

Middleware - Cloud Computing – Übung

Klaus Stengel, Johannes Behl, Tobias Distler,
Tobias Klaus, Christopher Eibel

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)

www4.cs.fau.de

Wintersemester 2013/14



Graph-Algorithmen mit MapReduce

Kurzeinführung Graphentheorie

Algorithmus zum Finden von Cliques

MapReduce Erweiterungen

Zusammenführen von Daten

Sortierung, Gruppierung und Partitionierung

Combiner

Zähler

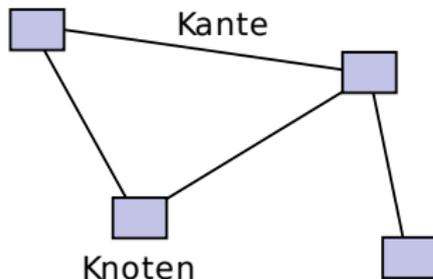


Graph-Algorithmen mit MapReduce
Kurzeinführung Graphentheorie
Algorithmus zum Finden von Cliquen

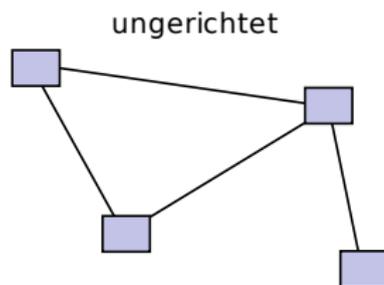
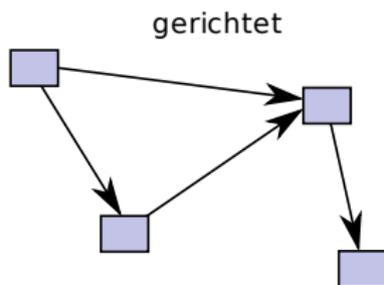
MapReduce Erweiterungen
Zusammenführen von Daten
Sortierung, Gruppierung und Partitionierung
Combiner
Zähler



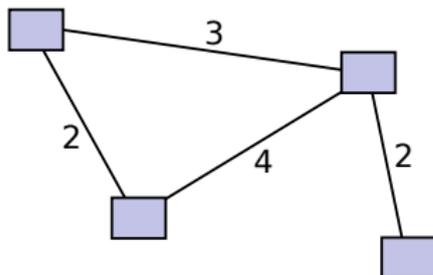
- Graphen bestehen aus
 - **Knoten** (englisch: Node, Vertex, Mehrzahl Vertices)
 - **Kanten** (englisch: Edge), die Knoten **verbinden**
 - **Grad** eines Knoten ist die Zahl der dort zusammenlaufenden Kanten



- Kanten können **gerichtet** oder **ungerichtet** sein
- Dementsprechend: Gerichter/ungerichteter Graph
- Gerichtete Kante verbindet **Start**-und **Endknoten**



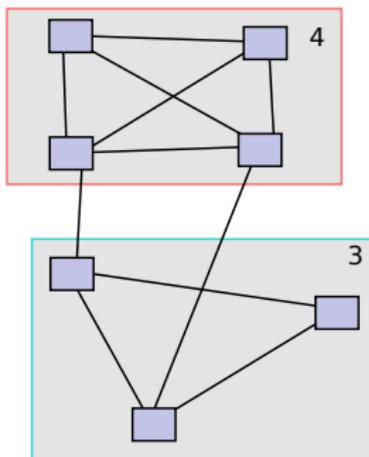
- Knoten sowie Kanten können **gewichtet** sein:
Zusätzliche Wertung an jeweiliges Objekt als Attribut angefügt



- Beispiel:
Knoten sind verschiedene Städte
Kanten stellen Wege zwischen Städten dar
Gewichtung entspricht Weglänge

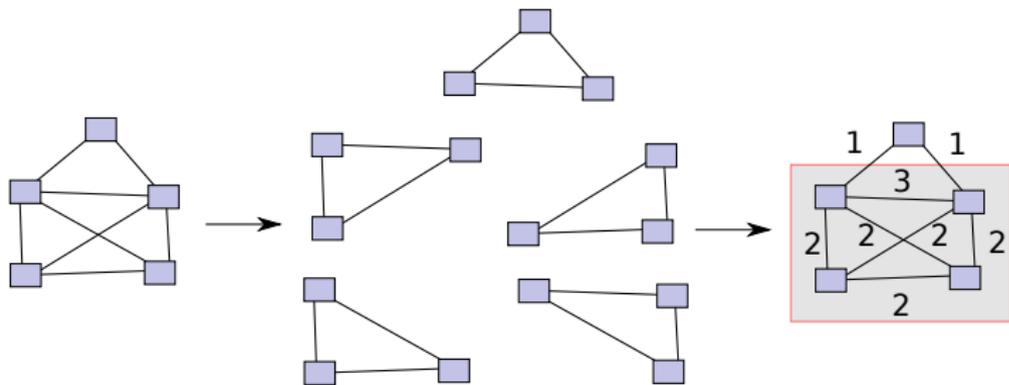
Algorithmus zum Finden von Cliques

- **Clique:** Teilmenge von Knoten eines ungerichteten Graphen, die alle untereinander durch Kanten verbunden sind.
- Algorithmus zum Finden von Cliques auf Basis des Artikels *Graph Twiddling in a MapReduce World*[Cohen09]
- Beispiel:



Algorithmus zum Finden von Cliquen

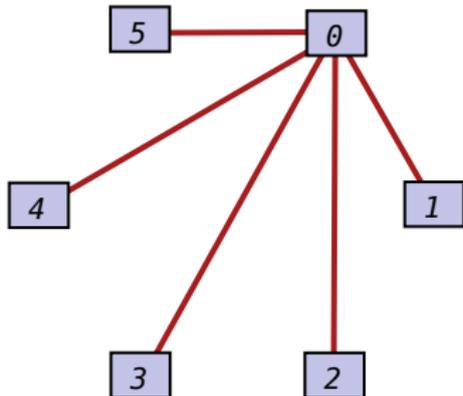
- Aufgabe: Finden von Cliquen einer bestimmten Mindestgröße
- Lösungsansatz basiert auf Finden von **Trägern**:
 - Finden von nichttrivialer Cliquen minimaler Größe: Dreiecke
 - Vollständiger Graph mit n Knoten enthält $\frac{n \cdot (n-1) \cdot (n-2)}{6}$ unterschiedliche Dreiecke
 - Jede Kante kommt in Dreiecken **mindestens** $n - 2$ mal vor
 - Weniger häufig auftretende Kanten können nicht Teil eines vollständigen Teilgraphen (Clique) sein.



Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten

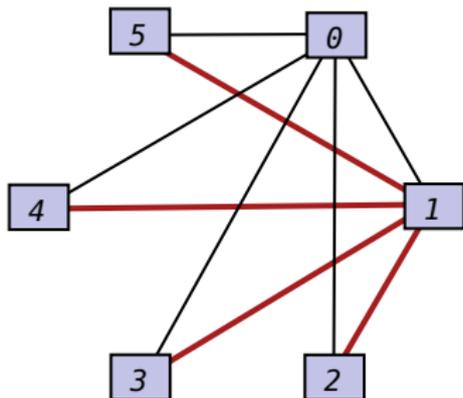
5



Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten

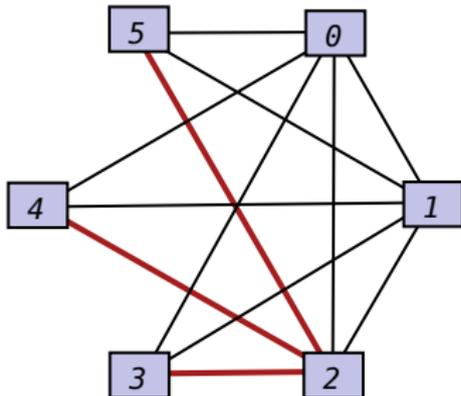
5+4



Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten

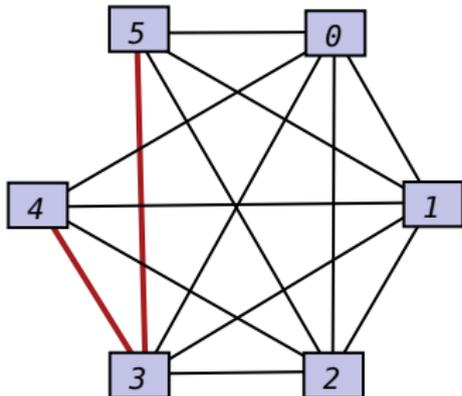
5+4+3



Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten

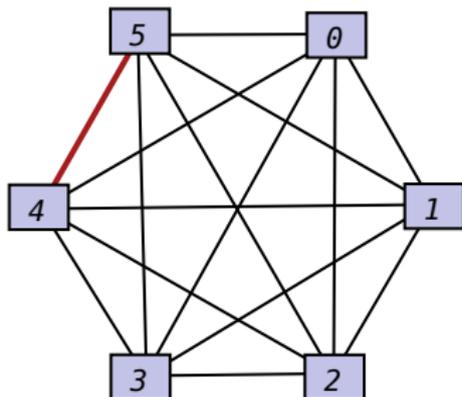
$$5+4+3+2$$



Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten

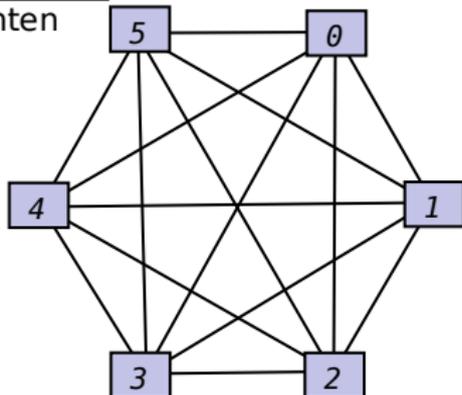
$$5+4+3+2+1$$



Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten

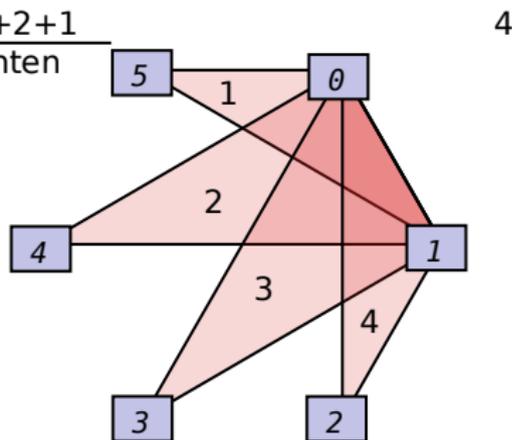
$$\frac{5+4+3+2+1}{=15 \text{ Kanten}}$$



Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten
 - $\frac{n(n-1)(n-2)}{6}$ Dreiecke
- Dreieck besteht aus 3 Kanten: $(n - 2)$ Überlagerungen

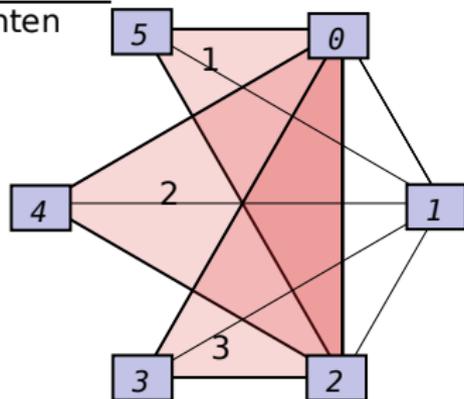
$$\frac{5+4+3+2+1}{=15 \text{ Kanten}}$$



Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten
 - $\frac{n(n-1)(n-2)}{6}$ Dreiecke
- Dreieck besteht aus 3 Kanten: $(n - 2)$ Überlagerungen

$$\frac{5+4+3+2+1}{=15 \text{ Kanten}}$$



$$4+3$$

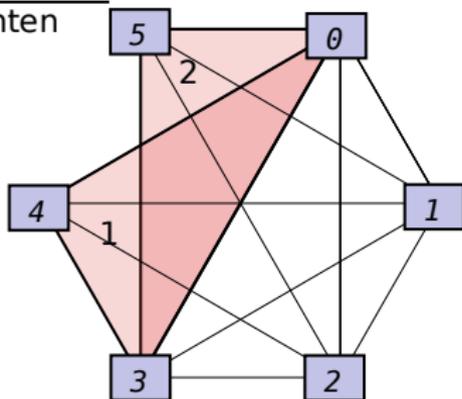


Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten
 - $\frac{n(n-1)(n-2)}{6}$ Dreiecke
- Dreieck besteht aus 3 Kanten: $(n - 2)$ Überlagerungen

$$\frac{5+4+3+2+1}{=15 \text{ Kanten}}$$

$$4+3+2$$

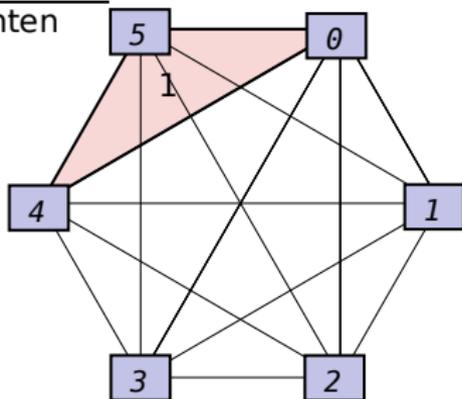


Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten
 - $\frac{n(n-1)(n-2)}{6}$ Dreiecke
- Dreieck besteht aus 3 Kanten: $(n-2)$ Überlagerungen

$$\frac{5+4+3+2+1}{=15 \text{ Kanten}}$$

$$4+3+2+1$$

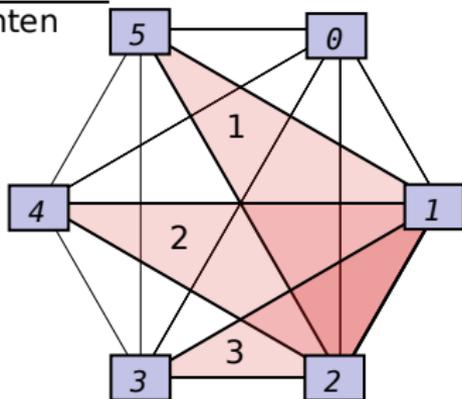


Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten
 - $\frac{n(n-1)(n-2)}{6}$ Dreiecke
- Dreieck besteht aus 3 Kanten: $(n - 2)$ Überlagerungen

$$\frac{5+4+3+2+1}{=15 \text{ Kanten}}$$

$$\frac{4+3+2+1}{3}$$

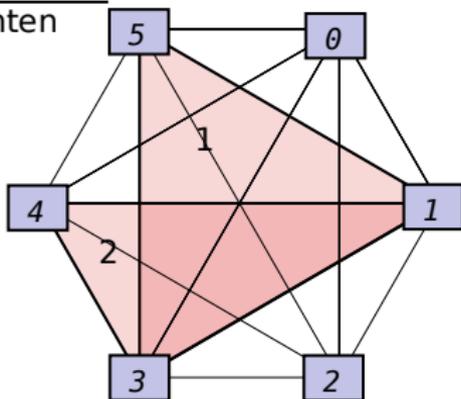


Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten
 - $\frac{n(n-1)(n-2)}{6}$ Dreiecke
- Dreieck besteht aus 3 Kanten: $(n - 2)$ Überlagerungen

$$\frac{5+4+3+2+1}{=15 \text{ Kanten}}$$

$$\begin{array}{l} 4+3+2+1 \\ 3+2 \end{array}$$

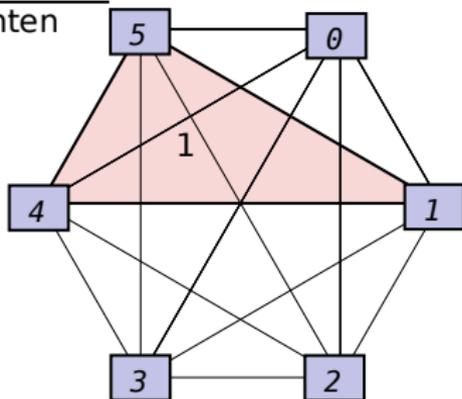


Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten
 - $\frac{n(n-1)(n-2)}{6}$ Dreiecke
- Dreieck besteht aus 3 Kanten: $(n-2)$ Überlagerungen

