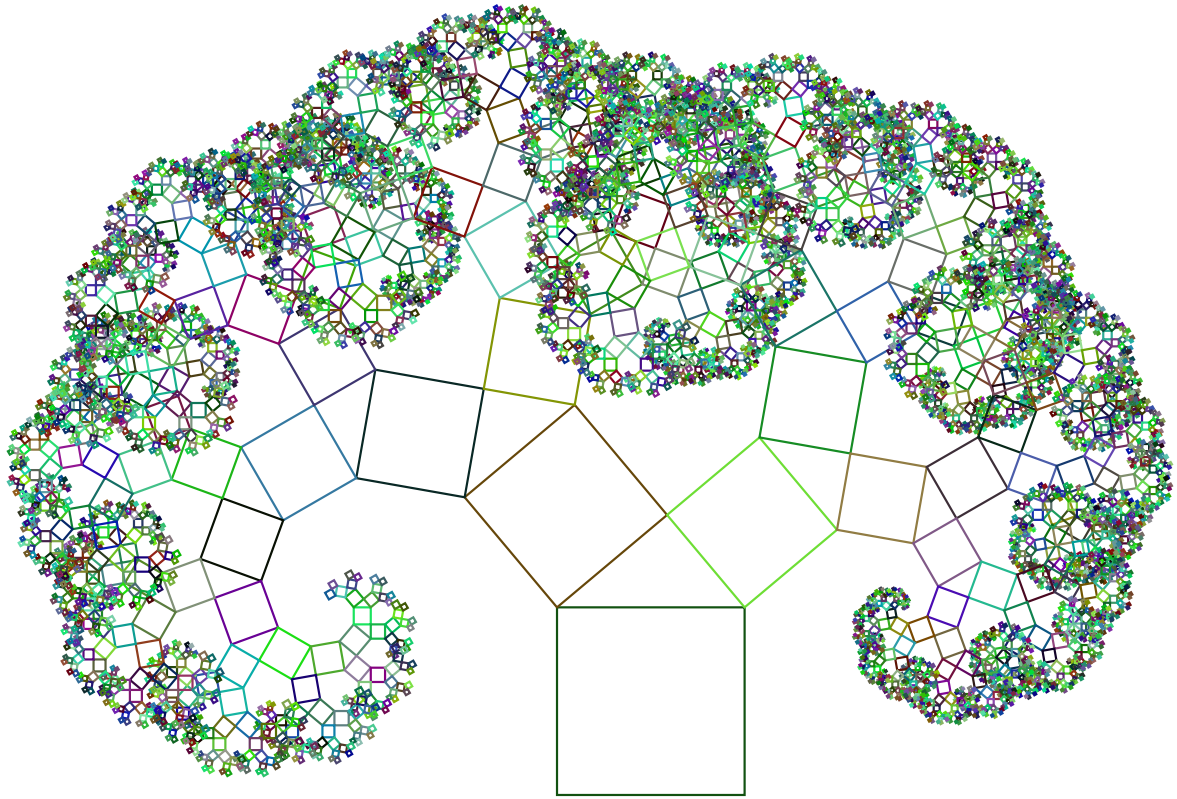
Karlsruher Institut für Technologie

10. Übung – Algorithmen I

INSTITUT FÜR THEORETISCHE INFORMATIK



Bäume

Charakterisierung von Bäumen

Definition: Ein ungerichteter Graph heißt Baum, wenn es von jedem Knoten zu jedem anderen Knoten genau einen Kantenweg gibt.
Satz: Für einen ungerichteten Graphen $G = (V, E)$ sind äquivalent:

1. G ist ein Baum.
2. G ist zusammenhängend und $|E| = |V| - 1$.
3. G ist zusammenhängend und kreislos.
4. G ist kreislos und $|E| = |V| - 1$.
5. G ist *maximal kreislos*: G ist kreislos und jede zusätzliche Kante zwischen nicht-adjazenten Knoten ergibt einen Kreis.
6. G ist *minimal zusammenhängend*: G ist zusammenhängend und bei Entfernen einer beliebige Kante zerfällt G.

