Τεχνική Αναφορά Εργασίας

Παναγιώτης Αναστασιάδης 22101

Όλα τα κομμάτια κώδικα που παρουσιάζονται, έχουν μεταφερθεί αυτούσια από τα αρχεία στη τεχνική αναφορά. Ενδεικτικά παραδείγματα εκτέλεσης δίνονται στο τέλος κάθε κεφαλαίου.

1. Common Neighbors

main method

Checking input arguments…

Ξεκινώντας με τη συνάρτηση της main, ελέγχουμε αν ο χρήστης έδωσε τουλάχιστον 2 εισόδους στο πρόγραμμα και εμφανίζεται μήνυμα σφάλματος που εξηγεί το τι πρέπει να δοθεί ως είσοδος κατά την εκτέλεση σε περίπτωση λιγότερων εισόδων.

```
if (args.length < 2) {
   System.err.println("Usage: Arguments must be <inputpath> <number-of-top-scores-to-be-displayed>");
    System.exit(1);
}
```

Spark Initialization

Ορίζουμε το spark να τρέχει σε local mode και βάζουμε το option [*], ώστε τα worker threads να είναι όσα και τα logical cores του μηχανήματος:

```
SparkConf sparkConf =newSparkConf().setAppName("Common Neighbors").setMaster("local[*]");
JavaSparkContext sc =newJavaSparkContext(sparkConf);
```

Reading from the input file path

Βάζουμε τις ακμές του δωσμένου γράφου σε ένα rdd “lines”. Έχουμε τη παραδοχή ότι το δοσμένο αρχείο περιέχει είτε ακμές μορφής <ακέραιος1 ακέραιος2> είτε σχολία μορφής <# comment>.

```
JavaRDD<String> lines = sc.textFile(args[0]);
```

Executing common neighbors calculation and printing the output

Στη συνέχεια εκτελούμε τη συνάρτηση common neighbors που μας υπολογίζει τη μετρική μας, δίνοντας της το input αριθμό k που όρισε ο χρήστης, για να μας επιστρέψει μια λίστα με

τα top-k αποτελέσματα που είναι της μορφής **(<αριθμός κοινών γειτόνων>, κόμβος1 ↔ κόμβος2).**

Παράδειγμα: (2,2 <-> 3), (2,1 <-> 4), …

Τέλος, εκτυπώνουμε τα αποτελέσματα και σταματάμε το Spark Context.

```
List<Tuple2<Integer, String>> cnScores = commonNeighbors(lines, Integer.parseInt(args[1]));

for(Tuple2<Integer, String> score : cnScores) {
    System.out.println(score);
}
sc.stop();
```

commonNeighbors method

Η συνάρτηση για τον υπολογισμό των κοινών γειτόνων.

Attaching lines to RDD of edges

Σε πρώτο βήμα φτιάχνουμε ένα RDD με όλα τα edges του γράφου μας. Αυτό κάνει attach ένα-ένα τα lines σε tuples της μορφής (node1, node2) που το καθένα χαρακτηρίζει μια υπάρχουσα ακμή του γράφου από node1 σε node2. Έχοντας ως παραδοχή από την εκφώνηση της εργασίας ότι ο γράφος είναι μη κατευθυνόμενος, είναι δεδομένο ότι θα πάρουμε και την ακμή node1 σε node2 αλλά και την αντίστροφη, από node2 σε node1. Επιπλέον αγνοούμε τις γραμμές με σχόλια (’#’).

Το RDD edges θα το ξαναχρησιμοποιήσουμε δυο φορές παρακάτω, οπότε και το κασάρουμε.

```
JavaPairRDD<String, String> edges = lines.flatMapToPair(s -> {
            String[] tokens = s.split(SPACE.pattern());
            ArrayList<Tuple2<String, String>> arrayList = new ArrayList<>();
            if (!s.contains("#")) {
                arrayList.add(new Tuple2<>(tokens[0], tokens[1]));
            }
            return arrayList.iterator();
        });
//cache the result
edges.cache();
```

Join the edges and filter the result

Κάνοντας join το RDD των edges με τον εαυτό του (όπως έχουμε δει και στο εργαστήριο για την άσκηση fof), είναι δεδομένο ότι θα πάρουμε ένα αποτέλεσμα από pairs της μορφής (common-neighbor, (node1, node2)), όπου το (node1, node2) θα περιέχει:

1. pairs μορφής (node1, node1)

2. ακμές node1-node2 που **δεν** υπάρχουν στο γράφο, άλλα οι κόμβοι έχουν κοινό γείτονα. Η ακμή node1-node2 θα έχει τόσα instances στο αποτέλεσμα όσοι και οι κοινοί γείτονες των δύο κόμβων. **Αυτές είναι και οι επιθυμητές ακμές που ψάχνουμε.**
3. ακμές node1-node2 που υπάρχουν **ήδη** στο γράφο και οι κόμβοι τους έχουν κοινό γείτονα.
4. το 2. και το 3. ξανά αλλά σε reverse μορφή, όπου η ακμή είναι node2-node1.

```
JavaPairRDD<String, Tuple2<String, String>> joinedEdges = edges.join(edges);
```

Στη συνέχεια, θέλουμε να φιλτράρουμε το αποτέλεσμα του join από το 1. και το 4. (από το 3. θα απαλλαχτούμε στη συνέχεια), αλλα και να αλλάξουμε τα pairs μορφής (common-neighbor, (node1, node2)) σε pairs απλουστευμένης μορφής (node1, node2).

Ο τρόπος για να το κάνουμε αυτό είναι μια flatMapToPair, όπου και κρατάμε μόνο τις ακμές όπου node1 < node2 δημιουργώντας pairs μορφής (node1, node2).

```
JavaPairRDD<String, String> tempResults = joinedEdges.flatMapToPair(s -> {
    ArrayList<Tuple2<String, String>> arrayList =newArrayList<>();
        if(Integer.parseInt(s._2()._1()) < Integer.parseInt(s._2()._2())) {
        arrayList.add(s._2());
    }
return arrayList.iterator();
});
```

Removing the existing edges

Έχουμε φτάσει πλέον πολύ κοντά στο τελικό αποτέλεσμα, αφού το tempResults αυτή τη στιγμή περιέχει instances από ακμές κόμβων που δεν υπάρχουν στο γράφο (κάθε instance = ένας κοινός γείτονας, όπως είπαμε και παραπάνω), αλλά και ακμές που υπάρχουν ήδη στο γράφο.

Θα αφαιρέσουμε τα ανεπιθύμητα pairs κάνοντας subtract το αρχείο edges (που περιέχει όλες τις υπάρχουσες ακμές) από το δικό μας αποτέλεσμα, ενώ με μια mapToPair θα φτιάξουμε ένα σύνολο από pairs μορφής (node1 ↔ node2, 1) που θα μας επιτρέψει μετά να μετρήσουμε τους κοινούς κόμβους για κάθε non-existing ακμή μέσω μιας reduceByKey.

```
JavaPairRDD<String, Integer> unconnectedEdgeInstances = tempResults.subtract(edges).mapToPair(s -> {
return newTuple2<String, Integer>(s._1() + " <-> " + s._2(), 1);
});

JavaPairRDD<String, Integer> scores =  unconnectedEdgeInstances.reduceByKey((v1, v2) -> v1 + v2);
```

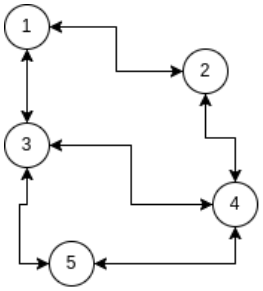
Sorting and returning the results

Θέλουμε να επιστρέψουμε τα top-k αποτελέσματα στον χρήστη, οπότε πρέπει να ταξινομήσουμε τα αποτελέσματα σε φθίνουσα σειρά με βάση το value, δηλαδή των αριθμό

των κοινών κόμβων. Για το λόγο αυτό, φτιάχνουμε ένα RDD με τα αποτελέσματα σε reversed μορφή (<αριθμός κοινών γειτόνων>, node1 ↔ node2) για να μπορέσουμε να εκτελέσουμε για το τέλος μια sortByKey και να επιστρέψουμε το αποτέλεσμα.

```
JavaPairRDD<Integer, String> reversedTuples = scores.mapToPair(s -> {
    return newTuple2<>(s._2(), s._1());
});
return reversedTuples.sortByKey(false).take(numOfDisplayedScores);
```

