Hauptstudiumspraktikum Algorithmen und Datenstrukturen

Starke Zusammenhangskomponenten

Dorothea Wagner und Thomas Willhalm
Universität Konstanz

Sommersemester 2000

Inhaltsverzeichnis

Praktikum Algorithmen und Datenstrukturen
Team Nr. 1

Dekomposition eines Graphen in stark zusammenhängende Komponenten

Eine Anwendung der Tiefensuche

Günther Hagleitner, Martin Oberhofer

Fachbereich Informatik und Informationswissenschaft
Universität Konstanz
Deutschland

Sommersemester 2000

Zusammenfassung Die vorliegende Arbeit befasst sich mit einer empirischen Analyse eines Algorithmus zur Bestimmung starker Zusammenhangskomponenten in gerichteten Graphen. Der untersuchte Algorithmus ist ein modifizierter Tiefensuchalgorithmus mit einer Laufzeit in $O(\mid V \mid + \mid E \mid)$. Wir ermitteln mit Hilfe linearer Regression auch Koeffizienten, die beschreiben, wie stark die Knoten- und Kantenzahl die Laufzeit beeinflussen.

1 Einleitung

Gegenstand unserer Untersuchung ist das Auffinden stark zusammenhängender Komponenten in gerichteten Graphen. Die Fragestellung ist interessant, weil eine Klasse von Algorithmen besteht, die eine Dekomposition der zu behandelnden Graphen in stark zusammenhängende Komponenten voraussetzt.
Wir stellen zunächst einen Algorithmus zur Lösung dieses Problems vor. Der Großteil dieses Artikels beschäftigt sich mit der Implementation dieser Lösung basierend auf einem generischen Tiefensuchalgorithmus und den Ergebnissen einer empirischen Analyse dieser Implementation.
In Abschnitt 2 führen wir grundlegende Begriffe und Definitionen ein. In den beiden darauffolgenden Abschnitten präsentieren wir den Algorithmus und die Implementation. In Abschnitt 5 stellen wir un-

sere Ergebnisse vor und im letzten Abschnitt bieten wir noch einen Ausblick.

2 Definitionen

In diesem Artikel verstehen wir unter einem Graphen stets einen einfachen und gerichteten Graphen. Wir beginnen mit folgender

Definition 1
Für einen Graphen G = (V, E) erklären wir eine Äquivalenzrelation $A \subset V \times V$ auf der Knotenmenge V durch folgende Eigenschaft:
Zwei Knoten v, w sind äquivalent, wenn ein gerichteter Pfad vom Knoten v zum Knoten w sowie ein Pfad vom Knoten w zum Knoten v existiert.

Das Nachprüfen der Eigenschaften einer Äquivalenzrelation sei hier ausgelassen.

Bezeichnung 1
Wir nennen die Äquivalenzklassen, die durch A auf der Knotenmenge gegeben sind, die stark zusammenhängenden Komponenten des gerichteten Graphen G.

Bezeichnung 2
Ein Graph heißt stark zusammenhängend, wenn die Knotenmenge V eine stark zusammenhängende Komponente ist.

3 Algorithmus

Algorithmus FindeStarkZusammenhängendeKomponente

Eingabe:
- G=(V,E) ein gerichteter Graph
- v ∈ V ein Knoten
- n Anzahl der Knoten in G

Ausgabe:
- markiert die stark zusammenhängenden Knoten in G

```
1.   {   //Initialisierung
2.       Für alle Knoten v des Graphen führe aus{
3.            v.tsNummer = 0;   //Nummer der Tiefensuche
4.            v.komponente = 0;   //Nummer der Komponente
5.       }
```

```
 6.          aktuelleKomponente = 0;
 7.          tsNummmer = n;
 8.          Solange ein Knoten v mit v.tsNumber = 0 existiert{
 9.                scc(v)
10.          }
11.      }
```

Prozedur scc(v)

```
 1.      {
 2.          v.tsNummer = tsNummer;
 3.          tsNummer = tsNummer-1;
 4.          Füge v in den Stack:
 5.          v.hoch = v.tsNummer;
 6.          Für alle Kanten (v,w) führe aus{
 7.              falls w.tsNummer = 0 dann{
 8.                  scc(w);
 9.                  v.hoch = max(v.hoch, w.hoch);
10.              }
11.              sonst{
12.                  falls w.tsNummer > v.tsNummer
                     und w.komponente = 0 dann{
                     // (v,w) ist eine Kreuzungs- oder Rückwärts-
                     // kante, die wir berücksichtigen müssen
                     v.hoch = max (v.hoch, w.tsNummer);
13.                  }
14.              }
15.          }
16.              falls v.hoch = v.tsNummer dann{
                 // v ist die Wurzel einer Komponente
                 aktuelleKomponente = aktuelleKomponente + 1;
17.              wiederhole{
                 //markiere die Knoten der neuen Komponente
18.                  lösche v vom Stack
19.                  x.komponente = aktuelleKomponente;
20.              } bis x = v;
21.              }
22.          }
```

Wir beschränken uns in diesem Artikel auf die Angabe der Idee des Korrektheitsbeweises. Der Algorithmus besucht alle Knoten in der Reihenfolge, die durch die Tiefensuche vorgegeben ist. Wie immer bei gerichteten Graphen muß die Rekursion gegebenenfalls öfter

aufgerufen werden, da nicht alle Knoten notwendigerweise in einem Durchlauf markiert werden. Der Algorithmus ermittelt iterativ die Komponente, die im Tiefensuchbaum unten links zu finden ist. Ob ein Knoten Wurzel dieser Komponente ist wird über die *hoch*-Werte entschieden. Die Laufzeit des Algorithmus liegt in $O(\mid V \mid + \mid E \mid)$. Für einen genauen Beweis des Algorithmus und der Laufzeit verweisen wir auf [Man89].

4 Implementation

Bei genauem Hinsehen entpuppt sich der Algorithmus als Anwendung des folgenden generischen Tiefensuchalgorithmus:

Algorithnmus: Generischer Tiefensuchalgorithmus

Eingabe: – G = (V,E) ein gerichteter Graph
– v ein Knoten
Ausgabe: – Die Ausgabe ist abhängig von Anwendung.

```
1.  {
2.  markiere v;
3.  Vorarbeit (v);
4.  Für alle Knoten v aus V führe aus{
5.      Wenn w nicht markiert ist, dann{
6.          Generische Tiefensuche(w);
7.      }
8.      Sonst wenn (v,w) Rückwartskante, dann{
9.          behandleRückwartskante(v,w);
10.     }
11.     Sonst wenn (v,w) eine Kreunzungskante ist{
12.         behandleKreunzungskante (v,w);
13.     }
14.     Nacharbeit (v,w);
15.     }
16. }
```

Dieser Algorithmus kann mit Hilfe des Entwurfsmusters “Schablonenmethode” aus [Gam95] an die jeweilige Aufgabenstellung angepasst werden. Dabei werden die von der Anwendung abhängigen Prozeduren Vorarbeit(v), Nacharbeit(v,w), behandleRückwärtskante(v,w) und behandleKreunzungskante(v,w) domänenbezogen implementiert.

Wir spezifizierten den Algorithmus FindeStarkZusammenhängende-Komponenten dementsprechend als spezielle Anwendung dieses generischen Algorithmus.
Die Prozedur Vorarbeit(v) weist in unserem Fall den Variablen Initialisierungswerte zu. In behandleRückwärtskante(v), behandleKreuzugskante(v) und Nacharbeit(v) werden die *hoch*-Werte gegebenenfalls aktualisiert. Zusätzlich wird in Nacharbeit(v) mittels den *hoch*-Werten noch bestimmt, ob der Knoten v die Wurzel einer stark zusammenhängenden Komponente im Tiefensuchwald ist und diese dann markiert.
Wir verwendeten bei der Implementation die Java-Bibliothek JDSL (siehe [jdsl]), welche den generischen Algorithmus zur Verfügung stellt.

5 Ergebnisse

5.1 Versuchsaufbau

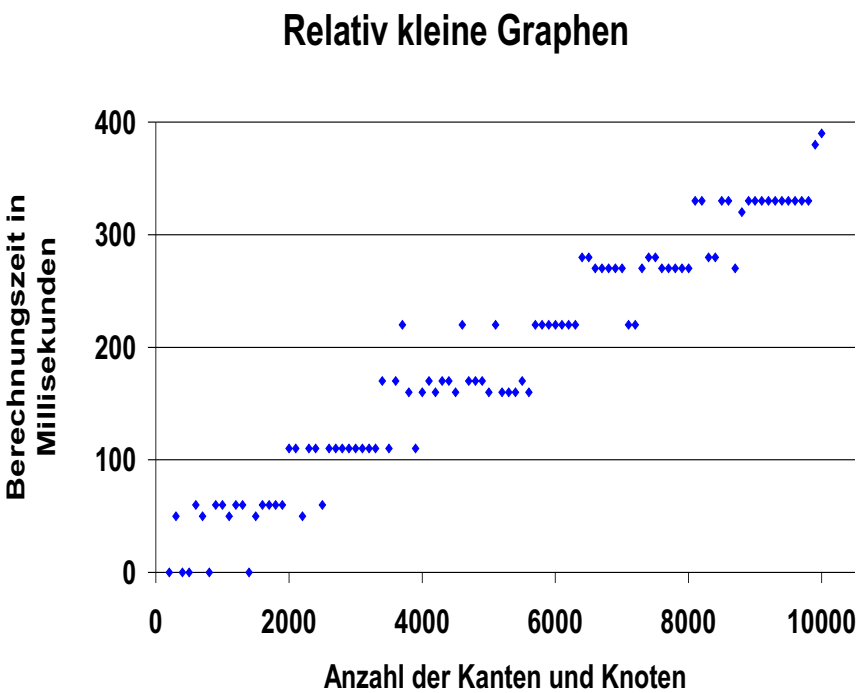
Für die von uns durchgeführten Tests verwendeten wir folgende Rechnerarchitektur:

- Prozessor: Athlon mit 700 MHZ
- RAM: 256 MB
- Betriebssystem: Win98
- Programmiersprache: Java (JDK1.2)

Als Datensätze verwendeten wir selbst erzeugte Graphenserien mit 100 bis 70000 Knoten und variierender Kantenzahl.
Bei der im nächsten Abschnitt folgenden Besprechung der Ergebnisse gehen wir auch auf die Struktur dieser Graphenserien ein.

5.2 Laufzeit

Unsere empirische Untersuchung verifizierte die Laufzeitannahme von $O(\mid V \mid + \mid E \mid)$. Insgesamt haben wir auf mehr als 40 verschiedenen Graphenserien getestet. Darunter waren Serien mit vollständigen Graphen, strukturierte Graphen (z.B. zwei Kanten pro Knoten) und Zufallsgraphen. Als Beispiele präsentieren wir im folgenden die Ergebnisse einiger dieser Graphenserien.

