (gerichtet/ungerichtet)
- Auffinden aller strengen Zusammenhangskomponenten
- Auffinden aller Artikulationspunkte
- Auffinden aller (direkten) Dominatoren eines gerichteten Graphen
- Topologische Ordnung eines Graphen

- Entscheidung, ob G planar bzw. eine Einbettung falls dies der Fall ist ($\rightarrow 5$)

3.2. Breitensuche

geg: $G = (V, E)$

ges: Systematisches Besuchen aller Knoten, wobei in die Breite gesucht werden soll.

Das Hauptprogramm der Breitensuche ist mit dem Hauptprogramm der Tiefensuche identisch. Sobald ein Knoten bearbeitet wurde, wird er als „besucht“ gekennzeichnet und aus der Warteschlange gelöscht. Danach werden alle noch nicht besuchten Nachbarknoten in eine Warteschlange hinzugefügt. Derjenige Knoten, welcher zuerst in die Warteschlange gekommen ist, hat höhere Priorität als später hinzugekommene. Dies wird solange wiederholt bis die Warteschlange leer ist. Am Anfang besitzt die Warteschlange lediglich den Knoten, von welchem die Suche beginnt.

Aufwand: $\mathcal{O}(|V| + |E|)$ bei Adjazenzlisten und $\mathcal{O}(|V|^2)$ bei Adjazenzmatrizen.

3.3. Dijkstra-Algorithmus (für ssat und ssst)

geg: $G = (V, E, \delta)$ ein gewichteter Graph (gerichtet oder ungerichtet) und $s \in V$.

ges: Alle kürzesten Wege von s nach $v, v \in V \setminus \{s\}$ (single source all targets shortest paths).

Vorgehen: Hierbei wird V in 3 disjunkte Mengen $B \cup U \cup R = V$ unterteilt. B ist die Menge der bereits besuchten Knoten für welche der kürzeste Weg induktiv feststeht. Die Menge R beschreibt die Menge der Randknoten, welche allesamt einen Schätzwert besitzen, U entspricht der Menge der noch unbesuchten Knoten. Der Dijkstra-Algorithmus ist ein gieriges Verfahren. Nach jeder Iteration wird B mit genau einem Knoten erweitert und zwar mit dem $v_0 \in R$, welches den kleinsten Schätzwert besitzt. Für diesen Knoten steht die Länge des kürzesten Wegs endgültig fest. Die Schätzwerte aller Nachbarn w von v_0 müssen unter Umständen aktualisiert werden, falls $w \notin B$, denn dann ist der kürzeste Weg bereits bekannt. Falls $w \in R$ kann es möglich sein, daß der kürzeste Weg nun über v_0 führt, falls $w \in U$ kann w in R aufgenommen werden und erhält nun einen Schätzwert. Das Verfahren ist dann beendet, wenn die Randknotenmenge R leer ist, denn dann gibt es keinen Nachbarn mehr, zu welchem ein Weg vom Startknoten s führen kann.

FOR ALL $v \in V$ **DO**

$d[v] := \infty;$

OD

$d[s] := t_0;$

```

B := {s};
R := {v ∈ V | (s, v) ∈ E};
FOR ALL v ∈ R DO d[v] := δ((s, v)); OD
WHILE R ≠ ∅ DO
    v0 := ein Knoten aus R mit d[v0] ≤ d[v] für alle
    v ∈ R;
    B := B ∪ {v0};
    R := R \ {v0};
    FOR ALL w ∈ V : (v0, w) ∈ E ∧ w ∉ B DO
        IF w ∈ R THEN
            IF d[w] ≥ d[v0] + δ((v0, w))
                d[w] := d[v0] + δ((v0, w));
            FI
        ELSE
            R := R ∪ {w};
            d[w] := d[v0] + δ((v0, w));
        FI
    OD
OD

```

Aufwand: Falls der Graph dicht ist, eignet sich die Verwendung einer einfachen Liste für die Randknotenmenge. Hierbei beträgt der Aufwand $\mathcal{O}(|V|^2)$. Ist der Graph allerdings licht, eignet sich die Verwendung eines Heaps für die Randknotenmenge, wobei der Aufwand dadurch auf $\mathcal{O}((|V| + |E|) \log(|V|))$ beschränkt ist. Achtung jedoch bei der Verwendung von Heaps bei dichten Graphen, denn dann kann der Aufwand auf $\mathcal{O}(|V|^2 \log(|V|))$ steigen.

