Karlsruher Institut für Technologie

Algorithmen I
Tutorium 33

Woche 9 | 22. Juni 2018

Daniel Jungkind (daniel.jungkind@student.kit.edu)

INSTITUT FÜR THEORETISCHE INFORMATIK

Inhalt

Graphen durchlaufen

Quiz

Ein binärer Heap ist ein (1,2)-Baum. **?**

Quiz

Ein binärer Heap ist ein (1,2)-Baum. **Falsch.**

Heaps sind nicht sortiert.

Quiz

Ein binärer Heap ist ein (1,2)-Baum. **Falsch.**

Heaps sind nicht sortiert.

Beim Einfügen in einen (a,b)-Baum kann es zu *balance*- und *fuse*-Operationen kommen. **?**

Quiz

Ein binärer Heap ist ein (1,2)-Baum. **Falsch.**

Heaps sind nicht sortiert.

Beim Einfügen in einen (a,b)-Baum kann es zu *balance*- und *fuse*-Operationen kommen. **Falsch.**

Beim Einfügen *split*, beim Löschen *balance*/*fuse*.

Quiz

Ein binärer Heap ist ein (1,2)-Baum. **Falsch.**

Heaps sind nicht sortiert.

Beim Einfügen in einen (a,b)-Baum kann es zu *balance*- und *fuse*-Operationen kommen. **Falsch.**

Beim Einfügen *split*, beim Löschen *balance*/*fuse*.

Adjazenzfelder eignen sich besser zum Traversieren von Graphen als Adjazenzmatrizen. **?**

Quiz

Ein binärer Heap ist ein (1,2)-Baum. **Falsch.**

Heaps sind nicht sortiert.

Beim Einfügen in einen (a,b)-Baum kann es zu *balance*- und *fuse*-Operationen kommen. **Falsch.**

Beim Einfügen *split*, beim Löschen *balance*/*fuse*.

Adjazenzfelder eignen sich besser zum Traversieren von Graphen als Adjazenzmatrizen. **Wahr.**

Quiz

Ein binärer Heap ist ein (1,2)-Baum. **Falsch.**

Heaps sind nicht sortiert.

Beim Einfügen in einen (a,b)-Baum kann es zu *balance*- und *fuse*-Operationen kommen. **Falsch.**

Beim Einfügen *split*, beim Löschen *balance*/*fuse*.

Adjazenzfelder eignen sich besser zum Traversieren von Graphen als Adjazenzmatrizen. **Wahr.**

Jeder kreisfreie Graph, bei dem es zu jedem Knoten höchstens einen Pfad gibt, ist ein Baum. **?**

Quiz

Ein binärer Heap ist ein (1,2)-Baum. **Falsch.**

Heaps sind nicht sortiert.

Beim Einfügen in einen (a,b)-Baum kann es zu *balance*- und *fuse*-Operationen kommen. **Falsch.**

Beim Einfügen *split*, beim Löschen *balance*/*fuse*.

Adjazenzfelder eignen sich besser zum Traversieren von Graphen als Adjazenzmatrizen. **Wahr.**

Jeder kreisfreie Graph, bei dem es zu jedem Knoten höchstens einen Pfad gibt, ist ein Baum. **Falsch.**

Es gibt auch Wälder.

Quiz

Ein binärer Heap ist ein (1,2)-Baum. **Falsch.**

Heaps sind nicht sortiert.

Beim Einfügen in einen (a,b)-Baum kann es zu *balance*- und *fuse*-Operationen kommen. **Falsch.**

Beim Einfügen *split*, beim Löschen *balance*/*fuse*.

Adjazenzfelder eignen sich besser zum Traversieren von Graphen als Adjazenzmatrizen. **Wahr.**

Jeder kreisfreie Graph, bei dem es zu jedem Knoten höchstens einen Pfad gibt, ist ein Baum. **Falsch.**

Es gibt auch Wälder.

Jeder ungerichtete, zusammenhängende, kreisfreie Graph ist ein Baum. **?**

Quiz

Ein binärer Heap ist ein (1,2)-Baum. **Falsch.**

Heaps sind nicht sortiert.

Beim Einfügen in einen (a,b)-Baum kann es zu *balance*- und *fuse*-Operationen kommen. **Falsch.**

Beim Einfügen *split*, beim Löschen *balance*/*fuse*.

Adjazenzfelder eignen sich besser zum Traversieren von Graphen als Adjazenzmatrizen. **Wahr.**

Jeder kreisfreie Graph, bei dem es zu jedem Knoten höchstens einen Pfad gibt, ist ein Baum. **Falsch.**

Es gibt auch Wälder.