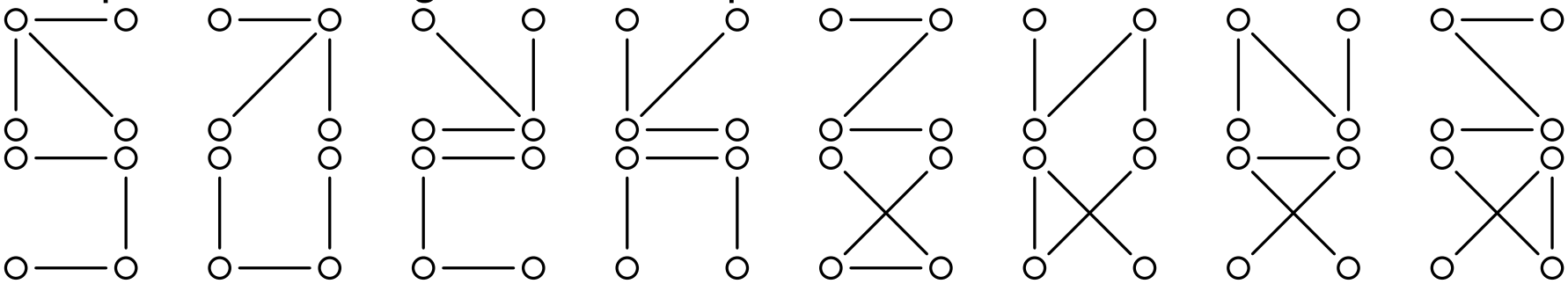
Satz von Cayley

Ist $G = (V, E)$ ein zusammenhängender ungerichteter Graph, dann heißt ein Untergraph (V, E') mit $E' \subseteq E$ ein G aufspannender Baum, wenn dieser ein Baum ist.

Satz von Cayley

Im vollständigen Graph K_n gibt es genau n^{n-2} verschiedene K_n aufspannende Bäume.

Beispiel: K_4 hat folgende 16 aufspannende Bäume:



Kürzeste Wege in Graphen

Bellman-Ford-Algorithmus

Wiederholung

- Ausgehend von *einem* Knoten s berechne kürzesten Wege-Baum

```
1:  procedure BellmanFord(s : NodeId) : NodeArray × NodeArray
2:      d = ⟨∞, ..., ∞⟩ : NodeArray of ℝ ∪ {−∞, ∞}
3:      parent = ⟨⊥, ..., ⊥⟩ : NodeArray of NodeId
4:      d[s] := 0; parent[s] := s
5:      for i := 1 to n − 1 do
6:          forall e ∈ E do relax(e)
7:
8:      forall e = (u, v) ∈ E do
9:          if d[u] + c(e) < d[v] then Error: "Graph contains negative
        cycle"
10: return (d, parent)
```

- Erinnerung Kante (u, v) relaxieren:
  1. wenn $d[u] + c(u, v) < d[v]$ dann $d[v] := d[u] + c(u, v)$, $\text{parent}[v] := u$

Bellman-Ford vs Dijkstra

- Dijkstra: *greedy* – wählt Knoten mit minimaler Distanz und relaxiert ausgehende Kanten
- Bellman-Ford: nicht *greedy* – relaxiert alle Kanten $n - 1$ mal

- Bellman-Ford funktioniert mit negativen Kantengewichten
- Bellman-Ford funktioniert nicht, falls G negativen Kreis enthält
  - aber detektiert solche Kreise und bricht ab

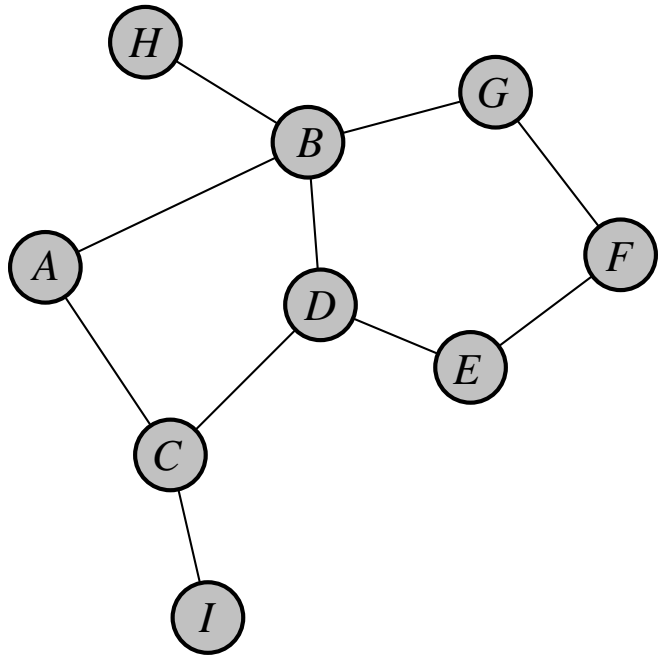
Durchmesser ungerichteter Graphen

Durchmesser ungerichteter Graphen

Intuitiv: Größter Abstand zweier Knoten im Graph.