$$\frac{5+4+3+2+1}{=15 \text{ Kanten}}$$

$$\begin{array}{l} 4+3+2+1 \\ 3+2+1 \end{array}$$

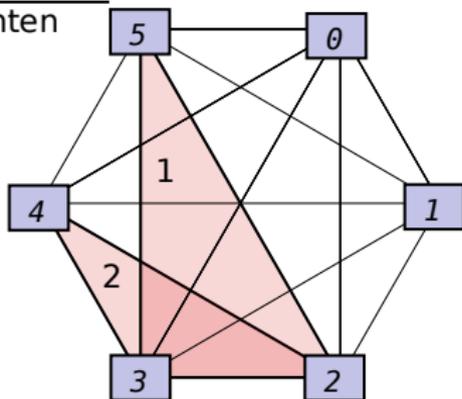


Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten
 - $\frac{n(n-1)(n-2)}{6}$ Dreiecke
- Dreieck besteht aus 3 Kanten: $(n-2)$ Überlagerungen

$$\frac{5+4+3+2+1}{=15 \text{ Kanten}}$$

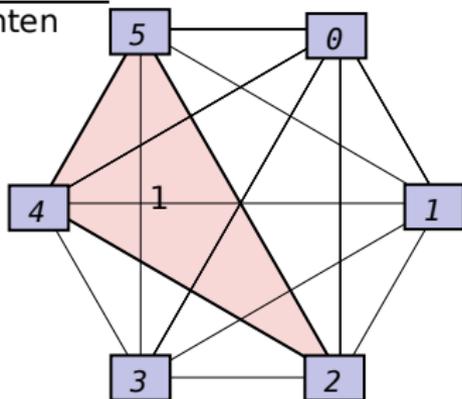
$$\begin{array}{r} 4+3+2+1 \\ 3+2+1 \\ 2 \end{array}$$



Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten
 - $\frac{n(n-1)(n-2)}{6}$ Dreiecke
- Dreieck besteht aus 3 Kanten: $(n-2)$ Überlagerungen

$$\frac{5+4+3+2+1}{=15 \text{ Kanten}}$$



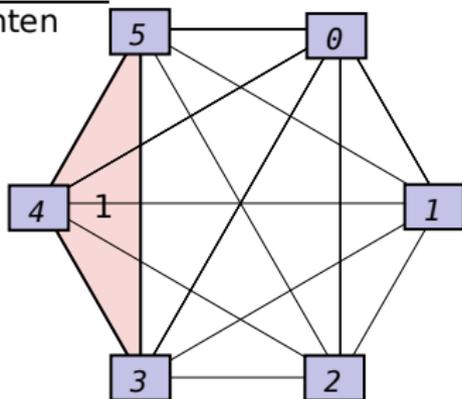
$$\begin{aligned} &4+3+2+1 \\ &3+2+1 \\ &2+1 \end{aligned}$$



Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten
 - $\frac{n(n-1)(n-2)}{6}$ Dreiecke
- Dreieck besteht aus 3 Kanten: $(n-2)$ Überlagerungen

$$\frac{5+4+3+2+1}{=15 \text{ Kanten}}$$



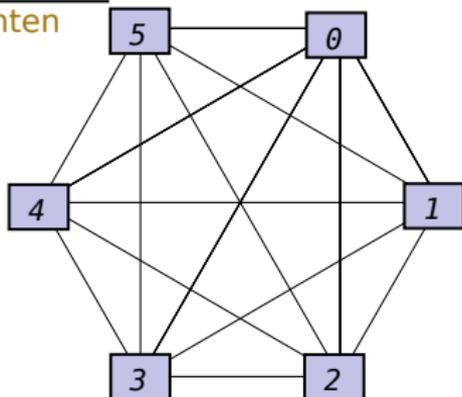
$$\begin{aligned} &4+3+2+1 \\ &3+2+1 \\ &2+1 \\ &1 \end{aligned}$$



Dreiecke im vollständigen Graph

- Vollständiger Graph mit n Knoten enthält:
 - $\frac{n(n-1)}{2}$ Kanten
 - $\frac{n(n-1)(n-2)}{6}$ Dreiecke
- Dreieck besteht aus 3 Kanten: $(n - 2)$ Überlagerungen

$$\begin{array}{r} 5+4+3+2+1 \\ \hline =15 \text{ Kanten} \end{array}$$



$$\begin{array}{r} 4+3+2+1 \\ 3+2+1 \\ 2+1 \\ 1 \\ \hline =20 \text{ Dreiecke} \end{array}$$

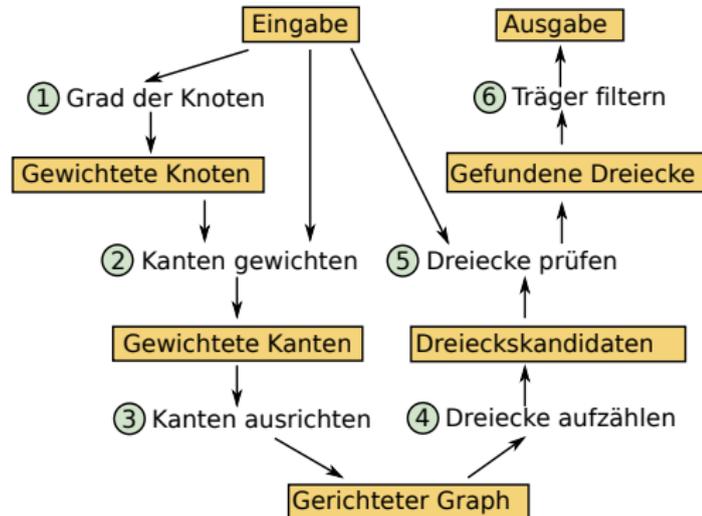
$$\begin{array}{r} 20 \times 3 \\ =60 \text{ Kanten} \end{array}$$

$$\frac{60 \text{ Kanten}}{15 \text{ Kanten}} = 4$$



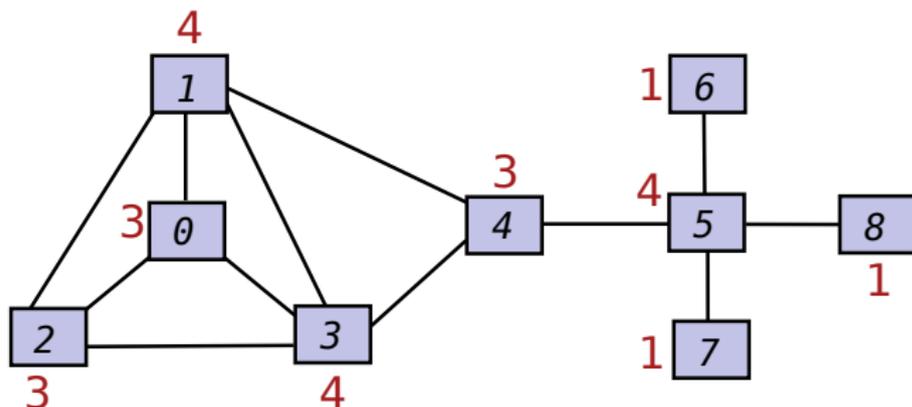
Implementierung mittels MapReduce

- Implementierung in einzeltem MapReduce-Schritt nicht möglich
- Zerlegung in mehrere hintereinander ausgeführte Schritte
- Ausnutzen der **Sortierphase** zum Zusammenführen interessanter Daten
- Ablauf:



