Middleware - Cloud Computing – Übung

Tobias Distler, Klaus Stengel, Timo Hönig

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

www4.informatik.uni-erlangen.de

Wintersemester 2010/11

Überblick

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

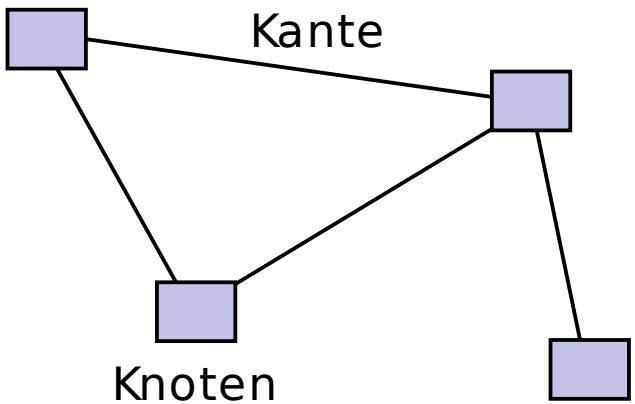
Programmieren mit Hadoop
- Überblick
- Jobs und Tools
- JAR-Dateien erstellen
- Ein- und Ausgabe, Serialisierung
- Zusammenführen von Daten
- MapReduce Erweiterungen
- Verteilte Ausführung
- Dateisysteme in Hadoop

Aufgabe 4

Überblick

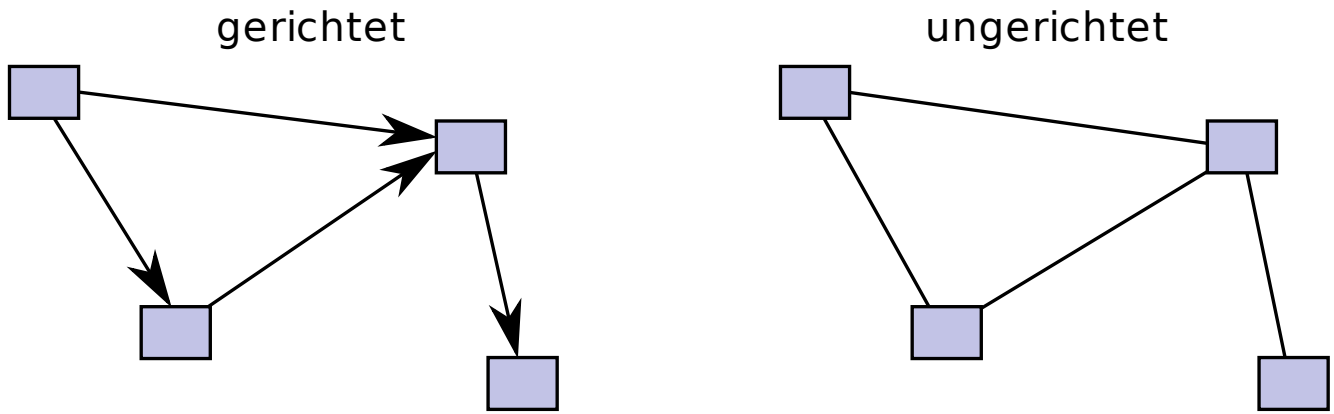
Einführung Graphentheorie

- 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



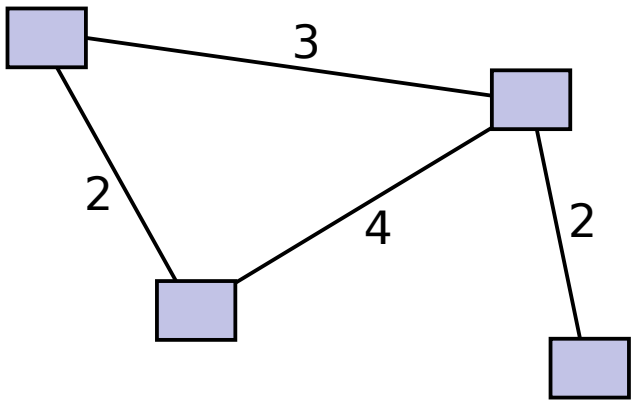
Einführung Graphentheorie

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



Einführung Graphentheorie

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

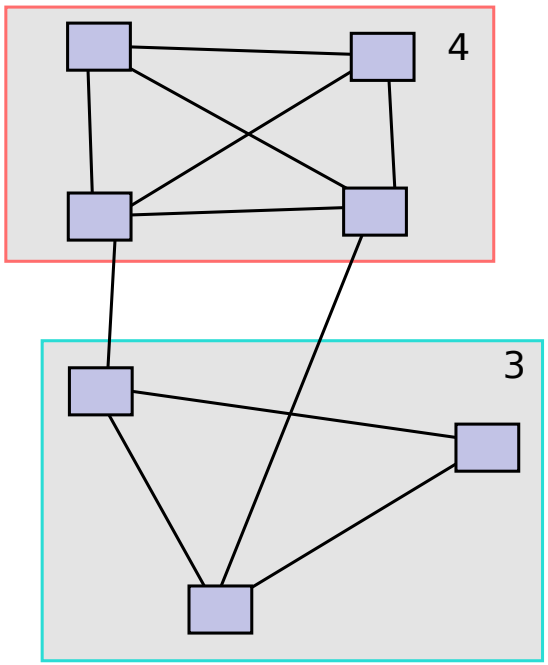


- **Beispiel**:
  Knoten sind verschiedene Städte
  Kanten stellen Wege zwischen Städten dar
  Gewichtung entspricht Weglänge

Überblick

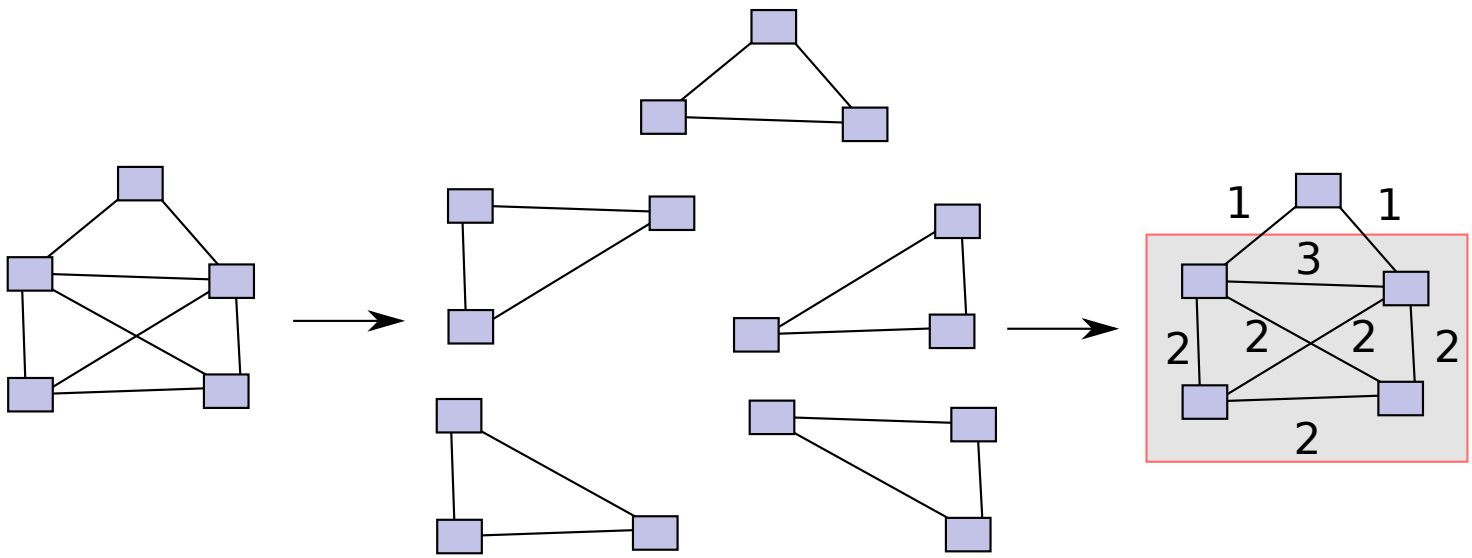
Algorithmus zum Finden von Cliquen

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



Algorithmus zum Finden von Cliquen

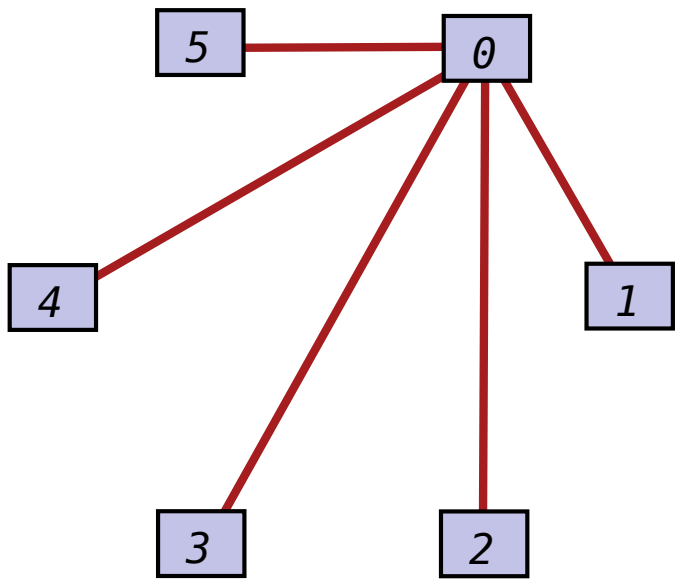
- **Aufgabe**: Finden von Cliquen einer bestimmten Mindestgröße
- Lösungsansatz basiert auf Finden von **Trägern**:
  - Finden von nichttrivialer Cliquen minimaler Größe: Dreiecke
  - Vollständiger Graph mit n Knoten enthält $\frac{n \cdot (n-1) \cdot (n-2)}{6}$ unterschiedliche Dreiecke
  - Jede Kante 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

