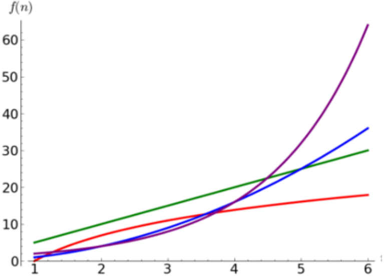
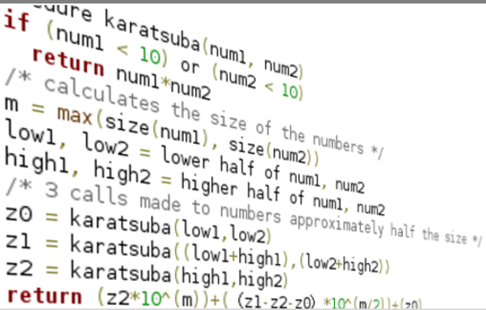
Tutorium Algorithmen 1

Simon Bischof (simon.bischof2@student.kit.edu) | 17. Juni 2013

Institut für theoretische Informatik, Prof. Sanders





Übungsblatt

- Bei Heapsort: aktuellen Schritt hinschreiben
- Pointerumbiegung bei Aufgabe 2(c) explizit hinschreiben

Programmierwettbewerb

- Siehe Übungsfolien 7
- Bei Fragen könnt ihr euch an mich wenden

Aus der Übung: Rot-Schwarz-Bäume

- Balancierte Suchbäume mit zusätzlichen Eigenschaften
- Knoten haben Farbe: Rot oder Schwarz
- Lokal feststellbar, welche Operationen zum Balancieren nötig sind
- Äquivalent zu (2,4)-Bäumen

Aus der Übung: Rot-Schwarz-Bäume

- Balancierte Suchbäume mit zusätzlichen Eigenschaften
- Knoten haben Farbe: Rot oder Schwarz
- Lokal feststellbar, welche Operationen zum Balancieren nötig sind
- Äquivalent zu (2,4)-Bäumen

Graphen

Konventionen

- $G = (V, E)$, V Kantenmenge, E Knotenmenge
- $n = |V|$, $m = |E|$, Knoten s, t, u, v, w, x, y, z
- Kanten $e \in E$: Knotenpaare (gerichtet) oder 2-elementige Knotenmenge (ungerichtet)
- ungerichtet $\rightarrow$ gerichtet durch bigerichtete Graphen

Konventionen

- $G = (V, E)$, V Kantenmenge, E Knotenmenge
- $n = |V|$, $m = |E|$, Knoten s, t, u, v, w, x, y, z
- Kanten $e \in E$: Knotenpaare (gerichtet) oder 2-elementige Knotenmenge (ungerichtet)
- ungerichtet $\rightarrow$ gerichtet durch bigerichtete Graphen

Konventionen

- $G = (V, E)$, V Kantenmenge, E Knotenmenge
- $n = |V|$, $m = |E|$, Knoten s, t, u, v, w, x, y, z
- Kanten $e \in E$: Knotenpaare (gerichtet) oder 2-elementige Knotenmenge (ungerichtet)
- ungerichtet $\rightarrow$ gerichtet durch bigerichtete Graphen

Konventionen

- $G = (V, E)$, V Kantenmenge, E Knotenmenge
- $n = |V|$, $m = |E|$, Knoten s, t, u, v, w, x, y, z
- Kanten $e \in E$: Knotenpaare (gerichtet) oder 2-elementige Knotenmenge (ungerichtet)
- ungerichtet $\rightarrow$ gerichtet durch bigerichtete Graphen

Kantenanfragen

- speichere Kanten in einer Hashtabelle
- unabhängig von restlicher Struktur

Kantenfolgerepräsentation

- Liste aller Kantenpaare
- Kompakt, gute I/O
- Wenig sinnvolle Operationen unterstützt

Kantenfolgerepräsentation

- Liste aller Kantenpaare
- Kompakt, gute I/O
- Wenig sinnvolle Operationen unterstützt

Adjazenzfeld

- V=1..n oder V=0..n-1
- Kantenfeld E speichert Ziele gruppiert nach Startknoten
- V speichert Index der ersten ausgehenden Kante
- Dummy-Eintrag V[n+1] speichert m+1

Adjazenzfeld

- V=1..n oder V=0..n-1
- Kantenfeld E speichert Ziele gruppiert nach Startknoten
- V speichert Index der ersten ausgehenden Kante
- Dummy-Eintrag V[n+1] speichert m+1