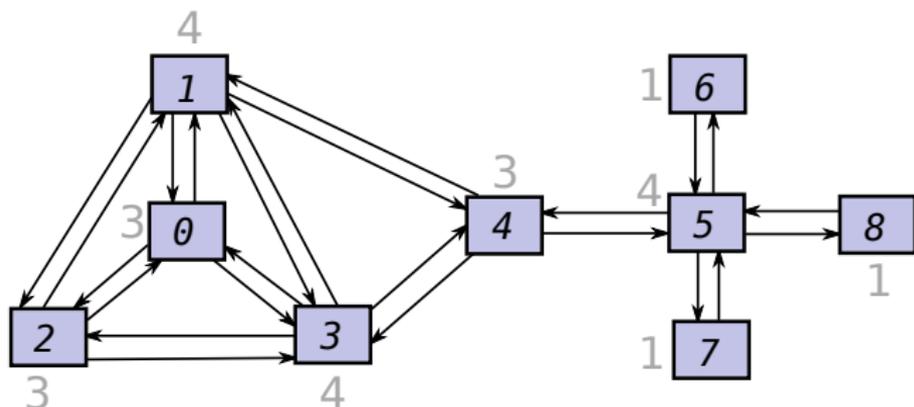
Schritt 1: Grad der Knoten bestimmen

- Abzählen der Kanten, die mit jedem Knoten verbunden sind (→ Grad)
- Gewichtung der Knoten nach Grad



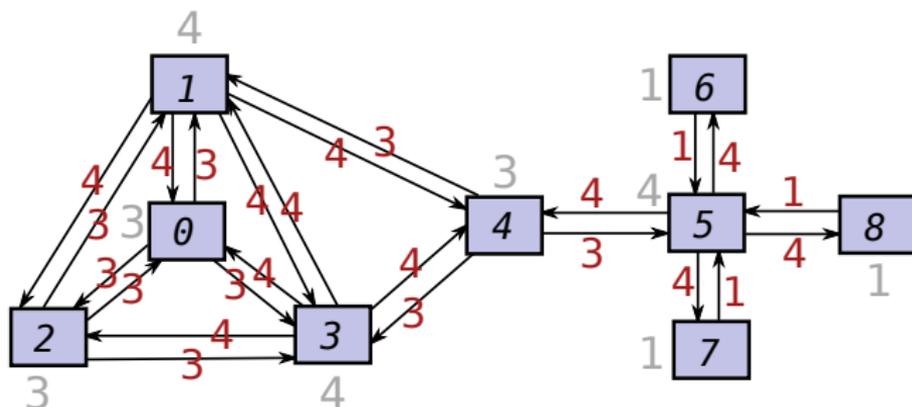
Schritt 2: Kanten gewichten

- Verwandlung in gerichteten Graph, mit Kanten in beide Richtungen
- Gewichtung der gerichteten Kanten nach Startknoten



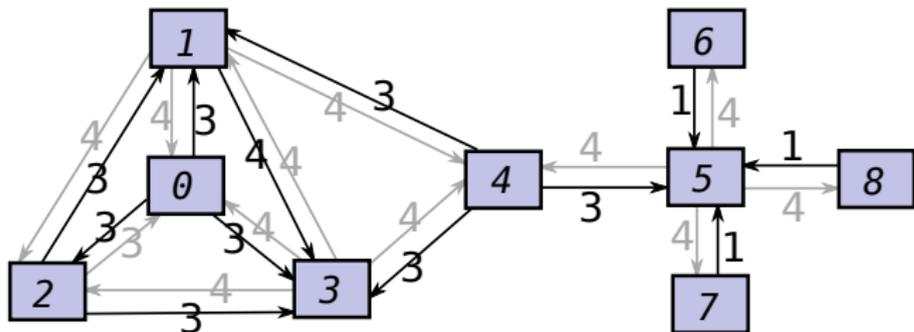
Schritt 2: Kanten gewichten

- Verwandlung in gerichteten Graph, mit Kanten in beide Richtungen
- Gewichtung der gerichteten Kanten nach Startknoten



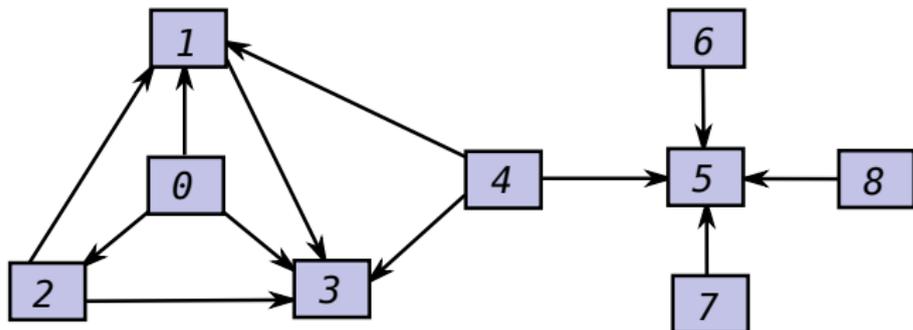
Schritt 3: Kanten ausrichten

- Entfernen von Kanten mit größerer Gewichtung ergibt gerichteten Graph
- Bei gleicher Gewichtung gewinnt Kante mit niedrigerem Startknoten
- Ausrichtung bestimmt Basis für Dreieckskandidaten



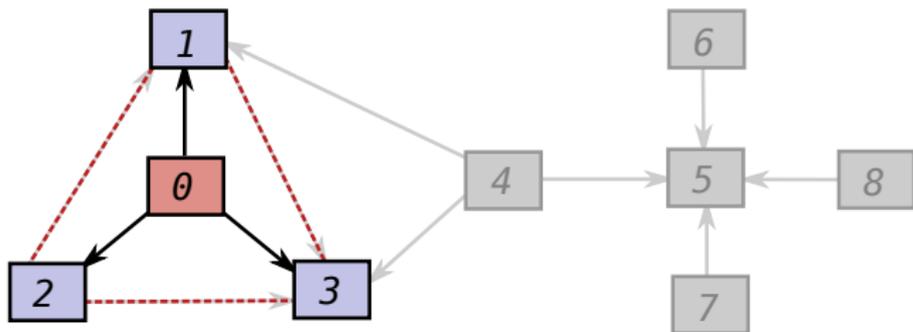
Schritt 3: Kanten ausrichten

- Entfernen von Kanten mit größerer Gewichtung ergibt gerichteten Graph
- Bei gleicher Gewichtung gewinnt Kante mit niedrigerem Startknoten
- Ausrichtung bestimmt Basis für Dreieckskandidaten



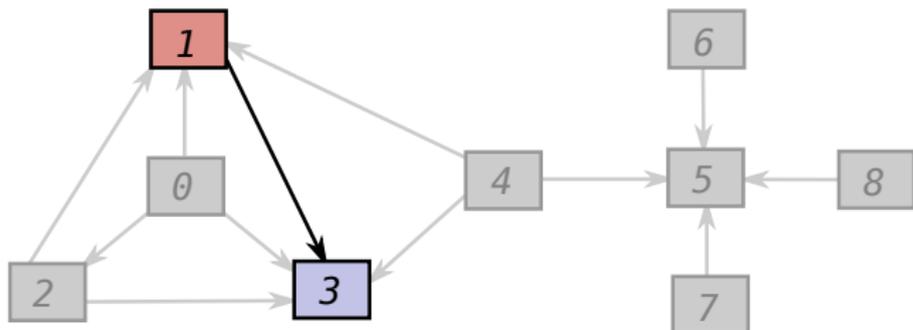
Schritt 4: Dreieckskandidaten aufzählen

- Aufzählen aller ausgehenden Verbindungen je Knoten
- Verbindung von Endpunkten ergibt mögliche Dreiecke



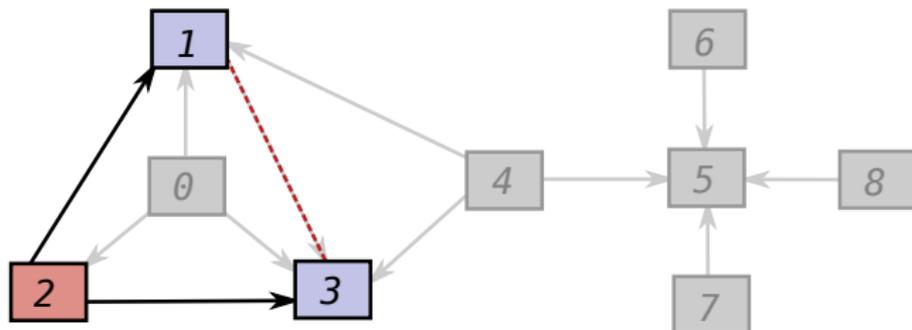
Schritt 4: Dreiecke aufzählen

- Aufzählen aller ausgehenden Verbindungen je Knoten
- Verbindung von Endpunkten ergibt mögliche Dreiecke