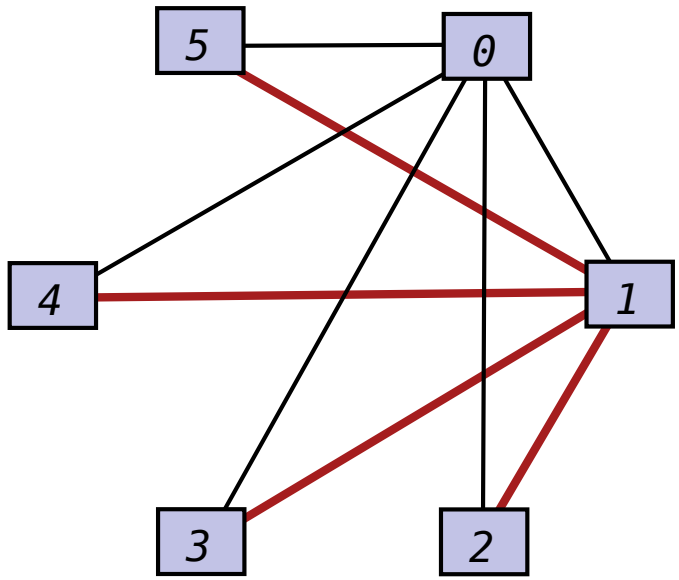
5



Dreiecke im vollständigen Graph

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

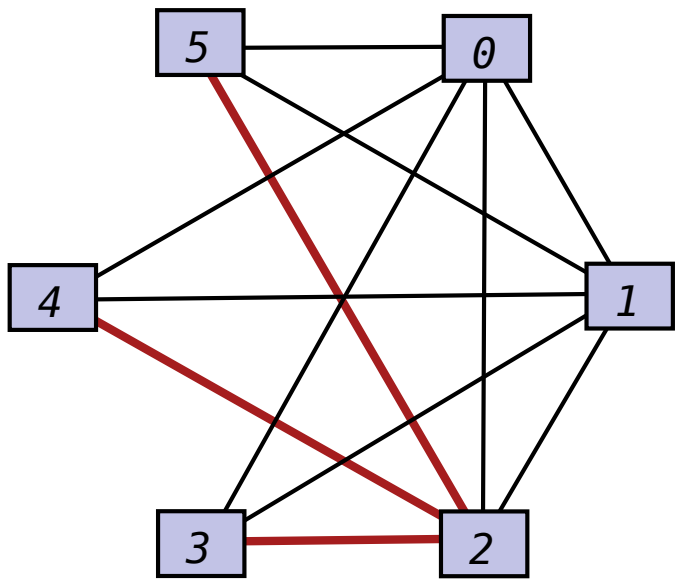
5+4



Dreiecke im vollständigen Graph

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

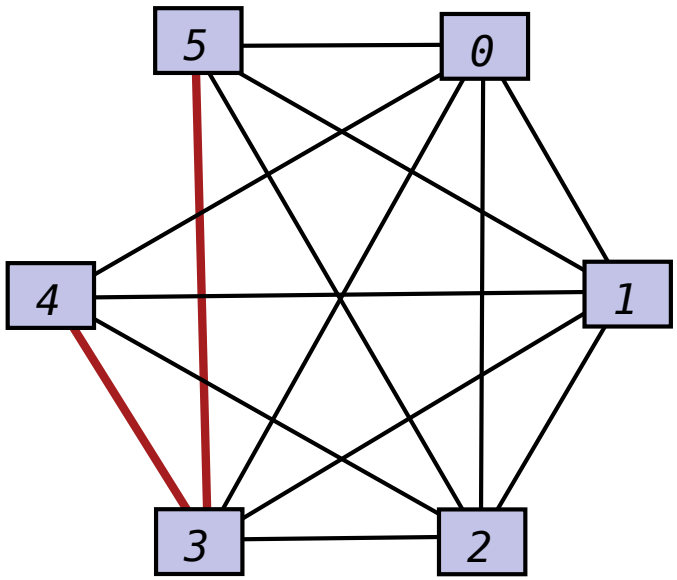
5+4+3



Dreiecke im vollständigen Graph

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

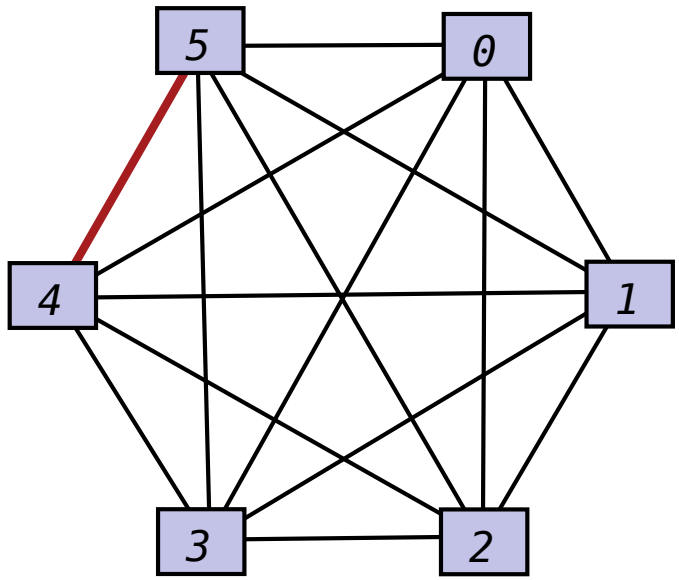
5+4+3+2



Dreiecke im vollständigen Graph

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

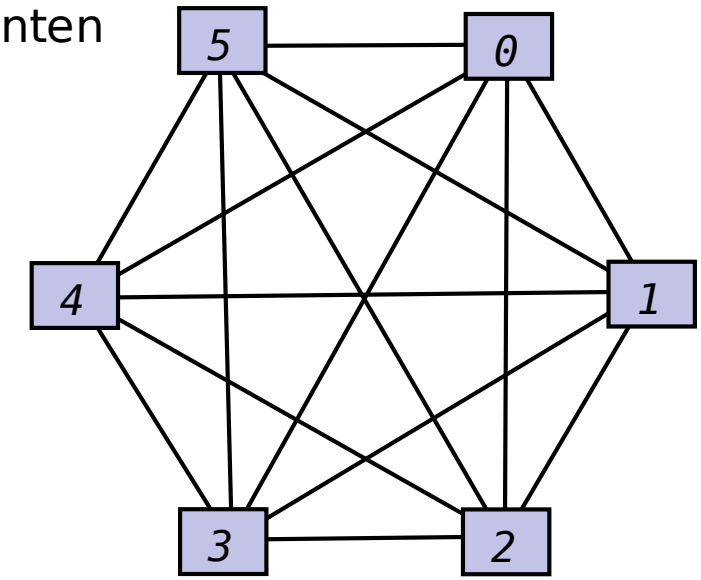
5+4+3+2+1



Dreiecke im vollständigen Graph

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

$5+4+3+2+1$
$=15$ Kanten

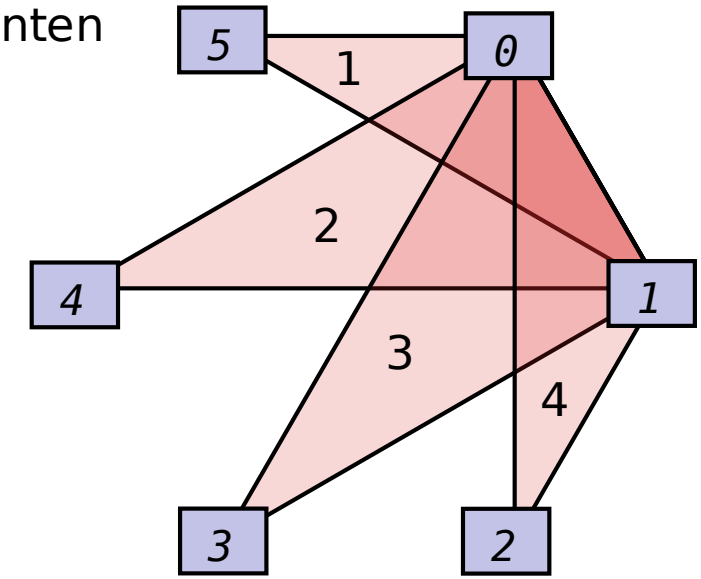


Dreiecke im vollständigen Graph

- 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

5+4+3+2+1
=15 Kanten

4

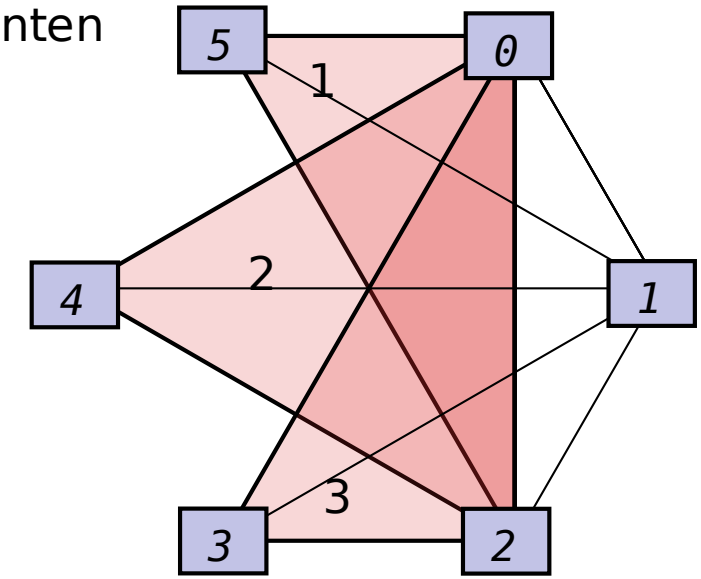


Dreiecke im vollständigen Graph

- 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

5+4+3+2+1
=15 Kanten

4+3

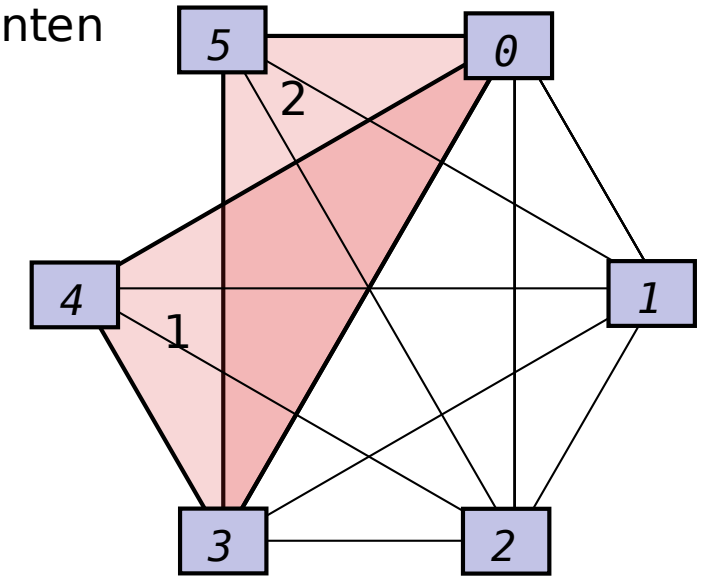


Dreiecke im vollständigen Graph

- 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

