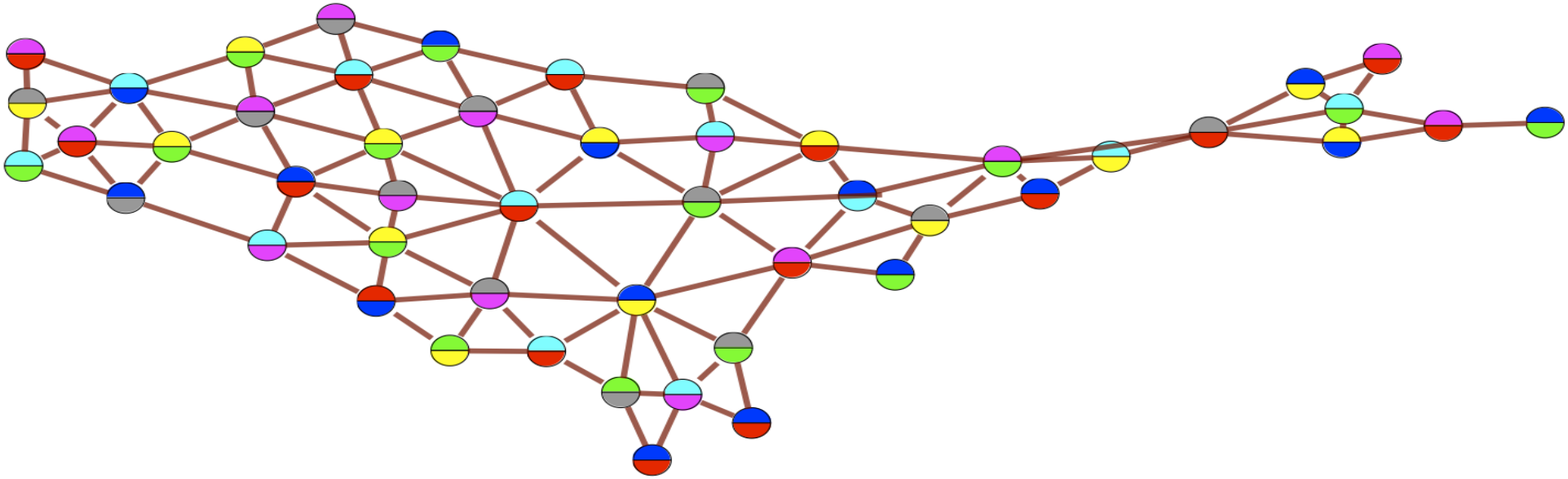
Karlsruher Institut für Technologie

Tutorium 10: Graphdarstellung und Traversierung

Holger Ebhart | 1. Juli 2015

TUTORIUM ZUR VORLESUNG ALGORITHMEN I IM SS15



Gliederung

8. Übungsblatt

- Heapsort mit swap funktioniert inplace
- bei Heapsort wird der Heap bei **jeder** Iteration kleiner
- beim Löschen aus (a,b)-Bäumen wird entweder fuse oder balance aufgerufen
- Eulerkreis ist ein Zyklus der jede Kante genau einmal traversiert
- In Graph G existiert ein Eulerkreis $\Leftrightarrow$ G ist zusammenhängend und $\forall v \in V$: deg(v) ist gerade
- Hamiltonkreis ist ein Zyklus der jeden Knoten einmal besucht (NP-complete)

8. Übungsblatt

- Heapsort mit swap funktioniert inplace
- bei Heapsort wird der Heap bei **jeder** Iteration kleiner
- beim Löschen aus (a,b)-Bäumen wird entweder fuse oder balance aufgerufen
- Eulerkreis ist ein Zyklus der jede Kante genau einmal traversiert
- In Graph G existiert ein Eulerkreis $\Leftrightarrow$ G ist zusammenhängend und $\forall v \in V$: deg(v) ist gerade
- Hamiltonkreis ist ein Zyklus der jeden Knoten einmal besucht (NP-complete)

8. Übungsblatt

- Heapsort mit swap funktioniert inplace
- bei Heapsort wird der Heap bei **jeder** Iteration kleiner
- beim Löschen aus (a,b)-Bäumen wird entweder fuse oder balance aufgerufen
- Eulerkreis ist ein Zyklus der jede Kante genau einmal traversiert
- In Graph G existiert ein Eulerkreis $\Leftrightarrow$ G ist zusammenhängend und $\forall v \in V$: deg(v) ist gerade
- Hamiltonkreis ist ein Zyklus der jeden Knoten einmal besucht (NP-complete)

