Algorithmen und Datenstrukturen II

1. Graphenalgorithmen

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 18.04.2020 15:57

Inhaltsverzeichnis

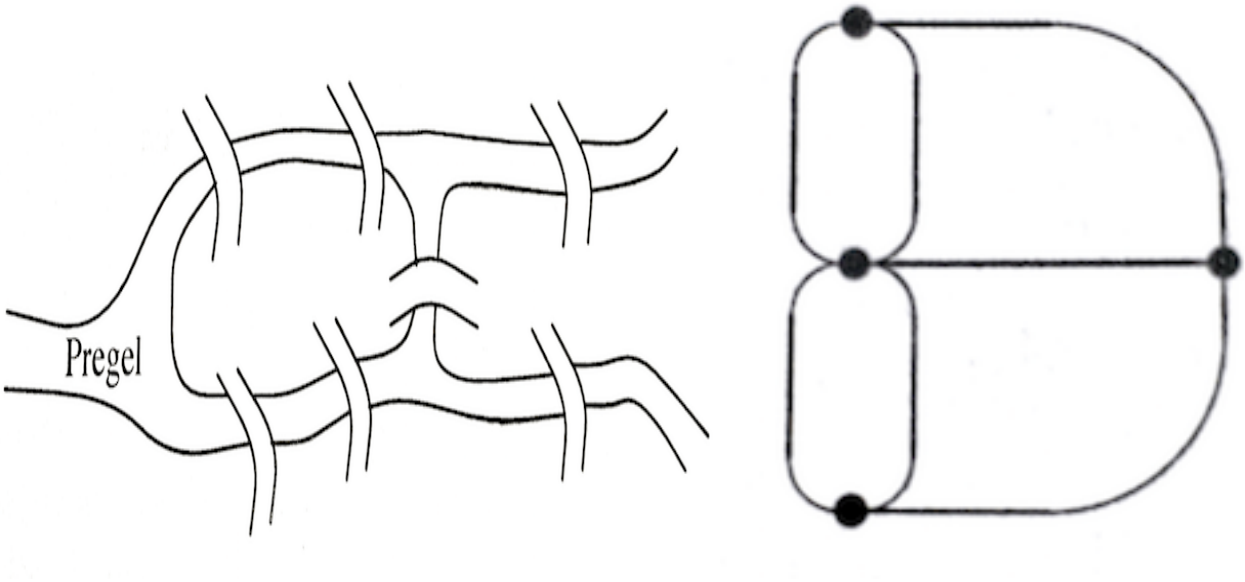
Wofür

- wie komme ich am schnellsten von München nach Hamburg?

- wie teile ich die Arbeitskräfte in meinem Team am Besten den Tätigkeiten zu für die sie am Besten geeignet sind?
- wann kann der Hausbau frühestens fertig sein, wenn die einzelnen Arbeiten in der richtigen Reihenfolge ausgeführt werden?
- wie besuche ich alle meine Kunden mit einer kürzest möglichen Rundreise?
- …

Das Königsberger Brückenproblem

Über alle Brücken genau einmal laufen und am Ausgangspunkt ankommen.



Unmöglich (1736, Euler).

Gerichteter Graph

- ein **gerichteter Graph** (engl: *digraph*) besteht aus
  - einer Menge $V = \{1, 2, ..., |V|\}$ von **Knoten** (engl: *vertices*) und
  - einer Menge von $E \subseteq V \times V$ von **Pfeilen** (engl: *edges*, *arcs*)
- ein Paar $(v, v') \in E$ heißt **Pfeil von** v **nach** v'
  - v heißt **Anfangsknoten** von (v, v')
  - v' heißt **Endknoten** von (v, v')
  - dann heißen v und v' auch **adjazent**
- Frage: Kann es nach obiger Definition *parallele* Pfeile geben?