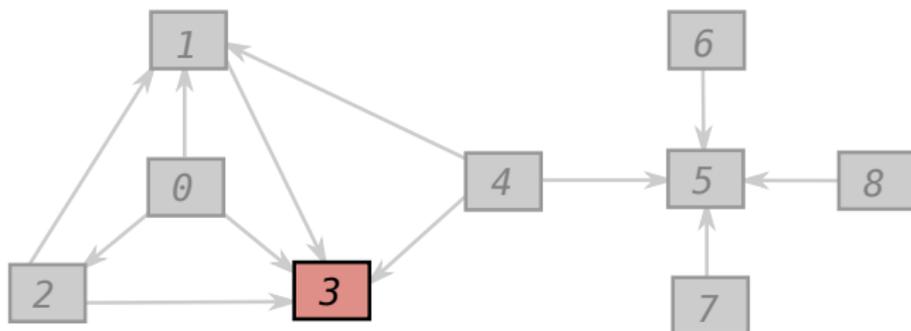
Schritt 4: Dreiecke aufzählen

- Aufzählen aller ausgehenden Verbindungen je Knoten
- Verbindung von Endpunkten ergibt mögliche Dreiecke



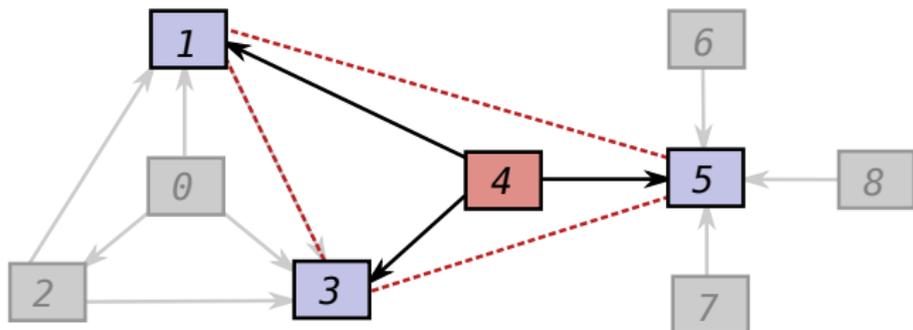
Schritt 4: Dreiecke aufzählen

- Aufzählen aller ausgehenden Verbindungen je Knoten
- Verbindung von Endpunkten ergibt mögliche Dreiecke



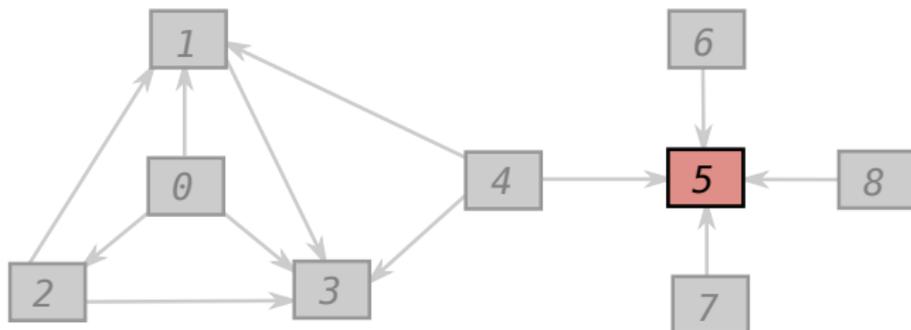
Schritt 4: Dreiecke aufzählen

- Aufzählen aller ausgehenden Verbindungen je Knoten
- Verbindung von Endpunkten ergibt mögliche Dreiecke



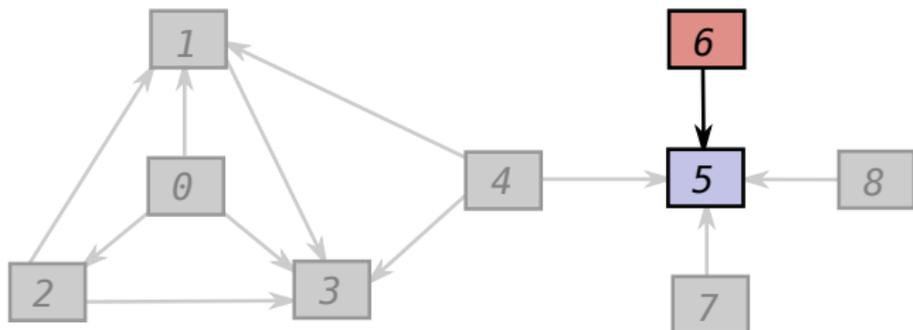
Schritt 4: Dreiecke aufzählen

- Aufzählen aller ausgehenden Verbindungen je Knoten
- Verbindung von Endpunkten ergibt mögliche Dreiecke



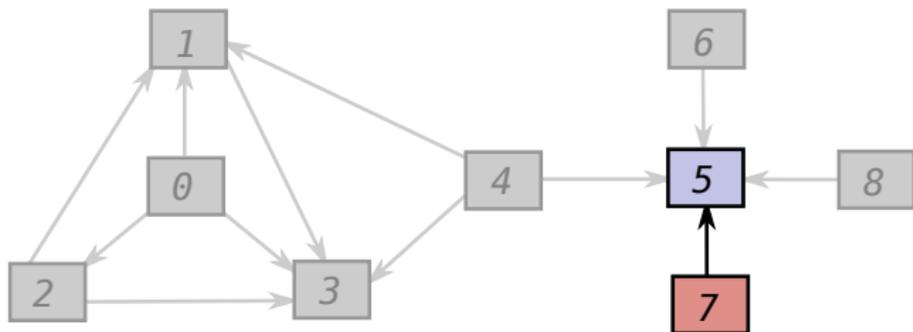
Schritt 4: Dreiecke aufzählen

- Aufzählen aller ausgehenden Verbindungen je Knoten
- Verbindung von Endpunkten ergibt mögliche Dreiecke



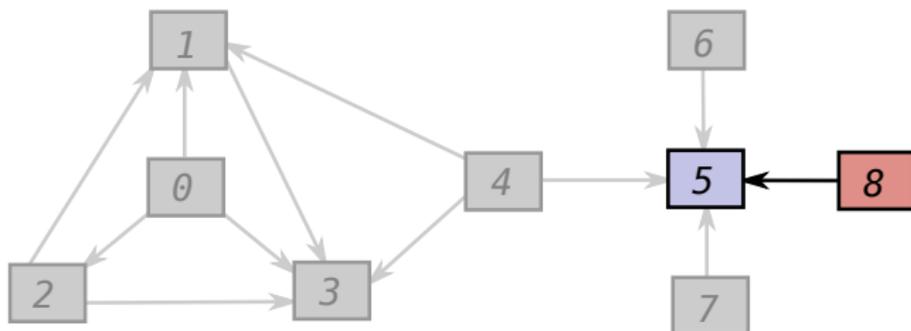
Schritt 4: Dreiecke aufzählen

- Aufzählen aller ausgehenden Verbindungen je Knoten
- Verbindung von Endpunkten ergibt mögliche Dreiecke



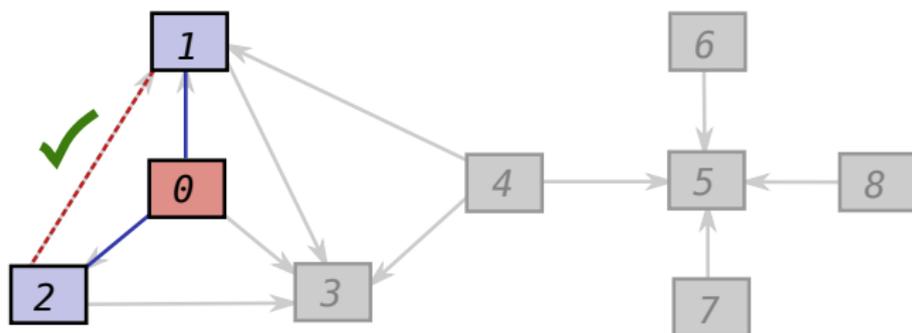
Schritt 4: Dreiecke aufzählen

- Aufzählen aller ausgehenden Verbindungen je Knoten
- Verbindung von Endpunkten ergibt mögliche Dreiecke



