# Graph Analytics mit der Apache Spark GraphFrame API - Primer

Im [ersten Beitrag](https://www.btelligent.com/blog/wofuer-sind-graph-analytics-werkzeuge-gut/) über Graph Analytics wurden einige Use Cases beschrieben. Im zweiten Teil möchte ich zeigen, wie Graphen in Apache Spark unter Verwendung der GraphFrame API verarbeitet werden, welche „eingebauten“ Algorithmen genutzt werden können und wo die Beschränkungen liegen.

Graphen bestehen aus einer Menge von Knoten und einer Menge von Kanten, die Verbindungen zwischen Knoten darstellen. Entsprechend werden Graphen in Spark aufgebaut: Ein GraphFrame-Objekt wird aus zwei DataFrame-Objekten zusammengesetzt (eine Tabelle mit den Knoten-Informationen sowie eine Tabelle mit den Kanten-Informationen).

## Vorbereitungen

Für die hier gezeigten Beispiele benutze ich pyspark. Um die GraphFrame API verwenden zu können, muss ich beim Starten von pyspark das entsprechende Package laden (Eine Liste der Releases findet man [hier](https://spark-packages.org/package/graphframes/graphframes)).

pyspark --packages graphframes:graphframes:0.7.0-spark2.4-s\_2.11

In pyspark muss dann noch die Klasse importiert werden:

from graphframes import GraphFrame

Als Beispiel möchte ich einen gerichteten Graphen aufbauen, der aus einer Auswahl von weltweiten Städten (Knoten) und den für jede Stadt die Verbindungen zu den nach der Luftlinie jeweils sieben nächstgelegenen Städten (Kanten) besteht. Als Grundlage verwende ich den öffentlich verfügbaren Datensatz worldcities.csv ([Quelle](https://simplemaps.com/data/world-cities)), der neben den Städtenamen u.a. auch die geographischen Koordinaten und den ISO-Ländercode enthält.

citiesDF.show(10)

+-----------+--------+---------+----+

| id|latitude|longitude|iso2|

+-----------+--------+---------+----+

| Malishevë| 42.4822| 20.7458| XK|

| Prizren| 42.2139| 20.7397| XK|

|Zubin Potok| 42.9144| 20.6897| XK|

| Kamenicë| 42.5781| 21.5803| XK|

| Viti| 42.3214| 21.3583| XK|

| Shtërpcë| 42.2394| 21.0272| XK|

| Shtime| 42.4331| 21.0397| XK|

| Vushtrri| 42.8231| 20.9675| XK|

| Dragash| 42.0265| 20.6533| XK|

| Podujevë| 42.9111| 21.1899| XK|

+-----------+--------+---------+----+

only showing top 10 rows

Durch einen Self Cross Join, der paarweisen Berechnung von Distanzen zwischen zwei Städten und einem Filter auf die jeweils sieben nächstgelegenen Städte bekomme ich folgenden DataFrame cityConnectsDF, der die Kantenmenge des Graphen bilden wird (hier eingeschränkt auf Berlin):

cityConnectsDF.where("src = 'Berlin' OR dst = 'Berlin'").show()

+-------------------+-------------------+------------+

| src| dst|cityDistance|

+-------------------+-------------------+------------+

| Magdeburg| Berlin| 129|

| Potsdam| Berlin| 26|

| Szczecin| Berlin| 125|

| Berlin| Potsdam| 26|

| Berlin| Cottbus| 105|

| Berlin| Szczecin| 125|

| Berlin|Gorzów Wielkopolski| 126|

| Berlin| Magdeburg| 129|

| Berlin| Leipzig| 149|

| Berlin| Zielona Góra| 157|

| Cottbus| Berlin| 105|

| Zielona Góra| Berlin| 157|

|Gorzów Wielkopolski| Berlin| 126|

+-------------------+-------------------+------------+

Als Schlüssel wird der Städtename verwendet. Damit bei der Erzeugung des GrapFrames die Schlüsselspalten erkannt werden, müssen diese in den zugunde liegenden DataFrames benannt werden: Im DataFrame citiesDF (Knoten) heißt die Spalte id, im DataFrame cityConnectsDF (Kanten) heissen die Spalten src (Quelle, „source“) bzw. dst (Ziel, „destination“). Diese Spalten werden dann verwendet, um die Verbindung zwischen Knoten und Kanten herzustellen.