Speicherung eines Graphen in einer Adjazenzmatrix

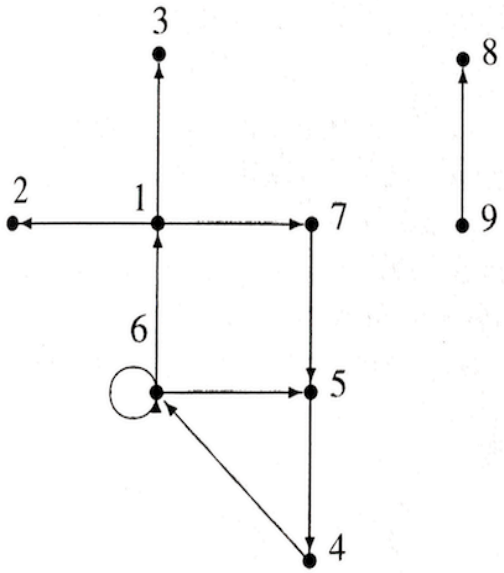
Ein Graph $G = (V, E)$ wird in einer Boole'schen $|V| \times |V|$-Matrix $A_G = (a_{ij})$ mit $1 \leq i \leq |V|, 1 \leq j \leq |V|$ gespeichert, wobei

$$a_{ij} = 0, \text{ falls } (i, j) \notin E$$

und

$$a_{ij} = 1, \text{ falls } (i, j) \in E$$

Beispiel: Graph als Adjazenzmatrix



(a)

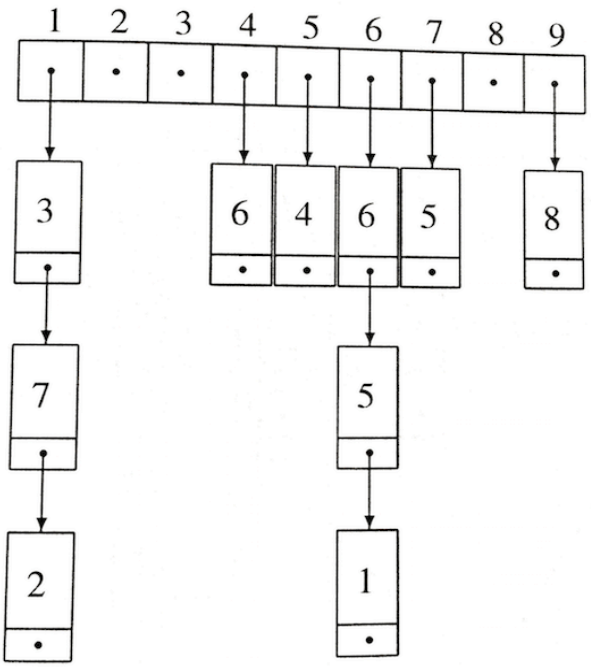
	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	1	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0
5	0	0	0	1	0	0	0	0	0
6	1	0	0	0	1	1	0	0	0
7	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	1	0

(b)

Speicherung relativ ineffizient (quadratischer Aufwand)

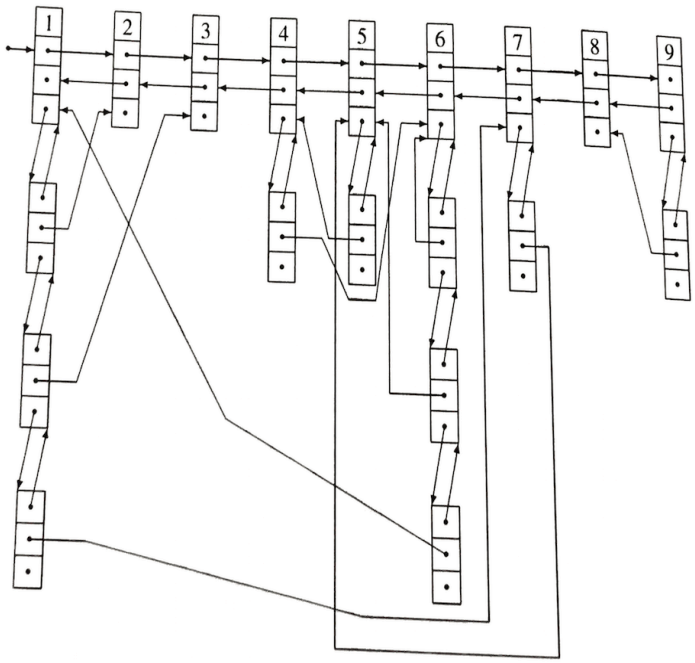
Speicherung in Adjazenzlisten

- für jeden Knoten eine lineare, verkettete Liste der von diesem ausgehenden Pfeile
- Beispiel:



Doppelt verkettete Pfeilliste

doubly connected arc list (DCAL)



Code im Livecoding-Repository

- Interface Graph
- Implementierung als Adjazenzmatrix
- Implementierung als Adjazenzliste
- Tests und Benchmarks
- Looper
- Einbindung einer Dot-Lib

Weitere Definitionen

- **Eingangsgrad** $indeg(v)$ ist Anzahl der in v einmündenen Pfeile
- **Ausgangsgrad** $outdeg(v)$ ist Anzahl der von v ausgehenden Pfeile
- Ein Digraph $G' = (V', E')$ ist ein **Teilgraph** von $G = (V, E)$, geschrieben $G' \subseteq G$ falls
  - $V' \subseteq V$ und $E' \subseteq E$
- für $V' \subseteq V$ **induziert** V' den Teilgraphen $(V', E \cap (V' \times V'))$, auch **Untergraph** genannt
- ein **Weg** von v nach v' mit $v, v' \in V$ ist der durch eine Folge von $(v_0, v_1, ..., v_k)$ von Knoten mit $v_0 = v, v_k = v'$ und $(v_i, v_{i+1}) \in E$ für $0 \leq i < k$ beschriebene Teilgraph (V', E')
  - die Anzahl der Pfeile in E' sind die **Länge** des Weges
  - ein Weg heißt **einfach**, wenn kein Knoten mehrfach besucht wird

Weitere Definitionen (2)

- ein **Zyklus** ist ein Weg von v nach v
- triviale Wege die aus einem Knoten und keinem Pfeil bestehen, werden nicht betrachtet
- ein Digraph heißt **zyklenfrei** oder **azyklisch** wenn er keine Zyklen enthält
- gibt es einen Weg von v nach v', ist v' von v **erreichbar**

Wälder und Bäume

- ein Digraph heißt **gerichteter Wald**, wenn E zyklenfrei und jeder Knoten einen Eingangsgrad von höchstens 1 hat
  - jeder Knoten mit $indeg(v) = 0$ heißt **Wurzel**

  - ein gerichteter Wald mit genau einer Wurzel heißt **gerichteter Baum (Wurzelbaum)**

- für einen Digraphen $G = (V, E)$ ist ein gerichteter Wald $W = (V, F)$ mit $F \subseteq E$ ein **spannender Wald** von G
  - falls W ein Baum ist, heißt er **spannender Baum** (*spanning tree*)

Ungerichtete Graphen

- wenn wir erzwingen, dass zwischen zwei Knoten entweder kein Pfeil oder in jeder Richtung ein Pfeil ist, können wir die Richtung weglassen
- ein solcher Graph $G = (V, E)$ für den gilt $(v, v') \in E \Leftrightarrow (v', v) \in E$ heißt **ungerichteter Graph** oder einfach **Graph**
- ein Paar $((v, v'), (v', v))$ heißt **Kante**
- der Grad $deg(v)$ ist dann gleich $indeg(v)$ und auch $outdeg(v)$
- ein ungerichteter Graph heißt zyklenfrei oder azyklisch, falls er keinen einfachen Zyklus mit wenigstens 3 Pfeilen enthält
  - Erinnerung: Bei einem *einfachen* Pfad darf kein Knoten mehrfach besucht werden