Jeder ungerichtete, zusammenhängende, kreisfreie Graph ist ein Baum. **Wahr.**

GRAPHEN DURCHLAUFEN

Hänsel und Gretel im Tiefensuchwald

Graphen durchlaufen

Karlsruher Institut für Technologie

- **Geg.**: Startknoten $s \in V$
- **Ziel**: Von s aus alle weiteren Knoten besuchen
- **Aber**: Keine **Doppel**besuche/Endlosschleifen $\Rightarrow$ **Merke** besuchte Knoten
- Am Ende wollen wir zu jedem Knoten nen Weg haben

Graphen durchlaufen

- **Geg.**: Startknoten $s \in V$
- **Ziel**: Von s aus alle weiteren Knoten besuchen
- **Aber**: Keine **Doppel**besuche/Endlosschleifen $\Rightarrow$ **Merke** besuchte Knoten
- Am Ende wollen wir **zu jedem Knoten nen Weg** haben

Graphen durchlaufen – Tiefensuche

Intuitive Implementierung: Tiefensuche, Beta-Version

function DFS($G = (V, E)$, $s \in V$)
    $visited = ($**false**$, \ldots,$ **false**$)$: **array of** Boolean

    procedure DFS-step($u \in V$)
        foreach $(u, v) \in E$ **do**
            if $\neg visited[v]$ **then**
                $visited[v] :=$ **true**
                visit(v)   *// Do something with v*
                DFS-step(v)

    visit(s), $visited[s] :=$ **true**
    DFS-step(s)

Graphen durchlaufen – Tiefensuche

Karlsruher Institut für Technologie

Intuitive Implementierung: Tiefensuche, Release-Candidate

function DFS($G = (V, E)$, $s \in V$)

  $visited = ($**false**$, ..., $**false**$)$: **array of** Boolean

  $parent = (\bot, ..., \bot)$: **array of** V, $d = (0, ..., 0)$: **array of** $\mathbb{N}_0$

  procedure DFS-step($u \in V$)

    foreach $(u, v) \in E$ **do**

      if $\neg visited[v]$ **then**

        $visited[v] :=$ **true**, $parent[v] := u$, $d[v] := d[u] + 1$

        visit(v, u)   // Do something with v and u

        DFS-step(v)

  visit(s), $visited[s] :=$ **true**, $parent[s] := s$

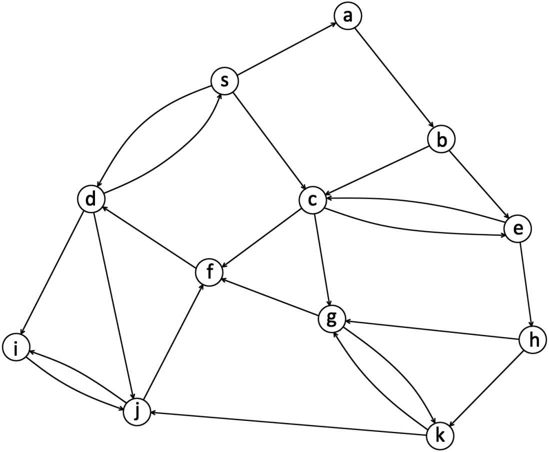
  DFS-step(s)

  return $(parent, d)$

Graphen durchlaufen – Tiefensuche

Aufgabe 1: Tiefe in freier Wildbahn

Führt auf diesem Graphen Tiefensuche von *s* ausgehend aus und malt die Laufwege hinein. Nachbarn werden in alphabetischer Reihenfolge besucht.

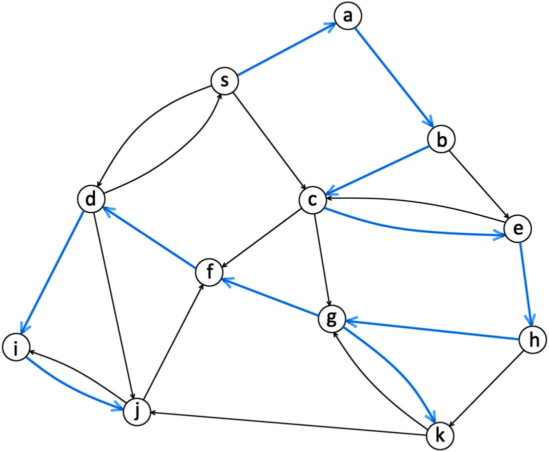


Graphen durchlaufen – Tiefensuche

Lösung zu Aufgabe 1

Besuchsreihenfolge:

s, a, b, c, e, h, g, f, d, i, j, k



Graphen durchlaufen – Tiefensuche

Karlsruher Institut für Technologie

- **Beobachtung**: Dringt schnell **tief** in den Graphen ein, anstatt sich „auszubreiten“ (daher der Name)
- **Laufzeit**?
- Etwas chaotische Laufwege – geht's auch organisierter?