Kantenfolge

+ speichere nur Kanten in Feld, Liste oder Suchbaum
+ schnelles Iterieren über Kanten
+ auch für dynamische Graphen geeignet
+ gut für I/O
- schlecht um über Knoten zu Iterieren
- nur Knoten mit Kanten können gespeichert werden
- unterstützt kaum Funktionen

Kantenfolge

+ speichere nur Kanten in Feld, Liste oder Suchbaum
+ schnelles Iterieren über Kanten
+ auch für dynamische Graphen geeignet
+ gut für I/O
- schlecht um über Knoten zu Iterieren
- nur Knoten mit Kanten können gespeichert werden
- unterstützt kaum Funktionen

Adjazenzmatrix

+ gut für dichte Graphen
+ einfache Kantenanfragen
+ für dynamische Graphen ideal
+ verbindet Graphentheorie mit LA (z.B. Potenzen)
- benötigt viel Speicherplatz
- langsame Navigation

Adjazenzmatrix

+ gut für dichte Graphen
+ einfache Kantenanfragen
+ für dynamische Graphen ideal
+ verbindet Graphentheorie mit LA (z.B. Potenzen)
- benötigt viel Speicherplatz
- langsame Navigation

Adjazenzfeld/-liste

+ einfache Navigation
+ einfache Kantenanfragen
+ Speicherplatz-effizient
+ für dynamische Graphen geeignet (Listen)
+ einfache Einfügen und löschen von Kanten (Listen)
- nur für statische Graphen (Felder)
- viele Cache-Misses (Listen)

Adjazenzfeld/-liste

+ einfache Navigation
+ einfache Kantenanfragen
+ Speicherplatz-effizient
+ für dynamische Graphen geeignet (Listen)
+ einfache Einfügen und löschen von Kanten (Listen)
- nur für statische Graphen (Felder)
- viele Cache-Misses (Listen)

Aufgabe

Stellt den Graphen an der Tafel in folgenden Formen dar:

- Kantenfolge
- Adjazenzmatrix
- Adjazenzfeld

BFS

Ziel ist es einen Baum im Graphen zu finden, der die kürzesten Pfade von einem Startknoten s aus angibt.

Dabei werden die Knoten in Ebenen eingeteilt und die Kanten werden folgendermaßen klassifiziert:

- tree
- cross
- backward
- forward

BFS

Breitensuche:

```
1: procedure bfs(s:Node)
2:     Q := {s}
3:     while Q ≠ {} do
4:         Q' := {}
5:         foreach v ∈ Q do
6:             Q' := Q' ∪ explore(v)
7:         Q := Q'
```

explore(v) liefert noch nicht betrachtete Knoten u zurück die eine Kante (v,u) haben

Aufgabe BFS

Führt BFS auf dem Graph an der Tafel aus.
Klassifiziert dabei die Kanten und gebt für jeden Knoten die Ebene an.
Gebt außerdem den entstandenen Baum an.
Der Startknoten sei durch A gegeben.
(Die Knoten werden wenn nötig in alphabetischer Reihenfolge betrachtet.)

DFS

- Gehe im Graphen zuerst rekursiv in die Tiefe
- baue parent-Zeiger auf um wieder zurück zu finden
- DFS ist Grundlage vieler Graphenalgorithmen
- auch hier ist eine Kantenklassifizierung möglich
- DFS-Nummerierung für topologische Sortierung
- Fertigstellungszeit ebenfalls als Nummerierung üblich

DFS - Kantenklassifizierung

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

DFS

```
 1: procedure initDFS()
 2:     init
 3:     foreach s ∈ V do
 4:         if s is not marked then
 5:             mark s
 6:             root(s)
 7:             dfs(s,s)
 8: procedure dfs(u,v:NodeID)
 9:     foreach (v,w) ∈ E do
10:         if w is marked then
11:             traverseNonTreeEdge(v,w)
12:         else
13:             traverseTreeEdge(v,w)
14:             mark w
15:             dfs(v,w)
16:     backtrack(u,v)
```