Topologische Sortierung

- betrachtet man einen Digraph als binäre Relation, dann
  - ist ein zyklenfreier Digraph eine Halbordnung
- eine topologische Sortierung
  - ist eine vollständige Ordnung über die Knoten
  - die mit der partiellen Ordnung verträglich ist
    * die durch die Pfeile ausgedrückt wird
- d.h. eine topologische Sortierung
  - ist eine Abbildung $ord : V \to \{1, ..., n\}$ mit $n = |V|$, so dass gilt

$$(v, w) \in E \Rightarrow ord(v) < ord(w)$$

- G ist genau dann zyklenfrei, wenn es für G eine topologische Sortierung gibt
  - $\Leftarrow$ ist klar, $\Rightarrow$ lässt sich durch Induktion beweisen (siehe Buch S. 597)
- Code im Livecoding-Repository zeigen:

```
func (g AdjLst) IsAcyclic() bool
```

- mit der Funktion `IsAcyclic` kann für einen Digraph in der Zeit $\mathcal{O}(|V| + |E|)$ berechnet werden, ob er zyklenfrei ist
- für AdjMat rein kopieren.

Transitive Hülle

- weitere interessante Fragestellung: Erreichbarkeit von Knoten
- es kann sinnvoll sein von vornherein alle Erreichbarkeiten explizit zu berechnen
- Definition
  - Digraph $G^* = (V, E^*)$ ist *reflexive, transitive Hülle* von $G = (V, E)$ wenn gilt:

$$(v, v') \in E^* \Leftrightarrow \text{ es gibt Weg von } v \text{ nach } v' \text{ in } G$$

Transitive Hülle allgemein

erster Ansatz

```
func (g AdjMat) TransClosure() Graph {
  closure := NewGraphAdjMat(len(g) - 1)

  for i := range g {
    copy(closure[i], g[i])
  }

  n := Node(len(closure) - 1)

  for i := Node(1); i <= n; i++ {
    closure.AddEdge(i, i)
  }
  ...

  for i := Node(1); i <= n; i++ {
    for j := Node(1); j <= n; j++ {
      if g[i][j] {
        for k := Node(1); k <= n; k++ {
          if g[j][k] {
            closure.AddEdge(i, k)
          }
        }
```

```
      }
    }
  }
  return closure
}
```

Problem?

Lösung

- Algorithmus könnte so oft durchlaufen werden, bis keine neuen Wege gefunden werden
- nicht sehr effizient
- durch die 3 verschachtelten Schleifen schon bei einem Durchlauf $\mathcal{O}(|V|^3)$

Effizientere Lösung

- Idee um es in einem Durchlauf zu schaffen:
- zum Finden eines Weges von i nach k betrachten wir nicht jede mögliche Zusammensetzung von Teilwegen
- ein Weg von i nach k ist
  - entweder ein Pfeil
  - oder ein Weg über einen Knoten j, für den gilt j ist dir größte Knotennummer auf dem Weg zwischen i und k (ohne i und k selbst)
- wir ermitteln nun Wege in Reihenfolge, so dass Wege zwischen i und j bzw. j und k nur Zwischenknoten mit einer kleineren Nummer als j benutzen
- Invariante für aufsteigende Werte von j:
  - für j sind alle Wege bereits bekannt, die nur Zwischenknoten mit Nummern $< j$ benutzen
- für $j = 1$ klar
- im nächsten Schritt werden die Wege über j verbunden
- im Code müssen dazu nur die äußeren beiden `for`-Schleifen vertauscht werden

```
for j := Node(1); j <= n; j++ {
    for i := Node(1); i <= n; i++ {
        if closure[i][j] {
            for k := Node(1); k <= n; k++ {
                if closure[j][k] {
                    closure.AddEdge(i, k)
                }
```

```
            }
        }
    }
}
```

- Laufzeit ist zunächst beschränkt durch $\mathcal{O}(|V|^3)$
- die innerste Schleife wird aber nur durchlaufen, wenn es einen Pfeil von i nach j gibt
- dieser Pfeil kann aus E oder erst E^* stammen
- d.h. die innerste Schleife wird insgesamt nur (E^*)-mal durchlaufen
- ein Durchlauf benötigt $\mathcal{O}(|V|)$ Schritte
- daraus ergibt sich eine Gesamtlaufzeit in $\mathcal{O}(|V|^2 + |E^*||V|)$