Adjazenzliste

- Knoten-Array mit doppelt verketterter Liste von ausgehenden Kanten
- Einfaches Löschen und Einfügen von Kanten
- platz-, cacheineffizient

Adjazenzliste

- Knoten-Array mit doppelt verketterter Liste von ausgehenden Kanten
- Einfaches Löschen und Einfügen von Kanten
- platz-, cacheineffizient

Adjazenzmatrix

- A$\in \{0,1\}^{n \times n}$; A(i,j)=1 $\Leftrightarrow$ (i,j)$\in$E
- platzeffizient für sehr dichte Graphen, platzineffizient sonst
- einfache Kantenanfragen, langsame Navigation
- verbindet lineare Algebra und Graphentheorie

Adjazenzmatrix

- A$\in \{0,1\}^{n\times n}$; A(i,j)=1 $\Leftrightarrow$ (i,j)$\in$E
- platzeffizient für sehr dichte Graphen, platzineffizient sonst
- einfache Kantenanfragen, langsame Navigation
- verbindet lineare Algebra und Graphentheorie

Customization

- Zuschneiden von Datenstrukturen für Algorithmen
- Manchmal implizite Repräsentation möglich

Kreativaufgabe

In dieser Aufgabe soll ein dynamisiertes Adjazenzfeld entwickelt werden. Mit anderen Worten: Gesucht ist eine Datenstruktur für gerichtete Graphen G = (V,E) mit folgenden Eigenschaften:

- Stabile und eindeutige KnotenIDs. Knoten sollen durch IDs eindeutig identifiziert werden. Diese IDs sollen Zahlen aus $\mathbb{N}_0$ sein. Dabei seien die KnotenIDs stabil, d.h. die ID eines Knotens ändere sich nie solange dieser Knoten existiert (nach Entfernen eines Knotens darf dessen ID jedoch neu vergeben werden).
- Eindeutige KantenIDs. Die Kanten sollen ebenfalls durch IDs eindeutig identifiziert werden. Allerdings müssen diese nicht unbedingt Zahlen aus $\mathbb{N}_0$ sein und sie müssen auch nicht stabil sein.
- Effizienter Wahlfreier Zugriff auf Knoten und Kanten. Es gibt die Operationen node(u : NodeID) : Handle of Node und edge(e : EdgeID) : Handle of Edge, die in O(1) Zeit einen Handle auf das Knoten bzw. Kantenobjekt zu einer Knoten- bzw. Kanten-ID liefern.

- Effiziente Navigation. Es gibt die Operationen firstEdge(v : NodeID) : EdgeID $\cup\{\perp\}$ und nextEdge(e : EdgeID) : EdgeID $\cup\{\perp\}$ mit deren Hilfe wie folgt über alle ausgehenden Kanten eines Knoten v iteriert werden kann in einem Graph G:
  for (EdgeID e := graph.firstEdge(v); e $\neq \perp$; e := nextEdge(e))
  h_e := G:edge(e) : Handle of Edge
  /* do something */
  end for
  Sowohl firstEdge als auch nextEdge dürfen höchstens O(1) Zeit brauchen.

- Amortisiert konstantes Einfügen von Knoten und Kanten. Es gibt Operationen insertNode : NodeID und insertEdge(u, v : NodeID) : EdgeID, die in amortisiert konstanter Zeit einen neuen Knoten bzw. eine neue Kanten von u nach v einfügen und jeweils die ID des neu erzeugten Elementes zurückliefern. Beide Operationen dürfen höchstens amortisiert konstante Zeit kosten.
- Amortisiert konstantes Entfernen von Knoten und Kanten. Es gibt Operationen deleteNode(v : NodeID) und deleteEdge(e : EdgeID), die einen Knoten bzw. eine Kante entfernen. Der Einfachheit halber darf ein Knoten dabei nur entfernt werden, wenn bereits alle seine Kanten entfernt worden sind. Beide Operationen dürfen höchstens amortisiert konstante Zeit kosten.

- Überlegen Sie sich, wie Sie diese Datenstruktur realisieren.
- Begründen Sie, warum die beschriebenen Operationen in Ihrer Realisierung das geforderte Laufzeitverhalten aufweisen.
- Wieviel Speicher kann ein Graph mit Ihrer Realisierung im schlimmsten Fall belegen (abhängig von aktuellen oder zwischenzeitlichen Werten von $|V|$ und $|E|$ und das nicht nur im O-Kalkül)? Wieviel im besten Fall? Vergleichen Sie mit dem Speicherverbrauch des statischen Adjazenzfeldes aus der Vorlesung.

