Tiefensuche, Breitensuche

Tiefesuche

Weiterführende Literatur:

- *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 6*, Seite 39-52 (PDF 32-45)
- *Wikipedia-Artikel „Tiefensuche“*
- Schneider, *Taschenbuch der Informatik*, Kapitel 6.2.2.2 Graphalgorithmen, Seite 185

Die Tiefensuche (englisch *depth-first search*, *DFS*) ist in der Informatik ein Verfahren zum Suchen von Knoten in einem Graphen.

depth-first search

DFS

Der Tiefendurchlauf ist das Standardverfahren zum Durchlaufen eines Graphen bei dem *jeder Knoten mindestens einmal* und *jede Kante genau einmal* besucht wird. Man geht vom jeweiligen Knoten *erst zu einem nicht besuchten Nachbarknoten* und setzt den Algorithmus dort *rekursiv* fort. Bei schon besuchten Knoten wird abgebrochen. Als Hilfsstruktur wird ein *Stack* (Stapelspeicher, Keller) verwendet. Der konkrete Durchlauf hängt von der Reihenfolge der Knoten in den Adjazenzlisten bzw. in der Adjazenzmatrix ab.[1]

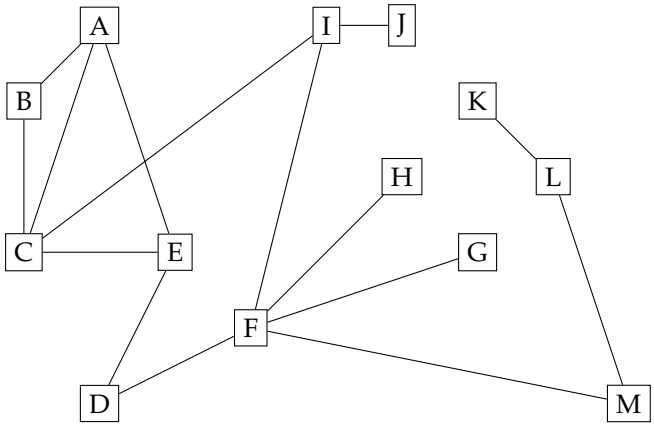
jeder Knoten mindestens einmal

jede Kante genau einmal

erst zu einem nicht besuchten Nachbarknoten

rekursiv

Stack



Startknoten A:

A:

Besuchte Knoten: A, B

Stack: A

B:

Besuchte Knoten: A, B, C

Stack: A, B

[1] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 6*, Seite 40.

C:

Besuchte Knoten: A, B, C, E

Stack: A, B, C

B: A, B, C

Stack: A, B, C

C: A, B, C, E

Stack: A, B, C, E

E: A, B, C, E, D

Stack: A, B, C, E

E: A und C schon markiert A, B, C, E, D

Stack: A, B, C, E, D

D: A, B, C, E, D, F

Stack: A, B, C, E, D

F: A, B, C, E, D, F, M

Stack: A, B, C, E, D, F

M: A, B, C, E, D, F, M, L

Stack: A, B, C, E, D, F, M

L: A, B, C, E, D, F, M, L, K

Stack: A, B, C, E, D, F, M, L

K: keine Kinder A, B, C, E, D, F, M, L, K

Stack: A, B, C, E, D, F, M, L

L: alle Kinder schon markiert A, B, C, E, D, F, M, L, K

Stack: A, B, C, E, D, F, M

M: alle Kinder schon markiert A, B, C, E, D, F, M, L, K

Stack: A, B, C, E, D, F

F: A, B, C, E, D, F, M, L, K, G

Stack: A, B, C, E, D, F

G: keine Kinder

F: A, B, C, E, D, F, M, L, K, G, H

Stack: A, B, C, E, D, F

H: keine Kinder

F: A, B, C, E, D, F, M, L, K, G, H, I

Stack: A, B, C, E, D, F

I: A, B, C, E, D, F, M, L, K, G, H, I, J

Stack: A, B, C, E, D, F, I

J: keine Kinder I: alle Kinder markiert F: alle Kinder markiert D: alle Kinder markiert E: alle Kinder markiert C: alle Kinder markiert B: alle Kinder markiert A: alle Kinder markiert

A, B, C, E, D, F, M, L, K, G, H, I, J

Stack: leer

ENDE

Implementierung der Tiefensuche

- Eine Möglichkeit, abzuspeichern, welche Knoten bereits besucht wurden → Boolean-Array