5+4+3+2+1
=15 Kanten

4+3+2

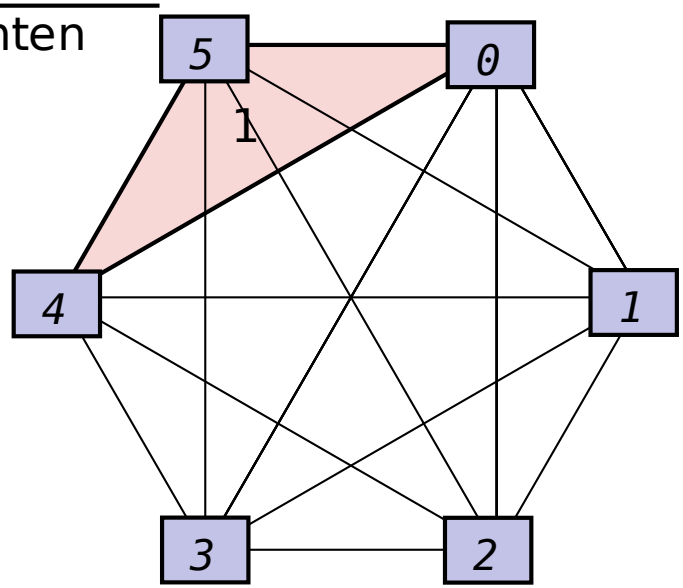


Dreiecke im vollständigen Graph

- 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

5+4+3+2+1
=15 Kanten

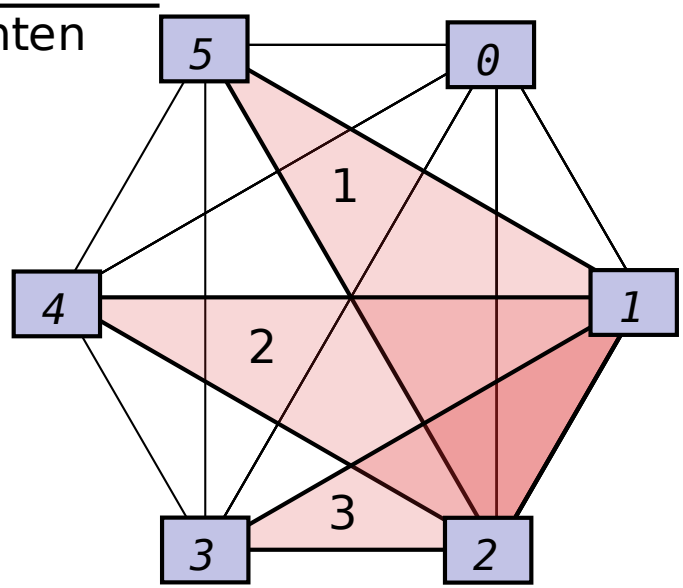
4+3+2+1



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

5+4+3+2+1
=15 Kanten

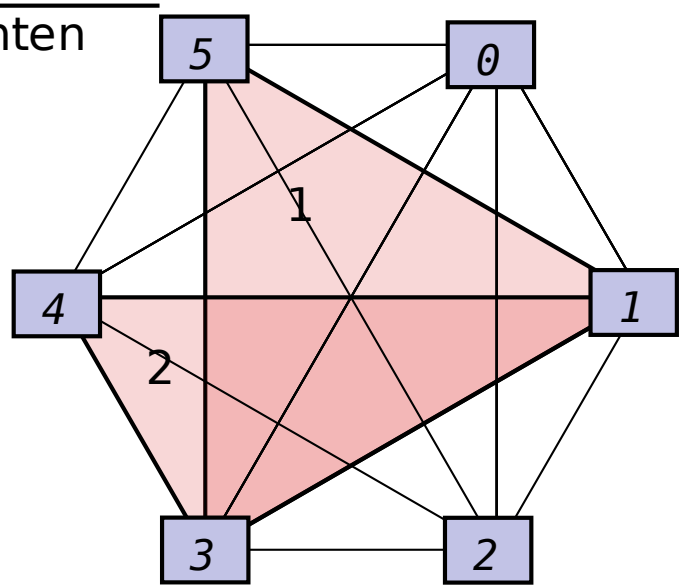


4+3+2+1
3

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

5+4+3+2+1
=15 Kanten

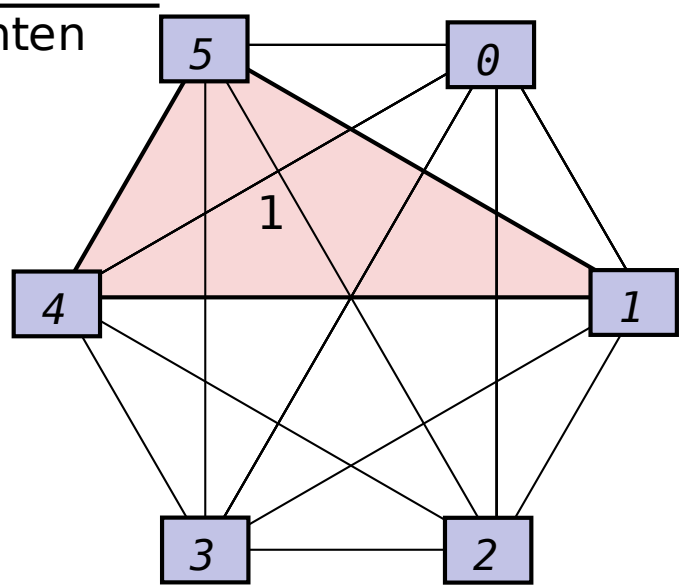


4+3+2+1
3+2

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

5+4+3+2+1
=15 Kanten

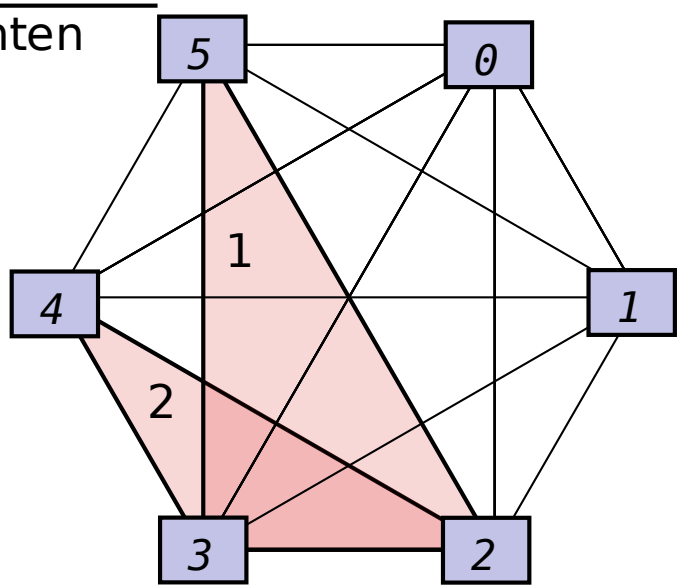


4+3+2+1
3+2+1

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

5+4+3+2+1
=15 Kanten

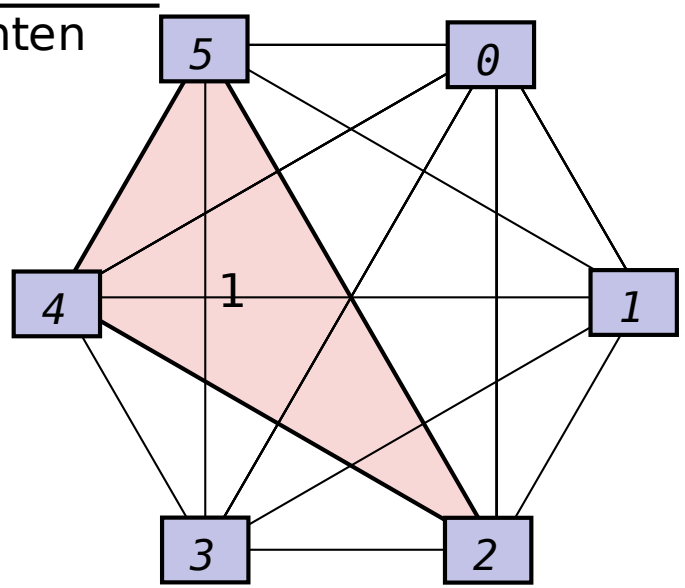


4+3+2+1
3+2+1
2

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

5+4+3+2+1
=15 Kanten

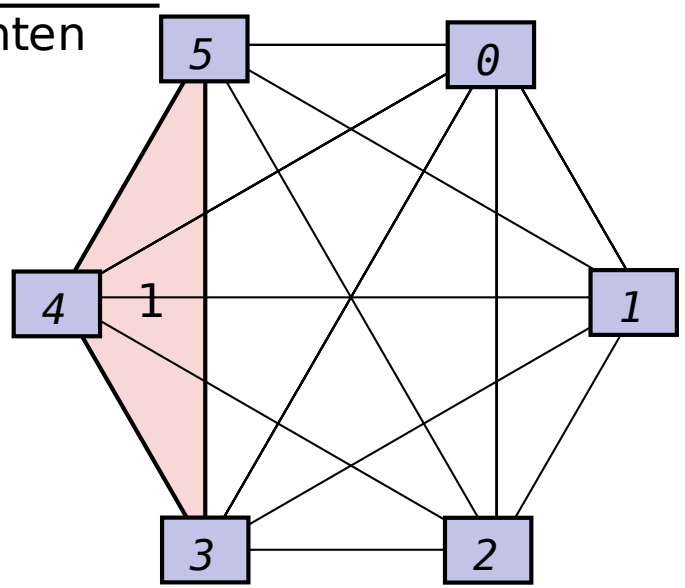


4+3+2+1
3+2+1
2+1

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

5+4+3+2+1
=15 Kanten

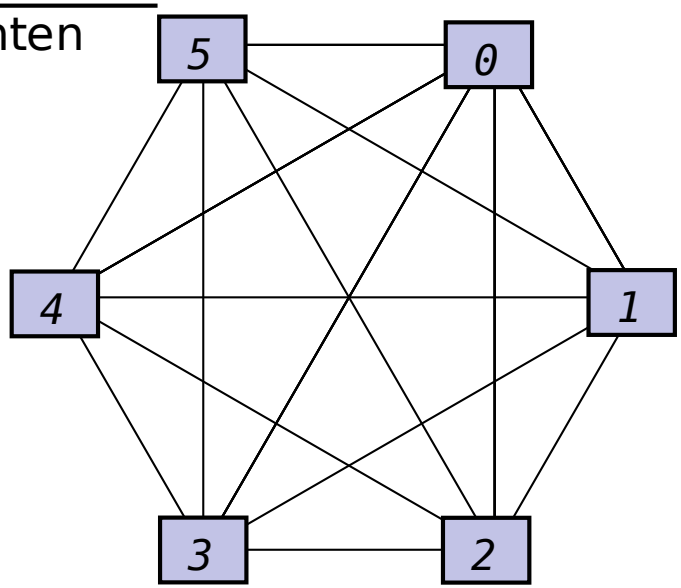


4+3+2+1
3+2+1
2+1
1

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

5+4+3+2+1
=15 Kanten