Für das single source single target-Problem (der kürzeste Weg von Knoten u zum Knoten v) scheint der Algorithmus nicht geeignet, da viele Knoten möglicherweise unnötig betrachtet werden, doch man kann den Aufwand verringern, wenn man zu G den Umkehrgraphen G' für v verwendet und nun immer abwechselnd Dijkstra auf G und G' anwendet, also von u aus auf G arbeitet und von v aus auf G' . Das Verfahren ist dann beendet, wenn es einen Knoten g gibt, welcher in beiden Besuchsmengen enthalten ist. Danach gibt es noch 2 Fälle zu unterscheiden. Entweder der kürzeste Weg von u nach v führt über g , wobei wir seine Länge schon kennen. Oder der kürzeste Weg führt direkt von der B -Menge zu G in die B von G' . Man kann nun die Randmenge von G mit der B -Menge von G' schneiden und testen, ob entsprechende Wege kürzer sind als das bisher bekannte Minimum.

3.4. Floyd-Algorithmus (apsp)

geg: $G = (V, E, \delta)$

ges: Für alle Knotenpaare $u, v \in V$ den kürzesten Weg von u nach v (all pairs shortest paths).

Vorgehen: Man könnte natürlich $|V|$ -mal den Dijkstra-Algorithmus starten. Es gibt aber auch eine elegantere Me-

thode, welche nach dem Prinzip des Dynamischen Programmierens vorgeht. Es werden nacheinander (für $i = 1 \dots |V|$) Graphen G_i gebildet, wobei $G_0 = G$ ist. Die Menge der Knoten seien nun $V = \{1 \dots |V|\}$. Dabei gilt für $G_i = (V_i, E_i, \delta_i)$, daß für $u, v \in V$ mit $u \leq v$ und $(u, v) \in E_i$ genau dann, wenn es einen Weg von u nach v gibt, so daß auf diesem Weg kein Knoten $> i$ vorkommt. $\delta_i((u, v))$ sei dann die Länge dieses kürzesten Weges. G_{i+1} läßt aus G und G_i bestimmen. Die Längen aller kürzesten Wege sind dann in $G_{|V|}$ enthalten.

```

for u := 1 to |V| do
    for v := 1 to |V| do
        D[u, v] := δ((u, v));
    od
od
for i := 1 to |V| do
    for u := 1 to |V| do
        for v := 1 to |V| do
            if D[u, v] > D[u, i] + D[i, v] then
                D[u, v] := D[u, i] + D[i, v];
            end if
        od
    od
od

```

Aufwand: Der Aufwand des Floyd-Algorithmus ist $\Theta(|V|^3)$

Bemerkung: Um die beteiligten Knoten jedes kürzesten Weges $u \rightarrow v$ zu erhalten, genügt es sich zu merken, welches $i \in V$ den kürzesten Weg als letztes modifiziert hat. Danach kann man rekursiv mit (u, i) und (i, v) weitermachen.

3.5. minimale Spannbäume

Spannender Baum: Sei $G = (V, E, \delta)$ ungerichtet, gewichtet gegeben. $G' = (V, E')$ heißt spannender Baum zu G falls G' ein Baum ist und $\forall e' \in E'$ gilt $e' \in E$.

Minimaler Spannbaum: Sei $G = (V, E, \delta)$ ungerichtet, gewichtet gegeben. $B = (V, E_B, \delta_{E_B})$ mit $E_B \subseteq E$ heißt minimaler Spannbaum von G : \iff

$$\sum_{e \in E_B} \delta(e) \leq \sum_{e' \in E_{B'}} \delta(e')$$

für jeden minimalen Spannbaum B' von B

Bemerkung: Der minimale Spannbaum muß nicht eindeutig sein.

Verfahren von Kruskal

geg: $G = (G, E, \delta)$ ungerichtet, gewichtet.

ges: Minimaler Spannbaum B von G .

