

09:41 Dienstag 9. Jan. 100 %

Fertig

Adjazenzliste bei ungerichteten Graphen

- Speichere für jeden Knoten eine linear verkettete Liste mit allen Nachbarknoten (Adjazenzliste).
- Kanten werden bei beiden Endknoten eingetragen.

The diagram shows an undirected graph with 5 nodes labeled v0 through v4. Node v0 has a self-loop and edges to v1 and v4. Node v1 has edges to v0, v2, and v4. Node v2 has an edge to v1 and an edge to v3. Node v3 has an edge to v2. Node v4 has edges to v0 and v1. To the right, the adjacency lists are represented as an array of arrays [0] through [4]. Each entry [i][j] indicates an edge from node i to node j. Red circles highlight specific entries: [1][1] (edge from v1 to v1), [1][2] (edge from v1 to v2), [1][4] (edge from v1 to v4), [2][1] (edge from v2 to v1), [2][3] (edge from v2 to v3), [2][4] (edge from v2 to v4), [4][0] (edge from v4 to v0), and [4][1] (edge from v4 to v1). Blue circles highlight [1][1], [1][2], and [1][4].

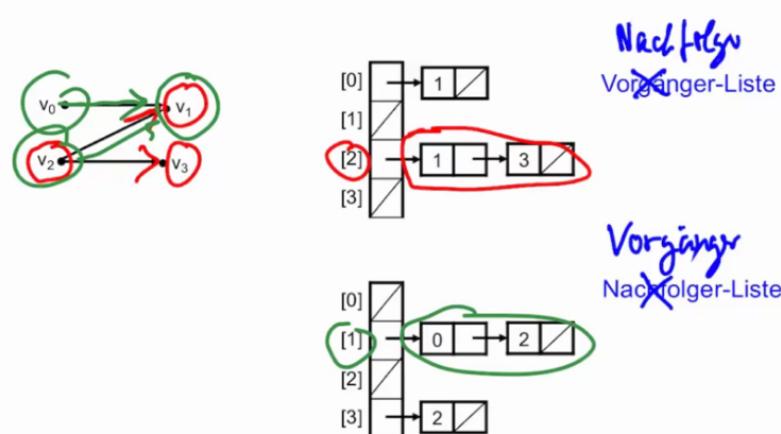
- Bei einem **gewichteten Graphen** wird noch zusätzlich das Kantengewicht eingetragen.
- Um die Dynamisierbarkeit der Datenstruktur zu verbessern, kann statt eines Feldes auch eine dynamische Datenstruktur wie ein binärer Suchbaum verwendet werden.

Prof. Dr. O. Bittel, HTWG Konstanz Algorithmen und Datenstrukturen – Einführung in Gra... WS 20/21 6-28

Fertig

Adjazenzliste bei gerichteten Graphen

- Bei einem gerichteten Graphen werden zwei getrennte Listen verwaltet: eine Vorgänger-Liste und eine Nachfolger-Liste.



Viewing Oliver Bittel's ap...

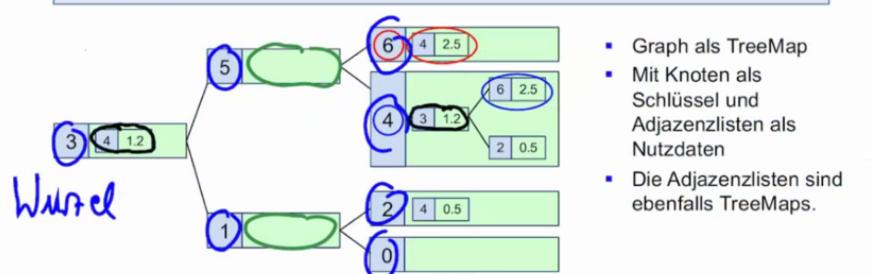
Fertig

Beispiel: ungerichteter und gewichteter Graph als TreeMap

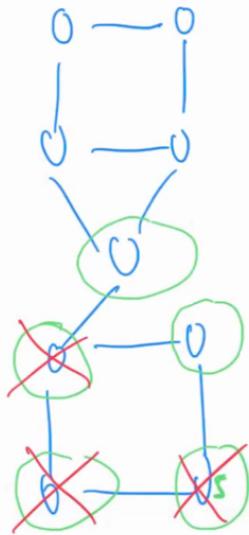
```

int n = 7; // Anzahl Knoten
Map<Integer, Map<Integer, Double>> g = new TreeMap<>(); // ungerichteter und gewichteter Graph
for (int i = 0; i < n; i++) {
    g.put(i, new TreeMap<>());
}
g.get(4).put(3, 1.2); // Kante von 4 nach 3 mit Gewicht 1.2 eintragen
g.get(3).put(4, 1.2);
g.get(4).put(6, 2.5); // Kante von 4 nach 6 mit Gewicht 2.5 eintragen
g.get(6).put(4, 2.5);
g.get(4).put(2, 0.5); // Kante von 4 nach 2 mit Gewicht 0.5 eintragen
g.get(2).put(4, 0.5);

```



- Graph als TreeMap
- Mit Knoten als Schlüssel und Adjazenzlisten als Nutzdaten
- Die Adjazenzlisten sind ebenfalls TreeMaps.