- **Frage:** Was ist der Abstand zweier Knoten im Graph?
- **Antwort:** Länge des kürzesten Pfades zwischen zwei Knoten.

- **Frage:** Was ist die Länge eines Pfades?
- **Antwort:** Die Anzahl Kanten des Pfades.

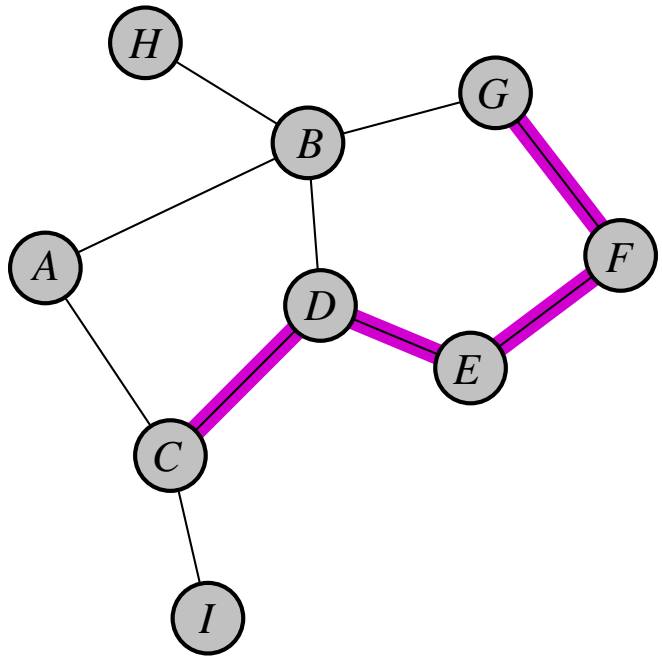


Durchmesser ungerichteter Graphen

Intuitiv: Größter Abstand zweier Knoten im Graph.

- **Frage:** Was ist der Abstand zweier Knoten im Graph?
- **Antwort:** Länge des kürzesten Pfades zwischen zwei Knoten.

- **Frage:** Was ist die Länge eines Pfades?
- **Antwort:** Die Anzahl Kanten des Pfades.



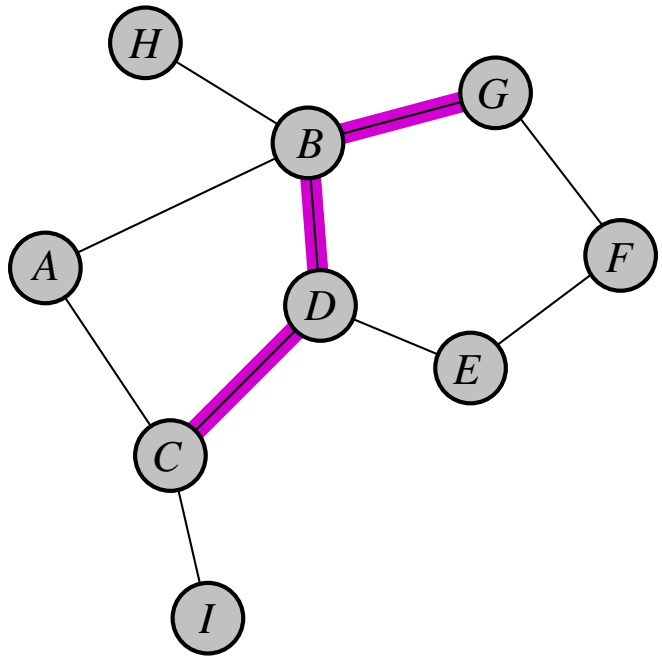
Ein Pfad von *C* nach *G*, Länge = 4

Durchmesser ungerichteter Graphen

Intuitiv: Größter Abstand zweier Knoten im Graph.