DFS-Baum

init:
parent := $\{\perp, \cdots, \perp\}$:Array of NodeID

root(s)
parent[s] := s

traverseTreeEdge(v,w):
parent[w] := v

DFS-Baum

init:
parent := $\{\perp, \cdots, \perp\}$:Array of NodeID

root(s)
parent[s] := s

traverseTreeEdge(v,w):
parent[w] := v

DFS-Baum

init:
parent := $\{\perp, \cdots, \perp\}$:Array of NodeID

root(s)
parent[s] := s

traverseTreeEdge(v,w):
parent[w] := v

DFS-Nummerierung

init:
dfsPos=1:$\mathbb{N}$
dfsNum[0...$|V|$]:Array of $\mathbb{N}$

root(s):
dfsNum[s] := dfsPos++

traverseTreeEdge(v,w):
dfsNum[w] := dfsPos++

DFS-Nummerierung

init:
dfsPos=1:$\mathbb{N}$
dfsNum[0...$|V|$]:Array of $\mathbb{N}$

root(s):
dfsNum[s] := dfsPos++

traverseTreeEdge(v,w):
dfsNum[w] := dfsPos++

DFS-Nummerierung

init:
dfsPos=1:$\mathbb{N}$
dfsNum[0...$|V|$]:Array of $\mathbb{N}$

root(s):
dfsNum[s] := dfsPos++

traverseTreeEdge(v,w):
dfsNum[w] := dfsPos++

DFS-Fertigstellungszeit

init:
fTime=1:$\mathbb{N}$
finishTime[0...$|V|$]:Array of $\mathbb{N}$

backtrack(u,v):
finishTime[v] := fTime++

DFS-Fertigstellungszeit

init:
fTime=1:$\mathbb{N}$
finishTime[0...$|V|$]:Array of $\mathbb{N}$

backtrack(u,v):
finishTime[v] := fTime++

Aufgabe DFS

Führt DFS auf dem Graph an der Tafel aus.
Klassifiziert dabei die Kanten und gebt für jeden Knoten die DFS-Nummerierung und die Fertigstellungszeit an.
Gebt außerdem den entstandenen Baum an.
Der Startknoten sei durch A gegeben.
(Die Knoten werden wenn nötig in alphabetischer Reihenfolge betrachtet.)

Kreativaufgabe

Gegeben sei ein gerichteter, stark zusammenhängender Graph in folgender Darstellung:

- Ein Knotenarray, das zu jedem Knoten v einen Eintrag mit seiner ID und einen Zeiger auf ein Array mit den von v ausgehenden Kanten enthält. Die Knoten haben eindeutige IDs.
- Das Kantenarray mit den ausgehenden Kanten von v enthält für jede Kante e einen Eintrag mit ihrer ID und der ID des Zielknotens der Kante.

Auf diesem Graphen soll nun eine BFS ausgeführt werden, wobei zu jedem Zeitpunkt nur $\mathcal{O}1$ zusätzlicher Speicher verwendet werden soll. Die Laufzeit darf dabei schlechter als bei der üblichen BFS sein. Die Art der Darstellung des Graphen soll während der BFS erhalten bleiben, es ist jedoch erlaubt z.B. die Knoten im Knotenarray zu permutieren. Gebt eine