Graphen durchlaufen – Tiefensuche

- **Beobachtung**: Dringt schnell **tief** in den Graphen ein, anstatt sich „auszubreiten“ (daher der Name)
- **Laufzeit**?

- Etwas chaotische Laufwege – geht's auch organisierter?

Graphen durchlaufen – Tiefensuche

- **Beobachtung**: Dringt schnell **tief** in den Graphen ein, anstatt sich „auszubreiten“ (daher der Name)
- **Laufzeit**? $\Theta(n + m)$

- Etwas chaotische Laufwege – geht's auch organisierter?

Graphen durchlaufen – Tiefensuche

- **Beobachtung**: Dringt schnell **tief** in den Graphen ein, anstatt sich „auszubreiten“ (daher der Name)
- **Laufzeit**? $\Theta(n+m)$

  In-place? **?**

- Etwas chaotische Laufwege – geht's auch organisierter?

Graphen durchlaufen – Tiefensuche

- **Beobachtung**: Dringt schnell **tief** in den Graphen ein, anstatt sich „auszubreiten“ (daher der Name)
- **Laufzeit**? $\Theta(n + m)$

  In-place? **Nein.** (wegen *visited*, *parent* und Rekursion)
- Etwas chaotische Laufwege – geht's auch organisierter?

Graphen durchlaufen – Tiefensuche

- **Beobachtung**: Dringt schnell **tief** in den Graphen ein, anstatt sich „auszubreiten“ (daher der Name)
- **Laufzeit**? $\Theta(n + m)$

  In-place? **Nein.** (wegen *visited*, *parent* und Rekursion)
- Etwas chaotische Laufwege – geht’s auch organisierter?

Graphen durchlaufen – Breitensuche

Organisierte Reihenfolge: Breitensuche (einfach)

procedure BFS($G = (V, E),\ s \in V$)
- $visited := ($**false**$, \ldots,$ **false**$)$
- $Q := \langle s \rangle$: Queue
- visit(s), $visited[s] :=$ **true**
- **while** $Q \neq \varnothing$ **do**
  - $u := Q$.popFront()
  - **foreach** $(u, v) \in E$ **do**
    - **if** $\neg visited[v]$ **then**
      - $visited[v] :=$ **true**
      - visit(v) *// Do something with v*
      - Q.pushBack(v)

Graphen durchlaufen – Breitensuche

Organisierte Reihenfolge: Breitensuche **mit Buchhaltung**

function BFS($G = (V, E),\ s \in V$) : $(parent, d)$

    $visited := ($**false**$, \ldots,$ **false**$)$, $parent := (\perp, \ldots, \perp)$, $d := (0, \ldots, 0)$
    $Q := \langle s \rangle$: Queue
    visit$(s, 0)$, $visited[s] :=$ **true**, $parent[s] := s$
    while $Q \neq \emptyset$ **do**
        $u := Q$.popFront()
        foreach $(u, v) \in E$ **do**
            if $\neg visited[v]$ **then**
                $visited[v] :=$ **true**, $parent[v] := u$, $d[v] := d[u] + 1$
                visit$(v, d[v])$ *// Do something with v and d[v]*
                Q.pushBack(v)

Graphen durchlaufen – Breitensuche

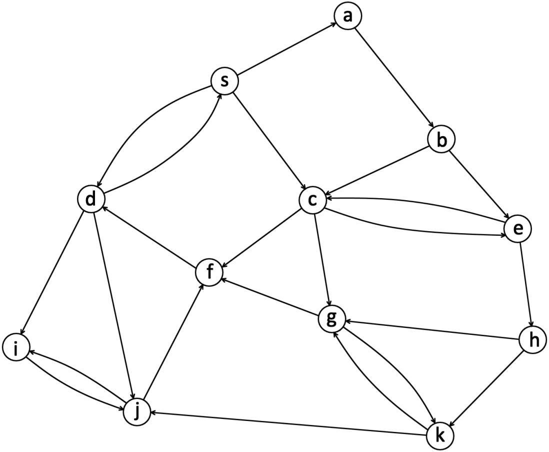
Organisierte Reihenfolge: Breitensuche **kompliziert** (siehe VL)

function BFS($G = (V, E),\ s \in V$) : ($parent, d$)
  $visited := ($**false**$, ..., $**false**$)$, $parent := (\bot, ..., \bot)$, $d := (0, ..., 0)$
  $Q := \langle s \rangle$, $Q' := \varnothing$ *// Extra queue Q'*
  visit(s, 0), $visited[s] :=$ **true**, $parent[s] := s$, $layer := 0$
  while $Q \neq \varnothing$ **do**
    $u := Q$.popFront()
    foreach $(u, v) \in E$ **do**
      if $\neg visited[v]$ **then**
        $visited[v] :=$ **true**, $parent[v] := u$, $d[v] := layer$
        visit(v, $layer$)   *// Do something with v and layer*
        Q'.pushBack(v)   *// Append to next-queue Q'*
    if $Q = \varnothing$ **then**
      $(Q, Q') := (Q', Q)$ *// New layer, so swap queues*
      $layer$++

Graphen durchlaufen – Breitensuche

Aufgabe 2: Volle Breitseite

Führt auf diesem Graphen Breitensuche von *s* ausgehend aus. Nachbarn werden in alphabetischer Reihenfolge besucht.