- **Frage:** Was ist der Abstand zweier Knoten im Graph?
- **Antwort:** Länge des kürzesten Pfades zwischen zwei Knoten.

- **Frage:** Was ist die Länge eines Pfades?
- **Antwort:** Die Anzahl Kanten des Pfades.



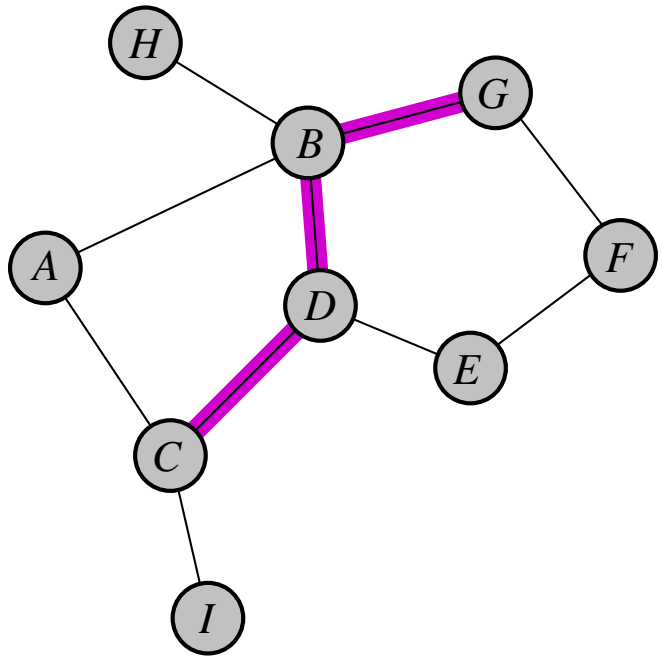
Kürzester Pfad von *C* nach *G*, Länge = 3

Durchmesser ungerichteter Graphen

Intuitiv: Größter Abstand zweier Knoten im Graph.

- **Frage:** Was ist der Abstand zweier Knoten im Graph?
- **Antwort:** Länge des kürzesten Pfades zwischen zwei Knoten.

- **Frage:** Was ist die Länge eines Pfades?
- **Antwort:** Die Anzahl Kanten des Pfades.



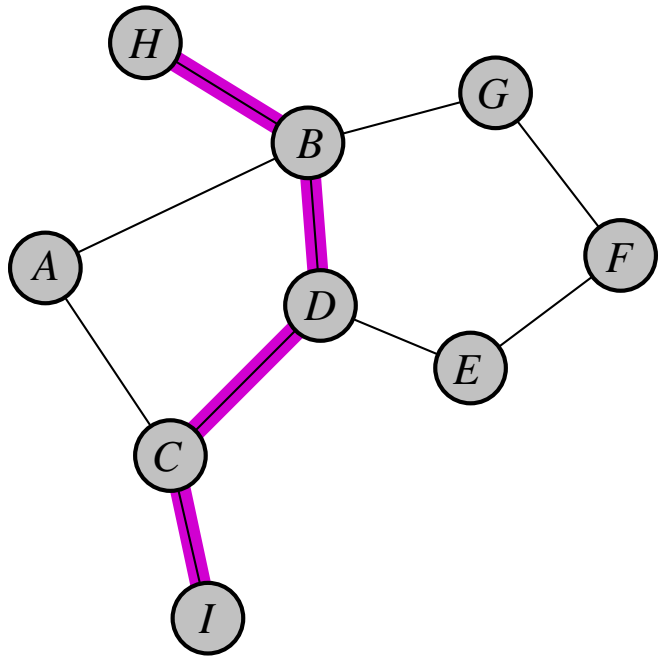
Abstand von *C* und *G*, Länge = 3

Durchmesser ungerichteter Graphen

Intuitiv: Größter Abstand zweier Knoten im Graph.

- **Frage:** Was ist der Abstand zweier Knoten im Graph?
- **Antwort:** Länge des kürzesten Pfades zwischen zwei Knoten.

- **Frage:** Was ist die Länge eines Pfades?
- **Antwort:** Die Anzahl Kanten des Pfades.