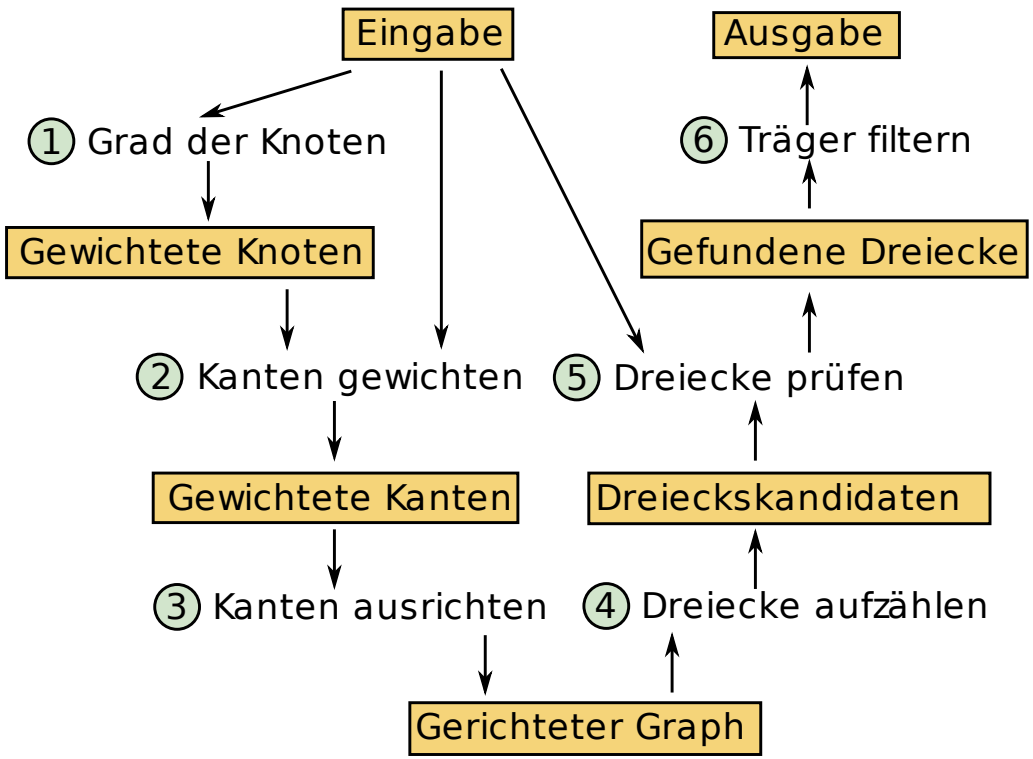
4+3+2+1
3+2+1
2+1
1
=20 Dreiecke

20x3
=60 Kanten

60 / 15 = 4

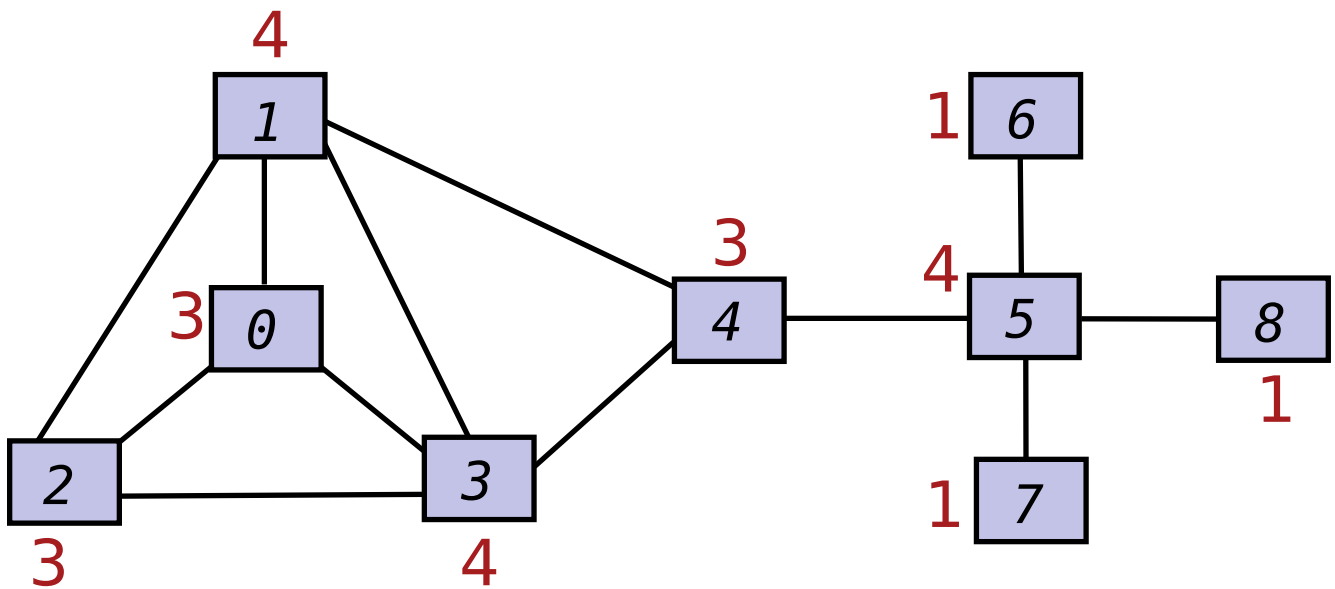
Implementierung mittels MapReduce

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



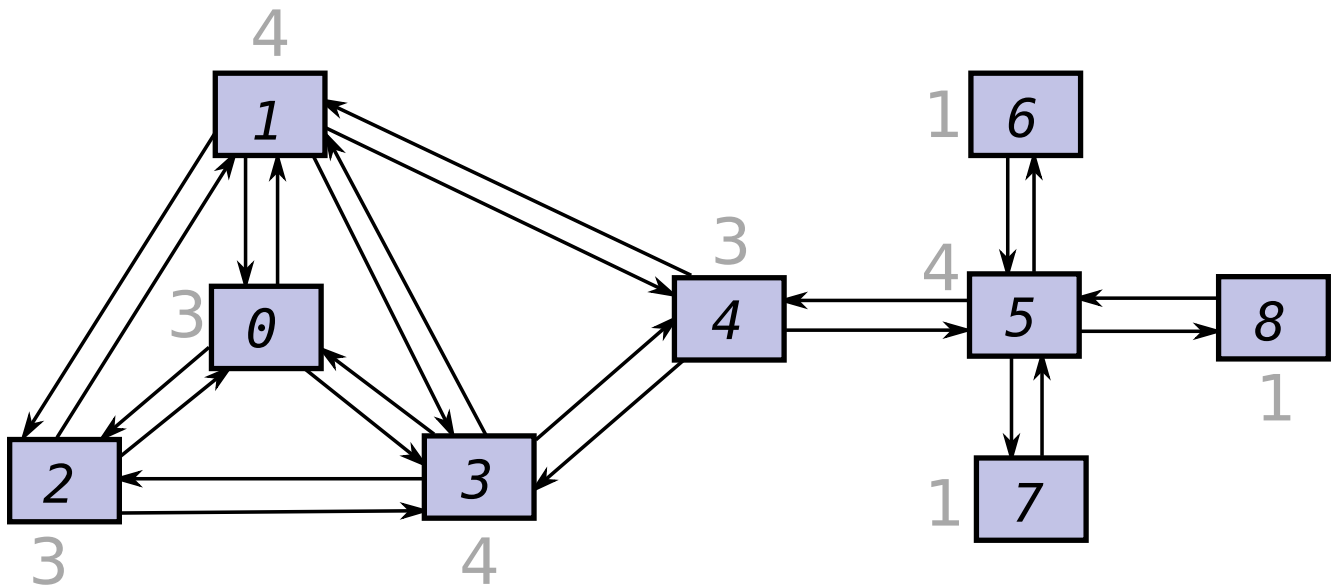
Schritt 1: Grad der Knoten bestimmen

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



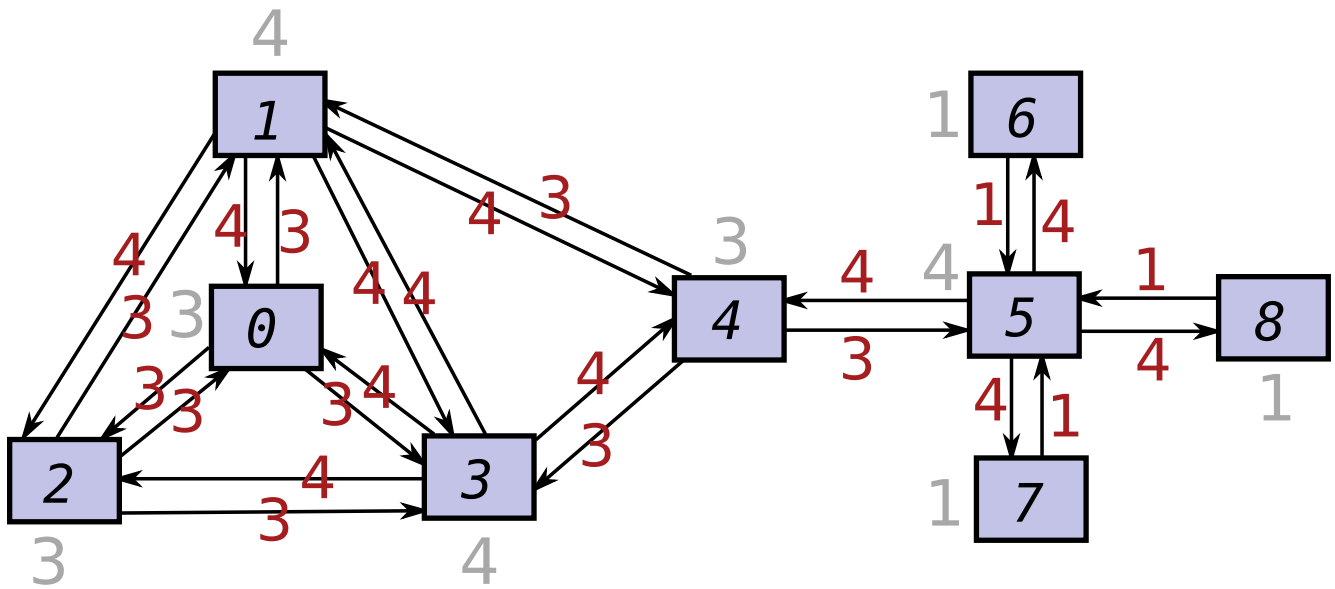
Schritt 2: Kanten gewichten

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



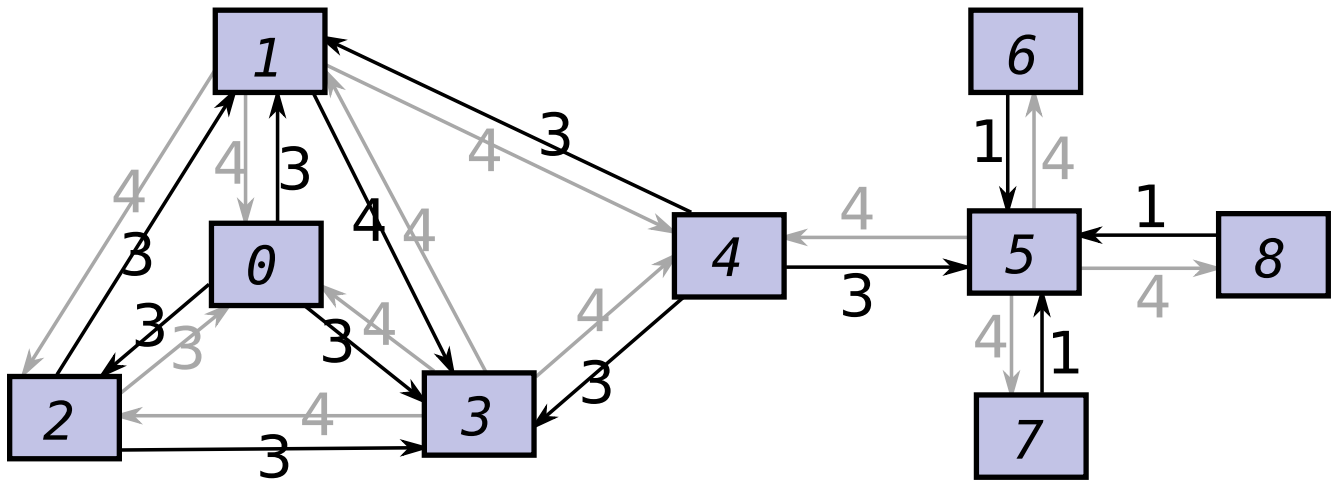
Schritt 2: Kanten gewichten

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



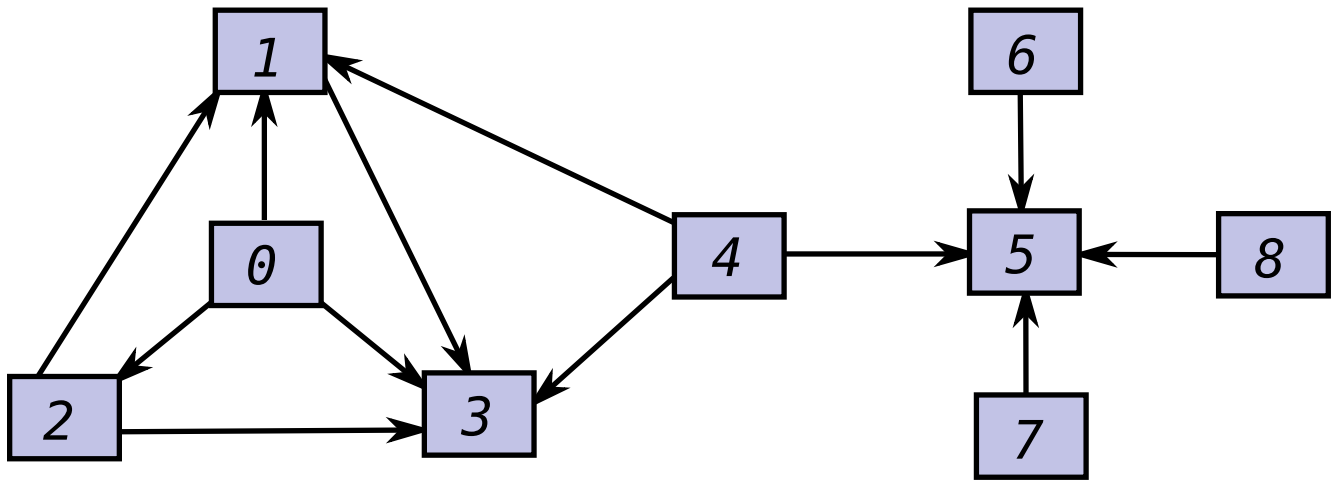
Schritt 3: Kanten ausrichten

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



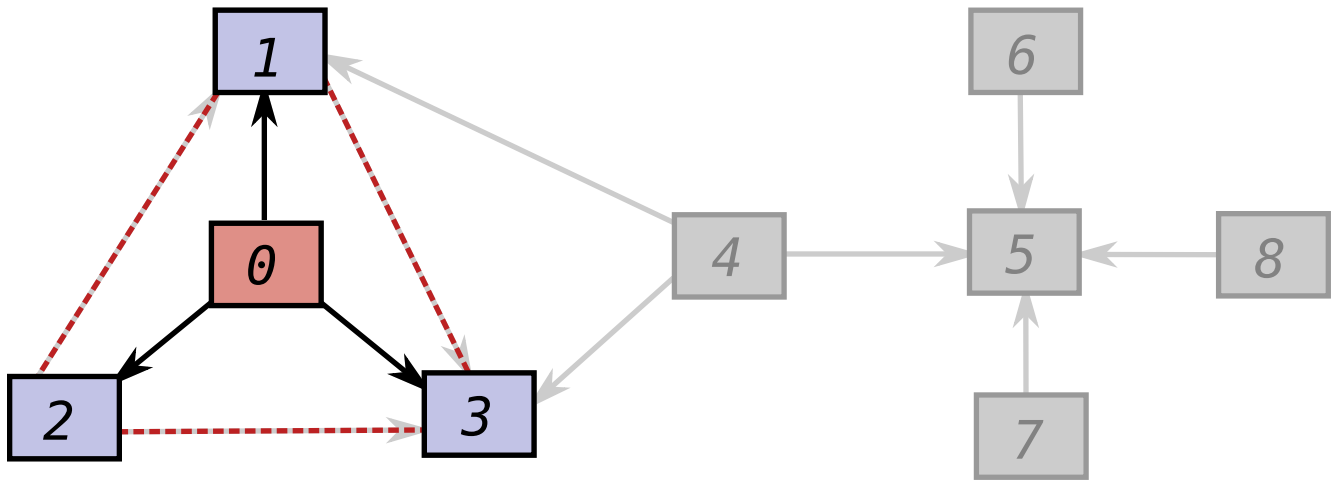
Schritt 3: Kanten ausrichten

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



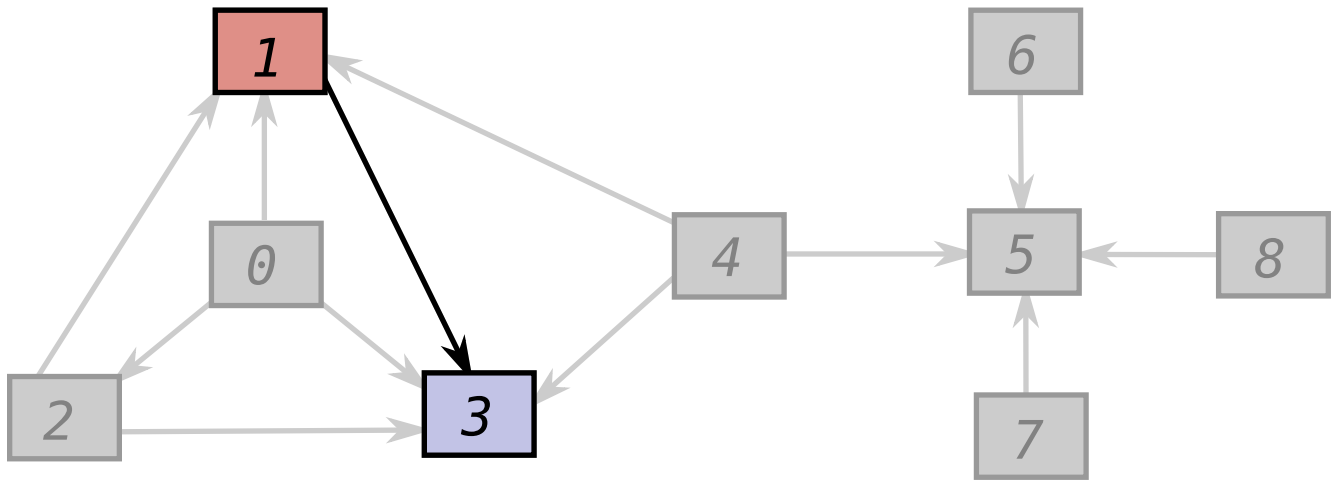
Schritt 4: Dreiecke aufzählen

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



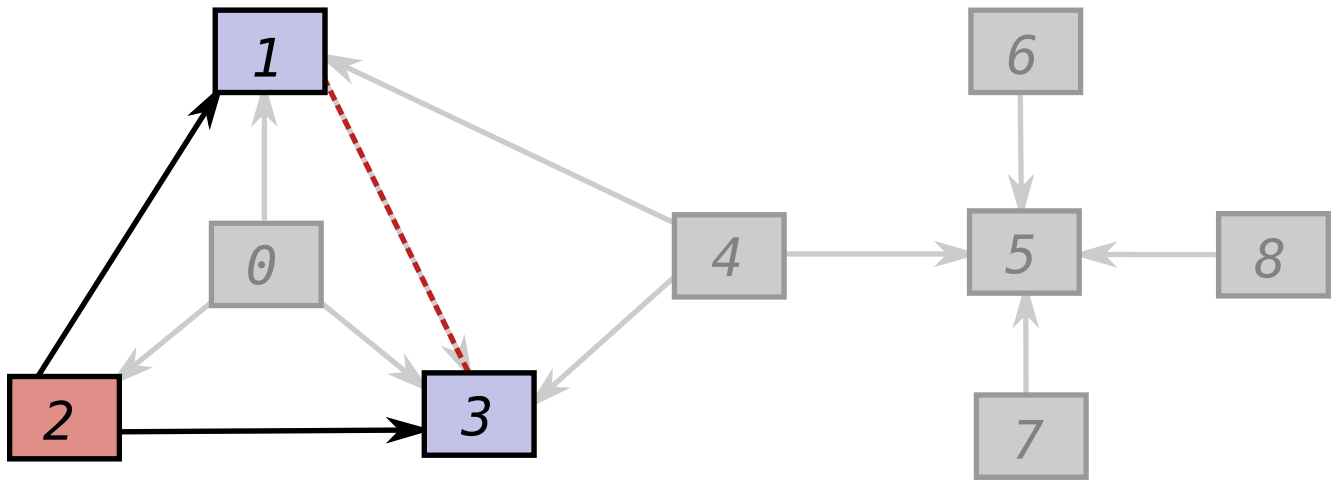
Schritt 4: Dreiecke aufzählen

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