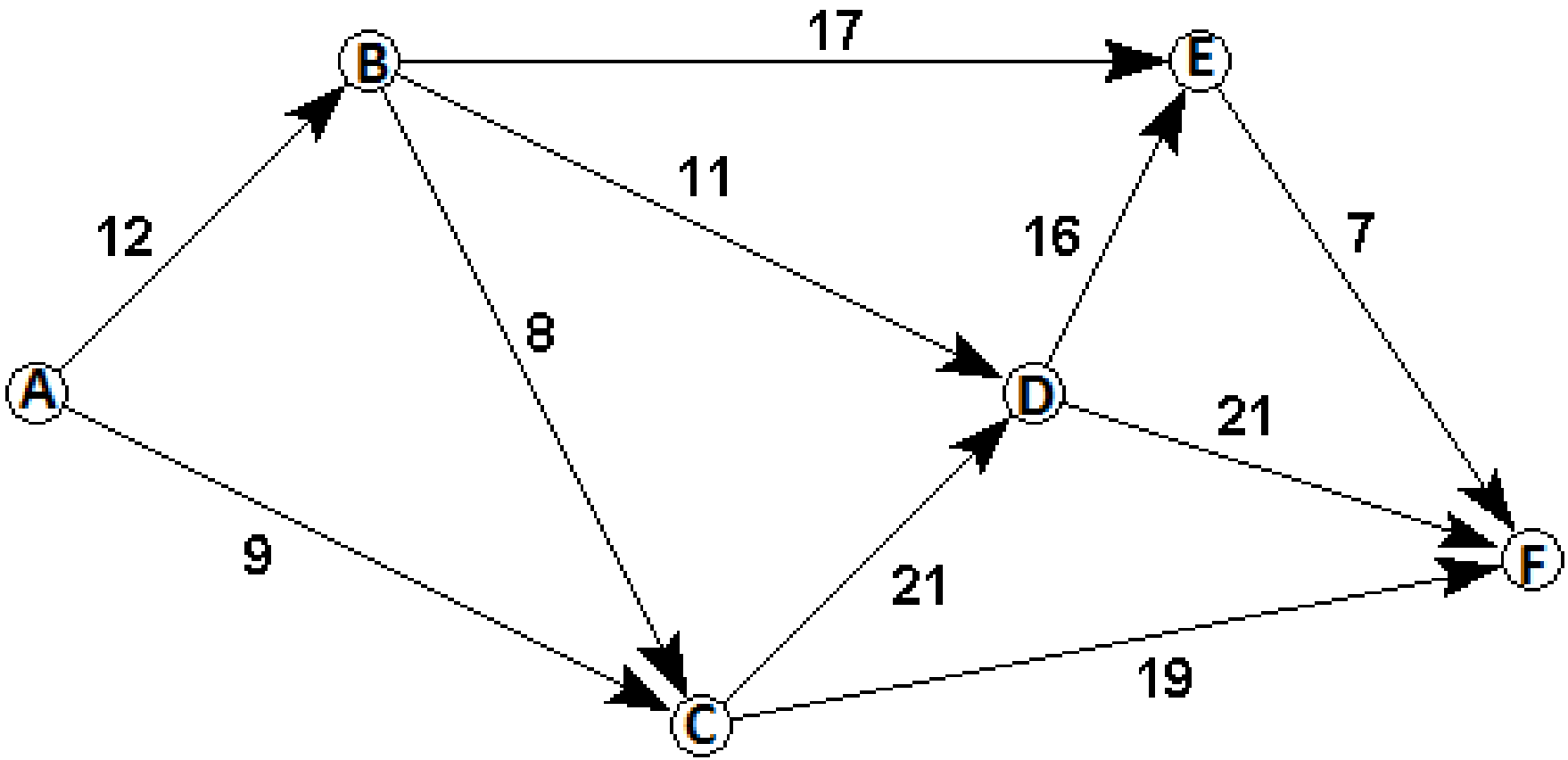
Pseudocode-Implementierung der BFS an, die diese Bedingungen erfüllt und begründet, warum diese Implementierung nur $\mathcal{O}1$ zusätzlichen Speicher benötigt. Es soll dabei für jeden Knoten eine unbekannte Funktion *f* aufgerufen werden, die als Eingabe die Knoten-ID und die Ebene (Entfernung zum Startknoten) des Knotens hat. Es kann angenommen werden, dass *f* während der Ausführung $\mathcal{O}1$ und nach der Ausführung keinen Speicher benötigt.

Lösung

```
1:  procedure BFS(NodeArray[1 . . . n] nodes, NodeID start)
2:      Finde i mit nodes[i].ID = start
3:      Vertausche nodes[i] mit nodes[1]
4:      Setze q1 := 1, q2 := 2, ebene := 0 und u := 2
5:      while q1 ≤ n do
6:          while q1 < q2 do
7:              call f(nodes[q1].ID, ebene)
8:              forall Kanten k im Kantenarray von nodes[q1] do
9:                  Suche i ∈ {u, . . . , n} mit nodes[i].ID = Ziel_ID
10:                 If i gefunden do
11:                     Vertausche nodes[i] mit nodes[u]
12:                     u + +
13:             q1 + +
14:         q2 := u, ebene + +
15: return
```

Zu Übungsblatt 9 - Dijkstra

Führen sie auf folgendem Graphen den Algorithmus von Dijkstra aus. Der Startknoten sei A.



Vielen Dank für eure Aufmerksamkeit!
Bis zum nächsten Mal.

stackoverflow.com