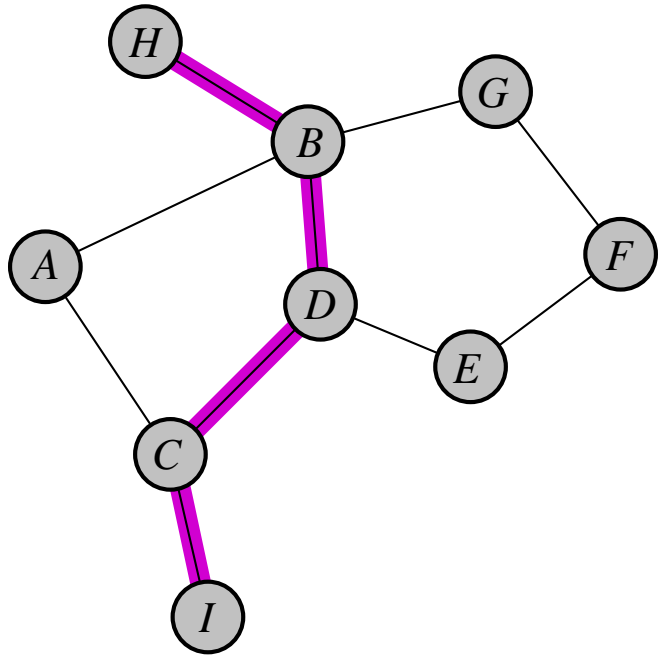
Kürzester Pfad von *I* nach *H*, Länge = 4

Durchmesser ungerichteter Graphen

Intuitiv: Größter Abstand zweier Knoten im Graph.

- **Frage:** Was ist der Abstand zweier Knoten im Graph?
- **Antwort:** Länge des kürzesten Pfades zwischen zwei Knoten.

- **Frage:** Was ist die Länge eines Pfades?
- **Antwort:** Die Anzahl Kanten des Pfades.



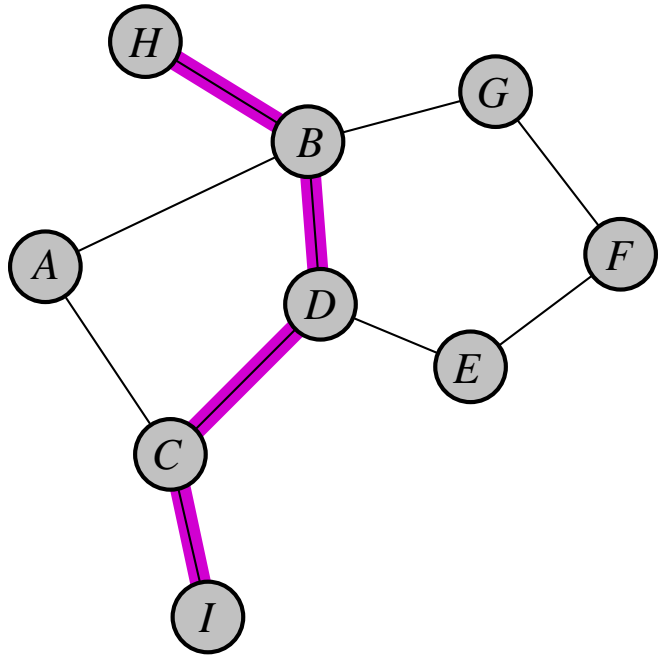
Kürzester Pfad von *I* nach *H*, Länge = 4
$\Rightarrow$ Abstand von *I* und *H* = 4

Durchmesser ungerichteter Graphen

Intuitiv: Größter Abstand zweier Knoten im Graph.

- **Frage:** Was ist der Abstand zweier Knoten im Graph?
- **Antwort:** Länge des kürzesten Pfades zwischen zwei Knoten.

- **Frage:** Was ist die Länge eines Pfades?
- **Antwort:** Die Anzahl Kanten des Pfades.



Kürzester Weg von *I* nach *H*, Länge = 4
$\Rightarrow$ Abstand von *I* und *H* = 4
Größter Abstand $\Rightarrow$ Durchmesser = 4

Durchmesser ungerichteter Graphen

Mathematische Definition

$$\textit{Durchmesser} := \max_{u,v \in V} \left(\min_{P \in \mathcal{P}(u,v)} |P| \right)$$

mit

- $\mathcal{P}(u, v)$ ist Menge aller Pfade zwischen u und v
- $|P|$ ist Anzahl der Kanten eines Pfades P

Durchmesser ungerichteter Graphen

Algorithmus

Idee:

1: $D = 0 : \mathbb{N}_{\geq 0}$
2: **for each** $u \in V$ **do**
3: Berechne kürzeste Pfade von u zu allen $v \in V \setminus \{u\}$
4: $D := \max\{D, \text{größte gefundene Pfadlänge}\}$
5: **return** D

Laufzeit: $O(n \cdot \text{Laufzeit Zeile 3})$

Durchmesser ungerichteter Graphen

Teilalgorithmus: Alle kürzesten Pfade von u nach $v \in V \setminus \{u\}$

in ungewichteten Graphen: Breitensuche plus Nachbearbeitung:

```
1: function maxAbstand(u: NodeID): Abstand
2:     (⊥, d) := bfs(u)
3:     result := max_{v∈V\{u}} d[v]
4:     return result
```

Durchmesser ungerichteter Graphen

Teilalgorithmus: Alle kürzesten Pfade von u nach $v \in V \setminus \{u\}$

Breitensuche plus Nachbearbeitung:

```
1: function maxAbstand(u: NodeID): Abstand
2:     (⊥, d) := bfs(u)
3:     result := max_{v∈V\{u}} d[v]
4:     return result
```

Laufzeit:

- BFS braucht $O(m)$
- Maximum berechenen braucht $O(n)$

$\Rightarrow$ Gesamt $O(n+m)$, falls zusammenhängend $= O(m)$, da $n \leq m$

Durchmesser ungerichteter Graphen

Algorithmus

Fertiger Algorithmus:

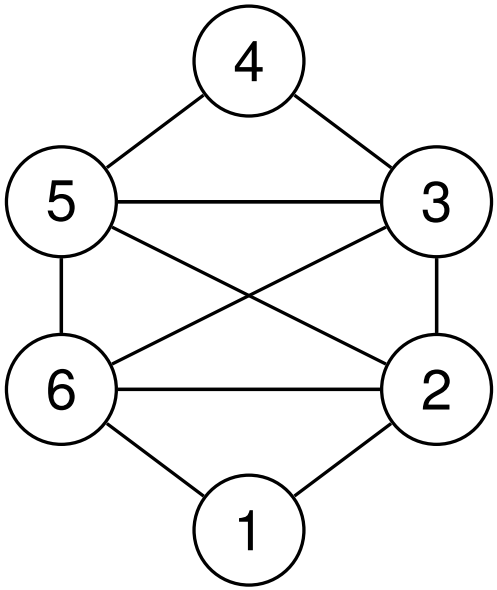
1: $D = 0 : \mathbb{N}_{\geq 0}$
2: **for each** $u \in V$ **do**
3: $\quad D := \max\{D, maxAbstand(u)\}$
4: **return** D

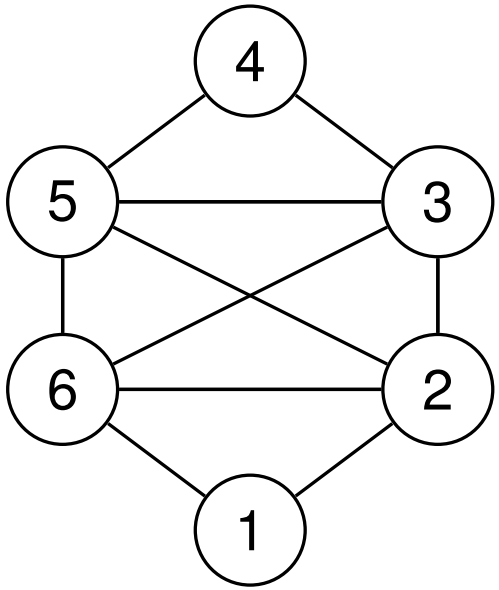
Laufzeit: $O(n \cdot$ Laufzeit Zeile 3$) = O(nm)$, für zusammenhängende, ungewichtete Graphen

k-Core Decomposition

k-core

Der größte knoteninduzierte Teilgraph $G' = (V', E')$ von G mit $\forall v \in V' : \text{Grad}(v) \geq k$ ist der k-core von G.