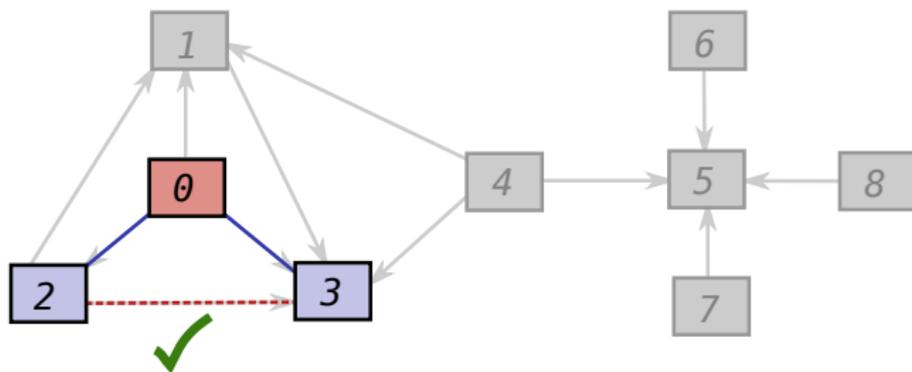
Schritt 5: Dreiecke prüfen

- Prüfen ob Dreieckskandidaten in Graphen existieren
- Nachweis durch Finden der in Schritt 4 ergänzten Kante im Graph
- Ausgabe aller bestätigten Dreiecke



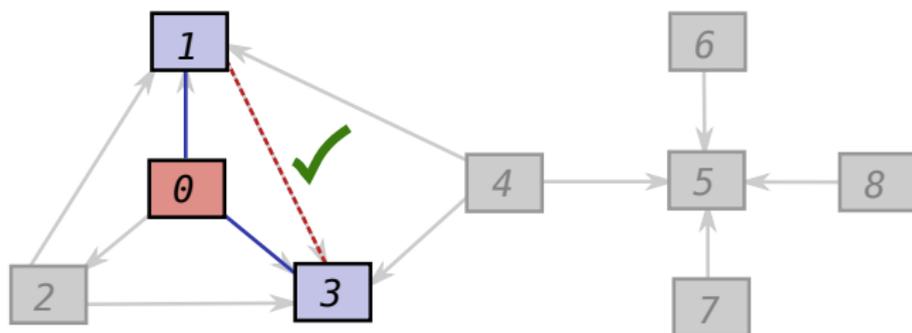
Schritt 5: Dreiecke prüfen

- Prüfen ob Dreieckskandidaten in Graphen existieren
- Nachweis durch Finden der in Schritt 4 ergänzten Kante im Graph
- Ausgabe aller bestätigten Dreiecke



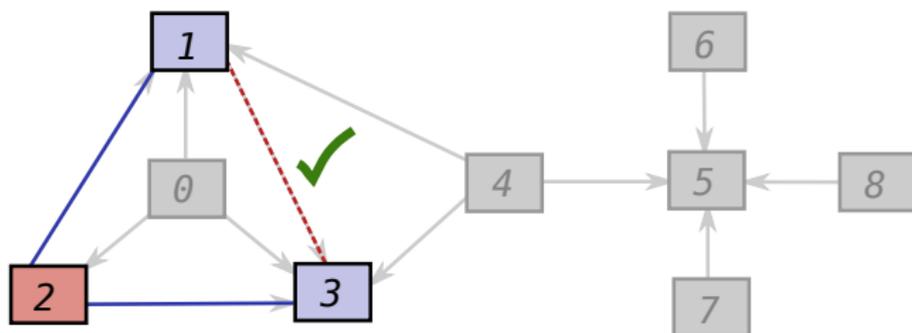
Schritt 5: Dreiecke prüfen

- Prüfen ob Dreieckskandidaten in Graphen existieren
- Nachweis durch Finden der in Schritt 4 ergänzten Kante im Graph
- Ausgabe aller bestätigten Dreiecke



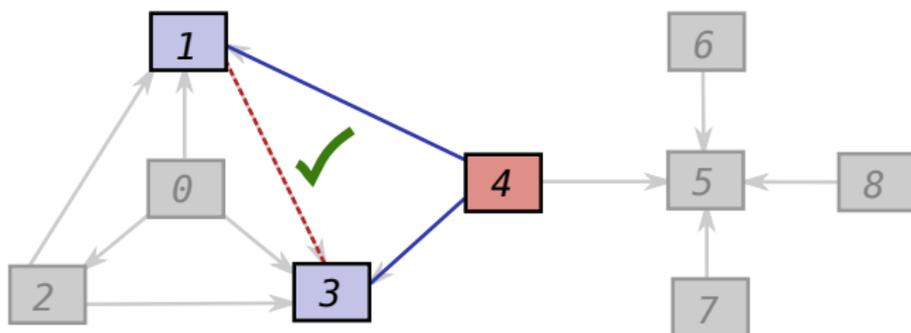
Schritt 5: Dreiecke prüfen

- Prüfen ob Dreieckskandidaten in Graphen existieren
- Nachweis durch Finden der in Schritt 4 ergänzten Kante im Graph
- Ausgabe aller bestätigten Dreiecke



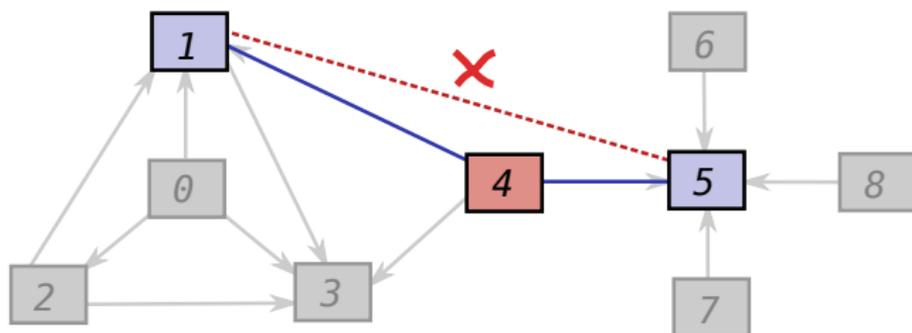
Schritt 5: Dreiecke prüfen

- Prüfen ob Dreieckskandidaten in Graphen existieren
- Nachweis durch Finden der in Schritt 4 ergänzten Kante im Graph
- Ausgabe aller bestätigten Dreiecke



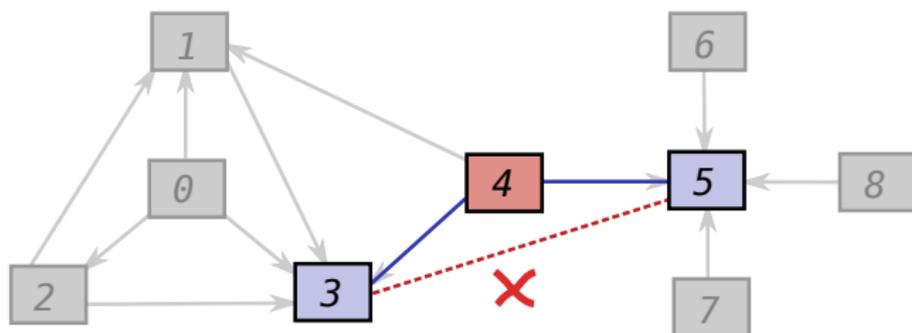
Schritt 5: Dreiecke prüfen

- Prüfen ob Dreieckskandidaten in Graphen existieren
- Nachweis durch Finden der in Schritt 4 ergänzten Kante im Graph
- Ausgabe aller bestätigten Dreiecke