Systemat. Besuchen
aller Knoten.

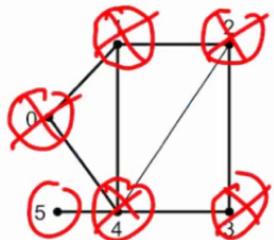
○ Kandidaten

✗ besucht

- Stack → Tiefensuche
- Queue → Breitensuche
- PrioQueue → Dijkstra

S7-6 schauen in video, um es auszufüllen

Breitensuche mit Queue



Bitte ausfüllen

Schlange



Besuchsreihenfolge:

1, 0, 2, 4, 3, 5

09:41 Dienstag 9. Jan.

$T = \mathcal{O}(|V| + |E|)$

$|V| = \text{Anz. Knoten}$

$|E| = \text{Anz. Kanten}$

Dichte Graphen

$|E| = \mathcal{O}(|V|^2)$

$\Rightarrow T = \mathcal{O}(|V|^2)$

Dünne Graphen

$|E| = \mathcal{O}(|V|)$

$\Rightarrow T = \mathcal{O}(|V|)$

Rekursive Tiefensuche (depth-first search)

```
void visitDF(Vertex v, Graph g) {  
    Set<Vertex> besucht = Ø;  
    visitDF(v, g, besucht);  
}  
  
void visitDF(Vertex v, Graph g, Set<Vertex> besucht) {  
    besucht.add(v);  
  
    // Bearbeite v:  
    println(v);  
  
    for (jeden adjazenten Knoten w von v)  
        if (!besucht.contains(w)) // w noch nicht besucht  
            visitDF(w, g, besucht);  
}
```

visitDF(v,g) startet von Knoten v eine Tiefensuche im Graph g.

besucht ist die Menge aller bereits besuchten Knoten.

Wichtig zur Vermeidung von Endlosschleifen.

visitDF(v, g, besucht) besucht Knoten v im Graphen g und ist rekursiv.

Keh. Aufr. $\geq 10.000 \rightarrow$ Stack-Over-flow

Iterative Tiefensuche mit einem Keller

```
void visitDF(Vertex v, Graph g) {  
    Set<Vertex> besucht = Ø;  
    visitDF(v, g, besucht);  
}  
  
void visitDF(Vertex v, Graph g, Set<Vertex> besucht) {  
    Stack<Vertex> stk;  
    stk.push(v);  
  
    while(!stk.empty()) {  
        v = stk.pop();  
        if (besucht.contains(v))  
            continue;  
  
        besucht.add(v);  
  
        // Bearbeite v:  
        println(v);  
  
        for (jeden adjazenten Knoten w von v)  
            if (!besucht.contains(w))  
                stk.push(w);  
    }  
}
```

visitDF startet von Knoten v eine Tiefensuche.

visitDF besucht Knoten v im Graphen g, wobei die bereits besuchten Knoten in besucht abgespeichert werden.

Im Keller stk werden alle Knoten verwaltet, die als nächstes zu besuchen sind.

Die Tiefensuche wird erreicht durch die LIFO-Organisation des Kellers.

Um die gleiche Besuchsreihenfolge wie bei der rekursiven Funktion zu erreichen, müssen in der for-Schleife die Nachbarn in umgekehrter Reihenfolge bearbeitet werden.

Beachte: Gleiche Knoten können mehrfach eingekellert werden. Das ließe sich durch eine weitere Knotenmarkierung verhindern:

- nicht besucht und nicht im Keller,
- nicht besucht und im Keller,
- besucht.

Allerdings würde sich eine andere Besuchsreihenfolge wie bei der rekursiven Funktion ergeben.

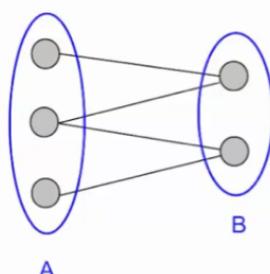
→ Nicht ArrayList sondern ArrayDeque

→ Iterative Tiefensuche ist nicht unbedingt schneller, vielleicht sogar langsamer, aber hier

hat man das Stackoverflow Problem nicht.