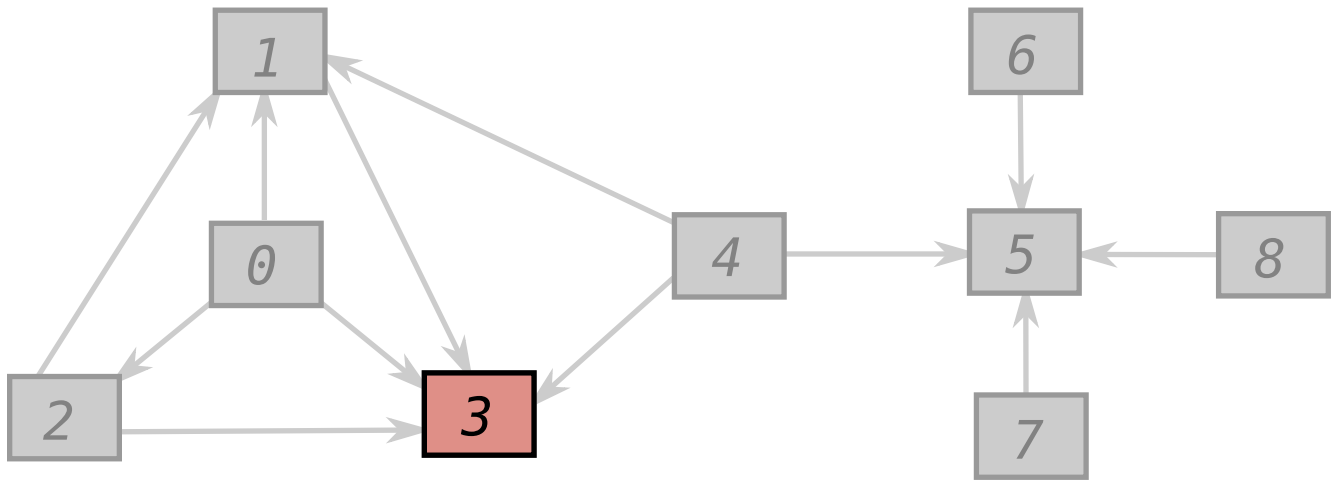
Schritt 4: Dreiecke aufzählen

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



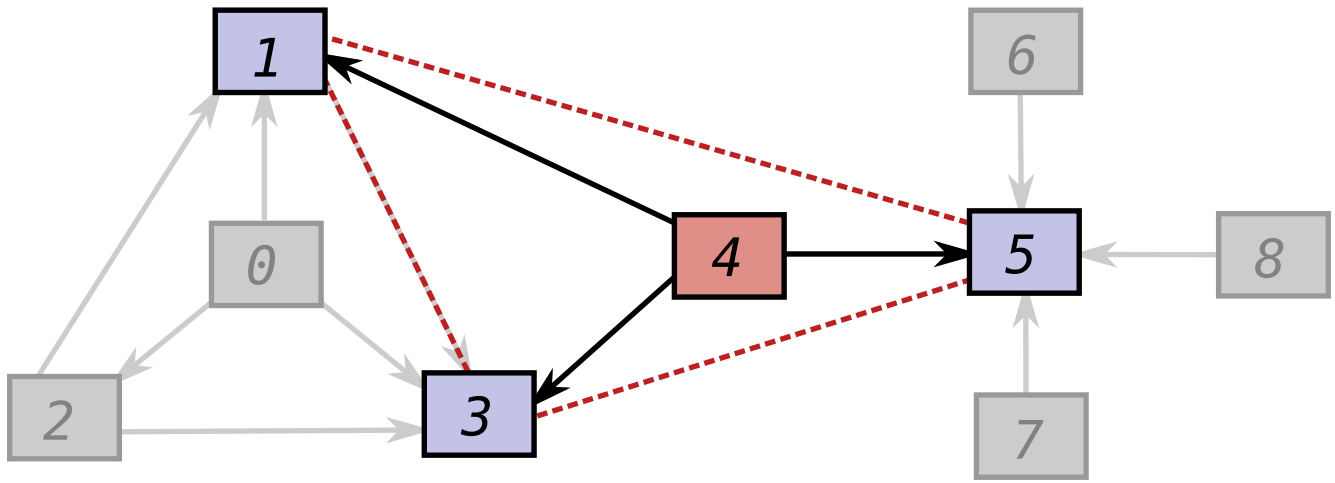
Schritt 4: Dreiecke aufzählen

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



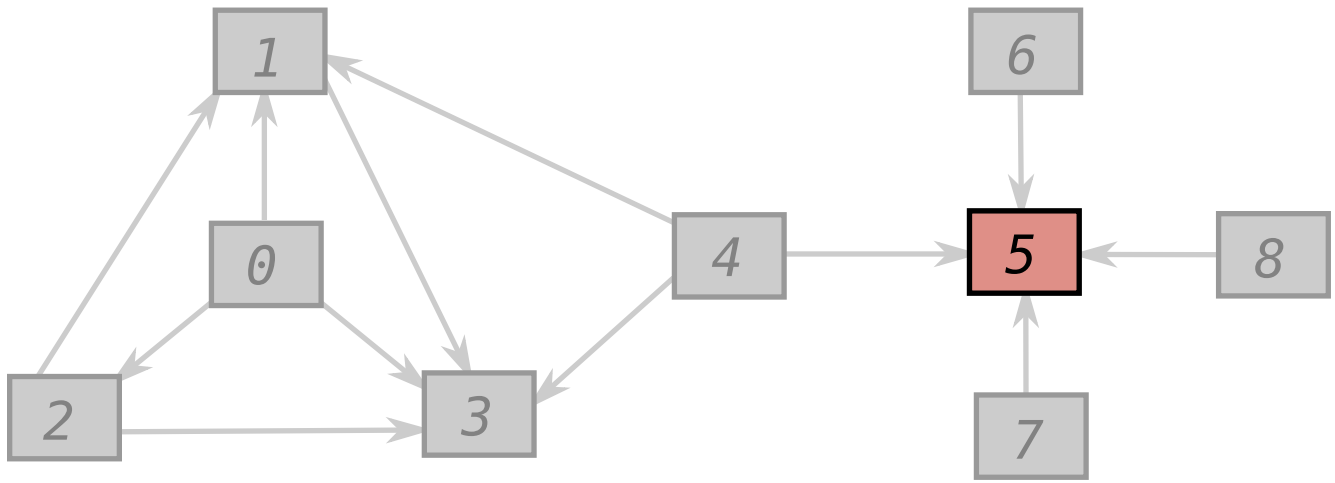
Schritt 4: Dreiecke aufzählen

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



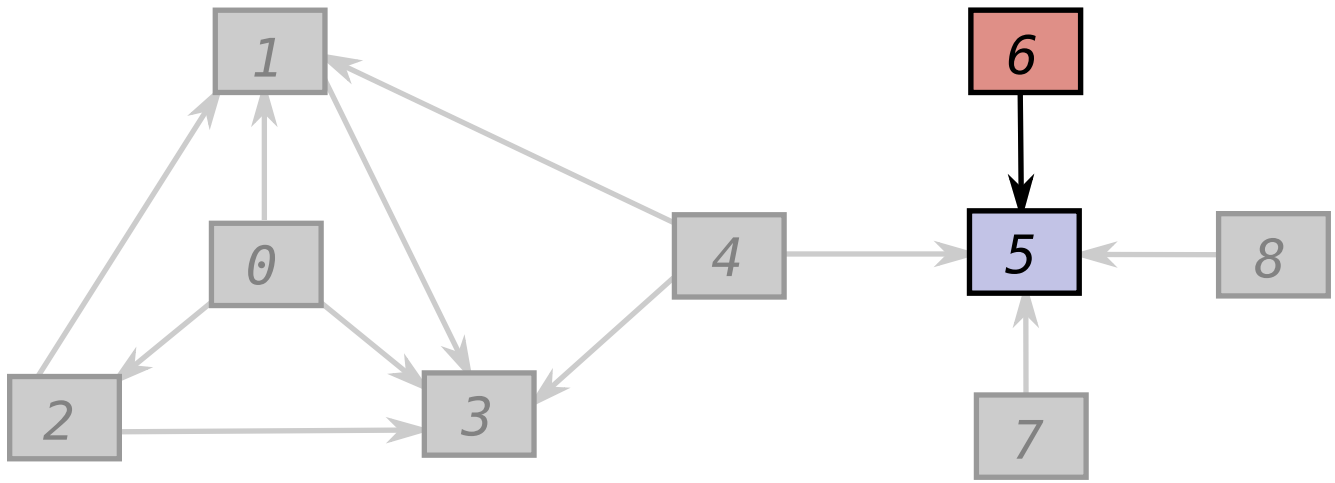
Schritt 4: Dreiecke aufzählen

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



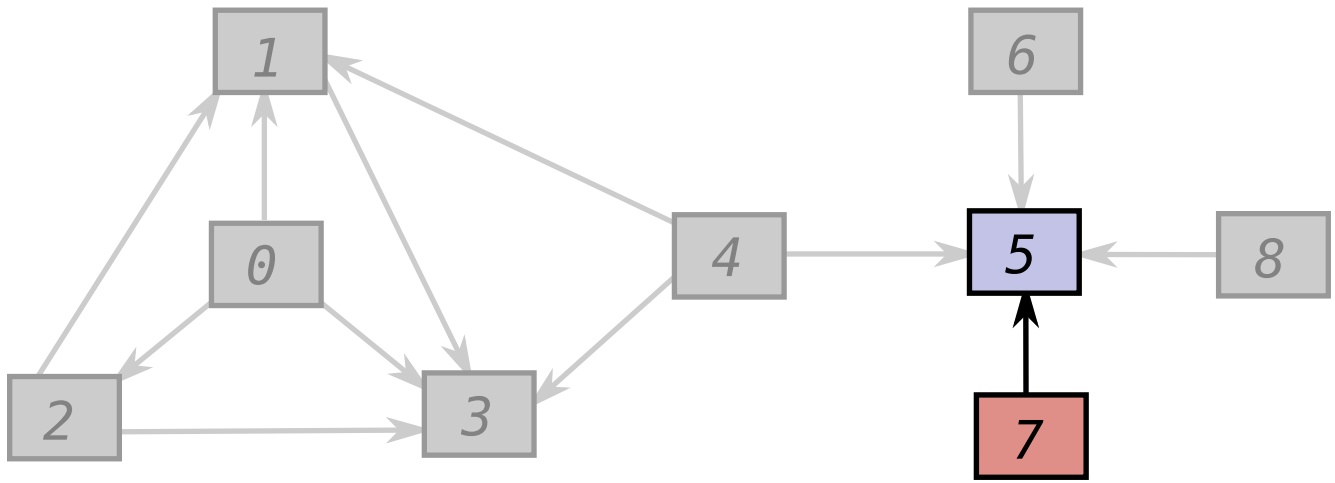
Schritt 4: Dreiecke aufzählen

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



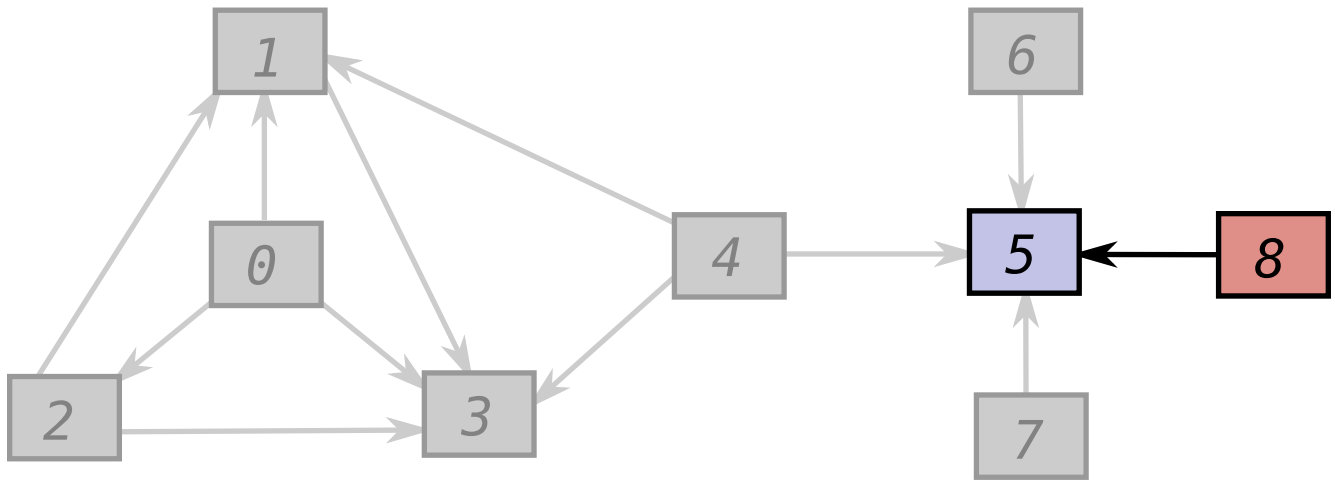
Schritt 4: Dreiecke aufzählen

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



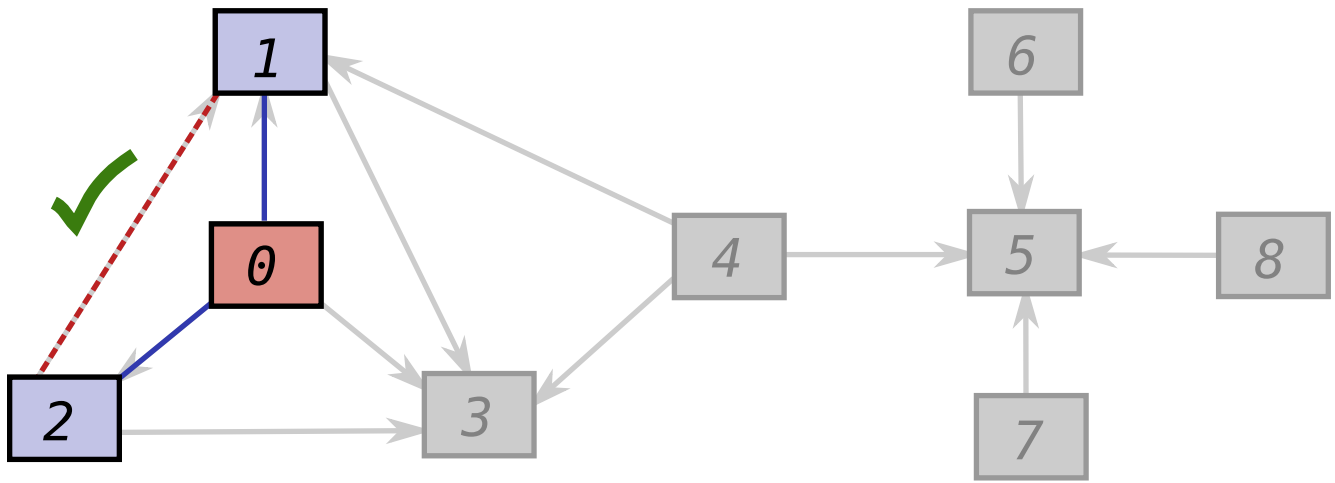
Schritt 4: Dreiecke aufzählen

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



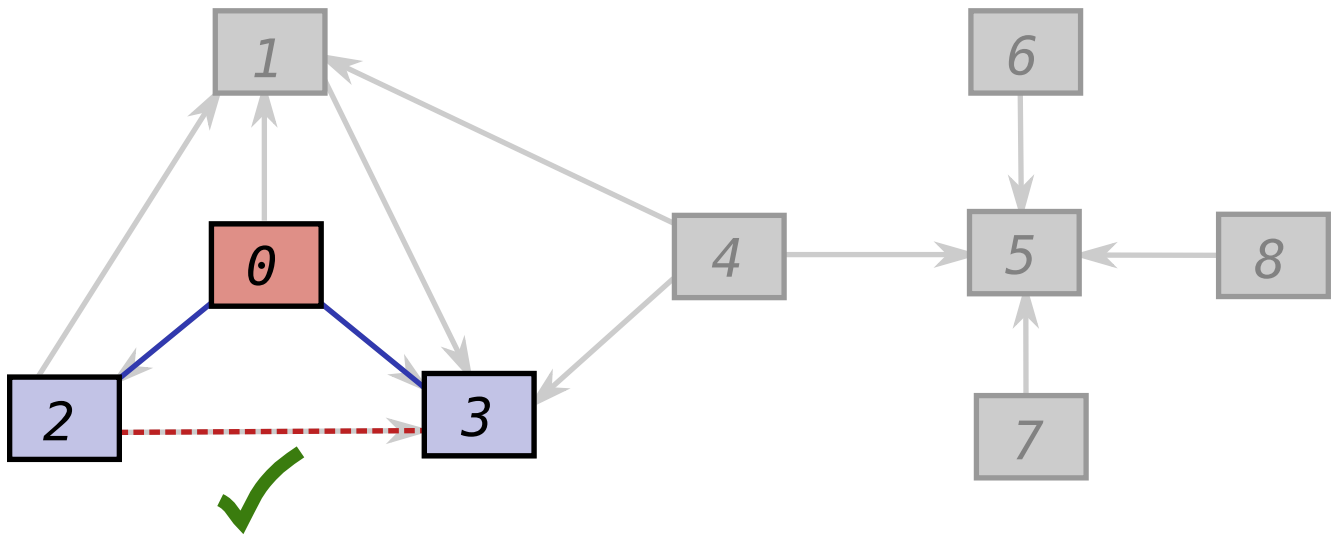
Schritt 5: Dreiecke prüfen

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



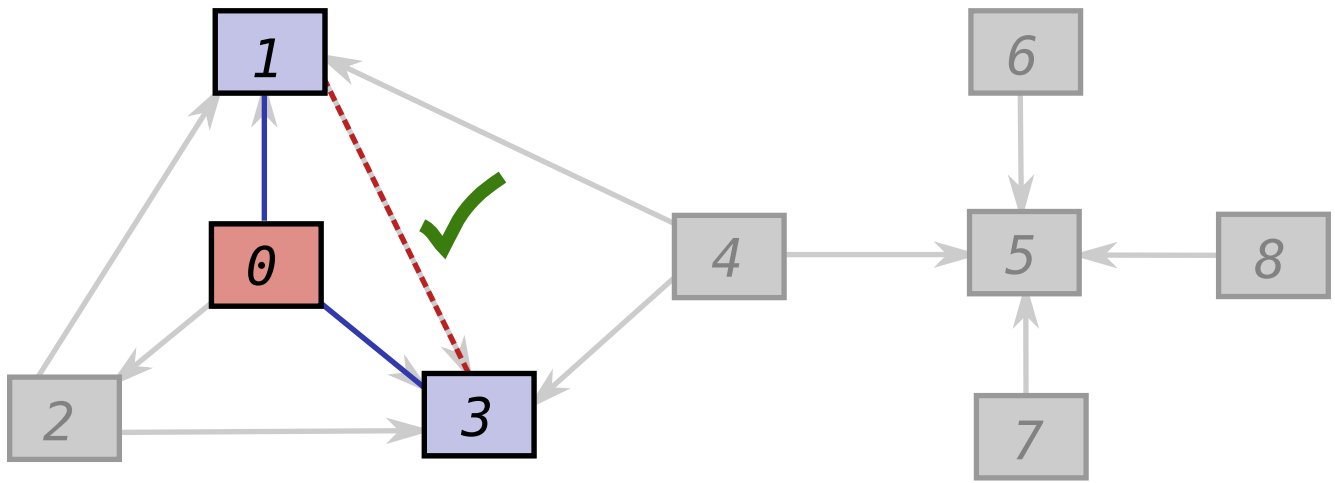
Schritt 5: Dreiecke prüfen

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