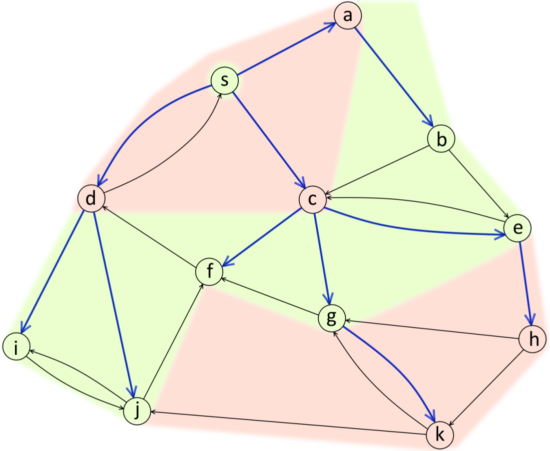


Graphen durchlaufen – Breitensuche

Lösung zu Aufgabe 2

Besuchsreihenfolge:

s, a, c, d, b, e, f, g, i, j, h, k



Graphen durchlaufen – Breitensuche

Karlsruher Institut für Technologie

- **Beobachtung**: **Breitet** sich schnell stark **aus** (daher der Name)
- Offensichtlich: Findet **kürzeste Pfade** (bei **ungewichteten** Kanten)
- **Laufzeit?**

Graphen durchlaufen – Breitensuche

- **Beobachtung**: **Breitet** sich schnell stark **aus** (daher der Name)
- Offensichtlich: Findet **kürzeste Pfade** (bei **ungewichteten** Kanten)
- Laufzeit?

Graphen durchlaufen – Breitensuche

- **Beobachtung**: **Breitet** sich schnell stark **aus** (daher der Name)
- Offensichtlich: Findet **kürzeste Pfade** (bei **ungewichteten** Kanten)
- **Laufzeit**?

Graphen durchlaufen – Breitensuche

- **Beobachtung**: **Breitet** sich schnell stark **aus** (daher der Name)
- Offensichtlich: Findet **kürzeste Pfade** (bei **ungewichteten** Kanten)
- **Laufzeit**? $\Theta(n + m)$

Graphen durchlaufen – Breitensuche

- **Beobachtung**: **Breitet** sich schnell stark **aus** (daher der Name)
- Offensichtlich: Findet **kürzeste Pfade** (bei **ungewichteten** Kanten)
- **Laufzeit**? $\Theta(n + m)$

  In-place **?**

Graphen durchlaufen – Breitensuche

- **Beobachtung**: **Breitet** sich schnell stark **aus** (daher der Name)
- Offensichtlich: Findet **kürzeste Pfade** (bei **ungewichteten** Kanten)
- **Laufzeit**? $\Theta(n + m)$

  In-place **Nein.** (wegen *visited*, Q und Q')

Graphen durchlaufen – Generell

- DFS/BFS finden **Pfade** von Startknoten *s* zu allen anderen erreichbaren Knoten
  $\Rightarrow$ *parent*-Array zum Rekonstruieren der Pfade
  ($parent[v]$: Vorgänger von *v* im Pfad zu *v*)
- DFS/BFS messen „**Distanz**“ der Knoten
  $\Rightarrow$ *d*-Array mit $d[v] =$ Anzahl Kanten auf dem Weg zu *v*

$\Rightarrow$ **Rückgabewerte** von BFS/DFS im Pseudocode benutzbar:
$(parent, d) := \text{BFS}(G, s)$ *// DFS similar*
// Now use $parent[\cdot]$ *and* $d[\cdot]$

- DFS/BFS finden **nur** alle von *s* **erreichbaren** Knoten

$\Rightarrow$ Um den **ganzen Graphen** abzudecken, müsst ihr von jedem noch nicht erreichten Knoten **extra** loslaufen *(„Tiefen-/Breitensuchwald“)*

Graphen durchlaufen – Generell

- DFS/BFS finden **Pfade** von Startknoten *s* zu allen anderen erreichbaren Knoten
  $\Rightarrow$ *parent*-Array zum Rekonstruieren der Pfade
  ($parent[v]$: Vorgänger von v im Pfad zu v)
- DFS/BFS messen „**Distanz**" der Knoten
  $\Rightarrow$ d-Array mit $d[v] =$ Anzahl Kanten auf dem Weg zu v