- Eine Methode, die für uns diese Markierung der Knoten als besucht übernimmt (und somit die eigentliche Tiefensuche durchführt) → Knoten als besucht eintragen, existierende Nachbarknoten suchen und prüfen, ob diese bereits besucht wurden, falls nicht: diese durch rekursiven Aufruf besuchen

- Eine Methode, um die Tiefensuche zu starten → Wenn ein übergebnisener Startknoten existiert, dann müssen erst alle Knoten als nicht besucht markiert werden und dann vom Startknoten aus das Besuchen der Knoten gestartet werden

[2]

Pseudocode: Tiefensuche mit explizitem Stack[3]

```
1  funktion dfs(G: Graph, k: Startknoten in G) {
2    S := leerer Stack;
3    lege k oben auf S;
4    markiere k;
5
6    solange S nicht leer ist fuehre aus {
7      a := entferne oberstes Element von S;
8      bearbeite Knoten a;
9
10     fuer alle Nachfolger n von a fuehre aus {
11       falls n noch nicht markiert fuehre aus {
12         lege n oben auf S;
13         markiere n;
14       }
15     }
16   }
17 }
```

[2] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 6*, Seite 45.
[3] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 6*, Seite 51 (PDF 45).

```
34
35      /**
36       * Durchlauf aller Knoten und Ausgabe auf der Konsole
37       *
38       * @param knotenNummer Nummer des Startknotens
39       */
40      public void besucheKnoten(int knotenNummer) {
41        besucht[knotenNummer] = true;
42        stapel.push(gibKnotenName(knotenNummer));
43        // Stapel ausgeben
44        System.out.println(Farbe.grün("Stapel: ") + stapel.toString());
45        while (!stapel.isEmpty()) {
46          // oberstes Element des Stapels nehmen und in die Route einfügen
47          String knotenName = stapel.pop();
48          System.out.println(Farbe.rot("besucht: ") + knotenName);
49          route.push(knotenName);
50
51          // alle nicht besuchten Nachbarn von w in den Stapel einfügen
52          for (int abzweigung = 0; abzweigung <= gibKnotenAnzahl() - 1; abzweigung++)
             ↪ {
53            if (matrix[gibKnotenNummer(knotenName)][abzweigung] > 0 &&
               ↪  !besucht[abzweigung]) {
54              besucht[abzweigung] = true;
55              stapel.push(gibKnotenName(knotenNummer));
56            }
57          }
58          // Stapel ausgeben
59          System.out.println(Farbe.grün("Stapel: ") + stapel.toString());
60        }
61        // Route ausgeben
62        System.out.println(Farbe.gelb("Route: ") + route.toString());
63      }
64
65      /**
66       * Start der Tiefensuche
67       *
68       * @param startKnoten Bezeichnung des Startknotens
69       */
70      public void starteTiefenSucheStapel(String startKnoten) {
71        int startnummer;
72        startnummer = gibKnotenNummer(startKnoten);
73
74        if (startnummer != -1) {
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/graph/algorithmen/TiefenSucheStapel.java`

Breitensuche

Weiterführende Literatur:

- *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 6*, Seite 53-64 (PDF 46-57)
- Schneider, *Taschenbuch der Informatik*, Kapitel 6.2.2.2 Graphalgorithmen, Seite 185
- *Wikipedia-Artikel „Breitensuche“*

breadth-first search
BFS

erst zu allen Nachbarknoten
bevor deren Nachbarn

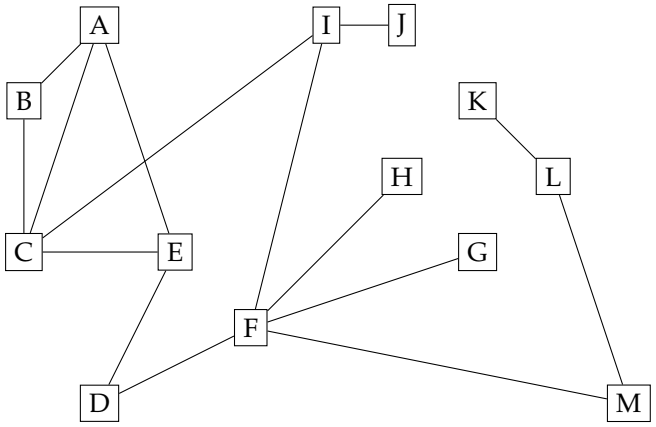
Der Breitendurchlauf (englisch *breadth-first search*, *BFS*)[4] ist Verfahren zum Durchlaufen eines Graphen bei dem jeder Knoten genau einmal besucht wird. Man geht von einem Knoten *erst zu allen Nachbarknoten bevor deren Nachbarn* besucht

[4] *Wikipedia-Artikel „Breitensuche“*.

werden. Bei schon besuchten Knoten wird abgebrochen. Als Hilfsstruktur wird eine *Queue* (*Warteschlange*) verwendet.

Queue (Warteschlange)

Der konkrete Durchlauf hängt von der Reihenfolge der Knoten in den Adjazenzlisten ab.



A: A, B, C, E w: B, C, E

B: alle Kinder markiert

A, B, C, E w: C, E

C: A, B und E markiert

A, B, C, E, I w: E, I

E: A und C markiert

A, B, C, E, I, D w: I, D

I: C markiert

A, B, C, E, I, D, F, J w: D, F, J

D: E, F sind markiert A, B, C, E, I, D, F, J w: F, J

F: I ist markiert A, B, C, E, I, D, F, J, M w: J, M, G, H

J: keine Kinder

A, B, C, E, I, D, F, J, M, G, H w: M, G, H

M: A, B, C, E, I, D, F, J, M, G, H, L w: G, H, L

G: keine Kinder A, B, C, E, I, D, F, J, M, G, H, L w: H, L

H: keine Kinder A, B, C, E, I, D, F, J, M, G, H, L w: L

L: A, B, C, E, I, D, F, J, M, G, H, L, K w: K

K: keine Kinder A, B, C, E, I, D, F, J, M, G, H, L, K w:

ENDE

Pseudocode Breitensuche mit Queue[5]