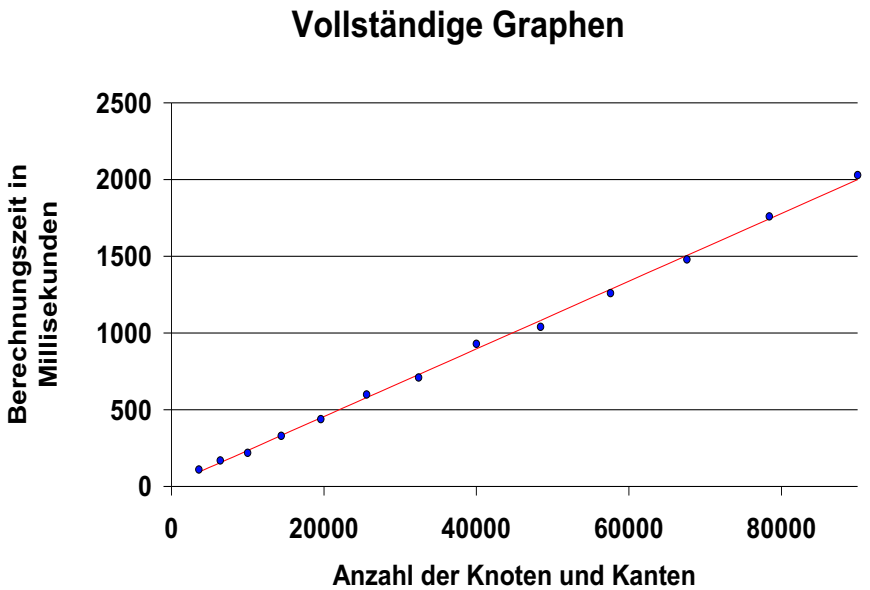


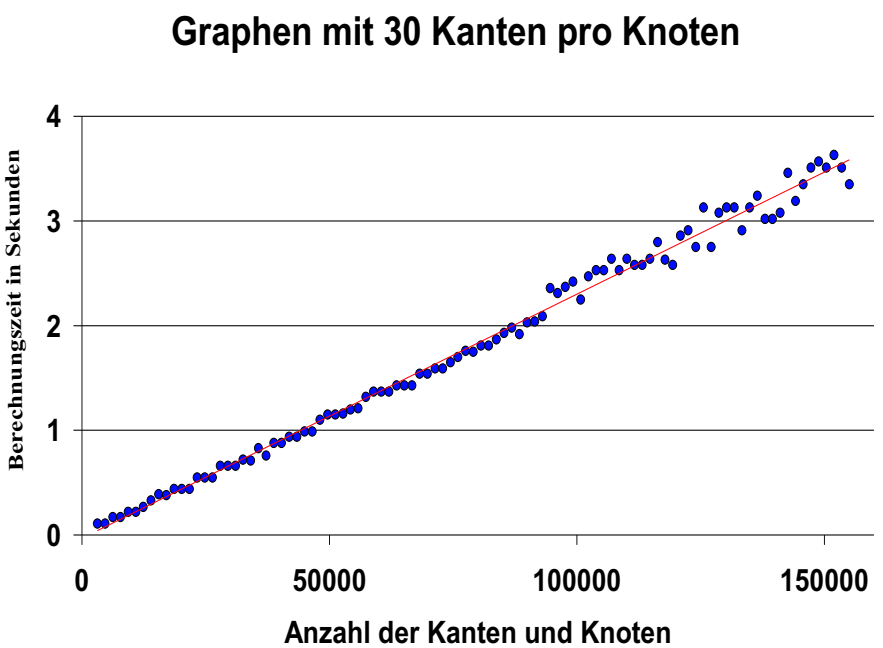
In obiger Graphik sieht man, daß die Laufzeitmessungen für kleine Graphen keinen Sinn macht, da die Verfälschungen durch die Auflösung die Ergebnisse zu stark beeinflussen, wie beispielsweise Laufzeiten von 0 zeigen. Für unsere Versuchsanordnung stellten wir fest, daß erst ab ungefähr 5000 Kanten und Knoten durch das Betriebssystem verursachte Schwankungen im Ergebnis vernachlässigbar werden.

Als obere Grenze für unsere Architektur erwiesen sich Graphen, die insgesamt 150000-180000 Knoten und Kanten haben, weil dann das System die Festplatte als virtuellen Arbeitsspeicher benutzte und diese Auslagerungen die Laufzeit negativ beeinbeträchtigen.

Als weiteres Problem erwies sich die automatische Speicherverwaltung von Java. Erst als wir die Speicherfreigabe zu einem genau definierten Punkt im Programmfluß unmittelbar vor der Prozedur SCC erzwungen haben, konnte die Verfälschung der Laufzeit ausgeschaltet werden.

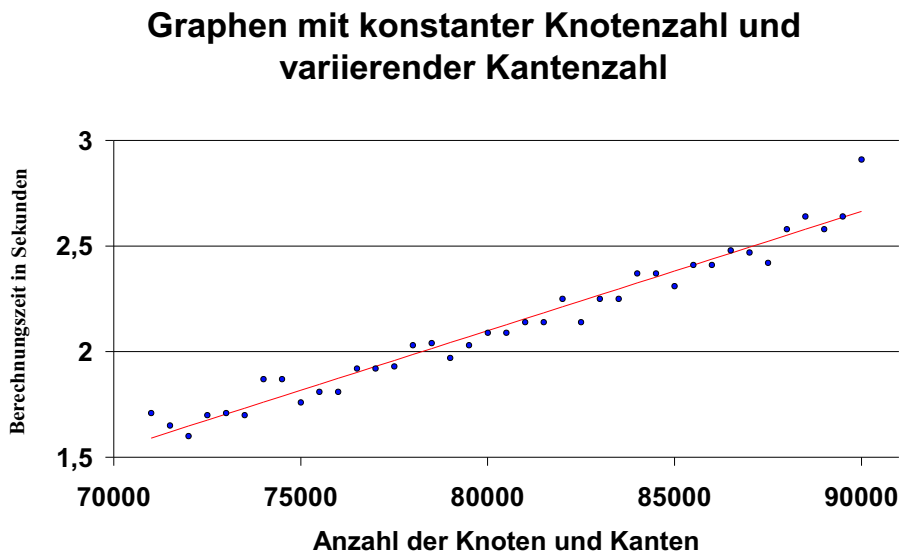
Nachdem wir diese Probleme eliminiert hatten, konnten wir gute Ergebnisse erzielen. Die nächsten zwei Graphiken illustrieren dies sehr deutlich, wobei die erste Graphik die Resultate einer Graphenserie mit vollständigen Graphen und die zweite Graphik die Resultate mit Graphen mit 30 zufällig gewählten Kanten pro Knoten wiedergibt.

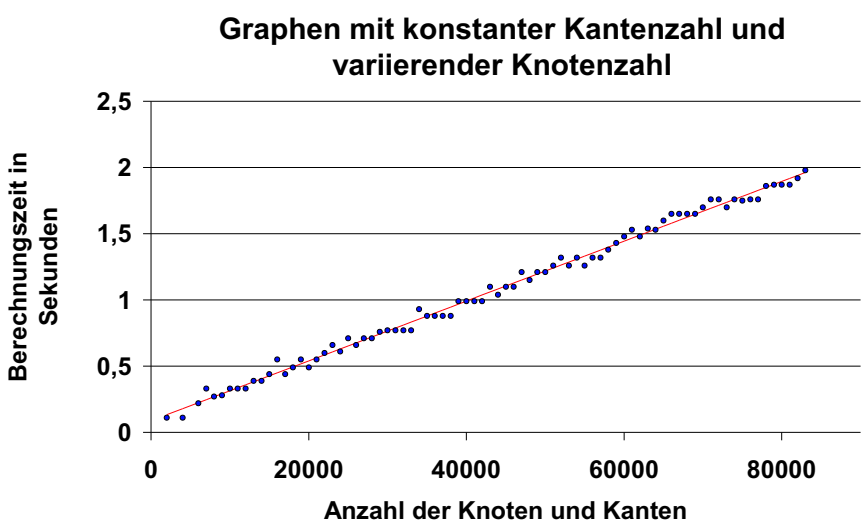

Vollständige Graphen
Berechnungszeit in Millisekunden
2500
2000
1500
1000
500
0
0
20000
40000
60000
80000
Anzahl der Knoten und Kanten


Graphen mit 30 Kanten pro Knoten
Berechnungszeit in Sekunden
4
3
2
1
0
0
50000
100000
150000
Anzahl der Kanten und Knoten

5.3 Detailanalyse

Ferner stellten wir Untersuchungen an, die ermitteln sollten wie das Verhältnis Kanten zu Knoten sich in der Laufzeit auswirkt.
Dazu erzeugten wir Graphenserien, die jeweils nur in der Knoten- oder Kantenzahl variierten. Die Ergebnisse zweier solcher Serien sind in den beiden folgenden Abbildungen dargestellt:





Mittels linearer Regression ermittelten wir die Steigungen der Ausgleichsgeraden. Durch die Implementation einer Routine, die die lineare Regression ausführt und die Steigung dieser Geraden ausgibt, haben wir diesen Vorgang automatisiert.
Der Steigungskoeffizient gibt an wie stark die Anzahl der Kanten beziehungsweise Knoten die Laufzeit in unserer Versuchsanordung beeinflussen. In den Steigungskoeffizienten geht natürlich als Konstante die Rechnerarchitektur ein, da die Anzahl der Operationen pro Sekunde, die der Rechner ausführen kann, plattformabhängig ist. Der Quotient aus diesen beiden Koeffizienten ist allerdings weniger von der Rechnerarchitektur abhängig, weil die Einflüsse der Plattform auf die Koeffizienten im Zähler und Nenner vergleichbar groß sind.
Für unsere Architektur erhielten wir folgende Ergebnisse: Bei konstanter Kantenzahl ist der Koeffizient c_1 für die Knoten $\approx 5{,}64$; bei konstanter Knotenzahl ist der Koeffizient c_2 für die Kanten $\approx 2{,}25$. Für das Verhältnis von c_1 und c_2 gilt:

$$\frac{c_1}{c_2} \approx 2{,}49.$$

Dies bedeutet, daß die Knotenzahl mehr als doppelt so stark wie die Kantenzahl in die Laufzeit eingeht. Dies Ergebnis scheint uns für diesen Algorithmus als Anwendung der generischen Tiefensuche plausibel zu sein.

6 Schluß

Wir haben nun empirisch die Laufzeit verifiziert und auf die Probleme einer solchen Analyse dargestellt.
Abschließend wollen wir noch darauf hinweisen, daß mit einer leichten Modifikation unseres präsentierten Algorithmus auch Graphen auf zweifach zusammenhängende Komponenten durchsucht werden können. Beides sind Fragestellungen innerhalb des Themenkomplexes Graphendekomposition, aus dem wir nur ein Beispiel empirisch untersuchten.

Literatur

[Man89] Manber, Udi *Introduction into Algorithms* (Addison-Wesley, Massachusetts 1989) p. 217-234

[Gam95] Gamma, Erich et al. *Design Patterns* (Addison-Wesley, Massachusetts 1995) p. 325-330

[jdsl] JDSL-Site *http://www.cs.brown.edu/cgc/jdsl/*

Praktikum Algorithmen und Datenstrukturen
Team Nr. 2

Starke Zusammenhangskomponenten

Aufspüren starker Zusammenhangskomponenten durch Zykelsuche in Graphen

Marco Gaertler, Roland Martin

Universität Konstanz

Sommersemester 2000

Zusammenfassung Ziel der Untersuchung ist es, ein Linearzeitalgorithmus zu finden, der in einem gerichteten Graphen starke Zusammenhangskomponenten findet. Unsere Strategie baut dabei auf gerichtete Zykelsuche im Graphen zusammen mit einer Union–Find–Struktur auf. Wir haben den gefundenen Algorithmus ebenfalls implementiert.