$\Rightarrow$ **Rückgabewerte** von BFS/DFS im Pseudocode benutzbar:
$(parent, d) := \text{BFS}(G, s)$ *// DFS similar*
// Now use $parent[\cdot]$ *and* $d[\cdot]$

- DFS/BFS finden **nur** alle von *s* **erreichbaren** Knoten

$\Rightarrow$ Um den **ganzen Graphen** abzudecken, müsst ihr von jedem noch nicht erreichten Knoten **extra** loslaufen *(„Tiefen-/Breitensuchwald")*

Graphen durchlaufen – Kantentypen

- Bei BFS/DFS „entlanggelaufene“ Kanten **bilden Baum** (da kein Knoten zweimal besucht!)

⇒ Teile Kanten ein:

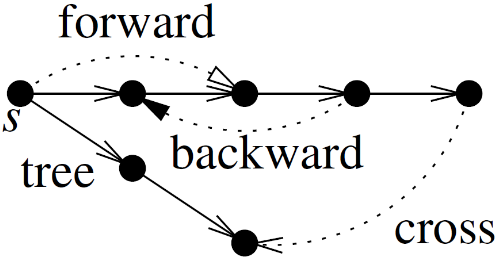
tree-:	„Entlanggelaufene“ Kanten des Baumes
cross-:	Kanten **zwischen** versch. „**Ästen**“ im Baum
backward-:	Kanten, die **rückwärts** zu (einer/mehreren) *tree*-Kanten laufen
forward-:	Kanten, die **mehrere** *tree*-Kanten „**überholen**“

Graphen durchlaufen – Kantentypen

- Bei BFS/DFS „entlanggelaufene“ Kanten **bilden Baum** (da kein Knoten zweimal besucht!)

⇒ Teile Kanten ein:

tree-: „Entlanggelaufene“ Kanten des Baumes
cross-: Kanten **zwischen** versch. **„Ästen“** im Baum
backward-: Kanten, die **rückwärts** zu (einer/mehreren) *tree*-Kanten laufen
forward-: Kanten, die **mehrere** *tree*-Kanten **„überholen“**

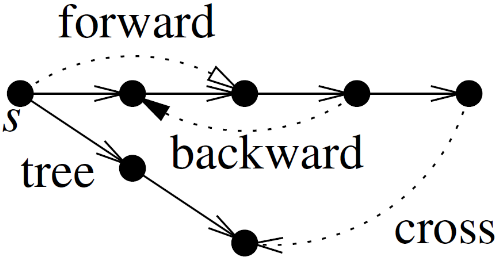


Graphen durchlaufen – Kantentypen

- Bei BFS/DFS „entlanggelaufene“ Kanten **bilden Baum** (da kein Knoten zweimal besucht!)

$\Rightarrow$ Teile Kanten ein:

tree-: „Entlanggelaufene“ Kanten des Baumes
cross-: Kanten **zwischen** versch. **„Ästen“** im Baum
backward-: Kanten, die **rückwärts** zu (einer/mehreren) *tree*-Kanten laufen
forward-: Kanten, die **mehrere** *tree*-Kanten „**überholen**“

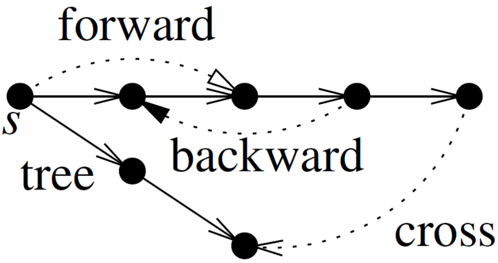


Graphen durchlaufen – Kantentypen

- Bei BFS/DFS „entlanggelaufene“ Kanten **bilden Baum** (da kein Knoten zweimal besucht!)

$\Rightarrow$ Teile Kanten ein:

tree-: „Entlanggelaufene“ Kanten des Baumes
cross-: Kanten **zwischen** versch. **„Ästen“** im Baum
backward-: Kanten, die **rückwärts** zu (einer/mehreren) *tree*-Kanten laufen
forward-: Kanten, die **mehrere** *tree*-Kanten „überholen“

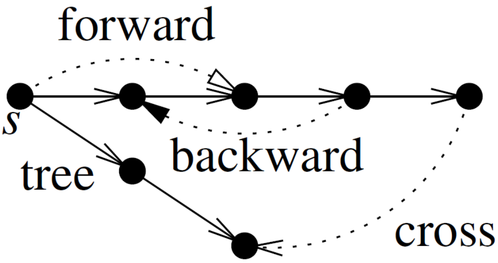


Graphen durchlaufen – Kantentypen

- Bei BFS/DFS „entlanggelaufene“ Kanten **bilden Baum** (da kein Knoten zweimal besucht!)