```
funktion bfs(G: Graph, k: Startknoten in G) {
  Q := leere Queue:
  fuege k in Q ein;
  markiere k;

  solange Q nicht leer ist fuehre aus {
    a := entferne vorderstes Element aus Q;
    bearbeite Knoten a;

    fuer alle Nachfolger n von a fuehre aus {
      falls n noch nicht markiert fuehre aus {
        fuege n hinten in Q ein;
        markiere n;
      }
    }
  }
}
```

```
   * @param knotenNummer Nummer des Startknotens
   */
  private void besucheKnoten(int knotenNummer) {
    besucht[knotenNummer] = true;
    liste.add(gibKnotenName(knotenNummer));
    // Liste gibMatrixAus
    System.out.println(Farbe.grün("Warteschlange: ") + liste.toString());
    while (!liste.isEmpty()) {
      // oberstes Element der Liste nehmen und in die Route einfügen
      String knotenName = liste.remove(0);
      System.out.println(Farbe.rot("besucht: ") + knotenName);
      route.add(knotenName);

      // alle nicht besuchten Nachbarn von knotenName in die Liste einfügen
      for (int abzweigung = 0; abzweigung <= gibKnotenAnzahl() - 1; abzweigung++)
      ↪ {
        if (matrix[gibKnotenNummer(knotenName)][abzweigung] > 0 &&
        ↪ !besucht[abzweigung]) {
          besucht[abzweigung] = true;
          liste.add(gibKnotenName(abzweigung));
        }
      }
      // Liste ausgeben
      System.out.println(Farbe.grün("Warteschlange: ") + liste.toString());
    }
    // Route ausgeben
    System.out.println(Farbe.gelb("Route: ") + route.toString());
  }

  /**
   * Start der Breitensuche
   *
   * @param startKnoten Bezeichnung des Startknotens
   */
  public void starteBreitenSuche(String startKnoten) {
    int startnummer;
    startnummer = gibKnotenNummer(startKnoten);

    if (startnummer != -1) {
      for (int i = 0; i <= gibKnotenAnzahl() - 1; i++) {
```

[5] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 6*, Seite 64 PDF (56).

```
75            besucht[i] = false;
76          }
77          besucheKnoten(startnummer);
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/graph/algorithmen/BreitenSucheWarteschlange.java`

Literatur

[1] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 6. Graphen*. https://www.studon.fau.de/file2635324_download.html.

[2] Uwe Schneider. *Taschenbuch der Informatik*. 7. Aufl. Hanser, 2012. ISBN: 9783446426382.

[3] *Wikipedia-Artikel „Breitensuche“*. https://de.wikipedia.org/wiki/Breitensuche.

[4] *Wikipedia-Artikel „Tiefensuche“*. https://de.wikipedia.org/wiki/Tiefensuche.