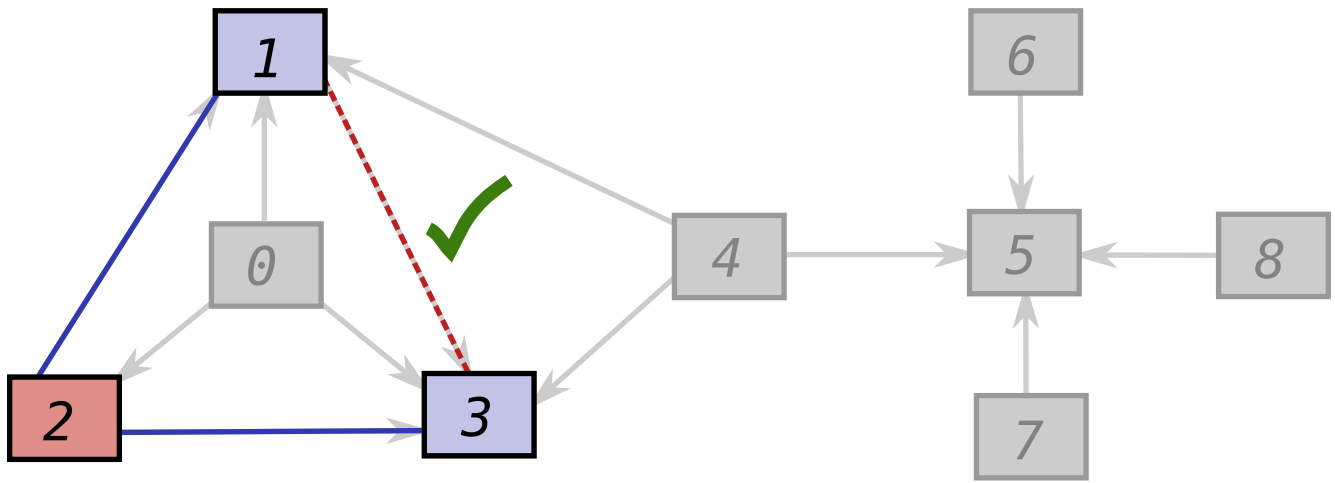
Schritt 5: Dreiecke prüfen

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



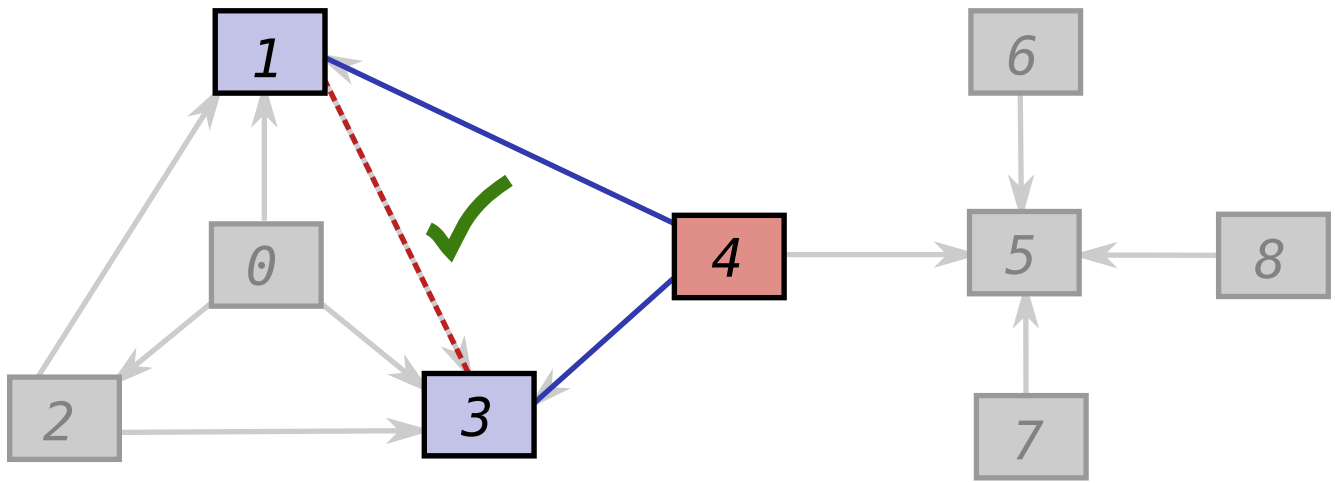
Schritt 5: Dreiecke prüfen

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



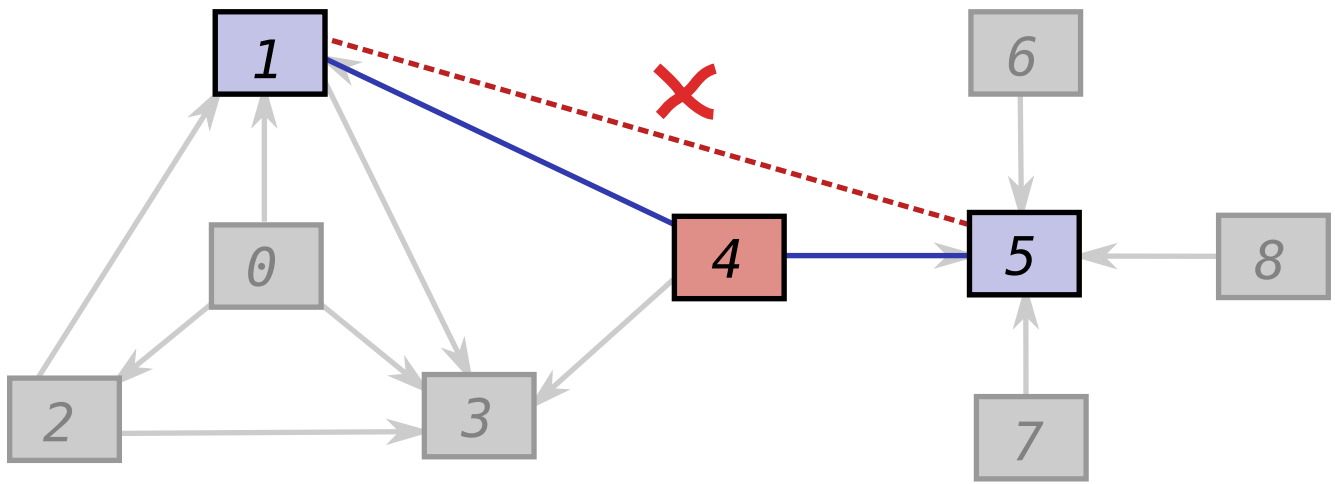
Schritt 5: Dreiecke prüfen

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



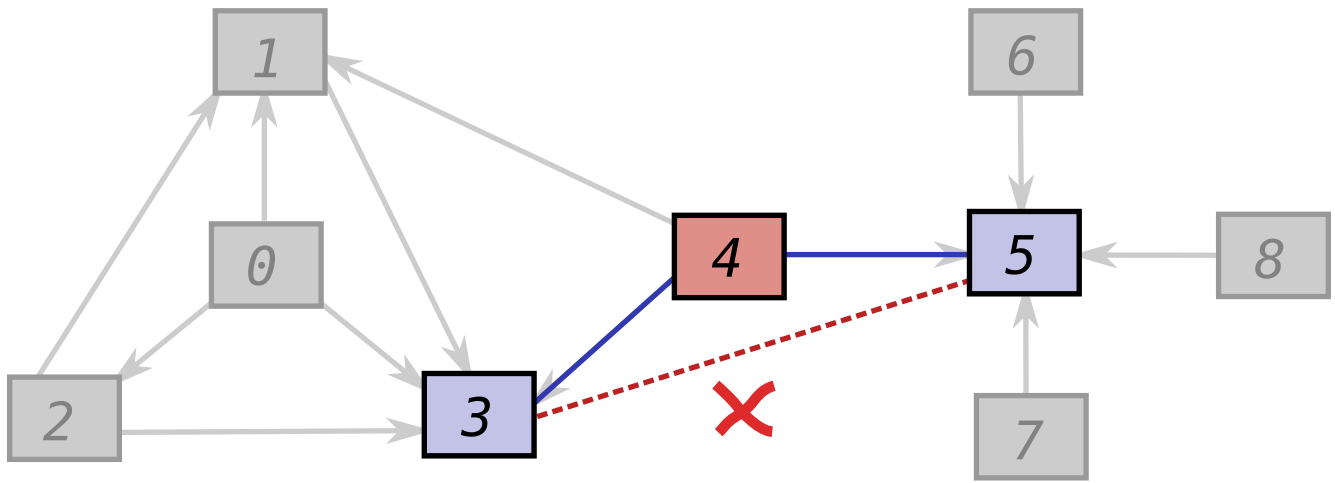
Schritt 5: Dreiecke prüfen

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



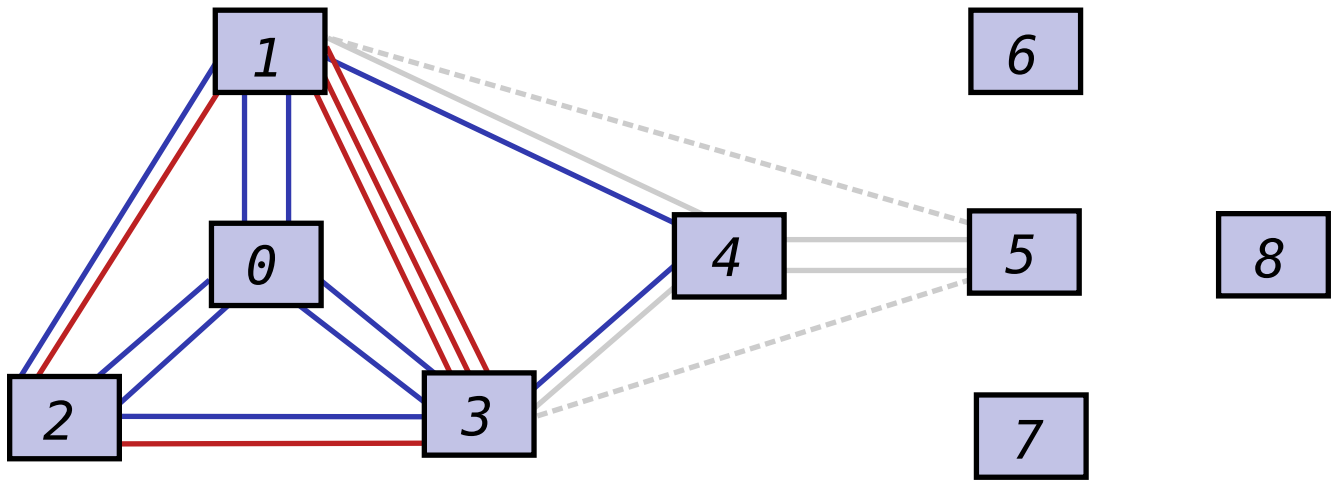
Schritt 5: Dreiecke prüfen

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



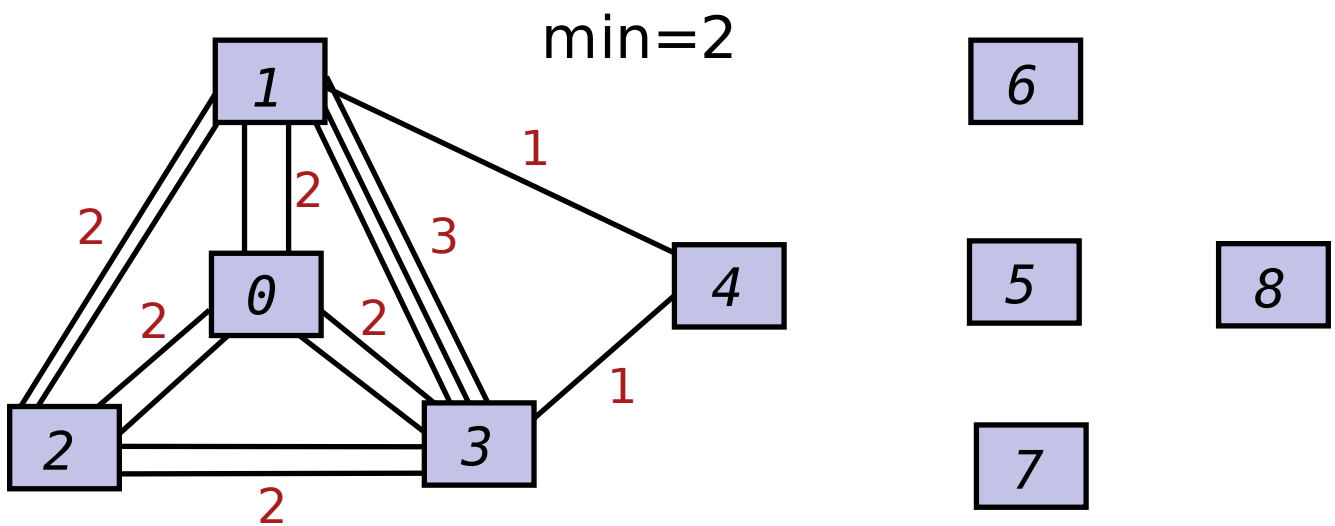
Schritt 6: Träger filtern

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



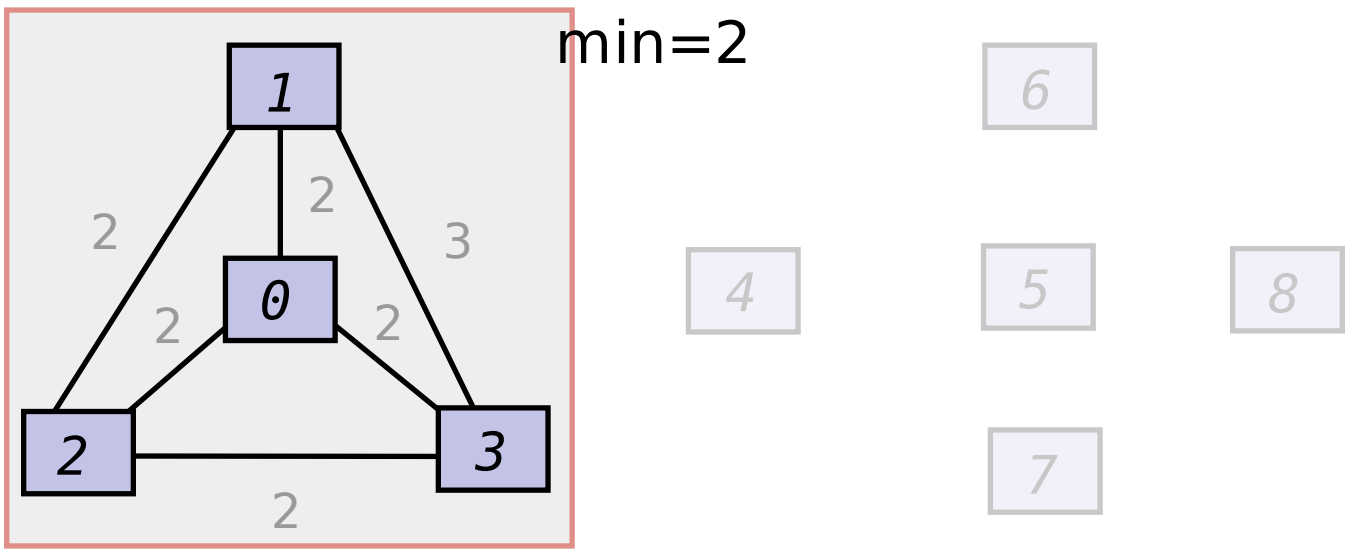
Schritt 6: Träger filtern

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



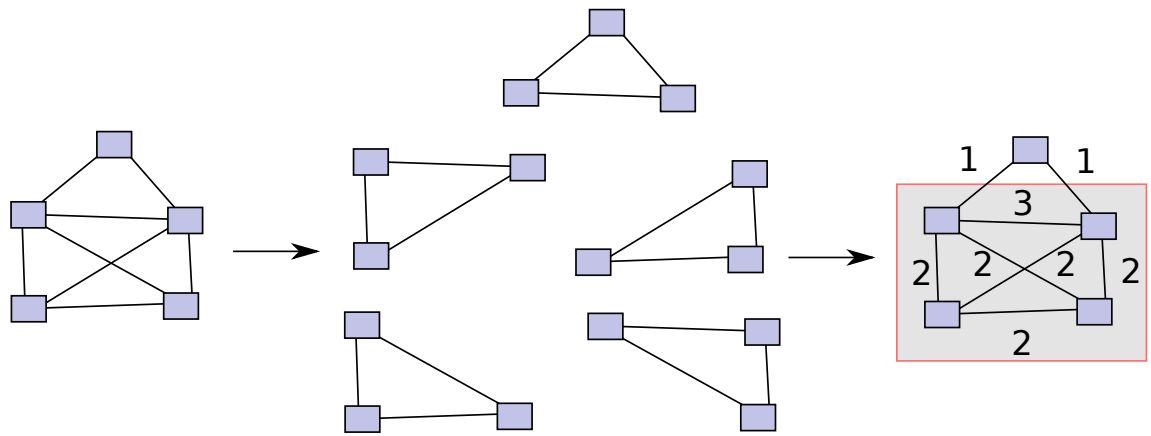
Schritt 6: Träger filtern

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



Träger und Cliquen

- 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 Cliquen mit mindestens 5 Knoten (= 3 Überlappungen) Graph nicht leer