$\Rightarrow$ Teile Kanten ein:

tree-: „Entlanggelaufene“ Kanten des Baumes
cross-: Kanten **zwischen** versch. **„Ästen“** im Baum
backward-: Kanten, die **rückwärts** zu (einer/mehreren) *tree*-Kanten laufen
forward-: Kanten, die **mehrere** *tree*-Kanten „**überholen**“

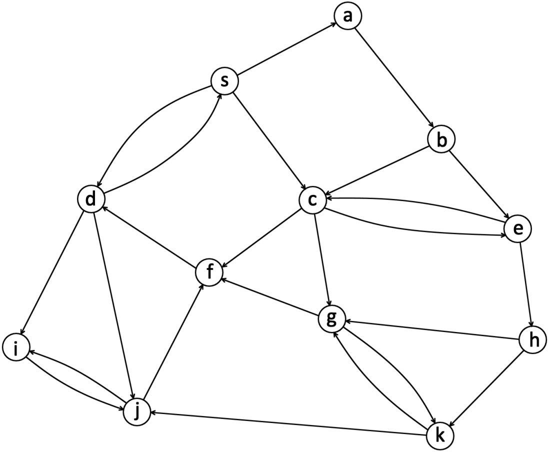


Graphen durchlaufen – Kantentypen

Karlsruher Institut für Technologie

Aufgabe 3: Die Graphschaft besichtigen

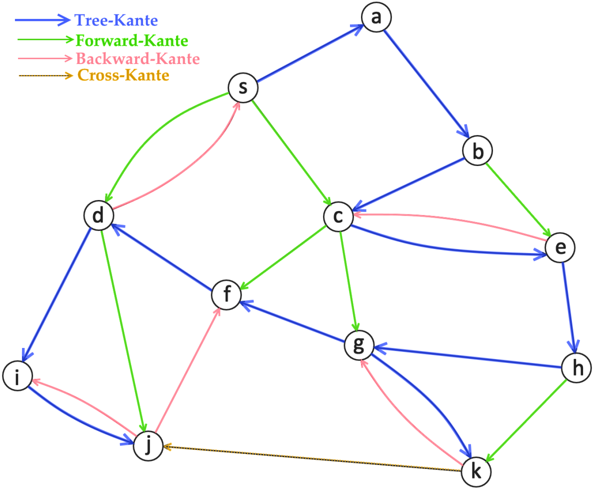
Betrachtet die vorhin durchgespielte Tiefen- und Breitensuche und klassifiziert jeweils alle Kanten entsprechend.



Graphen durchlaufen – Kantentypen

Lösung zu Aufgabe 3

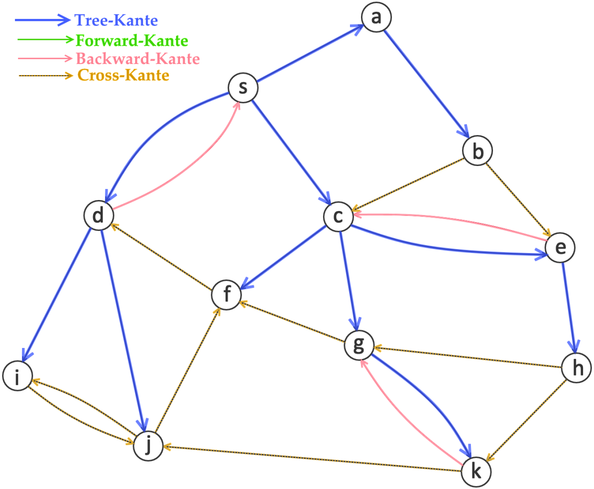
Für Tiefensuche:



Graphen durchlaufen – Kantentypen

Lösung zu Aufgabe 3

Für Breitensuche:



Graphen durchlaufen – Kantentypen

Karlsruher Institut für Technologie

Gibt es eine Art von Kante, die bei Breitensuche nicht auftreten kann? Falls ja, warum? **?**

Graphen durchlaufen – Kantentypen

Gibt es eine Art von Kante, die bei Breitensuche nicht auftreten kann? Falls ja, warum?

forward-Kanten können **nicht** auftreten.
(BFS bestimmt schon den Pfad mit kleinster Kantenanzahl.)

Graphen durchlaufen – Kantentypen

Gibt es eine Art von Kante, die bei Breitensuche nicht auftreten kann? Falls ja, warum?

forward-Kanten können **nicht** auftreten.
(BFS bestimmt schon den Pfad mit kleinster Kantenanzahl.)