1 Einleitung

Das Problem starke Zusammenhangskomponenten in einem gerichteten Graphen zu finden, taucht in verschiedenen Teilgebieten der Algorithmik auf. z.B. Visualisierung oder Dekomposition für weitere Verfahren auf den einzelnen Komponenten. Ziel der Untersuchung ist es, ein Linearzeitalgorithmus zu finden, der in einem Graphen starke Zusammenhangskomponenten findet. Gerichtete Zykelsuche zusammen mit einer modifizierten Union–Find–Struktur stellte dabei das Grundgerüst für unsere Implementation.

2 Problemstellung

Um Einblicke in die Thematik und Problemstellung zu bekommen, benötigen wir folgende Definitionen:

Definition 1
In einem gerichteten Graphen $G = (V, E)$ heißt eine Teilmenge V' von V stark zusammenhängend, *wenn für jedes Knotenpaar aus V' ein gerichteter Kreis in G existiert, auf dem beide liegen.*

Definition 2
Sei $G = (V, E)$ ein gerichteter Graph und $V_1, \ldots, V_n \subseteq V$. Dann heißt $V_1, \ldots, V_n$ eine Knotenpartition *von G, falls folgende Bedingungen erfüllt sind:*

- *Partitionen sind paarweise disjunkt*

$$i \neq j \quad \Longrightarrow \quad V_i \cap V_j = \emptyset$$

- *Vereinigung aller Partitionen ergibt ganz V*

$$\bigcup_{i=1}^{n} V_i = V.$$

Problem 3
Gegeben sei ein gerichteter Graph $G = (V, E)$. Gesucht ist eine Partition $V_1, \ldots, V_n$, so daß jedes V_i stark zusammenhängend und n minimal ist.

Bemerkung: Die Bedingung, daß n minimal ist, bedeutet gerade, daß die V_i knotenmaximal sind.

3 Technische Hilfsmittel

3.1 Tiefensuche

Möchte man einen Graphen komplett traversieren, so gibt es mehrere Strategien dies zu tun. Eine davon ist die Tiefensuche (**D**epth **F**irst **S**earch). Dabei geht man solange einen Weg entlang, bis man nicht mehr weiterkommt und sucht sich dann Alternativwege durch sogenanntes *Backtracking*.

Der Algorithmus DFS benötigt noch eine Knotenmarkierung, um zwischen besuchten und unbesuchten Knoten zu unterscheiden.

DFS(G, s)

1. für alle $v \in V$ führe aus
   setze $Nr(v) := 0$
2. setze $i := 0$, $v := s$, $Nr(s) := 1$

3. wiederhole
   (a) solange es zu v inzidente Knoten $w \in V$ mit $Nr(w) = 0$ gibt, führe aus
       wähle ein solches w und führe aus:
       $i := i + 1$, $Nr(w) := i$, $v := w$
   bis $v = s$ und $Nr(w) \neq 0$ für alle zu s inzidenten Knoten

Der Algorithmus bekommt einen Graphen $G = (V, E)$ und einen Startknoten s als Eingabeparameter. Beginned von s aus wird nun ein Weg zu einem inzidenten Knoten gewählt, dieser markiert und iterativ so fortgefahren, bis wir an einem Knoten angekommen, der keine unmarkierten Nachfolger mehr besitzt. Durch Backtracking wird dann von seinem Vorgänger aus wieder ein Weg zu einem inzidenten unmarkierten Knoten gesucht. Der Algorithmus endet, sobald alle Knoten markiert sind.
Bemerkung: Im gerichteten Fall muß die Inzidenz die Kantenrichtung respektieren. Dadurch ergibt sich im Allgemeinen kein Baum sondern ein Wald. D.h. eventuell kann es vorkommen, daß eine Tiefensuche, die bei einem bestimmten Knoten startet, nicht alle Knoten erreicht. Um aber den kompletten Graphen zu traversieren ist es notwendig, diese Prozedur von einem weiteren (unmarkierten) Knoten aus zu starten.

4 Union-Find

4.1 Abstrakte Beschreibung

Eine geeignete Datenstruktur zur Repräsentation von Partitionen einer Menge ist die Union-Find-Struktur. Hier wird ermöglicht, einzelne Elemente der Menge zu Partitionen zu gruppieren (MakeSet), zwei Partitionen zu einer neuen zu vereinigen (Union) und die Zuordnung der einzelnen Elemente zu ihren Partitionen festzustellen (Find).

4.2 Realisierung

Üblicherweise wird Union–Find durch einen Vorgängerwald realisiert. Dabei entspricht jede Partition einem Wurzelbaum. Die Wurzel fungiert hierbei als Repräsentant und alle anderen Elemente der Partition verweisen (direkt oder indirekt) auf diesen Repräsentanten.

Bei der Union–Operation werden nun beide Teilbäume verschmolzen, in der die Wurzel des ersten Teilbaums Nachfolger eines Knotens des zweiten Teilbaums wird.

Die Find–Operation nutzt die vorhandene Vorgängerrelation aus und hangelt sich so im Baum bis zur Wurzel nach oben und liefert diese Wurzel als Repräsentant der Partition, indem sich der Knoten befindet, zurück.

Allgemein benötigen m Union–Operationen und n Find–Operationen in beliebiger Reihenfolge $\mathcal{O}((n+m) \cdot G(n))$. Es gibt verschiedene Beschleunigungstechniken, die die Komplexität herabsetzen. Im Folgendem verwenden wir die Pfadkompression. Dabei werden bei jeder ausgeführten Find-Operation alle Elemente, die sich auf demWeg von dem gegebenen Element zur Wurzel befinden, direkt unter die Wurzel gehängt.

5 Entwicklung und Algorithmus

Zur Auffindung starker Zusammenhangskomponenten benutzen wir folgende intuitive Idee: Alle Knoten, die im Graphen auf einem gerichteten Kreis liegen, sind in der gleichen starken Zusammenhangskomponente. Um nun die Partition geeignet darzustellen, wird die Union-Find-Struktur benutzt. Dabei stellt am Anfang der Berechnung jeder Knoten eine eigene Partition dar. Jedesmal wenn ein gerichteter Zykel gefunden wird, werden die zu den Knoten gehörenden Partitionen mittels einer Union–Operation vereinigt. Dies liefert eine grobe Beschreibung der Vorgehensweise. Für eine detailiertere Beschreibung benötigen wir zusätzliche Hilfsstrukturen:

- einen Repräsentantenstack
- einen Hilfsstack zur effizienten Arbeit mit der Union–Find–Struktur
- eine Knotenmarkierung für endgültige Partition

Das Finden von Zykeln kann bekannterweise durch eine Tiefensuche realisiert werden. Wir verwenden aber nur eine Tiefensuche zur Berechnung aller relevanten Zykel. Die primäre Idee dabei besteht darin: Sobald der aktuelle Tiefensuchpfad einen DFS–markierten Knoten erreicht, schließt dieser einen gerichteten Zykel oder gehört zu einer schon fertigbearbeiteten Komponente. Im ersteren Fall können wir schließen, daß alle in diesem Pfad enthaltenen Knoten in einer Partition sind. Der letztere Fall dagegen impliziert das Betreten einer schonfertig bearbeiteten Komponente und löst deswegen einen Backtracking–Schritt aus.

Es ergibt sich folgende Abwandlung des DFS–Rekursionsschritts:

Prozedur FindSCC(v)

1. markiere den Knoten v

2. für alle Kanten $\{v, w\}$ führe aus
   (a) falls w nicht markiert ist
      i. make-set(w)
      ii. lege w auf den Repräsentanten-Stack
      iii. DFS(w)
      iv. falls top(Repräsentanten-Stack) $= w$,
      dann entferne w vom Stack
   (b) sonst
   vereinige v mit allen Knoten auf dem Repräsentanten-Stack
   bis Top(Repräsentanten-Stack) oberhalb oder gleich w

Problem: Ein Problem tritt bei mehrfachgeschachtelten Zykeln auf. Seien etwa $C_1, \ldots, C_k$ solche nicht kantendisjunkte Zykeln. Das Vereinigen der Knoten auf dem Stack kann zu einem Problem führen, wenn gewisse Zykeln $\{C_j;$ für gewisse j mit $1 \leq j \leq k\}$ schon vollständig abgearbeitet worden sind. Dann kann es passieren, daß bei der Vereinigung eines weiteren Zykels die Abbruchbedingung nicht mehr greift, da er selbst nicht mehr auf dem Stack liegt, sondern nur noch sein Repräsentant (siehe 9.1). Dieses Problem kann dadurch behoben werden, daß die Abbruchbedingung wie folgt abgeändert wird:

> *vereinige v mit allen Knoten auf dem Repräsentantenstack,*
> *bis top(Repräsentantenstack) oberhalb oder gleich find(w)*
> *ist*

Wie wir oben gesehen haben, ist das Ausführen der Find-Operationen nicht in konstanter Zeit. Wir geben nun eine "schnellere" Vorgehensweise an, in der wir auf die Find-Operationen verzichten.

Ein weiteres Problem tritt bei betreten fertig bearbeiteter Komponenten auf. In diesem Fall muß zusäztlich getestet werden, ob der Repräsentant überhaupt auf dem Stack liegt (siehe 9.2).

Festzustellen, ob eine Komponente fertig bearbeitet worden ist würde normalerweise mindestens auch ein Find benutzen.

6 Finaler Algorithmus

Zwei wesentliche Änderungen gegenüber den fehlerbehafteten Versionen treten hier in Aktion:

- Die Hilfsstruktur "PredStack". Dieser dient zum Aktualisieren der Union-Find-Struktur, damit kann man in konstanter Zeit testen, ob der Vorgänger eines Knoten der Repräsentant einer fertig bearbeiteten Komponente ist. (siehe Schritt 3(d)ii)

– Die Markierung “final”. Ein Komponente kann während des Algorithmuses nicht mehr wachsen, wenn ihr Repräsentant vom CycleStack genommen wird. Das Testen ob die aktuelle Komponete fertig ist (siehe Schritt 3(f)ii) konnte erst durch die aktualisierte Union-Find-Struktur (mittel PredStack) in konstanter Zeit realisiert werden)

FindSCC

1. markiere v
2. für alle $e : (v, w) \in E$ unmarkiert, führe aus
3. falls w unmarkiert ist
   (a) CycleStack.push(w)
   (b) PredStack.push(w)
   (c) DFS(w)
   (d) falls DFS(w) Repräsentant
      i. markiere w als final
      ii. führe aus: der Vorgänger aller Knoten oberhalb oder gleich w wird auf w gesetzt
   (e) falls CycleStack.top == w
      nehme w vom CycleStack
   (f) sonst
      i. sei u der Vorgänger von w
      ii. falls u nicht final führe aus:
         vereinige alle Knoten auf dem CycleStack mit w, die eine kleinere DFS-Nummerierung haben als w

7 Auswertung

Sämtliche Testreihen wurden auf einem System mit folgenden Eigenschaften durchgeführt:

Modell:	Sun (Sun Microsystems), Enterprise 4000/5000
CPU:	336 MHz SUNW,UltraSPARC-II
Hauptspeicher:	1024 MB
OS:	SunOS Release 5.6
JAVA:	1.2.1,
	Solaris VM (build Solaris_JDK_1.2.1_04, native threads, sunwjit)