Vorgehen: U sei die Menge der noch zu untersuchenden Kanten. \bar{E}_B sei die Menge der beteiligten Kanten in B .

```

U := E; B := ∅;
∀x ∈ V: x ist seine eigene Menge {x};
while (|B| < n - 1 and U ≠ ∅) {
  wähle e ∈ U mit kleinstem Wert δ(e);
  sei e = {x, y};
  if Find(x) ≠ Find(y) {
    B := B ∪ {e}; Union(x, y);
  }
  U := U \ U{e};
}
if (|B| < n - 1)
  Put('G ist nicht zusammenhängend');
else
  Put('B sind alle Kanten eines MSB');

```

Hierbei ist $\text{Find}(x)$ die zu x gehörende Knotenmenge und $\text{Union}(x, y)$ die Vereinigung der zu x und y gehörenden Knotenmengen. Wenn $\text{Find}(x) = \text{Find}(y)$ dann bildet die Kante $\{x, y\}$ einen Zyklus mit den in B aufgenommenen Kanten.

Aufwand: Der Aufwand des Kruskal-Algorithmus beträgt $\mathcal{O}(|E|(\log(|E|) + \log|V|))$, wenn für Union-Find-Operationen Bäume zur Darstellung der Knotenmengen verwendet werden.

4. Flüsse in Netzwerken

In vielen Anwendungen spricht man von einem Fluß verschiedenster Güter durch ein Verbundnetz. Dabei wird zum Ziel gesetzt diesen Fluß zu maximieren - so bildet ein Ölfeld beispielsweise eine Quelle von welcher Öl durch Ölleitungen von unterschiedlichen Kapazitäten zu einer Ö Raffinerie als Ziel fließen soll.

4.1. Netzwerk:

Ein Netzwerk $N = (G, q, s, c)$ ist ein gerichteter Graph $G = (V, E)$ mit Quelle q und Senke s mit $q, s \in V$ und der Kapazitätsfunktion $c : V^2 \rightarrow \mathbb{N}$, wobei $(u, v) \notin E \Rightarrow c((u, v)) = 0$.

4.2. Fluß:

Eine Funktion $f : V^2 \rightarrow \mathbb{Z}$ ist ein zulässiger Fluß \Leftrightarrow

- $\forall (u, v) \in V^2 : f((u, v)) \leq c((u, v))$ (Kapazitätsbedingung)

- $\forall (u, v) \in V^2 : f((u, v)) = -f((v, u))$ (Symmetriebedingung)

- $\forall v \in V \setminus \{q, s\} : f(u, v) := \sum_{v \in V} f((u, v)) = 0$ (Flußerhaltung)

4.3. Betrag eines Flusses f :

$$|f| := \sum_{v \in V} f((q, v))$$

Somit entspricht der Betrag eines Flusses der Flußmenge, welche von der Quelle weggeführt. Ziel ist es nun diesen Betrag zu maximieren.

4.4. Restnetzwerk eines Netzwerks $N = (G, q, s, c)$ und zulässigem Fluß f :

$N_f := ((V, E_f), q, s, c_f)$ ist das zugeordnete Restnetzwerk, wobei c_f die zugeordnete Restkapazität ist mit:

$$c_f((u, v)) = c((u, v)) - f((u, v))$$

4.5. Erweiterungspfad/zunehmender Weg

Ein Erweiterungsweg ist ein zyklenfreier Pfad p in N_f von q nach s . Ein zunehmender Weg entspricht genau einem solchen Erweiterungspfad.

4.6. Restkapazität eines Erweiterungspfades p

Die Restkapazität eines Erweiterungspfades p ist $c_f(p) = \min\{c_f((u, v)) \mid (u, v) \text{ liegt auf } p\}$

4.7. Eigenschaft 1:

Sei $N = (G, q, s, c)$ ein Netzwerk, f darauf ein zulässiger Fluß, N_f das zugehörige Restnetzwerk und g ein zulässiger Fluß auf N_f . Dann gilt $f + g$ ist ein zulässiger Fluß auf N mit $|(f + g)| = |f| + |g|$.

Dies bedeutet, daß sich der aktuelle Betrag des Netzwerks, welcher durch $|f|$ gegeben ist, immer durch $|g|$ erhöhen läßt, wobei g ein zulässiger Fluß des Restnetzwerks von N war. Es ist tatsächlich so, daß sich der Betrag des Netzwerks nicht mehr erhöhen läßt, falls es keinen Erweiterungspfad in N_f mehr gibt. Dazu benötigen wir jedoch den Begriff des Schnitts eines Netzwerks.

4.8. Schnitt eines Netzwerks N

Ein Schnitt (X, Y) ist eine Zerlegung der Knotenmenge $V = X \cup Y, X \cap Y = \emptyset$ und $q \in X, s \in Y$.