Gibt es eine Art von Kante, die bei Tiefensuche nicht auftreten kann? Falls ja, warum? **?**

Graphen durchlaufen – Kantentypen

Gibt es eine Art von Kante, die bei Breitensuche nicht auftreten kann? Falls ja, warum?

forward-Kanten können **nicht** auftreten.
(BFS bestimmt schon den Pfad mit kleinster Kantenanzahl.)

Gibt es eine Art von Kante, die bei Tiefensuche nicht auftreten kann? Falls ja, warum?

Bei Tiefensuche können **alle** Arten von Kanten auftreten.

Graphen durchlaufen – Kantentypen

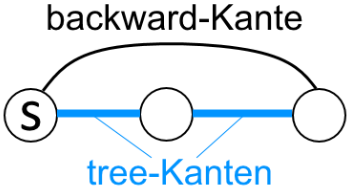
Karlsruher Institut für Technologie

Gibt es eine Art von Kante, die bei Tiefensuche **auf ungerichteten Graphen** nicht auftreten kann? Falls ja, warum? **?**

Graphen durchlaufen – Kantentypen

Gibt es eine Art von Kante, die bei Tiefensuche **auf ungerichteten Graphen** nicht auftreten kann? Falls ja, warum?

cross-Kanten können nicht auftreten:
Wäre nämlich schon **vorher** entlanggelaufen worden (da ungerichtet!).
Die einzigen Kanten, die hier das Ende eines Tiefensuch-Astes markieren können, sind *backward*-/*forward*-Kanten.
(Ob man die jetzt *backward*- oder *forward*- nennt, ist wurscht, sind ja faktisch **beides**.) Bsp. dazu:



Graphen durchlaufen – Kantentypen

Sind *cross*-Kanten eindeutig? Falls ja, warum? **?**

Graphen durchlaufen – Kantentypen

Sind *cross*-Kanten eindeutig? Falls ja, warum?

cross-Kanten sind genau dann eindeutig, wenn der zugehörige Baum eindeutig ist. $\Rightarrow$ I. A. **nicht** der Fall (da Nachbarn i. A. nicht in bestimmter Reihenfolge gewählt).

Graphen durchlaufen – Kantentypen

Sind *cross*-Kanten eindeutig? Falls ja, warum?

cross-Kanten sind genau dann eindeutig, wenn der zugehörige Baum eindeutig ist. $\Rightarrow$ I. A. **nicht** der Fall (da Nachbarn i. A. nicht in bestimmter Reihenfolge gewählt).

Nach welcher Strategie muss bei Tiefensuche die Reihenfolge der rekursiven Abstiege (also die Reihenfolge der Nachbarn) gewählt werden, damit keine *forward*-Kanten auftreten? **?**

Graphen durchlaufen – Kantentypen

Sind *cross*-Kanten eindeutig? Falls ja, warum?

cross-Kanten sind genau dann eindeutig, wenn der zugehörige Baum eindeutig ist. $\Rightarrow$ I. A. **nicht** der Fall (da Nachbarn i. A. nicht in bestimmter Reihenfolge gewählt).

Nach welcher Strategie muss bei Tiefensuche die Reihenfolge der rekursiven Abstiege (also die Reihenfolge der Nachbarn) gewählt werden, damit keine *forward*-Kanten auftreten?

Fangfrage! :P

Es gibt **keine** solche Strategie; *forward*-Kanten bei DFS in manchen Fällen unvermeidbar.

Graphen durchlaufen

Aufgabe 4: Best-Friend-Search
Das Kleine-Welt-Phänomen besagt, dass jeder Mensch mit jedem anderen über maximal sechs Ecken bekannt ist.

Wir betrachten einen *Freundschaftsgraphen* eines beliebigen Überwachungsnetzwerkes™, bei dem Menschen als Knoten und ihre Freundschaften als Kanten dargestellt sind. Weiter nehmen wir vereinfachend an, jeder Nutzer habe etwa 100 Freunde (dies entspricht dem Durchschnitt).

Überlegt euch ein Verfahren, mit welchem in einem Freundschaftsgraphen möglichst schnell zwischen zwei gegebenen Menschen ein „Bekanntheitspfad“ gefunden werden kann.

Graphen durchlaufen

Lösung zu Aufgabe 4

- **Intuitiv**: BFS vom Startmenschen, bis man den Zielmenschen gefunden hat
- Laufzeit: Pro Schicht **verhundertfacht** sich die Anzahl der Kanten
  ⇒ Zu überprüfende Kanten:
  $100^6 = 10^{12} \gg 7.5 \cdot 10^9 \approx$ #People on Earth!

⇒ **Besser**: **Zwei** BFS **parallel** vom Start und vom Ziel aus
⇒ Zu überprüfende Kanten: $2 \cdot 100^3 = 2 \cdot 10^6$
⇒ Um Faktor 10^6 **schneller**!