Überblick

Apache Hadoop

- Freie Implementierung eines MapReduce-Frameworks
- Besteht aus mehreren Teilprojekten
  - **Common**: Gemeinsame Infrastruktur
  - **HDFS**: Verteiltes Dateisystem
  - **MapReduce**: MapReduce-Framework
  - **ZooKeeper**: Koordinierungs-Dienst
- Implementierung in Java
- Bindings zu anderen Sprachen verfügbar (JNI)

Apache Hadoop API

- `org.apache.hadoop.conf`

  Verwaltung der Konfiguration

- `org.apache.hadoop.fs`

  Dateisystem-Abstraktion

- `org.apache.hadoop.io`

  Serialisierung und Deserialisierung von Daten

- `org.apache.hadoop.mapreduce`

  MapReduce-Framework

- `org.apache.hadoop.util`

  Verschiedene Hilfsklassen

Überblick

Hadoop Jobs

- Klasse `org.apache.hadoop.mapreduce.Job`
- Speichert Informationen zum Ablauf eines Jobs
  - Zur Ausführung benötigte Klassen (Mapper, Reducer...)
  - Ein- und Ausgabedateien
  - Jobspezifische Konfiguration
  - Ausführungszustand
- Änderbare Einstellungen:
  - Job-Bezeichnung

    ```
    void setJobName(String name)
    ```

  - `.jar`-Datei mit benötigten Klassen

    ```
    void setJarByClass(Class cls)
    ```

Hadoop Jobs

- Änderbare Einstellungen (Fortsetzung):
  - Klasse zur Verarbeitung des Eingabeformats

```
void setInputFormatClass(Class cls)
```

  - Klasse zum Schreiben der Ausgabedaten

```
void setOutputFormatClass(Class cls)
```

  - Mapper-Klasse und deren Key/Value-Ausgabeformate

```
void setMapperClass(Class cls)
void setMapOutpuKeyClass(Class cls)
void setMapOutputValueClass(Class cls)
```

  - Reducer-Klasse und deren Key/Value-Ausgabeformate

```
void setReducerClass(Class cls)
void setOutputKeyClass(Class cls)
void setOutputValueClass(Class cls)
```

Hadoop Jobs

- **Beispiel**:

```
class MyJob extends Job {
  [...]
  public MyJob(Configuration cfg, String name) {
    super(cfg, name);

    setJarByClass(getClass());

    setInputFormatClass(EdgeInputFormat.class);
    setOutputFormatClass(RecordOutputFormat.class);

    setMapperClass(Map.class);
    setMapOutputKeyClass(IntWritable.class);
    setMapOutputValueClass(LongWritable.class);

    setReducerClass(Reduce.class);
    setOutputKeyClass(NullWritable.class);
    setOutputValueClass(IntWritable.class);
  }
```

Hadoop Jobs

- **Beispiel** (Fortsetzung):

```
// class MyJob extends Job {
  public static class Map
      extends Mapper<NullWritable, Edge,
                     IntWritable, LongWritable> {
    @Override
    public void map(NullWritable k, Edge v, Context ctx) {
      [...]
    }
  }
  public static class Reduce
      extends Reducer<IntWritable, LongWritable,
                      NullWritable, IntWritable> {
    @Override
    public void reduce(IntWritable k,
                       Iterable<LongWritable> v, Context ctx) {
      [...]
    }
  }
}
```

Weitere Job-Einstellungen

- Konfigurationseinstellungen für Job abfragen/ändern

```
Configuration getConfiguration()
```

- Manipulation von Zählern:

```
Counters getCounters()
```

- Festlegen der Ein- und Ausgabedateien für Job:

```
static FileInputFormat.setInputPaths(Job job, Path path);
```

```
static FileOutputFormat.setOutputPath(Job job, Path path);
```

- Erfolgt indirekt über statische Methoden der verwendeten Klasse zur Formatierung der Ein- bzw. Ausgabe!

Job-Ausführung

- Starten des Jobs (asynchron!):

```
void submit()
```

- Nach Start können Job-Parameter nicht mehr verändert werden!
- Statusabfrage:

```
boolean isComplete()
```

```
boolean isSuccessful()
```

- Warten auf Ende des Jobs:

```
boolean waitForCompletion(boolean verbose)
```

Hadoop-Tools

- Java-Klasse `org.apache.hadoop.util.Tool`
  - Interface mit Einstiegspunkt für eigene Hadoop-Anwendungen

```
int run(String[] args)
```

- Java-Klasse `org.apache.hadoop.util.ToolRunner`
  - Initialisierung des Frameworks
  - Verarbeitung der Kommandozeile
  - Starten des angegebenen Tools

```
static int run(Configuration cfg, Tool t, String[] args)
```

Hadoop-Tools

- **Beispiel:**

```
public class MyApp extends Configured implements Tool {
  public int run(String[] args) throws Exception {
    System.out.println("Hello " + args[0] + "!");
    System.out.println(getConf().get("mycfg.setting"));
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(),
                             new MyApp(), args);
    System.exit(res);
  }
}
```

- **Aufruf**:

```
$ hadoop jar myapp.jar -D mycfg.setting=hi World
Hello World!
hi
```

Überblick

JAR-Dateien erstellen

- Hadoop-Programme müssen als `.jar`-Dateien vorliegen
- Aufruf:

```
$ hadoop jar myprogram.jar <parameter>
```

- `.jar`-Dateien werden im ZIP-Format gespeichert und enthalten
  - Java-Klassendateien (`.class`)
  - Meta-Informationen (`META-INF/MANIFEST.MF`)
- Manifest-Datei ermöglicht
  - Definieren von Abhängigkeiten und Classpath
  - Versionierung
  - Festlegen der zu verwendenden `main()`-Methode

JAR-Dateien erstellen

- **Beispiel** für Manifest:

```
Manifest-Version: 1.0
Main-Class: mw.MyApp
Class-Path: auxlib.jar
```

- Direktes starten der `.jar`-Datei (`Main-Class` muss definiert sein):

```
$ java -jar myapp.jar
```

- Erstellen eines JARs auf der Kommandozeile (siehe `man jar`):

```
$ jar cfe myapp.jar mw.MyApp mw/
```

JAR-Dateien mit Ant erstellen

- `ant` ist ein Programm zum automatischen Bauen von Programmen (vgl. `make`)
- Zu erstellenden Objekte mittels XML-Datei (`build.xml`) definiert
- Enthält Modul zum bauen von `.jar`-Dateien:

```
<?xml version="1.0" encoding="UTF-8" ?>
<project default="makejar" name="jar example">
  <target name="makejar">
    <jar destfile="myapp.jar">
      <manifest>
        <attribute name="Main-Class" value="mw.MyApp"/>
      </manifest>
      <fileset dir="bin/">
        <include name="mw/**"/>
      </fileset>
    </jar>
  </target>
</project>
```

- Erzeugen der `.jar-Datei` durch Aufruf von `ant`

Überblick

Ein-/Ausgabe (`org.apache.hadoop.io`)

- Objekte zur Bearbeitung durch MapReduce müssen in Dateien gespeichert werden können
- Zur Speicherung von Objekten muss bekannt sein, **welche** Daten **wie** abgelegt werden müssen
- Schnittstelle `WritableComparable` beschreibt Klassen, die
  - **serialisierbar** und **deserialisierbar** sind
  - untereinander **vergleichbar** sind
- Hadoop liefert Implementierung für elementare Datentypen mit:
  - `int` $\longrightarrow$ `IntWritable`
  - `long` $\longrightarrow$ `LongWritable`
  - ...
  - Platzhalter: `NullWritable`

Schnittstelle `WritableComparable`

- **Serialisierung**: Schreibt alle zur Rekonstruktion des Objekts in den Datenstrom `out`

```
public void write(DataOutput out)
```

- **Deserialisierung**: Rekonstruiert Zustand des Objekts aus Datenstrom `in`

```
public void readFields(DataInput in)
```

- **Vergleichs-Operator**: Vergleicht `this` mit `x` (siehe Java-Interface `Comparable`)

```
public int compareTo(WritableComparable x)
```

- **Hashing**: Ermittelt Hash-Wert über Daten

```
public int hashCode()
```

Schnittstelle `WritableComparable`

- **Beispiel**:

```
public class MyData implements WritableComparable {
  protected int node;
  protected long weight;

  public void write(DataOutput out)
                        throws IOException {
    out.writeInt(node);
    out.writeLong(weight);
  }
  public void readFields(DataInput in)
                        throws IOException {
    node = in.readInt();
    weight = in.readLong();
  }
  public int compareTo(MyData x) {
    return (weight < x.weight) ? -1 :
           ((weight == x.weight) ? 0 : 1);
  }
  [...]
```

Schnittstelle `WritableComparable`

- `hashCode()`-Methode von Java garantiert **keine** konsistenten Hash-Werte bei unterschiedlichen Objekten, trotz gleichem **Inhalt**
- Wird jedoch von `HashPartitionier` zum Partitionieren nach Schlüssel für die Reduce-Phase verwendet
- → Eigene Implementierung basierend auf Inhalt notwendig
- **Beispiel** (Fortsetzung):

```
  [...]
  public boolean equals(Object x) {
    if (! (o instanceof MyData)) return false;
    MyData z = (MyData)x;
    return (z.node == node) && (z.weight == weight);
  }

  public int hashCode() {
    return (node * 2873) ^ (int)(weight * 3782);
  }
}
```

Ein-/Ausgabe

- Einlesen der Daten über `InputFormat`
- Ausgabe über `OutputFormat`
- Hadoop enthält bereits Klassen für verschiedene Ein-/Ausgabeformate:
  - `TextInputFormat`: Einlesen aus Textdateien
  - `SequenceFileInputFormat`: Hadoop-internes Binärformat
  - ...`OutputFormat` entsprechend
- Für Übungsaufgabe vorgegeben: Lesen und schreiben von Binärdaten mit konstanter Satzlänge
  - `mw.cliquefind.io.RecordInputFormat`
  - `mw.cliquefind.io.RecordOutputFormat`
- InputFormat benötigt je abgeleiteten Typ eigene Spezialisierung, da beim Einlesen neue Instanzen erzeugt werden (via Reflection)

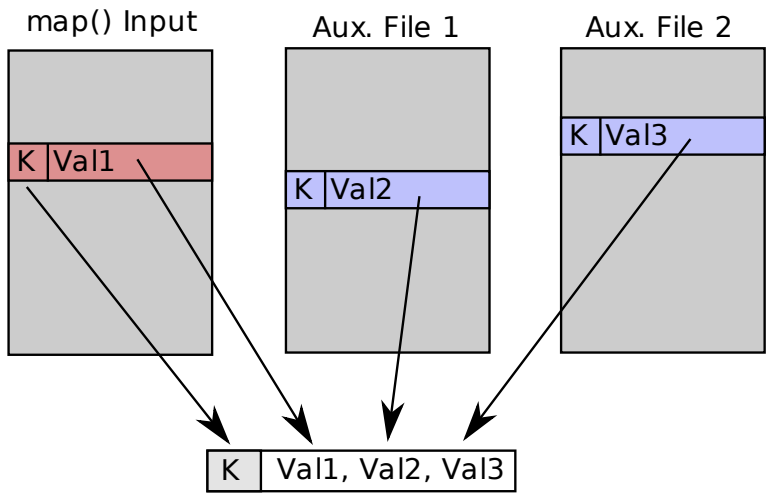
Überblick

Zusammenführen mit MapReduce

- 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

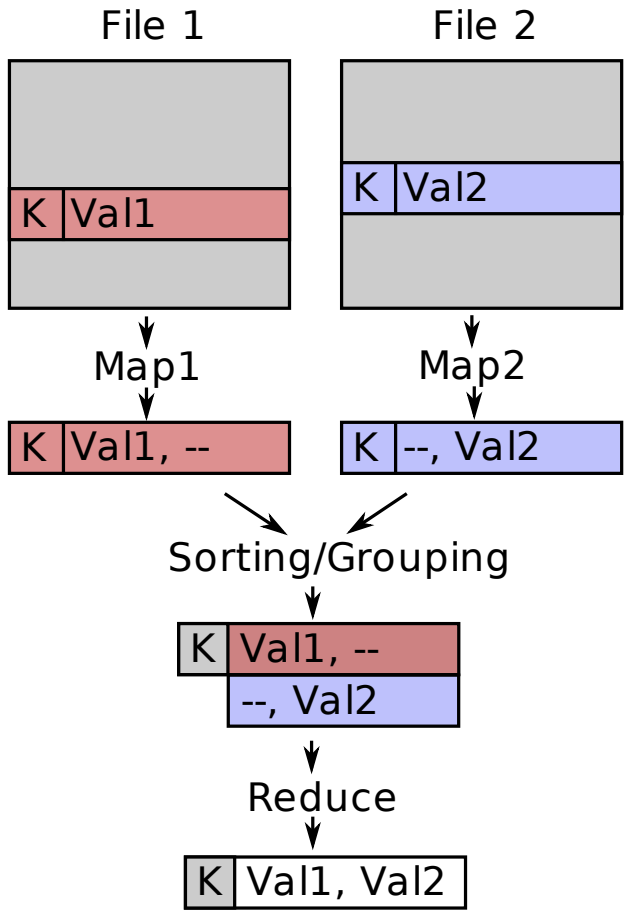
- Zusammenfassen von Daten während des Map-Vorgangs
- **Mapper** hält alle notwendigen Eingabedateien offen
- Durch Framework zugewiesener Datenbereich gibt zu suchende Schlüssel an
  - Suche des jeweiligen Schlüssels in anderen Dateien
- Voraussetzung für effiziente Ausführung: Daten bereits nach Schlüssel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