Die Kapazität eines Schnitts (X, Y) ist $c(X, Y) := \sum_{x \in X} \sum_{y \in Y} c((x, y))$. Analog ist dann auch der Fluß f über den Schnitt (X, Y) definiert durch $f(X, Y) := \sum_{x \in X} \sum_{y \in Y} f((x, y))$.

4.9. Eigenschaft 2:

Für jeden Schnitt (X, Y) in einem Netzwerk $N = ((V, E), q, s, c)$ gilt und Fluß f gilt:

$$f(X, Y) = |f|$$

4.10. Eigenschaft 3 (max-flow-mincut-theorem):

Sei f ein zulässiger Fluß in einem Netzwerk $N = (G, q, s, c)$. Dann sind folgende Aussagen äquivalent:

1. f ist maximaler Fluß aus N .
2. Das Restnetzwerk enthält keinen Erweiterungspfad.
3. Es gilt $|f| = c(X, Y)$ für einen Schnitt (X, Y) von N .

Dies sind die Vorüberlegungen zum nachfolgenden Verfahren.

4.11. Ford-Fulkerson Algorithmus:

```

forall  $(u, v) \in E$  {
     $f((u, v)) := 0$ ;
     $f((v, u)) := 0$ ;
}
while  $(\exists p \text{ in } N)$  {
     $r := c_f(p)$ ;
    erhöhe  $f$  entlang  $p$  um  $r$ .
}

```

Der angegebene Algorithmus hat jedoch den Nebeneffekt, daß die Laufzeit von den Kapazitäten abhängen. Man kann einfach ein Beispiel konstruieren, bei welchem es möglich ist, daß der Algorithmus um einiges länger läuft, als er müßte, denn es wird keine Information darüber bereitgestellt, welcher Pfad denn nun genau ausgewählt werden soll. Durch die Edmonds-Karp-Strategie, welche besagt, daß man nur kürzeste Erweiterungspfade wählen soll, kann letzteres vermieden werden, indem man mittels Breitensuche einen zur Senke führenden Weg findet. Durch diese Vorgehensweise ist garantiert, daß der maximale Fluß nach höchstens $|V||E|$ hinzugefügten Pfaden, ermittelt werden kann.

Somit ergibt sich ein Gesamtaufwand von $\mathcal{O}(|V||E|^2)$. Eine abgewandelte Form dieser Vorgehensweise beinhaltet das Suchen desjenigen Pfades, welcher den Fluß um den größten Betrag vergrößert. Dabei wird die Breitensuche

ebenfalls abgewandelt und den Knoten so Prioritäten vergeben, daß immer derjenige Knoten bevorzugt wird, welcher den maximalen Flußzuwachs aufweist. Jedoch ist dieser Zuwachs immer durch den unmittelbaren Vaterknoten beschränkt. Wird nach Pfaden gesucht, welche den Betrag des Netzes maximal erhöhen, so ergibt sich ein Aufwand von $\mathcal{O}(|V|^2 \log(f^*))$ bei Adjazenzmatrizen und $\mathcal{O}((|E| + |V|) \log(|V|) \log(f^*))$ bei Adjazenzlisten, wobei f^* die Kosten des Flusses sind.

5. Planarität von Graphen

Für die graphische Darstellung eines Graphen ist es oft hilfreich zu wissen, ob es möglich ist den Graphen überkreuzungsfrei in die Ebene einzubetten. 1974 entwickelten Tarjan und Hopcroft einen effizienten, doch sehr komplizierten Algorithmus, welcher auf der Tiefensuche beruht.

geg: Graph G .

ges: Das Testen, ob G planar ist und falls ja, die Angabe einer Einbettung für G .

Vorgehen: Im Groben läßt sich der Algorithmus in 2 Phasen unterteilen. Zunächst wird ein Zyklus C von G berechnet, dessen Beseitigung G in mehrere Zusammenhangskomponenten zerfallen läßt. Für diese Zusammenhangskomponenten wird rekursiv überprüft, ob sie alle ebenfalls planar sind. Der zweite Teil des Algorithmus überprüft, ob es Einbettungen für diese Zusammenhangskomponenten mit $G \setminus C$ gibt und berechnet diese, falls möglich.