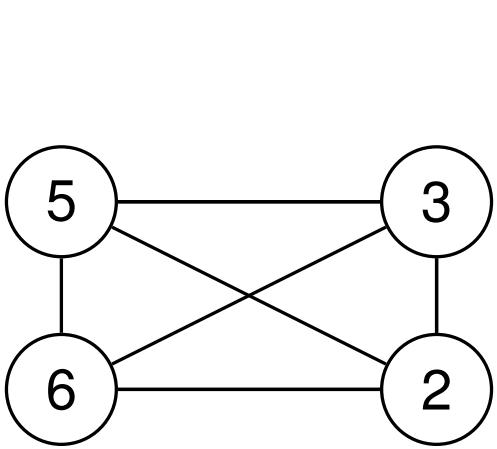


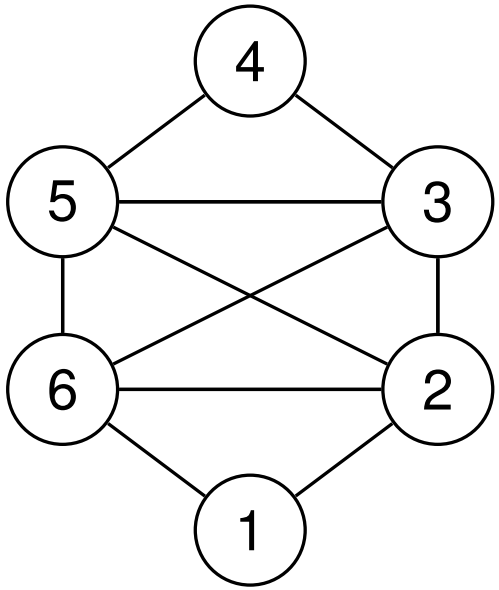


ist 2-core von

k-core

Der größte knoteninduzierte Teilgraph $G' = (V', E')$ von G mit $\forall v \in V' : \text{Grad}(v) \geq k$ ist der k-core von G.

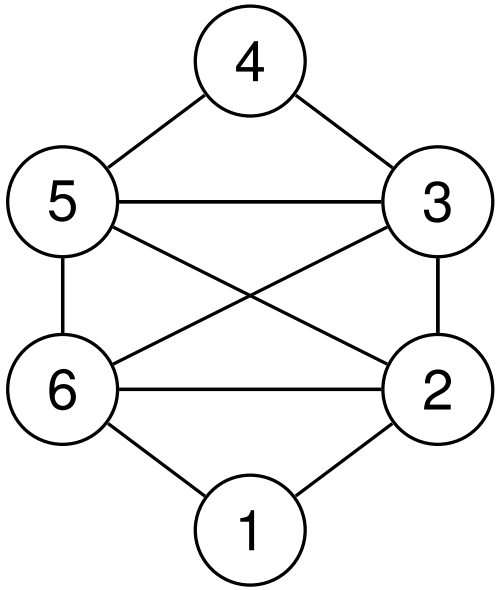




ist 3-core von

k-core

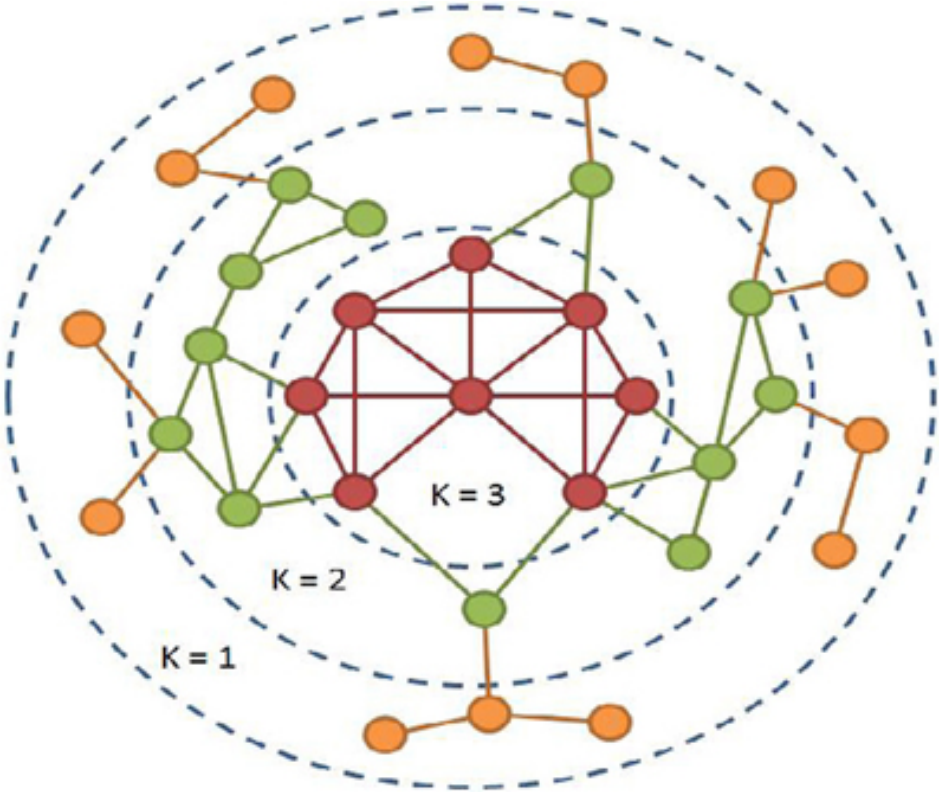
Der größte knoteninduzierte Teilgraph $G' = (V', E')$ von G mit $\forall v \in V' : \text{Grad}(v) \geq k$ ist der k-core von G.