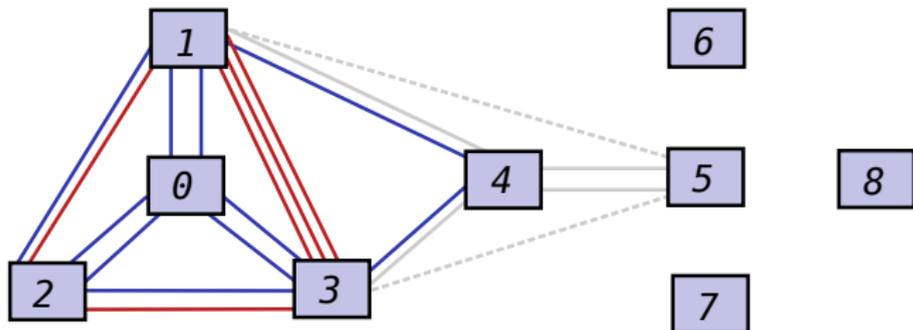
Schritt 5: Dreiecke prüfen

- Prüfen ob Dreieckskandidaten in Graphen existieren
- Nachweis durch Finden der in Schritt 4 ergänzten Kante im Graph
- Ausgabe aller bestätigten Dreiecke



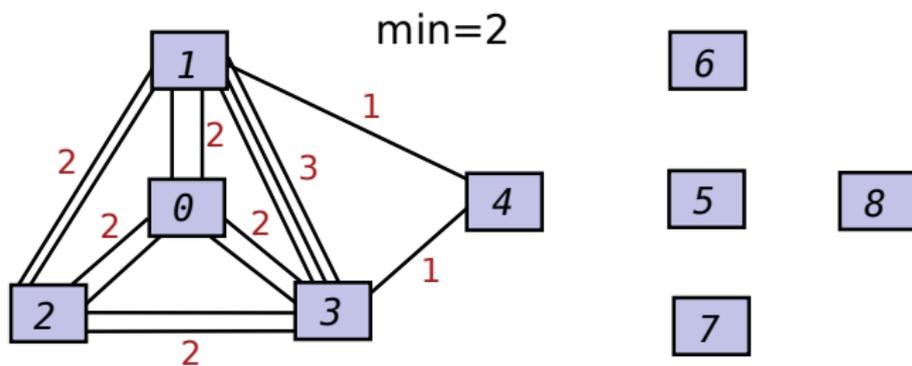
Schritt 6: Träger filtern

- Zählen der Kantenüberlagerungen in der Menge der validierten Dreiecke
- Herausfiltern aller Kanten mit zu wenigen Überlagerungen
- Beispiel: $n \geq 2$ entspricht Cliques mit mindestens 4 Knoten



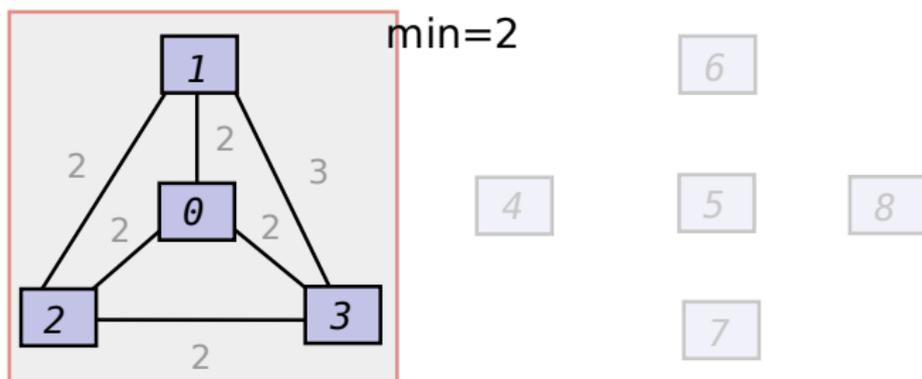
Schritt 6: Träger filtern

- Zählen der Kantenüberlagerungen in der Menge der validierten Dreiecke
- Herausfiltern aller Kanten mit zu wenigen Überlagerungen
- Beispiel: $n \geq 2$ entspricht Cliques mit mindestens 4 Knoten



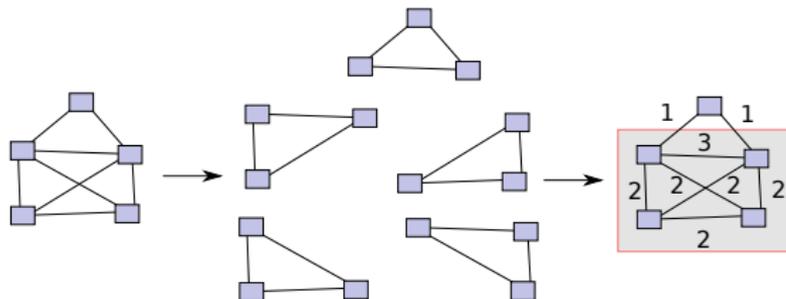
Schritt 6: Träger filtern

- Zählen der Kantenüberlagerungen in der Menge der validierten Dreiecke
- Herausfiltern aller Kanten mit zu wenigen Überlagerungen
- Beispiel: $n \geq 2$ entspricht Cliques mit mindestens 4 Knoten



Träger und Cliques

- Algorithmus isoliert nur **Träger**: $n - 2$ Überlagerungen zwar notwendig, aber nicht hinreichend
- Nicht an der Clique beteiligte Kanten erhöhen Zahl der Überlagerungen an den Rändern
- Filtern von Kanten mit zu wenigen Überlagerungen reduziert Anzahl der Dreiecke im Graph
- **Mehrfache Iteration notwendig**, bis keine Kanten mehr wegfallen
- Beispiel: Bei suche nach Cliques mit mindestens 5 Knoten (= 3 Überlappungen) Graph nicht leer



Graph-Algorithmen mit MapReduce

Kurzeinführung Graphentheorie

Algorithmus zum Finden von Cliques

MapReduce Erweiterungen

Zusammenführen von Daten

Sortierung, Gruppierung und Partitionierung

Combiner

Zähler

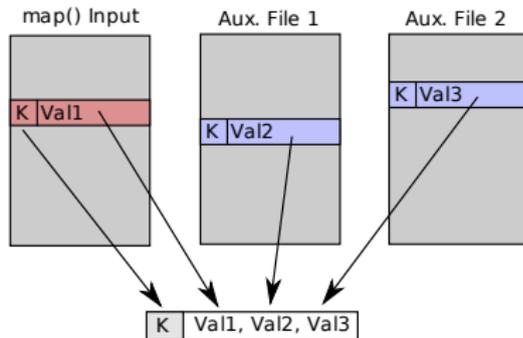


- Bei der Analyse von Daten häufig auftretendes Problem:
Zusammenführen von Informationen nach bestimmtem Schlüssel
- Datenbanken: **Join**-Operation
(`SELECT * FROM x INNER JOIN y USING (c)`)
- Möglichkeiten zur Implementierung:
 - Map-side Join
 - Reduce-side Join
- Zur Umsetzung beider Varianten notwendig:
 - Einlesen mehrerer unterschiedlicher Eingabedateien
 - Evtl. unterschiedliche Mapper je Datentyp



Map-side Join

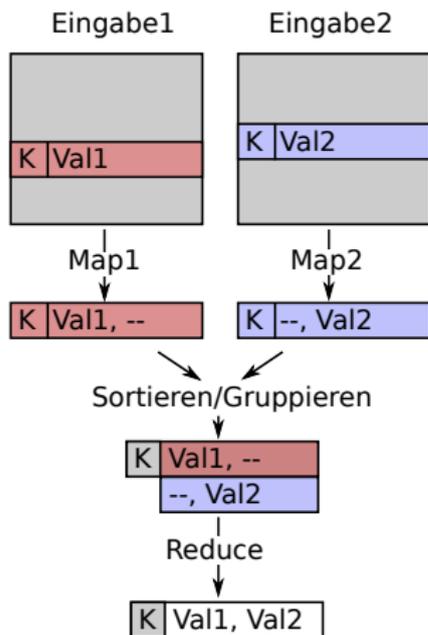
- Zusammenfassen von Daten während des Map-Vorgangs
- **Mapper** hält alle notwendigen Eingabedateien offen
- Durch Framework zugewiesener Datenbereich gibt zu suchende Schlüssel an
 - Suche des jeweiligen Schlüssels in anderen Dateien
- Voraussetzung für effiziente Ausführung: Daten bereits nach Schlüssel **vorsortiert** (→ siehe z.B. **Merge-Algorithmus**)
- Mapper gibt fertig zusammengeführte Daten aus