Man kann zunächst annehmen, daß der G 2-zusammenhängend¹ ist, denn ein Graph ist genau dann planar, wenn seine 2-Zusammenhangskomponenten planar sind. Es wird nun ein Kreis C berechnet, dessen Kanten allesamt Baumkanten einer zuvor durchgeführten Tiefensuche sind, bis auf eine - dies ist eine Rückwärtskante. Wie erwähnt, zerfällt G nun durch Wegnahme aller Knoten und Kanten von C unter Umständen in Teilgraphen $G_1 \dots G_n$, die allesamt in paarweise unterschiedlichen Zusammenhangskomponenten liegen. Nun sei G'_i der Graph, der entsteht, wenn man G_i mit C vereinigt und alle Kanten in G hinzunimmt, deren induzierte Knoten in C und G_i liegen. Es wird nun rekursiv überprüft, ob alle G'_i ihrerseits planar sind. Alle Kanten G'_i , welche mit C verbunden sind, müssen sich in der äußeren Facette von G'_i befinden, wobei G_i entweder innerhalb oder außerhalb von C eingebettet werden muß.

Es muß nun überprüft werden, ob sich die Einbettungen aller G'_i vereinbaren lassen, denn es kann vorkommen, daß

¹2-zusammenhängend bedeutet, daß der Graph durch Entfernen einer beliebigen Kante dieselbe Anzahl von Zusammenhangskomponenten besitzt, wie vor dem Entfernen.

dies nicht möglich ist, denn die G_i besitzen alle mindestens 2 mit C verbundene Kanten, da G 2-zusammenhängend ist. Um dies zu gewährleisten wird der sog. „Interlace Graph“ gebildet, dessen Knoten die Teilgraphen $G_1 \dots G_2$ repräsentieren und genau dann Kanten zwischen zwei Knoten gezogen werden, falls Einbettungen der entsprechenden Teilgraphen auf unterschiedlichen Seiten von G liegen müssen. Ist der „Interlace Graph“ bipartit, so gibt eine überkreuzungsfreie Einbettung für G .

Aufwand: Dieser Algorithmus läuft in $\mathcal{O}(|V| + |E|)$. Eine detaillierte Begründung würde den Rahmen sprengen.

6. Schnitt von Graphen

Seien zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ gegeben. Dann ist der Schnitt beider Graphen $G_3 = (V_3, E_3)$ mit $V_3 = V_1 \cap V_2$ und $E_3 = E_1 \cap E_2$. Dabei wird jedem Knoten und jeder Kante aus G_1 und G_2 eine eindeutige ID zugewiesen. Es wird angenommen, daß ein Graph als 2 AVL-Bäumen dargestellt wird. Der Graph besitzt einen geordneten AVL-Baum aller Knoten, wobei die ID der Knoten als Vergleichskriterium des Baums herangezogen wird, analog gibt es einen AVL-Baum aller Kanten. Um nun den Schnitt G_3 zu bekommen, würde man ähnlich wie bei der Sammelpphase von Mergesort beide Bäume Inorder durchlaufen und einen neuen Knoten- und Kantenbaum für G_3 herstellen. Das Verfahren besitzt insgesamt einen Aufwand von $\mathcal{O}(|V| \log(|V|) + |E| \log(|E|))$

7. Schleifenstruktur in Flußgraphen

Gegeben sei ein Flußgraph $G = (V, E)$, welcher Information über ein Programm liefert. Die Knoten repräsentieren Berechnungen, die Kanten entsprechen dem Kontrollfluß des Programms. Wir bezeichnen mit $u \in V$ den Dominator von $v \in V$, falls jeder Pfad vom Anfangsknoten des Flußgraphen zum Knoten v zwangsläufig über u führt. Jeder Knoten dominiert sich demnach selbst. Desweiteren ist u der direkte Dominator von v , falls u v dominiert, $u \neq v$ gilt und jeder weitere Dominator von v u dominiert.

Wenn man den Dominator-Baum² von G bereits berechnet hat, ist die Menge der Schleifen des Flußgraphen interessant. Jede Schleife des Flußgraphen hat einen eindeutigen Eintrittspunkt, den sog. „Header“. Dieser Header dominiert alle Knoten der Schleife. Desweiteren muß es vom Header mindestens einen Weg zurück zum Header geben. Um nun alle diese Schleifen des Flußgraphen ausfindig zu machen, sucht man nach Kanten (u, v) , so daß v u dominiert, man spricht von einer sog. „Rückwärtskante“. Nun kann man für eine Rückwärtskante (u, v) die natürliche Schleife von

²Ein Knoten eines Dominatorbaumes besitzt als Kinder alle Knoten, welche er direkt dominiert.