ist 4-core von

k-core

Core Decomposition



source: http://journal.frontiersin.org/article/10.3389/fnagi.2012.00013/

Berechnung des k-cores

Procedure core($G = (V, E)$, $k : \mathbb{N}$)
    $G' = (V', E') := G$
    while ($\exists v \in V'$ with $\text{Grad}_{G'}(v) < k$)
        Remove v from V' and all its incident edges from E'
    return G'

Frage: Laufzeit für diesen Algorithmus?

a) $O(n + m)$
b) $O(n \log n + m)$
c) $O(n^2 + m)$

Berechnung des k-cores

```
Procedure core(G = (V[1..n+1], E[1..m]), k : ℕ)    // Adjazenzfeld
    Q : FIFO
    d : Array[1..n] of ℕ
    for (v := 1; v ≤ n; ++v)
        d[v] := V[v+1] − V[v]
        if (d[v] < k) Q.pushBack(v)
    while (!Q.empty())
        v := Q.popFront()
        for (e := V[v]; e < V[v+1]; ++e)
            w := E[e]
            d[w] − −
            if (d[w] = k − 1) Q.pushBack(w)
    return subgraph induced by the nodes v with d[v] ≥ k
```

Berechnung des k-cores

```
Procedure core(G = (V[1..n+1], E[1..m]), k : ℕ)    // Adjazenzfeld
    Q : FIFO
    d : Array[1..n] of ℕ
    for (v := 1; v ≤ n; ++v)
        d[v] := V[v+1] − V[v]
        if (d[v] < k) Q.pushBack(v)
    while (!Q.empty())
        v := Q.popFront()
        for (e := V[v]; e < V[v+1]; ++e)
            w := E[e]
            d[w] − −
            if (d[w] = k − 1) Q.pushBack(w)
    return subgraph induced by the nodes v with d[v] ≥ k
```

Laufzeit: $O(n+m)$

k-core Decomposition

```
Procedure coreDec(G = (V, E))
    deg : Array[1..n] of ℕ                  // Array für den Knotengrad, mit 0 initialisiert
    active : Array[1..n] of {0, 1}          // Array für aktive Knoten, mit 1 initialisiert
    C : Array[1..n] of ℕ                    // Array für die Core-Struktur, mit 0 initialisiert
    Q : Addressable Priority Queue
    foreach {u, v} ∈ E do                                  // Knotengrad berechnen
        deg(u) += 1
        deg(v) += 1
    foreach v ∈ V do                                       // Prioritätsliste initialisieren
        Q.insert(v, deg(v))
    while (!Q.empty())
        v := Q.extractMin()
        C[v] := deg(v)                                     // Core-Nummer zuweisen
        active(v) := 0                                     // Knoten wurde entfernt
        foreach {v, u} ∈ E and active[v] do // Knotengrad der Nachbarn updaten
            deg[u] -= 1
            Q.decreaseKey(u, deg[u])
    return C
```

k-core Decomposition: Laufzeit

- $O((n+m) \cdot \log(n))$ mit Binary Heap als Priority Queue
- aber: Keys sind ganzzahlig (Knotengrade) und beschränkt ($< n$)
- also Implementierung mit Bucket Priority Queue in $O(n+m)$ möglich
- Details siehe Lösung zu Übungsblatt 7