- Nutzen des Sortiervorgangs zum zusammenfügen passender Schlüssel
- Sortierung werden Daten von **unterschiedlichen Mappern** zugeführt
- Mapper bilden unterschiedliche Eingabedaten auf gleichen Ausgabetypp für Sortierung und Reduce-Phase ab
- Sortierung/Gruppierung führt gleiche Schlüssel und damit zusammengehörige Daten zu einer Gruppe zusammen
- **Reducer** erzeugt zusammengesetzten Datensatz



Reduce-side Join



Graph-Algorithmen mit MapReduce

Kurzeinführung Graphentheorie

Algorithmus zum Finden von Cliquen

MapReduce Erweiterungen

Zusammenführen von Daten

Sortierung, Gruppierung und Partitionierung

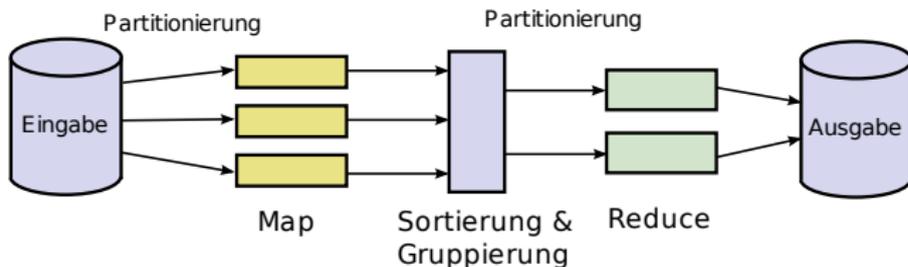
Combiner

Zähler



Sortierung, Gruppierung und Partitionierung

■ MapReduce-Grundkonzept:

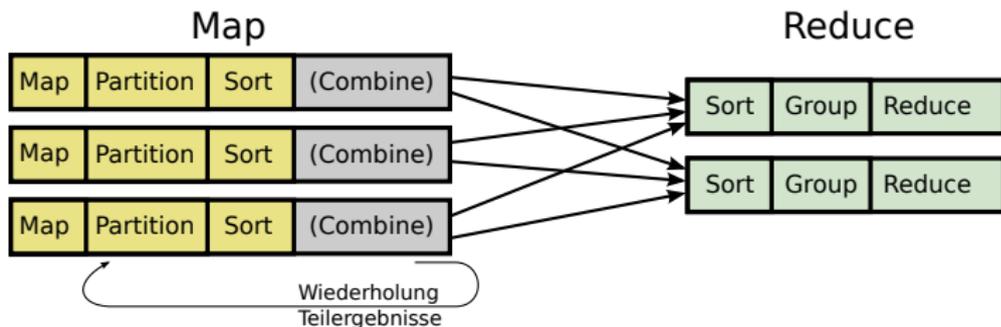


- Sortierung nicht parallelisierbar
- → Datenabhängige Partitionierung erlaubt paralleles Sortieren
- Datenübergabe zwischen Mapper und Reducer aufwändig
- → Reduktion der zu übertragenden Datenmenge mittels „Combiner“



Sortierung, Gruppierung und Partitionierung

- **Sortierung:** Reihenfolge der Schlüssel/Wert-Paare
- **Gruppierung:** Schlüssel, die beim Reduce zusammengefasst werden
- **Partitionierung:** Zuordnung der Schlüssel zu Reduce-Instanzen mittels Hash-Funktion
- Implementierung in Hadoop:



- Schnittstelle in Hadoop:
 - Sortierung/Gruppierung: `org.apache.hadoop.io.RawComparator`,
Achtung: arbeitet auf (binären) Rohdaten
 - Partitionierung: `org.apache.hadoop.mapreduce.Partitioner`

- Bei Implementierung zu beachtende Eigenschaften:
 - Sortierung gibt **strengere** oder gleichwertige Ordnung gegenüber Gruppierung vor
 - Hash-Funktion im Partitionierer generiert für alle Schlüssel einer Gruppe **gleichen** Hash-Wert



- Klasse `WritableComparator` ermöglicht Vergleich von Objekten, die die `WritableComparable` Schnittstelle implementieren
- Deserialisierung der Binärdaten und Zuführen zu `compare()`
- Ohne Überschreiben der `compare()`-Methode wird `compareTo()` der übergebenen Objekte verwendet
- Beispiel:

```
public class Sorting extends WritableComparator {
    public Sorting() {
        super(Edge.class, true);
    }

    // Standardimplementierung
    public int compare(WritableComparable a,
                      WritableComparable b) {
        return a.compareTo(b);
    }
}
```



- Standard-Implementierung `HashPartitioner` verwendet `hashCode()`-Methode von `Object` zur Partitionierung
- **Allerdings:** Standard-Implementierung von `hashCode()` nicht ausreichend, muss bei Implementierung von `Writable`-Schnittstelle überschrieben werden
- Beispiel: Kanten nach linkem Knoten gruppieren

```
public class Partition
    extends Partitioner<Edge, Writable> {

    @Override
    public int getPartition(Edge k, Writable v, int max) {
        return (k.getLeft().hashCode() & Integer.MAX_VALUE) % max;
    }
}
```



- Je nach Aufgabenstellung bereits teilweise Ausführung von Reduce-Schritt auch am Ende des Mappers möglich
- **Voraussetzungen:**
 - Reduce-Schritt darf nicht von **Vollständigkeit** einer Gruppe abhängen
 - Reducer muss **weniger** Ausgabedaten produzieren als Eingabedaten gelesen werden

→ Reduktion der zu sortierenden Datenmenge

- Implementierung in Hadoop mittels **Combiner**



Combiner zur Beschleunigung

- Schnittstelle des Combiners entspricht der des Reducers
- **Beispiel:** Wörter zählen

```
public class Combiner extends Reducer<Text, IntWritable,  
                                     Text, IntWritable> {  
  
    public void reduce(Text k, Iterable<IntWritable> v,  
                       Context ctx)  
        throws IOException, InterruptedException {  
        int count = 0;  
  
        for (IntWritable c: v) {  
            count += c.get();  
        }  
        context.write(k, new IntWritable(count));  
    }  
}
```

- Hier: Reducer identisch mit Combiner



- Hadoop ermöglicht Zählen von Ereignissen mittels `org.apache.hadoop.mapreduce.Counters`
- Von Hadoop selbst verwendet zum Anfertigen verschiedener Statistiken (z.B. Anzahl gelesener/geschriebener Bytes)
- Funktioniert auch bei verteilter Ausführung knotenübergreifend
- Erlauben nur erhöhen des Zählerstandes
- Identifizierung mittels `Enums`
 - `Enum`-Klasse bildet **Zählergruppe**
 - `Enum`-Einträge bildet einzelne **Zähler** der Gruppe



■ Beispiel:

```
public class MyClass {
    public static enum MyGroup{
        VERTEX_COUNT, EDGE_COUNT
    }

    public void map(Key k, Value v, Context ctx) {
        Counter vc = ctx.getCounter(MyGroup.VERTEX_COUNT);
        Counter ec = ctx.getCounter(MyGroup.EDGE_COUNT);
        vc.increment(2);
        ec.increment(1);
    }

    public void printVertexCount(Job job) {
        Counters cts = job.getCounters();
        Counter ct = cts.findCounter(MyGroup.VERTEX_COUNT);
        System.out.println("Vertices: " + ct.getValue());
    }
}
```



- [Cohen09]: Jonathan Cohen. Graph Twiddling in a MapReduce World. In *Computing in Science & Engineering Volume 11 Issue 4*, S. 29-41, ISSN 1521-9615, IEEE Computer Society, 2009
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5076317
(Kopie: /proj/i4mw/pub/aufgabe4/Cohen-Graph_Twiddling.pdf)

- Apache Hadoop Dokumentation
<http://hadoop.apache.org/common/docs/r0.20.2/index.html>

- Apache Hadoop API Javadoc
<http://hadoop.apache.org/common/docs/r0.20.2/api/index.html>