7.1 Auswertung I: vorgegebene Graphen

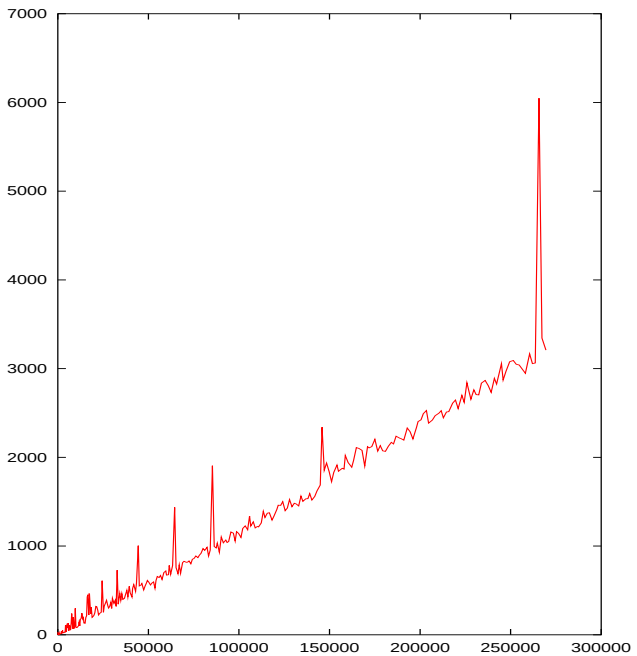


Abb. 1 Messungen der vorgegebenen Graphen

Die Diagramm 1 und 2 zeigen unsere Messungen bezüglich den vorgegebenen Graphen (graphA2.jar). Dabei wurde auf der X–Achse die Anzahl der Knoten plus die Anzahl der Kanten aufgetragen, auf der Y–Achse die benötigte Zeit in Millisekunden unseres Algorithmuses. Auffällig sind gemeinsame Spitzen, deren Ursache wir jedoch noch nicht weiter untersucht haben. Ebenso kann man im Vergleich erkennen, daß nicht jeder Ausreißer signifikant ist, sondern durch temporäre Effekte (weiter Prozesses, u.ä.) entstanden ist.

7.2 Auswertung II: eigene Testreihe

Das Diagramm 3 zeigt unsere Messung bezüglich unserer eigene Graphserie. Dabei wurde auf der X–Achse die Anzahl der Knoten plus

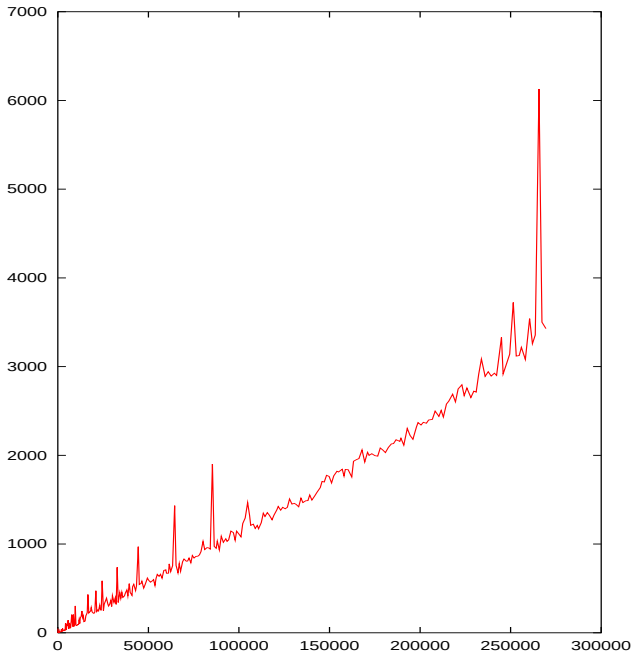


Abb. 2 Messungen der vorgegebenen Graphen

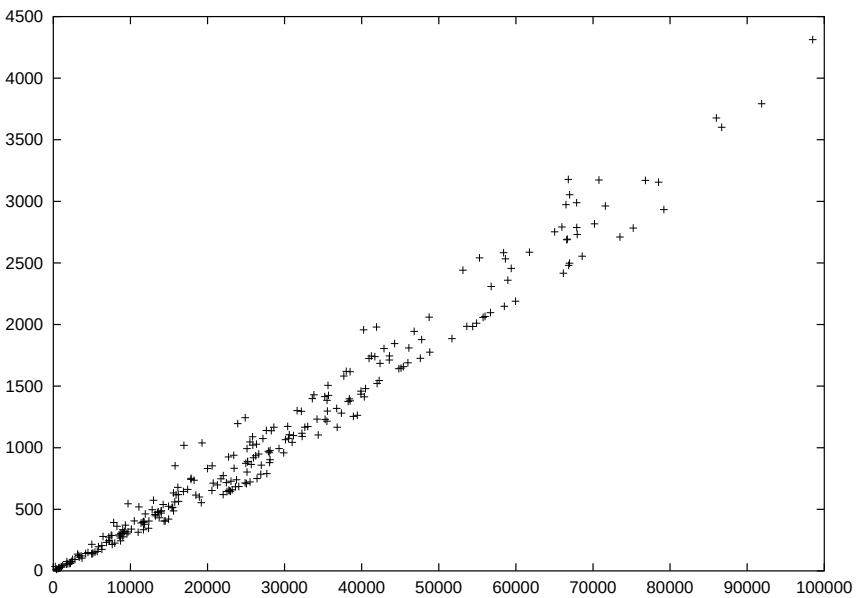


Abb. 3 Messungen der eigenen Graphen

die Anzahl der Kanten aufgetragen, auf der Y–Achse die benötigte Zeit in Millisekunden unseres Algorithmuses. Die Berechnungen wurden auf dem Rechner “ramsen” ausgeführt.

Die Graphen sind alle zufällig erstellt worden, aber mit der Beschränkung, daß höchstens 20 mal soviele Kanten vorhanden sind wie Knoten. Die kontrete Anzahl den Kanten wurden auch wieder per Zufall bestimmt. Deswegen erhält man auch nicht mehr eine so gerade Linie, sondern eher ein Feld. Aus dem Diagramm sollte aber erkennbar sein, daß sowohl die Schranken als auch ein Median linear verläuft.

8 Ausblick: Zweifachzusammenhangskomponenten

Sei $G = (V, E)$ ein ungerichteter Graph. Wir betrachten folgende Relation $\sim$ auf E:

$$e \sim e' : \iff e \text{ und } e' \text{ liegen auf einem einfachen Kreis.}$$

Diese Relation $\sim$ wird unter der Benutzung folgender Beobachtung zu einer Äquivalenzrelation: eine Kante e liegt stets mit sich selber auf einem einfachen Kreis. Dies soll nur für unsere Relation gelten, jedoch nicht im Allgemeinen (da sonst Bäume nicht azyklisch wären).

Definition 4
Die von der Äquivalenzrelation $\sim$ induzierte Kantenpartition nennt man die Zweifachzusammenhangskomponente *von G.*

Bemerkung: Zweifachzusammenhangskomponenten kann man auch wie folgt beschreiben: Es sind die maximalen kanteninduzierten Subgraphen von G, die nach entfernen eines beliebigen Knotens noch zusammenhängend sind.

Dies bedeutet, daß ein Knotenpaar in einer Zweifachzusammenhangskomponente durch einen Kreis verbunden ist. Dieses Problem ist dual zu unserem Problem, wenn man die Knoten durch Kanten ersetzt und die Kantenrichtung ignoriert.

Mittels der letzten Bemerkung können wir unseren Algorithmus – mit leichten Modifikationen – zur Suche nach den Zweifachzusammenhangskomponenten eines ungerichteten Graphen benutzen.

Leider kann man weder den ungerichteten Graphen benutzen noch zu einem "dualen" Graphen übergehen, wobei sich dual darauf bezieht Kanten durch Knoten zu bezeichnen und umgekehrt. Ein Problem ist, dass man kantendisjunkte Zykel, die aber nicht knotendisjunkt sind, von nicht kantendisjunkten Zykeln zu trennen hat.

Literatur

[BETT99] Guiseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph drawing, algorithms for the visualization of graphs.* Prentices-Hall, Inc., 1999.

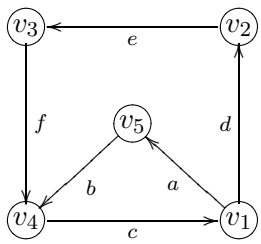
[Fou94] L. R. Foulds. *Graph theory applications.* Springer-Verlag, 1994.

[GT98] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in JAVA.* Wiley, 1998.

[Jun99] Dieter Jungnickel. *Graphs, Networks and Algorithms.* Springer-Verlag, 5 edition, 1999.

9 Anhang

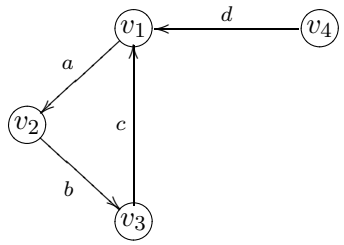
9.1 Beispiel: geschachtelte Zykel



Startknoten v_1:

Algorithmus:	Repräsentanten–Stack
DFS(v_1)	$\underline{v_1}$
markiere v_1	$\underline{v_1}$
Kante a und Knoten v_5 sind unmarkiert	$\underline{v_1}$
push v_5	$v_1, \underline{v_5}$
DFS(v_5)	$v_1, \underline{v_5}$
markiere v_5	$v_1, \underline{v_5}$
Kante b und Knoten v_4 sind unmarkiert	$v_1, \underline{v_5}$
push v_4	$v_1, v_5, \underline{v_4}$
DFS(v_4)	$v_1, v_5, \underline{v_4}$
markiere v_4	$v_1, v_5, \underline{v_4}$
Kante c ist unmarkiert	$v_1, v_5, \underline{v_4}$
Knoten v_1 ist markiert	$v_1, v_5, \underline{v_4}$
vereinige alle Knoten auf dem Repr.–Stack oberhalb oder gleich v_1 mit v_1	$\{\underline{v_1}, v_5, v_4\}$
Kante d und Knoten v_2 sind unmarkiert	$\{\underline{\boldsymbol{v_1}}, v_5, v_4\}$
push v_2	$\{\boldsymbol{v_1}, v_5, v_4\}, \underline{v_2}$
DFS(v_2)	$\{\boldsymbol{v_1}, v_5, v_4\}, \underline{v_2}$
markiere v_2	$\{\boldsymbol{v_1}, v_5, v_4\}, \underline{v_2}$
Kante e und Knoten v_3 sind unmarkiert	$\{\boldsymbol{v_1}, v_5, v_4\}, \underline{v_2}$
push(v_3)	$\{\boldsymbol{v_1}, v_5, v_4\}, v_2, \underline{v_3}$
DFS(v_3)	$\{\boldsymbol{v_1}, v_5, v_4\}, v_2, \underline{v_3}$
markiere v_3	$\{\boldsymbol{v_1}, v_5, v_4\}, v_2, \underline{v_3}$
Kante f ist unmarkiert	$\{\boldsymbol{v_1}, v_5, v_4\}, v_2, \underline{v_3}$
v_4 ist markiert	$\{\boldsymbol{v_1}, v_5, v_4\}, v_2, \underline{v_3}$
vereinige alle Knoten auf dem Repr.–Stack oberhalb oder gleich v_4 mit v_4	?