Reduce-side Join

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

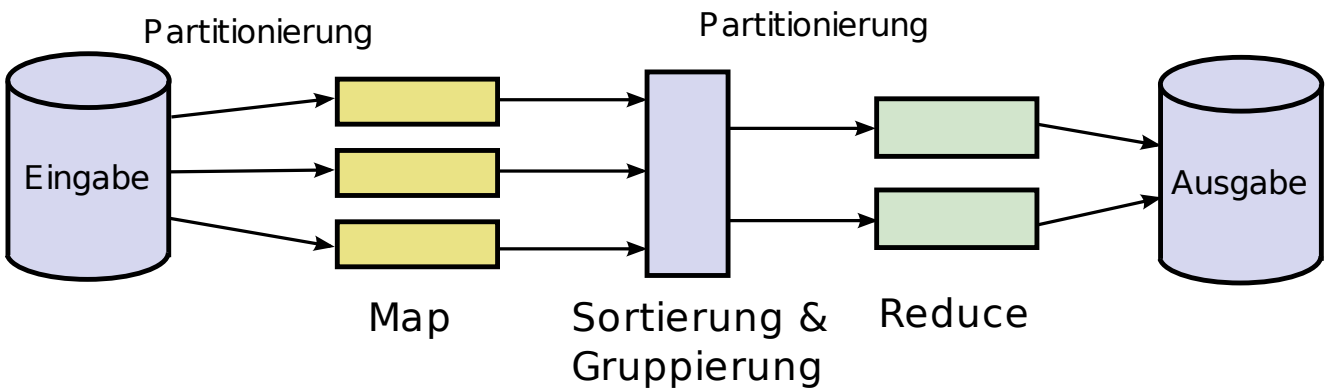
Reduce-side Join



Überblick

Sortierung, Gruppierung und Partitionierung

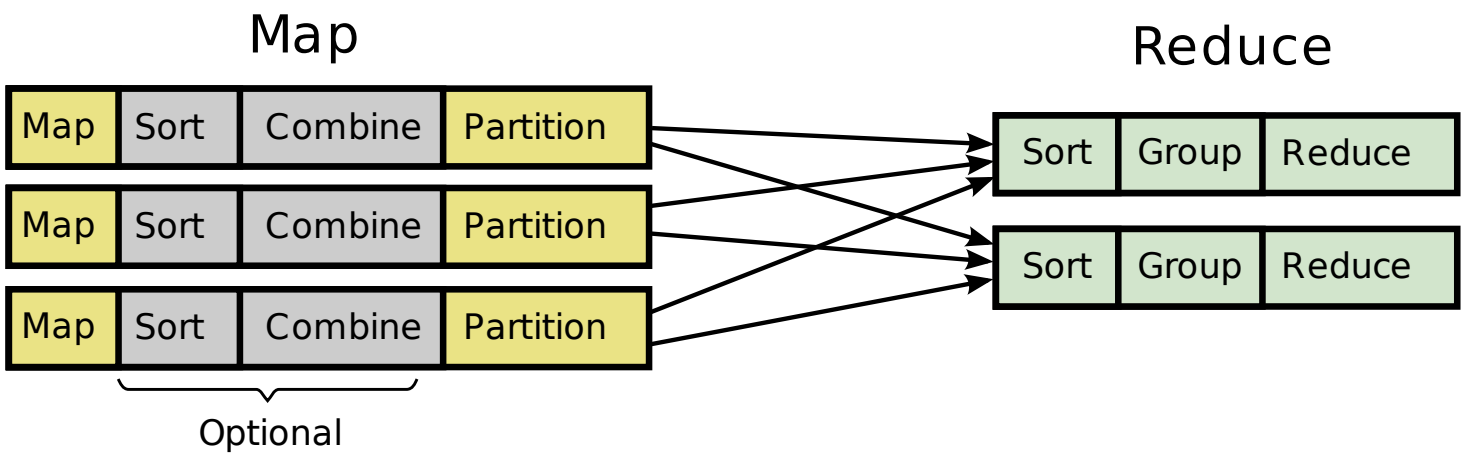
- MapReduce-Grundkonzept:



- **Problem**: Sortierung nicht parallelisierbar
- Datenabhängige Partitionierung erlaubt paralleles Sortieren

Sortierung, Gruppierung und Partitionierung

- **Sortierung**: Reihenfolge der Schlüssel/Wert-Paare
- **Gruppierung**: Schlüssel, die beim Reduce zusammengefasst werden sollen
- **Partitionierung**: Verteilung der Schlüssel auf Reducer mittels Hash-Funktion



Sortierung, Gruppierung und Partitionierung

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

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

`RawComparator`-Implementierung

- Klasse `WritableComparator` ermöglich 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);
  }
}
```

`Partitioner`-Implementierung

- Standard-Implementierung `HashPartitioner` verwendet `hashCode()`-Methode von `Object` zur Partitionierung
- **Achtung**: 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;
  }
}
```

Combiner zur Beschleunigung

- 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

  $\longrightarrow$ Reduktion der zu sortierenden Datenmenge
- Implementierung in Hadoop mittels **Combiner**

Combiner zur Beschleunigung

- Schnittstelle des Combiners entsprecht 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

Job-Konfiguration

- Festlegen von `Combiner`, `Sorting`, `Grouping` und `Partitioner` bei Job-Konfiguration
- **Beispiel**:

```
// Partitoner
setPartitionerClass(Partition.class);

// Grouping
setGroupingComparatorClass(Grouping.class);

// Sorting
setSortComparatorClass(Sorting.class);

// Combiner
setCombinerClass(Combine.class);
```

Hadoop Counter

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

Hadoop Counter

- **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());
  }
}
```

Überblick

Ausführung von MapReduce-Jobs

- Hadoop unterstützt zwei verschiedene Ausführungsmodi
  - Lokal
  - Verteilt
- Standardmäßig lokale, sofortige Ausführung
- Verteilte Ausführung erfordert
  - Dienste für Job-Verteilung und Ausführung
  - Hadoop Filesystem (HDFS)
  - Konfiguration (`/etc/hadoop/conf`)

Hadoop Dienste

- **Jobtracker**
  - Verwaltet und verteilt MapReduce-Jobs
  - Nur eine zentrale Instanz
- **Tasktracker**
  - Zuständig für die Ausführung von Jobtracker zugewiesener Jobs
  - Läuft auf jedem Knoten, der Jobs bearbeiten soll
- **Namenode**
  - Verwaltet Dateien und deren Speicherorte im verteilten Dateisystem HDFS
  - Eine zentrale, primäre Instanz
  - Sekundärer Namenode ermöglicht schnellere Wiederherstellung
- **Datanode**
  - Zuständig für die Speicherung von Datenblöcken des HDFS
  - Läuft auf jedem Knoten, der Speicherplatz zur Verfügung stellen soll

Hadoop Dienste

- Kommunikation der Dienste untereinander erfolgt per HTTP
- Web-Server liefert HTML-Seiten zur Statusabfrage
  - Status Jobtracker:

    `http://hostname:50030/`
  - Status Namenode/HDFS:

    `http://hostname:50070/`
- Zugriff kann derzeit nicht beschränkt werden
- Betrieb daher **nur** in privatem Netz oder hinter entsprechender Firewall

Konfiguration

- Setzen von Konfigurationsparametern
  - Auf der Kommandozeile:

```
$ hadoop <command> -D dfs.data.dir=/mnt/hadoop
```

  - XML-Dateien:

```
<configuration>
  <property>
    <name>dfs.data.dir</name>
    <value>/mnt/hadoop</value>
  </property>
</configuration>
```

- Verfügbare Konfigurationsdateien
  - `core-site.xml`: Generelle Einstellungen
  - `hdfs-site.xml`: HDFS-Spezifische Einstellungen
  - `mapred-site.xml`: MapReduce Einstellungen

Konfiguration

- Typische Konfiguration:
  - **Ein** Knoten für Jobtracker und Namenode
  - Restliche Knoten: Gleichzeitig Tasktracker und Datanode
  - Erlaubt Nutzung lokaler Datenspeicherung
- Namenode (`core-site.xml`)

```
<name>fs.default.name</name> <value>hdfs://131.188.42.111/</value>
```

- Jobtracker (`mapred-site.xml`)

```
<name>mapreduce.jobtracker.address</name> <value>131.188.42.111</value>
```

- Liste von Tasktracker/Datanode-Adressen in `conf/slaves` (kein XML!)

```
131.188.42.112
131.188.42.113
[...]
```

HDFS-Konfiguration

- `hadoop.tmp.dir`: Basisverzeichnis zur Speicherung lokaler Daten (`core-site.xml`)
- `dfs.namenode.name.dir`: Ort zur lokalen Speicherung von Metadaten auf dem Namenode (`hdfs-site.xml`)
  - Standardwert: `${hadoop.tmp.dir}/dfs/name`
  - Muss vor Benutzung „formatiert“ werden

```
$ hadoop namenode -format
```

- `dfs.datanode.data.dir` zur lokalen Speicherung von HDFS Datenblöcken (`hdfs-site.xml`)
  - Standardwert: `${hadoop.tmp.dir}/dfs/data`
- `dfs.replication`: Gibt an, wie oft Datenblöcke redundant gespeichert werden sollen
  - Standardwert: `3`

MapReduce-Konfiguration (`mapred-site.xml`)

- `mapred.tasktracker.map.tasks.maximum`,
  `mapred.tasktracker.reduce.tasks.maximum`:

  Maximale Anzahl parallel ablaufender Map/Reduce-Tasks **je Knoten**

- `mapred.map.tasks`, `mapred.reduce.tasks`:

  Hinweis an Partitionierer über die gewünschte Zahl von Teilen für Mapper/Reducer

- `mapred.min.split.size`, `mapred.max.split.size`:

  Grenzen für Partitionierer bezüglich Größe der pro Mapper/Reducer verarbeiteten Teile

Konfiguration aus API-Sicht

- Java-API: `org.apache.hadoop.conf.Configuration`
- Einstellungen können zur Laufzeit abgefragt und verändert werden:

```
public void changeBlocksize(Configuration mycfg) {
  long sz = mycfg.getLong("mapred.min.split.size");
  mycfg.setLong("mapred.min.split.size", sz * 10);
}
```

- Beliebige eigene Definitionen möglich:

```
Configuration mycfg = new Configuration(getConf());
int ni = mycfg.getInt("mytool.num_iterations", 100);
```

```
$ hadoop jar mytool.jar -D mytool.num_iterations=500
```

Überblick

Dateisystem-Abstraktion

- Hadoop abstrahiert über verschiendene „Dateisysteme“
- URI-Schema zur Unterscheidung
- Unterstützt werden unter anderem:
  - Lokales Dateisystem (`file://...`)
  - HDFS (`hdfs://...`)
  - Amazon S3 / Walrus (`s3://...`, `s3n://...`)
  - HTTP (`http://...`)
- Zugriff über einheitliche Java-API
- Ein- und Ausgaben von MapReduce-Jobs können beliebige Dateisysteme direkt verwenden

Dateisystem API (`org.apache.hadoop.fs`)

- Repräsentation von Pfaden mittels `Path`
- Operationen auf Objekte im Dateisystem mittels `FileSystem`
- Komplexere Operationen über statische Methoden in `FileUtil`
- **Beispiel**: Löschen einer Datei

```
public void delete(Configuration cfg, String name) {
  try {
    Path p = new Path(name);
    FileSystem fs = p.getFileSystem(cfg);

    fs.delete(p, true); //recursive delete

  } catch(Exception e) {
    System.err.println("Delete failed for " + name);
  }
}
```

Dateisystemzugriff mit FS Shell

- Hadoop bringt Werkzeug zum Zugriff auf Dateisysteme mit
- Aufruf mit:

```
$ hadoop fs <command> ...
```

- Mögliche Unterbefehle:
  - `-ls <path>`: Auflisten von Dateien
  - `-cp <src> <dst>`: Kopieren von Dateien
  - `-getmerge <src> <dest>`: Zusammenfügen und ins lokale Dateisystem kopieren
  - `-cat <src> <...>`: Zusammenfügen und auf Standardausgabe ausgeben
  - `-rmr <path>`: Rekursives löschen
- Siehe **Hadoop FS Shell Guide**

Überblick

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

Programmieren mit Hadoop
- Überblick
- Jobs und Tools
- JAR-Dateien erstellen
- Ein- und Ausgabe, Serialisierung
- Zusammenführen von Daten
- MapReduce Erweiterungen
- Verteilte Ausführung
- Dateisysteme in Hadoop

Aufgabe 4

Hadoop im CIP-Pool lokal starten

- Hadoop benötigt zur Ausführung die Umgebungsvariable `JAVA_HOME` zur Angabe der zu verwendenden Java-Runtime
- Im CIP-Pool liegt diese unter `/local/java`
- Setzen der Umgebungsvariable (`bash`-Shell):

```
$ export JAVA_HOME=/local/java
```

Visualisierung der Graphen

- Zur Visualisierung der Graphen ist das Program `tulip` im CIP installiert
- Die Binärdateien mit den Kanten können mit dem Werkzeug `/proj/i4mw/pub/aufgabe4/edge2tlp` in das Tulip-Format konvertiert werden
- **Beispiel**:

```
$ cd /proj/i4mw/pub/aufgabe4
$ ./edge2tlp small.bin ~/small.tlp
$ tulip ~/small.tlp
```

Start der verteilten Hadoop-Instanzen

- Gestarteten Hadoop-Dienste sind ohne besondere Vorkehrungen öffentlich erreichbar
- Vorgefertigtes Image und dazugehöriges Script zum Starten der notwendigen Dienste
- Benschränkt Zugriffe automatisch auf den Rechner, welche die Dienste gestartet hat
- Anpassung der Konfigurationsdateien durch Kopieren der Vorgabedateien in eigenes Verzeichnis
- **Beispiel**:

```
$ cp -a /proj/i4mw/pub/aufgabe4/defaultcfg mycfg
$ vi mycfg/core-site.xml
$ /proj/i4mw/pub/aufgabe4/conf-hadoop mycfg/     \
    131.188.42.1 131.188.42.2 [...]
```

Verwendung der verteilten Hadoop-Instanzen

- Login auf den VMs als Benutzer `root` mit SSH-Key möglich
- SSH-Key erst ins eigene Home kopieren und `chmod 600` ausführen
- Kopieren notwendiger `.jar`- und Datendateien
- Hadoop-Tools müssen als Benutzer `hadoop` gestartet werden, gegenüber lokaler Ausführung ansonsten keine Besonderheiten

  → Benutzerwechsel mit `su`
- **Beispiel**:

```
$ scp -i root-ssh-key myapp.jar root@master:/tmp/
$ ssh -i root-ssh-key root@master
$ su - hadoop
$ cd /tmp
$ hadoop jar myapp.jar [...]
```

Literatur

- [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`