## Den GraphFrame erstellen, modifizieren und abfragen

Der GraphFrame-Konstruktor hat nur zwei Parameter: Einen DataFrame für die Knoten und einen für die Kanten. In unserem Beispiel erzeugen wir den GraphFrame also so:

cityGraph = GraphFrame(citiesDF, cityConnectsDF)

Der Graph ist ziemlich gross, was wir einfach mit folgender Abfrage feststellen können. Dabei werden die Methoden vertices() und edges() verwendet, die aus einem GraphFrame den Knoten- bzw. den Kanten-DataFrame extrahieren.

print("The cityGraph has "\

+ str(cityGraph.vertices.count()) + " nodes and "\

+ str(cityGraph.edges.count()) + " edges.")

The cityGraph has 5063 nodes and 35473 edges.

Angenommen, wir interessieren uns nur für Städte aus Deutschland, Österreich und der Schweiz. Dann können wir einfach einen „Subgraphen“ des cityGraph durch Einschränkung auf die entsprechenden Knoten bilden. Die Methode filterVertices() löscht automatisch alle Kanten, die einen gelöschten Knoten treffen.

citySubgraph = cityGraph.filterVertices("iso2 IN ('DE','AT','CH')")

print("The citySubgraph has "\

+ str(citySubgraph.vertices.count()) + " nodes and "\

+ str(citySubgraph.edges.count()) + " edges.")

The citySubgraph has 88 nodes and 496 edges.

Eine wichtige Information für einen Knoten ist der Grad („Degree“), das ist die Anzahl der Kanten, die den Knoten treffen. Bei gerichteten Graphen unterscheidet man noch in Eingangsgrad („In-Degree“) und Ausgangsgrad („Out-Degree“).

Angenommen, in unserem Städte-Beispiel können wir von einer Stadt nur in eine andere reisen, wenn sie durch eine Kante (in der richtigen Richtung) verbunden sind. Dann gibt der Ausgangsgrad für eine Stadt an, in wie viele Städte ich von dort aus direkt reisen kann.

citySubgraph.outDegrees\

.where("id IN ('Frankfurt','Stralsund')").show()

+---------+---------+

| id|outDegree|

+---------+---------+

|Stralsund| 2|

|Frankfurt| 7|

+---------+---------+

Von den ursprünglich sieben nächsten Nachbarn sind durch die Einschränkung auf die deutschsprachigen Länder für Stralsund fünf Nachbarstädte weggefallen, weil sie in Polen liegen. Frankfurt am Main dagegen hat alle sieben nächsten Nachbarn in Deutschland.

## Graphen-Algorithmen

Die GraphFrame API wurde 2016 vorgestellt und ist daher eine relativ neue Komponente von Apache Spark. Wie wir gesehen haben, basieren GraphFrames auf DataFrames. In Apache Spark 1.x gab es schon die GraphX API, die auf RDDs basiert. Ähnlich wie DataFrame-Operationen Spark-intern auf RDD-Operationen zurückgeführt werden, werden auch GraphFrame-Operationen auf GraphX-Operationen zurückgeführt. Dies trifft insbesondere auf die verfügbaren Algorithmen zu.

Derzeit ist die Liste der Algorithmen für GraphFrames noch eingeschränkt. Insbesondere gibt es noch keine Visualisierungsalgorithmen. Dies wird sich – hoffentlich – in der Zukunft ändern, entweder durch Weiterentwicklung der API oder durch 3rd-Party Entwicklungen.

„Out-of-the-box” stehen für GraphFrames u.a. folgende Algorithmen zur Analyse von Graphen zu Verfügung (die vollständige Liste sowie die Dokumentation gibt es [hier](http://graphframes.github.io/graphframes/docs/_site/api/scala/index.html#org.graphframes.GraphFrame)):

* Breadth-first search (BFS): Bestimmung kürzester Wege zwischen zwei Knoten
* Verbundene Komponenten: Bestimmung der zusammenhängenden Teile eines Graphen und Zuordnung der Knoten zu diesen Teilen
* Label Propagation: Auffinden von Communities in Graphen, d.h. von Knotenmengen, deren Mitglieder viel untereinander kommunizieren und wenig mit Mitgliedern anderer Communities, vergleichbar mit der Clusteranalyse
* PageRank: Bewertung der „Wichtigkeit“ von Knoten im Netzwerk. Der Algorithmus wurde ursprünglich von Larry Page und Sergei Brin entwickelt und für Google eingesetzt.
* Kürzeste Wege: Berechnung kürzester Wege von allen Knoten zu einer Menge vorgegebener Knoten
* Auffinden struktureller Muster in Graphen („Motif finding“)