9.2 Beispiel: fertig–bearbeitete Komponente



Startknoten v_1:

Algorithmus:	Repräsentanten–Stack
DFS(v_1)	$\underline{v_1}$
markiere v_1	$\underline{v_1}$
Kante a und Knoten v_2 sind unmarkiert	$\underline{v_1}$
push v_2	$v_1, \underline{v_2}$
DFS(v_2)	$v_1, \underline{v_2}$
markiere v_2	$v_1, \underline{v_2}$
Kante b und Knoten v_3 sind unmarkiert	$v_1, \underline{v_2}$
push v_3	$v_1, v_2, \underline{v_3}$
DFS(v_3)	$v_1, v_2, \underline{v_3}$
markiere v_3	$v_1, v_2, \underline{v_3}$
Kante c ist unmarkiert	$v_1, v_2, \underline{v_3}$
Knoten v_1 ist markiert	$v_1, v_2, \underline{v_3}$
vereinige alle Knoten auf dem Repr.–Stack oberhalb oder gleich v_1 mit v_1	$\{\underline{v_1}, v_2, v_3\}$
neuer StartkKnoten v_4; push(v_4)	$\underline{v_4}$
DFS(v_4)	$\underline{v_4}$
markiere v_4	$\underline{v_4}$
Kante d ist unmarkiert	$\underline{v_4}$
Knoten v_1 ist markiert	$\underline{v_4}$
vereinige alle Knoten auf dem Repr.–Stack oberhalb oder gleich v_1 mit v_1	?

Praktikum Algorithmen und Datenstrukturen
Team Nr. 3

Starke Zusammenhangskomponenten

Stefan Klinger[1], **Odilo Oehmichen**[2], **Jonas Ritter**[3]

[1] klinger@fmi.uni-konstanz.de
[2] oehmiche@fmi.uni-konstanz.de
[3] ritter@fmi.uni-konstanz.de

Sommersemester 2000

Zusammenfassung Wir werden uns hier mit dem Auffinden von Starken Zusammenhangskomponenten in gerichteten Graphen befassen. Dazu betrachten wir einen linearen Algorithmus, dessen Implementation in Java sowie dessen Laufzeitverhalten.
Diese Arbeit entstand auf Grund eines Software-Praktikums an der Universität Konstanz am Lehrstuhl für Informatik von Frau Prof. Dr. Wagner.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Der Diplom-Studiengang Mathematik mit Schwerpunkt Informatik an der Universität Konstanz verlangt im Hauptstudium ein Software-Praktikum. Diesjähriges Thema am Lehrstuhl von Frau Prof. Dr. Wagner waren starke Zusammenhangskomponenten.
Unser Aufgabe bestand darin, einen in Pseudo-Code verfaßten linearen Algorithmus (siehe [1]) in Java zu programmieren und die Linearität der Implementation experimentell zu zeigen.

1.1 Definitionen

Stark zusammenhängend: Ein gerichteter Graph heißt stark zusammenhängend, falls für jedes Paar von Knoten v und w, ein Pfad von v nach w und von w nach v existiert, d.h. jeder Knoten kann von jedem anderen erreicht werden.
Starke Zusammenhangenskomponente(SCC): Eine SCC eines Graphen G ist eine maximale Untermenge der Knotenmenge von G, so daß der von ihr induzierte Subgraph stark zusammenhängend ist.

2 Der Algorithmus

Der Algorithmus arbeitet auf der Grundlage eines Tiefensuch-Verfahrens. Er erwartet als Eingabe einen gerichteten Graphen und gibt nach Abarbeitung aller Knoten die starken Zusammenhangskomponenten und die berechneten High-values aus. Dazu benutzt er die folgende Datenstruktur:

- akt_comp: speichert die aktuelle Komponentennummerierung (wird mit 0 initialisiert)
- Stack s
- DFS_N: aktuelle DFS-Nummer (wird mit n initialisiert)

Ausserdem erhält jeder Knoten im Graph folgende Eigenschaften:

- eine DFS-Nummer DFS-Nr (wird mit 0 initialisiert)
- eine Komponentenzugehörigkeitsnummer Comp_Nr (wird mit 0 initialisiert)
- einen High-value High

Für jeden Knoten im Graphen wird die Prozedur SCC aufgerufen. Diese Prozedur wird im Folgenden beschrieben: Nachdem sowohl die DFS-Nr von v als auch dessen High-value auf die aktuelle DFS-Nummer DFS_N gesetzt wurde und diese um eins erniedrigt wurde, wird v auf den Stack s gelegt. Nun wird für alle aus v herauslaufenden Kanten geprüft, ob es sich bei diesen um Kanten handelt, die schon zu besuchten Knoten (sprich Cross- oder Back-egdes) führen. Dies kann anhand der DFS-Nr des Endknoten ermittelt werden: Ist diese 0, so wurde der Knoten noch nicht besucht,so daß nun erst einmal die Prozedur SCC auf diesem Knoten ausgeführt wird.
Wurde der Knoten schon besucht, so wird, falls der Knoten noch zu keiner Komponente gehört (Comp-Nr ist 0) und seine DFS-Nr größer ist als die des aktuellen Knotens, der High value auf die DFS-Nr von w gesetzt, falls diese größer ist. Wurden alle aus v auslaufenden Kanten betrachtet, so wird überprüft, ob es sich um die Wurzel einer Komponente handelt. Dies ist der Fall, falls keine Kante zu einem Knoten führt, der über v liegt, d.h. der High-value von v mit seiner DFS-Nummer übereinstimmt. Handelt es sich also um eine Wurzel, so werden alle Knoten seiner Komponente markiert. Dies ist leicht möglich, da sie alle noch auf dem Stack liegen und nur gepoppt werden müssen.

3 Implementation in Java

Bei der Implementation wurde die Klassenbibliothek JDSL verwendet, deshalb lehnt sich das Klassendesign an diese an. Fundament der Architektur, wie auch des Algorithmus, bildet eine Tiefensuche auf gerichteten Graphen, DirectedDFS. Von ihr wird das Kernstück, die Klasse SCC welche den Algorithmus implementiert, abgeleitet. Sie überschreibt die folgenden Methoden der Klasse DirectedDFS: *startVisit(Vertex v), finishVisit(Vertex v)* sowie die vier Arten von Kantentraversierung *traverseBackEdge(Edge e, Vertex from), traverseForwardEdge(...), traverseCrossEdge(...)* und *traverseTreeEdge(...).*

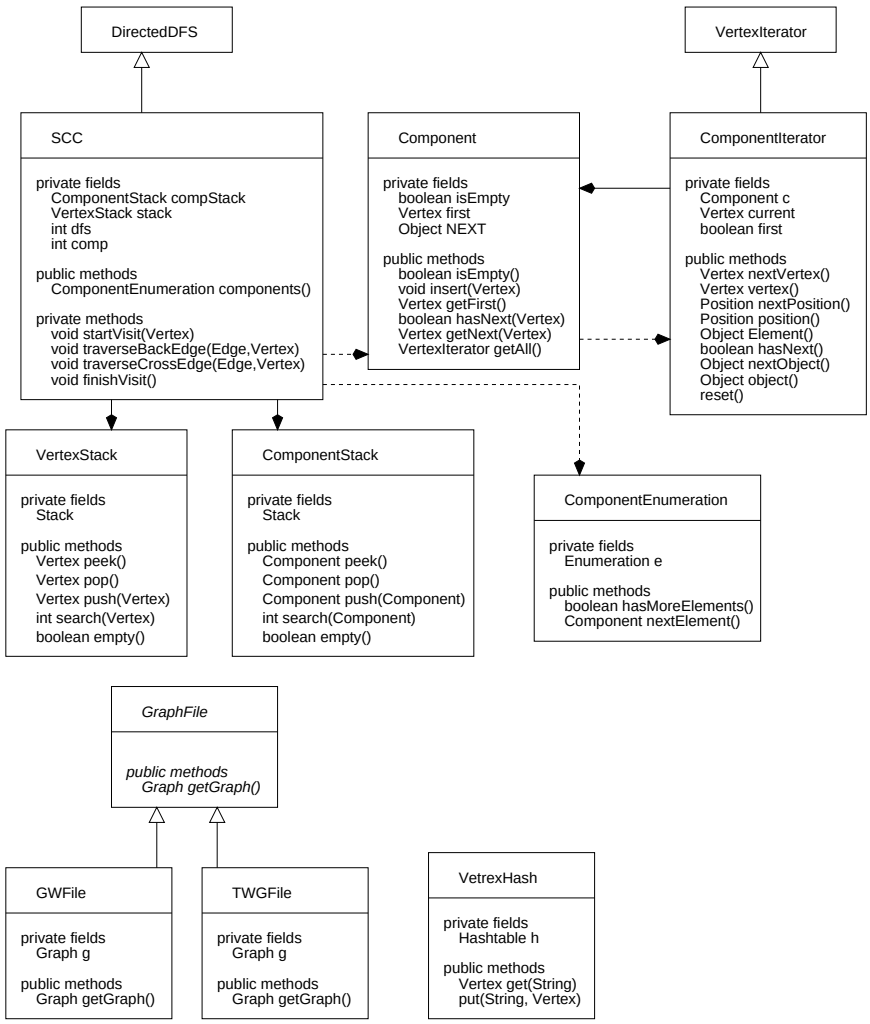


Abb. 1 Klassen Design

In diesen Methoden werden die notwendigen Operationen des Algorithmus ohne Einfluß auf die Tiefensuche hinzugefügt. Die gefundenen Komponenten werden in Instanzen der Klasse Component verwaltet, welche die Knoten in einer einfach verketteten Liste speichert. Diese Klasse bietet neben dem Einfügen von Knoten (*insert(Vertex v)*), dem Abfragen des ersten Knotens in der Komponente (*getFirst()*), dem Überprüfen, ob es weitere Knoten in der Komponente gibt (*hasNext(Vertex v)*, sowie dem Abfragen des Nächsten Knotens (falls vorhanden) (*getNext(Vertex v)*), die Möglichkeit alle Knoten der Komponente aufzuzählen(*getAll()*). Letzteres wird mit Hilfe eines vom

JDSL-Interface VertexIterator abgeleiteten Iterator (ComponentIterator) realisiert. Des weiteren stehen dem SCC Algorithmus typsichere Stackvarianten für Knoten (VertexStack) und Komponenten (ComponentStack) zur Verfügung. Der Stack für die Knoten ist der im Algorithmus beschriebene. Der Komponenten-Stack dient der Speicherung der gefundenen Komponenten und zur Erzeugung einer Komponenten-Enumeration nach Beendigung des Algorithmus. Eine weitere Gruppe von Klassen kümmert sich um das Auslesen von .jar Files und das Erzeugen von Graphen.

4 Laufzeitanalyse und Ergebnisse

4.1 Allgemein

Wie in [1] schon gezeigt, benötigt der Algorithmus Linearzeit in der Anzahl der Knoten und Kanten. Wir mußten also nur noch zeigen, daß unsere Implementation dies auch tut.

Dazu ließen wir den Algorithmus auf einer Vielzahl von Graphen operieren und beobachteten das dabei auftretende Laufzeitverhalten. Wir verwendeten unterschiedliche Klassen von Graphen, darunter auch zufällige Graphen, von deren Struktur wir keine weitere Kenntnis hatten[1]. Die Laufzeittests wurden dabei auf einem Rechner im Netz der Fakultät für Mathematik und Informatik der Universität Konstanz durchgeführt.

4.2 Eingabedaten

Die zu untersuchenden Graphen einer Klasse lagen in Form eines jar-Archives vor, in dem jeweils eine Datei einen Graphen repräsentierte. Die Graphen waren als Folge von Kanten in den Dateien gespeichert, wobei in jeder Zeile genau eine Kante durch die Integer Identifikationsnummern ihrer Endknoten beschrieben ist.