(u, v) als v vereinigt mit der Menge von Knoten, welche u erreichen können, ohne v zu besuchen.

geg: Flußgraph G und eine Rückwärtskante $e = (u, v)$

ges: Die natürliche Schleife von e .

Zunächst die Einfüge-Prozedur `insert`

Vorgehen:

```

procedure insert( $m \in V$ );
if  $m \notin loop$  then begin
     $loop := loop \cup \{m\}$ ;
    push  $m$  auf den stack;
end;

```

Nun folgt das Hauptprogramm:

```

 $stack := \emptyset$ ;
 $loop := \{d\}$ ;
insert( $v$ );
while  $stack \neq \emptyset$  do
    pop( $m$ );
    for alle Vorgänger  $p$  von  $m$  do
        insert( $p$ );
    end
end

```

Aufwand: Der Algorithmus besitzt im schlechtesten Fall eine Laufzeit von $\mathcal{O}(|V|)$, denn die natürliche Schleife einer Kante kann alle Knoten beinhalten.

8. Schwierige Probleme

Dieser Abschnitt ist eine Auflistung von Graphenproblemen, für welche es momentan keine effizienten Verfahren gibt ($P = NP$ -Problem). Folgende Liste bietet eine Übersicht:

- **k -Clique:** Das Problem, ob G eine Clique der Größe k besitzt, ist für $k > 2$ NP -vollständig.
- **k -Coloring.** Das Problem zu entscheiden, ob ein Graph k -färbbar ist, ist für $k > 2$ NP -vollständig.
- **Traveling Salesperson Problem:** Gegeben seien n Städte, mit bestimmten Abständen, wobei der Abstand von Stadt A nach B gleich dem Abstand von B nach A ist für alle Städte. Nun ist die Rundreise gesucht, welche alle Städte genau einmal besucht und minimal lang sein soll. Dieses Konstruktionsproblem ist NP -hart. Das Entscheidungsproblem, ob es überhaupt eine Rundreise gibt, ist NP -vollständig.
- **Subgraph-Isomorphie:** Gegeben seien 2 Graphen G_1 und G_2 . Zu entscheiden, ob entweder G_1 ein Subgraph von G_2 ist oder umgekehrt, ist NP -vollständig.

- **Max-Cut:** Gegeben sei $G = (V, E, \delta)$ ungerichtet. Eine Unterteilung der Knotenmenge V in $S \subseteq V$ und $T = V \setminus S$ zu finden, so daß $\sum_{u \in S} \sum_{v \in T} \delta((u, v))$ maximal ist für alle Unterteilungen von V ist NP -vollständig.
- **Minimale Überkreuzungen:** Eine Einbettung für einen nicht-planaren Graphen zu finden, so daß die Anzahl der Überkreuzungen minimal ist, ist NP -vollständig.
- **Geradliniges Einbetten:** Das Problem einen Graphen in die Ebene einzubetten und dabei nur gerade Kanten zu ziehen ist NP -vollständig.

Literatur

- [1] A. V. Aho, R. Sethi, und J. D. Ullman. *Compilers - Principles, Techniques, and Tools*. Addison Wesley Longman, 1988.
- [2] V. Claus. *Theoretische Informatik III(swt)*. 2001.
- [3] I. F. Cruz und R. Tamassia. How to Visualize a Graph: Specification and Algorithms. Seiten 1–71, January 1994.
- [4] U. Hertrampf. *Theoretische Informatik II(swt)*. 2000.
- [5] M. Kaufmann und D. Wagner. *Drawing Graphs*. Springer, 2001.
- [6] W. Kocay. *The Hopcroft Tarjan Planarity Algorithm*. Computer Science Depart, 2001.
- [7] M. Maier. *Proseminar Algorithmen: SS 2001*. Universitaet Passau, 2001.
- [8] C. Papadimitriou. *Computational Complexity*. Addison Wesley Longman, 1994.
- [9] R. Sedgewick. *Algorithmen in C*. Addison-Wesley, Bonn, Muenchen,Paris, 1992.