Graphen durchlaufen

Karlsruher Institut für Technologie

Lösung zu Aufgabe 4

- **Intuitiv**: BFS vom Startmenschen, bis man den Zielmenschen gefunden hat
- Laufzeit: Pro Schicht **verhundertfacht** sich die Anzahl der Kanten
  $\Rightarrow$ Zu überprüfende Kanten:
  $100^6 = 10^{12} \gg 7.5 \cdot 10^9 \approx$ #People on Earth!

$\Rightarrow$ **Besser: Zwei** BFS **parallel** vom Start und vom Ziel aus
$\Rightarrow$ Zu überprüfende Kanten: $2 \cdot 100^3 = 2 \cdot 10^6$
$\Rightarrow$ Um Faktor 10^6 **schneller**!

Graphen durchlaufen

Lösung zu Aufgabe 4

- **Intuitiv**: BFS vom Startmenschen, bis man den Zielmenschen gefunden hat
- Laufzeit: Pro Schicht **verhundertfacht** sich die Anzahl der Kanten
  $\Rightarrow$ Zu überprüfende Kanten:
  $100^6 = 10^{12} \gg 7.5 \cdot 10^9 \approx$ #People on Earth!

$\Rightarrow$ **Besser: Zwei** BFS **parallel** vom Start und vom Ziel aus
$\Rightarrow$ Zu überprüfende Kanten: $2 \cdot 100^3 = 2 \cdot 10^6$
$\Rightarrow$ Um Faktor 10^6 **schneller**!

Graphen durchlaufen

Aufgabe 5: Tiefensuche revisited
Implementiert Tiefensuche nicht-rekursiv als Pseudocode. Das asymptotische Laufzeitverhalten von eigentlicher Tiefensuche darf hierbei nicht überschritten werden.

Graphen durchlaufen

Lösung zu Aufgabe 5

Recursion-Faking mittels Stack:

procedure DFS$\big(G = (V, E),\ s \in V\big)$
- $S := \langle s \rangle$: Stack
- $visited := ($**false**, ..., **false**$)$
- visit(s), $visited[s] :=$ **true**
- **while** $S \neq \varnothing$ **do**
  - $u := S.\text{popBack}()$
  - **foreach** $(u, v) \in E$ **do**
    - **if** $\neg visited[v]$ **then**
      - visit(v) *// Do something with v*
      - $visited[v] :=$ **true**
      - $S.\text{pushBack}(v)$

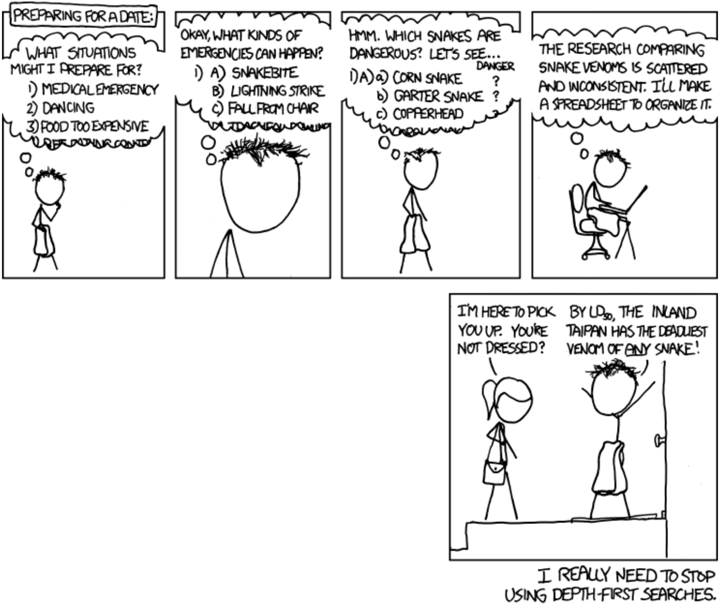
Graphen durchlaufen

Lösung zu Aufgabe 5

Zum Vergleich: Breitensuche mit Queue

procedure BFS$(G = (V, E),\ s \in V)$
- $Q := \langle s \rangle$: Queue
- $visited := ($**false**$, \ldots,$ **false**$)$
- visit(s), $visited[s] :=$ **true**
- **while** $Q \neq \varnothing$ **do**
  - $u := Q$.popFront()
  - **foreach** $(u, v) \in E$ **do**
    - **if** $\neg visited[v]$ **then**
      - visit(v) *// Do something with v*
      - $visited[v] :=$ **true**
      - Q.pushBack(v)

$\Rightarrow$ Der Apfel fällt nicht weit vom Tiefensuchbaum... :P



http://xkcd.com/761