4.3 Laufzeitmessung

Die Klasse OTFTester läßt den Algorithmus auf Graphklassen im oben angegebenen Format arbeiten. Für jeden Graphen wird dabei die Anzahl der Knoten, der Kanten und der ermittelten starken Zusammenhangskomponenten, sowie die benötigte Zeit in Millisekunden ausgegeben.

[1] Wir erhielten, wie auch alle anderen Teilnehmer des Praktikums, die zwei Graphklassen graphA.jar und graphA2.jar zum Test unserer Implementation.

Zur Messung der benötigten Zeit wird zunächst der Graph geladen und eine Instanz des Algorithmus erzeugt. Dann wird die Systemzeit der Java-Machine in Millisekunden abgefragt und in der Variablen t_0 gespeichert. Der Algorithmus startet sofort danach. Sobald er terminiert, wird erneut die Systemzeit in der Variablen t_1 gespeichert. Die Differenz $t_1 - t_0$ ergibt die Laufzeit. Auf diese Art und Weise wird nur die Laufzeit der Methode, die den Algorithmus anstößt gemessen. Es wird also nicht erfaßt wieviel Systemzeit der Algorithmus tatsächlich verbraucht, was insbesondere auf Multitasking- oder gar Multiuser-Systemen wesentlich weniger sein kann, je nachdem wie oft der ausführende Prozess von anderen Prozessen verdrängt wurde (siehe dazu auch [2]). Eine bessere Methode zur Erfassung der Laufzeit stand uns leider nicht zur Verfügung.

4.4 Darstellung der Ergebnisse

Die so gewonnenen Daten wurden aufbereitet und mit Hilfe von GNUplot graphisch dargestellt. Einige der so erzeugten Diagramme sind im Folgenden abgebildet.

4.5 Ergebnisse

Betrachtet man die Laufzeitdiagramme der beiden Graphklassen graphA und graphA2, so läßt sich zwar eine lineare Laufzeit beobachten (s. Abb 2), jedoch fallen teilweise starke Schwankungen der Laufzeit auf (s. Abb 3).

Zuerst vermuteten wir als Ursache dieser Schwankungen Aktivitäten anderer User auf dem gleichen Computer. Diese These hatte jedoch keinen Bestand, da die gleichen Schwankungen auch auftraten, wenn während der Laufzeit kein anderer User eingeloggt war.

Um zu untersuchen, ob diese Schwankungen von den bearbeiteten Graphen, und damit von deren Struktur (Anzahl der (starken) Zusammenhangskomponenten, Vollständigkeit, Baum, etc.) und Größe (Anzahl der Knoten oder Kanten) abhiengen, ließen wir den Algorithmus wiederholt auf dem gleichen Graph arbeiten, und protokollierten das Laufzeitverhalten der einzelnen Durchläufe. Dabei wurde der vollständige Zyklus Laden, Algorithmus instanziieren, Algorithmus starten mit Laufzeitmessung, Ausgabe, jedes Mal vollständig ausgeführt.

Bei einem ersten Graphen mit je 2000 Knoten und Kanten beobachteten wir ein sehr seltsames Laufzeitverhalten: Die Laufzeitkurve beschreibt ein Zickzackmuster, der Algorithmus schien abwechselnd

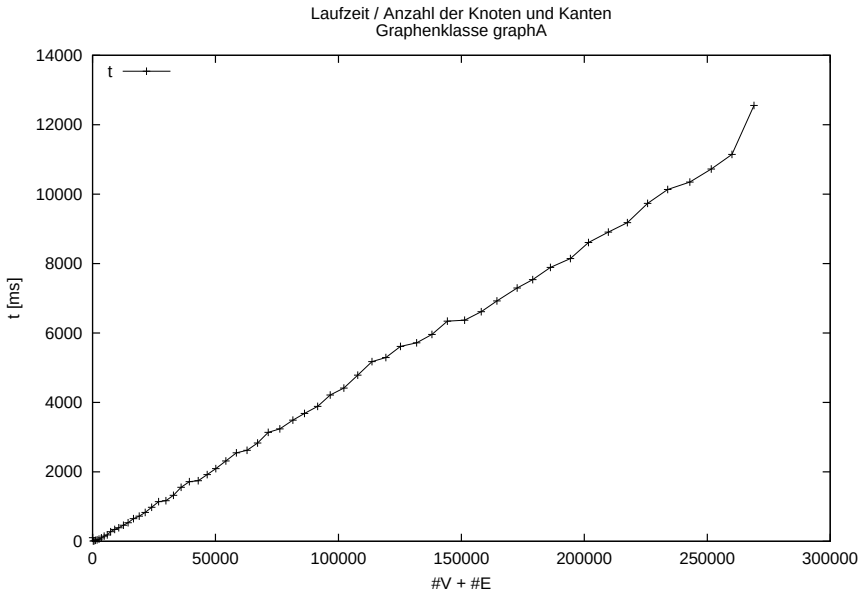


Abb. 2 Laufzeitdiagramm der Graphklasse graphA

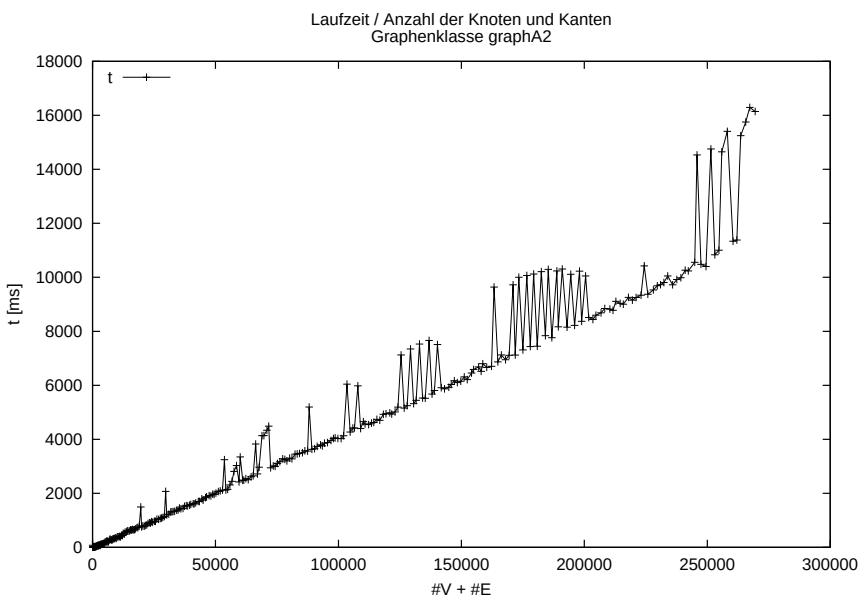


Abb. 3 Laufzeitdiagramm der Graphklasse graphA2

viel und wenig Laufzeit zu benötigen. Der erste Durchlauf bentigte außerdem exorbitant viel Laufzeit, was sich allerdings damit erklären läßt, daß die Java Virtual Machine (JVM) beim ersten Durchlauf noch nicht benötigte Klassen nachlädt, sie dann aber bei späteren Durchläufen bereits im Speicher hat.

Als Ursache für das Zickzackmuster vermuteten wir die Garbage Collection (GC) der JVM. Diese wird gelegentlich von der JVM ausgeführt, um nicht mehr benötigte Objekte vom Speicher zu entfernen. Eine genauere Aussage als 'gelegentlich' läßt sich nicht treffen, da die GC fast vollständig unter Kontrolle der JVM steht. Eine denkbare Erklärung für das beobachtete Verhalten wäre die Überlagerung der Frequenz, in der die GC ausgeführt wird, mit der Frequenz, die gerade einem Durchlauf des Algorithmus entspricht. Die GC läßt sich leider nicht abschalten, so daß man immer mit diesem Unsicherheitsfaktor rechnen muß. An dieser Stelle entzieht das Java-Konzept dem Programmierer die notwendige Kontrolle über die Maschine auf der das Programm läuft.

Allerdings kann man eine GC erzwingen. Also wurde die GC direkt vor dem Start des Algorithmus, aber noch außerhalb der Zeitmessung, explizit aufgerufen. Doch das Laufzeitverhalten änderte sich nicht wesentlich. Vielmehr erschienen die Zacken nun in einer größeren Regelmäßigkeit (s. Abb. 4). Auch hier schwankte die Laufzeit zwischen 100ms und 150ms. Dies zeigt, daß die GC zwar einen Einfluß auf die Laufzeit hat, aber noch nicht Ursache für unser eigentliches Problem ist.

Da man in die JVM nicht hineinsehen kann, waren wir auf Vermutungen angewiesen, und mehr aus Spielerei als mit sinnvollen Hintergedanken schalteten wir den Just In Time Compiler (JIT) der JVM ab. Der JIT übersetzt beim Laden einer Klasse deren Java Byte-Code in native Maschinensprache, so daß der Byte-Code nicht mehr interpretiert werden muß. Wir erwarteten das gleiche Ergebnis, jedoch entsprechend größer skaliert. Doch Abb. 5 zeigt eine insgesamt wesentlich höhere Laufzeit (zwischen 1320ms und 1400ms) und trotzdem nur Laufzeitschwankungen von 100ms, also im Verhältnis zur Gesamtlaufzeit wesentlich geringere Schwankungen als beim Versuch mit JIT. Eine schlüssige Interpretation fällt schwer, doch vermuten wir, daß sich die JVM gelegentlich 'mal eben' ein paar Millisekunden mehr genehmigt.

In einem weiteren Versuch verwendeten wir einen wesentlich größeren Graphen (89612 Knoten und 179378 Kanten bei 25000 starken Zusammenhangskomponenten). Wie man in Abb. 6 sieht, schwankt die Laufzeit zwischen 15700ms und 16300ms, also um 600ms. Das be-

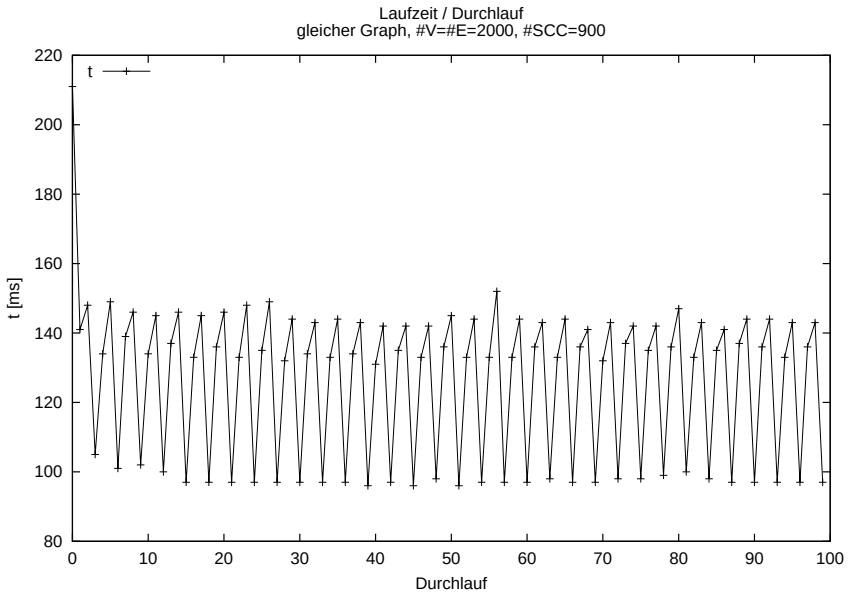


Abb. 4 Laufzeiten eines kleinen Graphen

deutet bei 67mal mehr Knoten und Kanten nur 6fach größere Laufzeitschwankungen.