Graphtraversierung

- Systematisches Durchsuchen eines Graphen
- Klassifizierung von Kanten als Baum-, Vorwärts-, Quer- und Rückwärtskanten

Graphtraversierung

- Systematisches Durchsuchen eines Graphen
- Klassifizierung von Kanten als Baum-, Vorwärts-, Quer- und Rückwärtskanten

Breitensuche

- Wähle Startknoten s
- Berechnet Abstand zu s (pro Kante Abstand 1)
- Berechnet Baum Schicht für Schicht
- Keine Vorwärtskanten (warum?)
- Speichert Distanz zu s und Vorgänger (parent[s]=s)

Breitensuche

- Wähle Startknoten s
- Berechnet Abstand zu s (pro Kante Abstand 1)
- Berechnet Baum Schicht für Schicht
- Keine Vorwärtskanten (warum?)
- Speichert Distanz zu s und Vorgänger (parent[s]=s)

Breitensuche

- Wähle Startknoten s
- Berechnet Abstand zu s (pro Kante Abstand 1)
- Berechnet Baum Schicht für Schicht
- Keine Vorwärtskanten (warum?)
- Speichert Distanz zu s und Vorgänger (parent[s]=s)

Breitensuche

- Wähle Startknoten s
- Berechnet Abstand zu s (pro Kante Abstand 1)
- Berechnet Baum Schicht für Schicht
- Keine Vorwärtskanten (warum?)
- Speichert Distanz zu s und Vorgänger (parent[s]=s)

Tiefensuche

- Wähle Startknoten s
- versuche soweit wie möglich zu gehen
- Keine weitere Kante: Backtracken

Tiefensuche

- Wähle Startknoten s
- versuche soweit wie möglich zu gehen
- Keine weitere Kante: Backtracken

Tiefensuche

- Wähle Startknoten s
- versuche soweit wie möglich zu gehen
- Keine weitere Kante: Backtracken

Markierungen etc.

- Markiere besuchte Knoten
- Verwalte Zähler dfsPos, finishingTime : 1..n
- beim Besuchen: dfsNum[w]=dfsPos++
- beim Abschließen: finishTime[w]=finishingTime++

Markierungen etc.

- Markiere besuchte Knoten
- Verwalte Zähler dfsPos, finishingTime : 1..n
- beim Besuchen: dfsNum[w]=dfsPos++
- beim Abschließen: finishTime[w]=finishingTime++

Markierungen etc.

- Markiere besuchte Knoten
- Verwalte Zähler dfsPos, finishingTime : 1..n
- beim Besuchen: dfsNum[w]=dfsPos++
- beim Abschließen: finishTime[w]=finishingTime++

Klassifikation von Kanten

type (v,w)	dfsNum[v]< dfsNum[w]	finishTime[w]< finishTime[v]	w is marked
tree	yes	yes	no
forward	yes	yes	yes
backward	no	no	yes
cross	no	yes	yes

DAG/Topologische Sortierung

- G DAG $\Leftrightarrow$ DFS finded keine Rückwärtskante
- Dann liefert t(w):=n-finishingTime[w] eine topologische Sortierung

DAG/Topologische Sortierung

- G DAG $\Leftrightarrow$ DFS finded keine Rückwärtskante
- Dann liefert t(w):=n-finishingTime[w] eine topologische Sortierung

starke Zusammenhangskomponenten

- Starke Zusammenhangskomponenten Betrachte die Relation $\overset{*}{\leftrightarrow}$ mit $u \overset{*}{\leftrightarrow} v$ falls $\exists$ Pfad $\langle v, \ldots, u \rangle$ und $\exists$ Pfad $\langle u, \ldots, v \rangle$
- $\overset{*}{\leftrightarrow}$ ist Äquivalenzrelation (warum?)
- Die Äquivalenzklassen von $\overset{*}{\leftrightarrow}$ bezeichnet man als starke Zusammenhangskomponenten.

starke Zusammenhangskomponenten

- Starke Zusammenhangskomponenten Betrachte die Relation $\overset{*}{\leftrightarrow}$ mit $u \overset{*}{\leftrightarrow} v$ falls $\exists$ Pfad $\langle v, \ldots, u\rangle$ und $\exists$ Pfad $\langle u, \ldots, v\rangle$
- $\overset{*}{\leftrightarrow}$ ist Äquivalenzrelation (warum?)
- Die Äquivalenzklassen von $\overset{*}{\leftrightarrow}$ bezeichnet man als starke Zusammenhangskomponenten.