Bipartiter Graph

- Ein ungerichteter Graph $G = (V, E)$ heißt **bipartit**, falls sich V disjunkt in A und B zerlegen lässt, so dass für jede Kante $(u, v) \in E$ gilt:
 - $u \in A$ und $v \in B$ oder
 - $u \in B$ und $v \in A$.



Es gibt nur Kanten zwischen A und B.

$$V = A \cup B \quad u. \quad A \cap B = \emptyset$$

Tiefensuche zur Prüfung der Bipartitheit

- Besuche mit Tiefensuche alle Knoten und färbe abwechselnd mit rot oder grün ein bzw. prüfe ob Färbung OK.

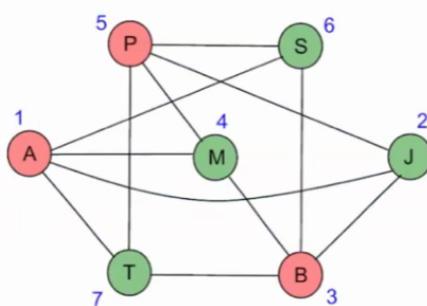
```
void pruefeBipartheit() {
    for (jeden Knoten v)
        if (v hat noch keine Farbe)
            färbe(v, "rot")
    print("Graph ist bipartit");
}
```

```
void färbe(Vertex v, Farbe f) {
    if (v bereits gefärbt) {
        if (Farbe von v ist nicht f) {
            print("Graph ist nicht bipartit");
            beende Prüfung;
        }
    } else {
        färbe v mit f ein;
        for (jeden Nachbarn w von v)
            färbe(w, flip(f));
    }
}
```

flip("rot") = "grün" und
flip("grün") = "rot".

A B

Tiefensuche mit alphabetischer Reihenfolge.
Reihenfolge der Färbung in blau.



Tiefensuche zur Prüfung der Bipartitheit

- Besuche mit Tiefensuche alle Knoten und färbe abwechselnd mit rot oder grün ein bzw. prüfe ob Färbung OK.

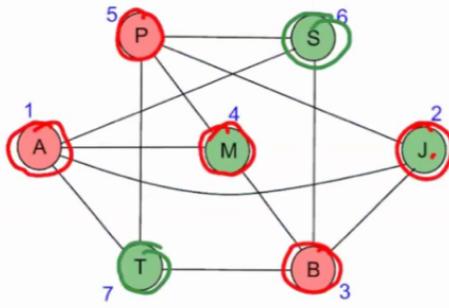
```
void pruefeBipartitheit() {
    for (jeden Knoten v)
        if (v hat noch keine Farbe)
            färbe(v, "rot")
    print("Graph ist bipartit");
}
```

```
void färbe(Vertex v, Farbe f) {
    if (v bereits gefärbt) {
        if (Farbe von v ist nicht f) {
            print("Graph ist nicht bipartit");
            beende Prüfung;
        }
    } else {
        färbe v mit f ein;
        for (jeden Nachbarn w von v)
            färbe(w, flip(f));
    }
}
```

flip("rot") = "grün" und
flip("grün") = "rot".

A B

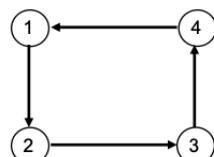
Tiefensuche mit alphabetischer Reihenfolge.
Reihenfolge der Färbung in blau.



Eigenschaften und Anwendungen

Eigenschaften:

- Falls der Graph einen Zyklus enthält, dann existiert keine topologische Sortierung und umgekehrt.



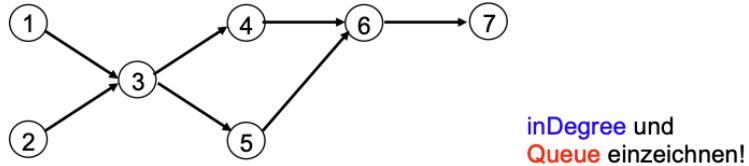
- Falls ein Graph topologisch sortiert werden kann, dann ist im allgemeinen die topologische Sortierung nicht eindeutig.

Beispiele für Anwendungen:

- Stelle fest, ob es eine Durchführungsreihenfolge für die Aktivitäten oder auch Prozesse in einem Vorranggraphen gibt.
- Prüfe, ob Vererbungsgraph oder include-Graph zyklenfrei ist.

→ Schau video hier für Graph

Idee für topologische Sortierung



- Speichere für jeden Knoten v:
 $\text{inDegree}[v]$ = Anzahl der noch nicht besuchten Vorgänger
- Halte alle noch nicht besuchten Knoten v mit $\text{inDegree}[v] == 0$ als Kandidat in einer Queue q.
- Besuche immer nächsten Knoten aus der Queue q.

→ siehe video für Kommentar