4.6 Fazit

Offensichtlich liegen also nicht Laufzeitschwankungen des Algorithmus, sondern vielmehr Laufzeitschwankungen der JVM vor. Woher die Regelmäßigkeit der Schwankungen kommt, wissen wir nicht. Dies müßte man in einer weitreichenden Analyse der JVM klären.

Wenn man die beschriebenen Schwankungen – die ohnehin durch die JVM und nicht durch den Algorithmus erzeugt werden – ignoriert, sieht man daß die Laufzeit unserer Implementation durchaus linear in der Anzahl der Knoten und Kanten des betrachteten Graphen ist.

Literatur

1. Udi Manber, *Introduction To Algorithms, A creative Approach* (Addison-Wesley, 1989) Seiten 226ff
2. Abraham Silberschatz, Peter B. Galvin, *Operating System Concepts, 5th edition* (Addison-Wesley, 1998)

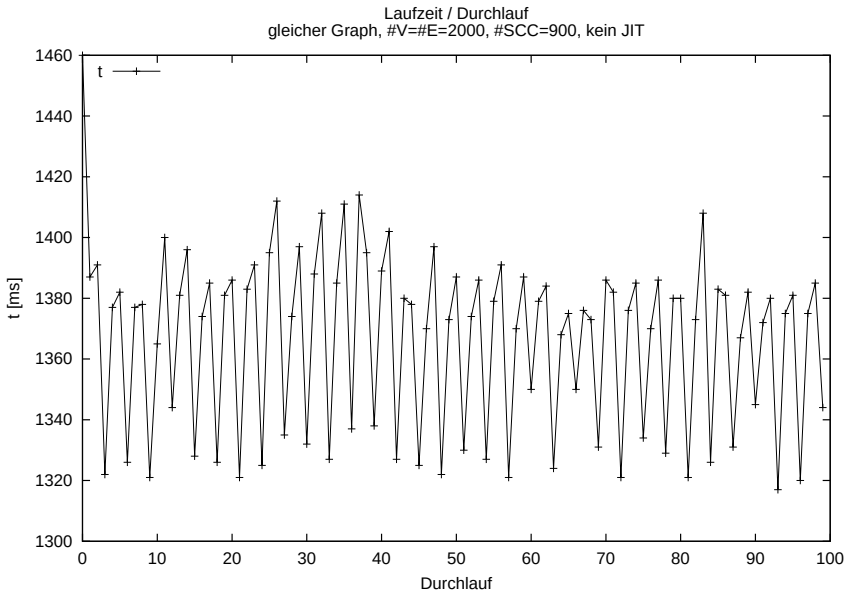


Abb. 5 Laufzeiten eines kleinen Graphen ohne JIT

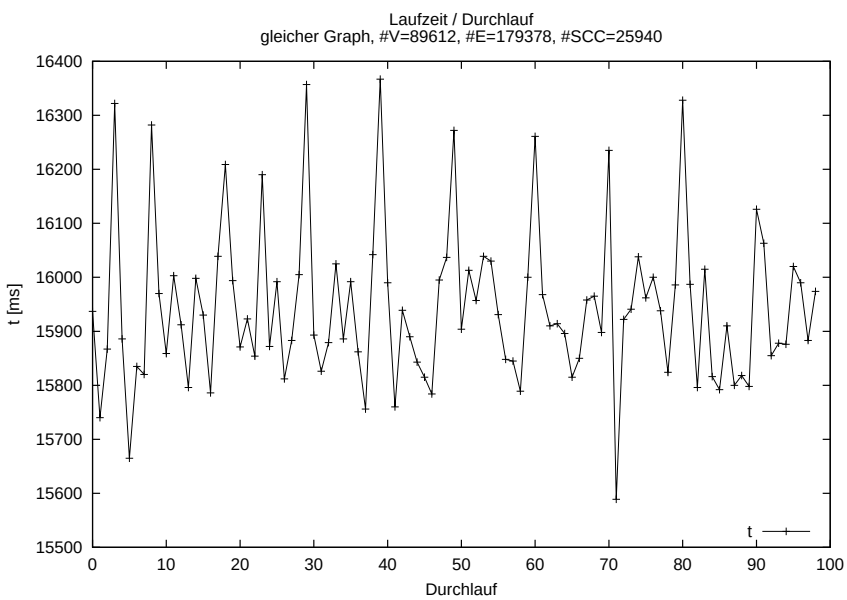


Abb. 6 Laufzeiten eines großen Graphen

Praktikum Algorithmen und Datenstrukturen
Team Nr. 4

SCC

Eine Linearzeitimplementation eines Algorithmus zur Berechnung starker Zusammenhangskomponenten in gerichteten Graphen

Michael Baur, Daniel Fleischer, Robert Schmaus

Universität Konstanz

Sommersemester 2000

Zusammenfassung
Dieses Paper stellt einen Algorithmus vor, der in linearer Laufzeit (in Abhängigkeit der Anzahl der Knoten und Kanten) die starken Zusammenhangskomponenten eines gerichteten Graphen berechnet.

1 Definitionen und Zielsetzungen

Definition 1 *Ein gerichteter Graph $G = (V, E)$ heißt **stark zusammenhängend**, wenn es zwischen je zwei Knoten $u, v \in V$ in beide Richtungen einen gerichteten Weg gibt.*

Definition 2 *Eine **starke Zusammenhangskomponente** eines Graphen ist ein maximaler Subgraph, der stark zusammenhängend ist.*

Zwei Knoten liegen also genau dann in der selben Zusammenhangskomponente, wenn sie auf einem (nicht notwendig einfachen) gerichteten Kreis liegen. Die Relation

$v \sim w \ :\Leftrightarrow v$ und w liegen in derselben Zusammenhangskomponente

ist eine Äquivalenzrelation auf den Knoten des Graphen. Im Folgenden wird nun ein Algorithmus vorgestellt, der die Zusammenhangskomponenten des Graphen berechnet – also die Äquivalenzklassen

bezüglich der Äquivalenzrelation $\sim$. Folglich besteht der Graph nach der Bearbeitung aus der disjunkten Vereinigung seiner Zusammenhangskomponenten. Der Algorithmus ist so implementiert, dass er die Zusammenhangskomponenten in linearer Zeit berechnet.

2 Algorithmus und Implementation

2.1 Union Find

Funktionsweise von `UnionFind`
`UnionFind` stellt eine Implementation des Union-Find-Algorithmus mit Pfadkompression dar. Sie ist speziell für die Verwendung auf Graphen zugeschnitten, deren Knoten vom Typ `Vertex` sind und das Interface `Decorable` erfüllen. `UnionFind` stellt außer den obligatorischen Methoden

```
makeset(Vertex v),
union(Vertex v, Vertex w) und
find(Vertex v)
```

zusätzlich die Methode `sizeOfSet(Vertex v)` zur Verfügung, die die Kardinalität der Menge zurückgibt, die `v` enthält.

`UnionFind` impementiert seine Mengen durch Baumstrukturen, die direkt auf den Graphen, der bearbeitet werden soll, gesetzt werden, ohne den Graphen selbst dabei zu verändern. Jeder dieser Wurzelbäume repräsentiert eine `UnionFind`-Menge, wobei die Wurzel des Baumes zum Repräsentanten der jeweiligen Menge wird.

Jeder Knoten, der in eine `UnionFind`-Menge aufgenommen wird, erhält in der Methode `makeset` zwei „Dekorationen“, `vor` und `number`. Der Schlüssel `vor` enthält einen Zeiger auf den Vorgänger im (virtuellen) `UnionFind`-Baum, wobei die Wurzel auf sich selbst zeigt. In der Dekoration `number` eines Knotens `v` ist die Kardinalität des Teilbaumens mit Wurzel `v` zu finden.

Die Pfadkompression ist in der `find`-Methode und bewirkt, dass bei jedem Aufruf von `find` der Zeiger jedes „angefassten“ Knotens auf die Wurzel des jeweiligen Baumes gesetzt wird.

Laufzeit von `UnionFind`
Die Methode `makeset` hat natürlich konstante Laufzeit.

Die Methoden `union` und `sizeOfSet` sind (in natürlicher Weise) Methoden, deren Komplexität in $\mathcal{O}(1)$ liegt, sofern sie als Argument

bereits Repräsentanten einer Menge erhalten. Da dies jedoch nicht verlangt wird, führen beide Methoden selbst zu Beginn entsprechend Ein- oder Zweimal die `find`-Methode aus, so dass ihre Komplexität im Wesentlichen durch `find` dominiert wird.

Wie von Hopcroft & Ullman (1973) gezeigt wurde, ist der Gesamtaufwand von n Operationen vom Typ `makeset`, `union` und `find` in $\mathcal{O}(n \cdot G(n))$, wo $G(n) = \min\{y \mid F(y) \geq n\}$ und $F(n)$ rekursiv folgendermassen definiert ist:

$$F(0) := 1$$
$$F(y) := 2^{F(y-1)} \text{ für } y \geq 1.$$

$G(n)$ ist damit eine so langsam wachsende Funktion, dass in realistischen Anwendungen $G(n)$ als konstant angesehen werden kann (etwa ist $G(2^{65\,536}) = 5$, vgl. [2]).

2.2 Der Algorithmus zu `SCC`

Funktionsweise von `SCC`
Der benutzte Algorithmus zur Bestimmung der starken Zusammenhangskomponenten setzt sich im wesentlichen aus drei Teilalgorithmen zusammen. Die ersten beiden werden unten in Pseudocode vorgestellt. Der erste Algorithmus `Execute()` benutzt den zweiten Algorithmus `SCC(v)` und der wiederum den Algorithmus `UnionFind()` mit seinen Methoden `makeset(v)`, `union(v,w)` und `find(v)` (`v`,`w` Knoten des Graphen). Die Algorithmen benutzen folgende Datenstrukturen:

- `Repraesentanten-Stack`: speichert Repräsentanten-Knoten der starken Zusammenhangskomponenten
- `Union-Stack`: dient dazu, zu entscheiden, ob ein schon besuchter Knoten zu einer schon vorhandenen Zusammenhangskomponente gehört
- `DFS-Nummer`: jeder zum ersten Mal besuchte Knoten erhält diesen integer-Wert, der anschließend um eins erhöht wird

Hier nun die ersten beiden Algorithmen in Pseudocode:

```
Execute():
 1. solange der Graph einen Knoten v hat,
    der keine DFS-Nummer besitzt
 2.   SCC(v)
 3.   leere den Repraesentanten-Stack
 4.   leere den Union-Stack
```

```
SCC(v):
 1. setze DFS-Nummer von v auf n, setze n = n+1
 2. makeset(v)
 3. lege v auf den Repraesentanten-Stack
 4. lege v auf den Union-Stack
 5. solange aus v Kanten zu Knoten w herauszeigen
 6.   falls w eine DFS-Nummer besitzt
 7.     falls w auf dem Union-Stack liegt
 8.       solange DFS-Nummer von w kleiner als DFS-Nummer
          vom obersten Knoten des Repraesentanten-Stack
 9.         nimm obersten Knoten x des Repraesentanten-
            Stack herunter, union(x,w)
10.   ansonsten SCC(w)
11.     falls w der oberste Knoten des
        Repraesentanten-Stack ist
12.       nimm w vom Repraesentanten-Stack herunter
13.       solange DFS-Nummer von w kleinergleich
          DFS-Nummer vom obersten Knoten des Union-Stack
14.         nimm obersten Knoten vom Union-Stack herunter
```