An unserem Städte-Beispiel soll nun der BFS-Algorithmus gezeigt werden. Ich möchte im Subgraphen (Deutschland, Österreich, Schweiz) die kürzeste Reise von Frankfurt nach Basel finden. „Kurz“ bezieht sich dabei zunächst auf die benötigte Anzahl der Schritte (von Stadt zu Stadt). Die Syntax dafür ist einfach:

paths = citySubgraph.bfs("id = 'Frankfurt'", "id = 'Basel'")

paths.printSchema()

root

|-- from: struct (nullable = false)

| |-- id: string (nullable = true)

| |-- latitude: float (nullable = true)

| |-- longitude: float (nullable = true)

| |-- iso2: string (nullable = true)

|-- e0: struct (nullable = false)

| |-- src: string (nullable = true)

| |-- dst: string (nullable = true)

| |-- cityDistance: integer (nullable = true)

|-- v1: struct (nullable = false)

| |-- id: string (nullable = true)

| |-- latitude: float (nullable = true)

| |-- longitude: float (nullable = true)

| |-- iso2: string (nullable = true)

|-- e1: struct (nullable = false)

| |-- src: string (nullable = true)

| |-- dst: string (nullable = true)

| |-- cityDistance: integer (nullable = true)

|-- v2: struct (nullable = false)

| |-- id: string (nullable = true)

| |-- latitude: float (nullable = true)

| |-- longitude: float (nullable = true)

| |-- iso2: string (nullable = true)

|-- e2: struct (nullable = false)

| |-- src: string (nullable = true)

| |-- dst: string (nullable = true)

| |-- cityDistance: integer (nullable = true)

|-- v3: struct (nullable = false)

| |-- id: string (nullable = true)

| |-- latitude: float (nullable = true)

| |-- longitude: float (nullable = true)

| |-- iso2: string (nullable = true)

|-- e3: struct (nullable = false)

| |-- src: string (nullable = true)

| |-- dst: string (nullable = true)

| |-- cityDistance: integer (nullable = true)

|-- to: struct (nullable = false)

| |-- id: string (nullable = true)

| |-- latitude: float (nullable = true)

| |-- longitude: float (nullable = true)

| |-- iso2: string (nullable = true)

Das Ergebnis ist ein DataFrame, der je nach Länge der gefundenen Wege eine unterschiedliche Struktur hat. In unserem Fall sind vier Schritte nötig, und der Data Frame enthält für jede der fünf Städte (Knoten: Spalten from, v1, v2, v3, to) und jeden der vier Schritte (Kanten: Spalten e0, e1, e2, e3) eine Spalte. Diese Spalten enthalten jeweils alle verfügbaren Knoten- bzw. Kanteninformationen als struct.

Der Algorithmus hat drei solcher Wege gefunden. Mit normaler DataFrame Syntax kann ich diese Wege anzeigen, zusätzlich berechne ich für jeden Weg die Gesamtdistanz:

paths.select(concat(col("from.id"), lit(", ")\

,col("v1.id"), lit(", ")\

,col("v2.id"), lit(", ")\

,col("v3.id"), lit(", ")\

,col("to.id")\

).alias("path"),\

(col("e0.cityDistance")\

+ col("e1.cityDistance")\

+ col("e2.cityDistance")\

+ col("e3.cityDistance"))\

.alias("totalDistance")\

).show(3,False)

+-------------------------------------------------+-------------+

|path |totalDistance|

+-------------------------------------------------+-------------+

|Frankfurt, Mainz, Karlsruhe, Freiburg, Basel |311 |

|Frankfurt, Mannheim, Karlsruhe, Freiburg, Basel |293 |

|Frankfurt, Heidelberg, Karlsruhe, Freiburg, Basel|297 |

+-------------------------------------------------+-------------+

Es stellt sich heraus, dass der Weg über Mannheim, Karlsruhe und Freiburg nach Kilometern der kürzeste ist.

## Zusammenfassung

Mit der Apache Spark GraphFrame API steht ein Werkzeug für die Analyse von Graphen zur Verfügung, das durch verteiltes Rechnen skalierbar und auch für große Graphen geeignet ist und sich zudem leicht bedienen lässt. Anhand eines praktischen Beispiels wurde gezeigt, dass schon wenige Zeilen Code ausreichen, um einen Graphen in Spark zu erzeugen, zu manipulieren und auszuwerten.

Der verwendete pyspark Code steht hier (bitte Download-Link einfügen) als Jupyter Notebook zum Download zur Verfügung.