`SCC(v)` ermittelt alle starken Zusammenhangskomponenten, welche von `v` aus erreichbar sind. `Execute()` ruft `SCC(v)` jeweils für alle noch nicht besuchten Knoten auf, so dass nach Ablauf des Algorithmus alle Knoten besucht wurden und somit auch alle starken Zusammenhangskomponenten gefunden wurden. Hauptsächlich im Algorithmus `SCC(v)` geschieht die Arbeit. Er geht rekursiv vor: beginnend bei `v` startet er eine Tiefensuche und vergibt dabei jeweils die `DFS-Nummer` an die erstmals besuchten Knoten. Trifft er bei der Tiefensuche auf einen Knoten `w`, der schon eine `DFS-Nummer` hat – also einen schon früher besuchten Knoten – so muss er entscheiden, ob die Kante zu diesem Knoten `w` bewirkt, dass bisher vorhandene Zusammenhangskomponenten vereinigt werden – nämlich dann, wenn durch diese Kante ein gerichteter Kreis im schon besuchten Subgraphen zustande kommt. Diese Entscheidung werden wir weiter unten im Zusammenhang mit dem `Union-Stack` behandeln. Jetzt wollen wir zunächst annehmen, dass durch die Kante ein gerichteter Kreis zustande kam. In diesem Fall müssen alle Komponenten, welche mindestens einen Knoten aus diesem Kreis enthalten, vereinigt werden. Der Algorithmus geht dabei so vor, dass er `w` als 'Repräsentant' für diese Komponenten wählt. Es genügt jetzt alle Knoten dieses Kreises mit `w` zu vereinigen, die sich auf dem `Repraesentanten-Stack` befinden und eine größere `DFS-Nummer` haben als `w`. An dieser Stelle sei darauf hingewiesen, dass `w` selbst gar nicht auf dem `Repraesentanten-Stack` liegen muss.

Es genügt aber die obigen Komponenten im Kreis mit `w` zu vereinigen, denn damit vereinigt man diese auch insbesondere mit dem eigentlichen Repräsentanten der Zusammenhangskomponente, in der sich `w` befindet. Alle Repräsentanten im Kreis ohne `w` sind jetzt keine Repräsentanten mehr und werden vom `Repraesentanten-Stack` genommen.

Im Fall, dass oben kein gerichteter Kreis zustande kam, muss an den schon bestehenden Komponenten nichts geändert werden; der Knoten `w` wird deshalb in der Tiefensuche einfach ignoriert.

Ob bestehende Komponenten vereinigt werden müssen oder nicht, dazu dient der `Union-Stack`. Liegt der Knoten `w` auf dem `Union-Stack`, dann wird vereinigt, liegt er nicht auf dem Stack, dann wird auch nicht vereinigt. Der Union-Stack sagt also aus, ob ein Kreis zustande kam, oder nicht. Dies funktioniert so: (Mit DFS-Stack ist im folgenden der Stack gemeint, den die Tiefensuche induziert.) Wird ein Knoten erstmals besucht, dann wird er auf den DFS-Stack gelegt, gleichzeitig auf den `Repraesentanten-Stack` (, da er sein eigener Repräsentant ist) und auch auf den `Union-Stack`. Wird ein Knoten `w` vom DFS-Stack herunter genommen und dieser Knoten ist kein Repräsentant, so bleiben die beiden Stacks `Repraesentanten-Stack` und `Union-Stack` unverändert. Ist dieser Knoten `w` hingegen ein Repräsentant, so wird dieser Knoten vom `Repraesentanten-Stack` heruntergenommen. Ebenfalls werden alle Knoten oberhalbgleich von `w` (d.h. einschließlich `w`) vom `Union-Stack` herunter genommen. Sobald nämlich ein Repräsentant vom DFS-Stack herunter genommen wird, kann dessen Komponente nicht mehr erreicht werden, da diese Komponente keine auswärts gerichtete Kanten mehr hat, die noch nicht besucht wurden. Somit können auch alle Knoten oberhalbgleich von `w` auf dem `Union-Stack` nicht mehr erreicht werden. Damit erfüllt der `Union-Stack` genau die oben genannte Aufgabe.

Laufzeit von `SCC`
Die lineare Laufzeit des Algorithmus lässt sich schnell erkennen. Jeder Knoten wird in der äußeren Schleife von `Execute()` und in `SCC(v)` nur einmal berührt; ebenso wird auch jede Kante höchstens einmal besucht. Somit arbeitet der Algorithmus linear in der Anzahl der Kanten plus Knoten.

3 Laufzeitstatistik

3.1 `MultiSCC`

Um das Laufzeitverhalten unseres Algorithmus statistisch auszuwerten, wurde die Klasse `MultiSCC` erstellt. Sie erlaubt es, ganze Listen von Graphdateien mit `SCC` zu bearbeiten und speichert die Bearbeitungszeit für die Graphen zusammen mit der jeweiligen Knoten- und Kantenzahl in eine Logdatei.

Um störende Einflüsse (etwa von gleichzeitig auf dem Rechner laufenden Programmen) zu minimieren, ist es mit `MultiSCC` möglich, jeden Graphen mehrfach zu bearbeiten, um einen Mittelwert für die jeweilige Laufzeit zu berechnen.

3.2 Laufzeitstatistik von `SCC`

Eine unproblematisch lineare Laufzeit von `SCC` ergibt sich bei der Bearbeitung von Graphen mit $m \in \mathcal{O}(n)$ [1].

Die Schaubilder in Abb. 1 stammen von der Bearbeitung einer Liste von Graphen, mit $|E| = 3 \cdot |V|$, die von einem Zufallsgenerator erzeugt wurden. Die Bearbeitung erfolgte mit `MultiSCC`, wobei die Laufzeit pro Graph aus 10 Durchläufen des Algorithmus gemittelt ist. Links sehen wir das Laufzeitverhalten bezüglich n, rechts das Laufzeitverhalten bezüglich m.

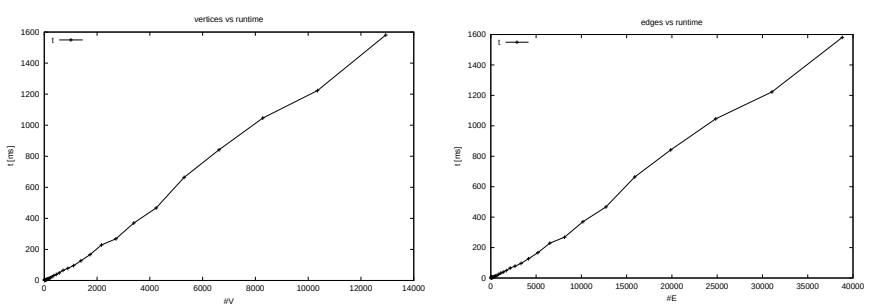


Abb. 1: Graphen mit $m \in \mathcal{O}(n)$

[1] Wie üblich bezeichnet für alle Graphen jeweils V die Knoten- und E die Kantenmenge. Immer ist $n := |V|$ und $m := |E|$.

Dass die Laufzeit von `SCC` tatsächlich linear in der Anzahl der Knoten *plus* Anzahl der Kanten ist, zeigt sich in den Diagrammen der nebenstehenden Abbildung 2. Bearbeitet wurden Graphen, deren Kantenanzahl quadratisch in der Anzahl der Knoten wächst. Jeder Graph durchlief `SCC` 10 Mal; die Diagramme zeigen die durchschnittliche Laufzeit. Das obere Schaubild stellt die Laufzeit des Programms in Abhängigkeit der Knotenanzahl dar – wie zu erwarten ergibt sich kein lineares Verhalten. Dieses zeigt sich erst im unteren Schaubild, wo die Laufzeit in Abhängigkeit von $\#V + \#E$ dargestellt wird.

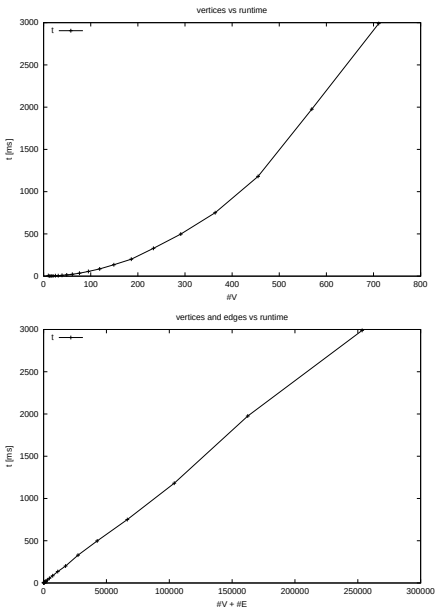


Abb. 2: Graphen mit $m \in \mathcal{O}(n^2)$

Um eine Vergleichbarkeit mit anderen Gruppen, die denselben Algorithmus zu impementieren hatten, zu gewähren, stellen wir an dieser Stelle noch die Statistik der Graphen aus den beiden `.jar`-Files zur Verfügung, die allen Gruppen zugänglich waren.

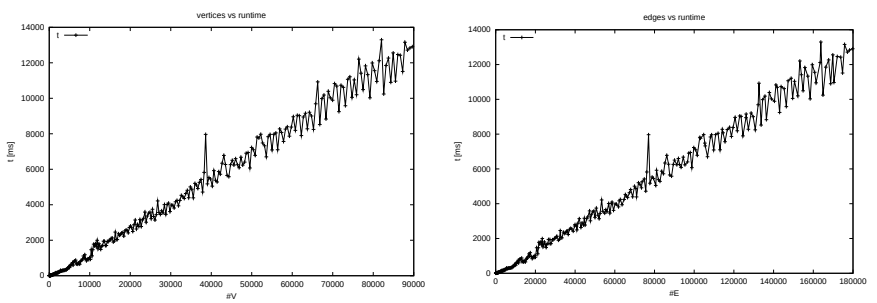


Abb. 3: Graphen aus dem `.jar`-File `graphA2.jar`

Die Diagramme in Abbildung 3 zeigen das Laufzeitverhalten von `SCC` wieder in Abhängigkeit von n und m, wobei $m \in \mathcal{O}(n)$ (bearbeitet wurden die Graphen aus dem File `graphA.jar`). Die Kurve ist zwar alles andere als glatt ... was vor allem daran liegt, dass jeder Graph nur einmal bearbeitet wurde, und sich so Unregelmäßigkeiten durch gleichzeitig laufende Programme u.ä. besonders gravierend zeigen.

Alles in allem zeigt aber das Diagramm deutlich ein lineares Verhalten von `SCC`.

Literatur

1. *Data Structures Library in Java (JDSL).* `tux.cs.brown.edu/cgc/jdsl`.
2. D. Wagner, *Entwurf und Analyse von Algorithmen.* Vorlesungsskript, Fachgruppe Informatik der Universität Konstanz, Wintersemester 1999/2000.
3. T. Willhalm, *Praktikum Algorithmen und Datenstrukturen, Blatt 1.* `www.fmi.uni-konstanz.de/~wilhalm/lehre/prad_S00/`, Sommersemester 2000.