Demonstration example given a small graph



edges RDD

```
[(2,1), (1,3), (4,2), (3,4), (5,3), (4,5), (1,2), (3,1), (2,4), (4,3), (3,5), (5,4)]
```

joinedEdges -> pairs of type (common-neighbor-of-a-b, (a, b))

```
[(4,(2,2)), (4,(2,5)), (4,(2,3)), (4,(5,2)), (4,(5,5)), (4,(5,3)), (4,(3,2)), (4,(3,5)),
 (4,(3,3)), (2,(1,1)), (2,(1,4)), (2,(4,1)), (2,(4,4)), (5,(3,3)), (5,(3,4)), (5,(4,3)),
 (5,(4,4)), (3,(4,4)), (3,(4,1)), (3,(4,5)), (3,(1,4)), (3,(1,1)), (3,(1,5)), (3,(5,4)),
 (3,(5,1)), (3,(5,5)), (1,(3,3)), (1,(3,2)), (1,(2,3)), (1,(2,2))]
```

tempResults -> removing pairs (a, a) and and pairs (a, b) where a > b

```
[(2,5), (2,3), (3,5), (1,4), (3,4), (4,5), (1,4), (1,5), (2,3)]
```

existing edges!!

unconnectedInstances -> removing existing edges (subtract) and creating pairs of (a <-> b, 1) (mapToPair)

```
[(1 <-> 5,1), (2 <-> 3,1), (2 <-> 3,1), (1 <-> 4,1), (1 <-> 4,1), (2 <-> 5,1)]
```

reversedTuples -> after reducing the previous elements and reversing the tuples

```
[(2,2 <-> 3), (2,1 <-> 4), (1,2 <-> 5), (1,1 <-> 5)]
```

2. Jaccard Coefficient

main method

Πανομοιότυπη με αυτή των common neighbors.

Printing the results

Εκτυπώνουμε τα top αποτελέσματα σε μορφή <jaccard-coefficient, node1 ↔ node2> με ακρίβεια 5 δεκαδικών ψηφίων. Παράδειγμα: 0.47170, 3547 <-> 108418

```
List<Tuple2<Double, String>> jcScores =jaccardCoefficient(lines, Integer.parseInt(args[1]));

//print the results
for(Tuple2<Double, String> score : jcScores) {
    System.out.println(String.format("%.5f", score._1()) + ", " + score._2());
}
```

findCommonNeighborsScores method

Η μετρική jaccard coefficient, με βάση τον τύπο της, χρειάζεται τον υπολογισμό του αριθμού των κοινών γειτόνων.

Για το σκοπό αυτό δημιουργούμε τη μέθοδο findCommonNeighborsScores, η οποία βασίζεται επακριβώς στον υπολογισμό των κοινών γειτόνων, έτσι όπως το πραγματοποιήσαμε στο πρώτο ερώτημα.

Λαμβάνει σαν arguments το rdd των existing edges του γράφου, καθώς και το αποτέλεσμα του join των edges με τον εαυτό τους.

Η υλοποίηση είναι ακριβώς η ίδια με αυτή που περιγράφεται στο κεφάλαιο ***Common Neighbors*** και συγκεκριμένα στις 2 παραγράφους ***Join the edges and filter the result*** και ***Removing the existing edges.***

Η συνάρτηση επιστρέφει ένα RDD με pairs της μορφής

(node1 ↔ node2, number-of-common-neighbors) που αφορά κάθε unconnected ακμή.

jaccardCoefficient method

Creating the RDD of edges, caching and joining…

Ακολουθώντας ίδια λογική με το πρόγραμμα του προηγούμενο ερωτήματος, φτιάχνουμε ένα RDD με τα όλα τα edges του γράφου (tuples μορφής (node1, node2)), θεωρώντας τον ως μη κατευθυνόμενο και αγνοώντας τα σχόλια, δηλαδή τις γραμμές που ξεκινάνε με #.

Στη συνέχεια, κασάρουμε το αποτέλεσμα και υπολογίζουμε το join των edges με τον εαυτό τους φτιάχνοντας το joinedEdges.

Calculating total number of neighbors for every node

Για τη συγκεκριμένη μετρική είναι απαραίτητο να βρούμε το σύνολο των γειτόνων για κάθε κόμβο ξεχωριστά. Δεδομένου ότι ο γράφος είναι μη κατευθυνόμενος μπορούμε με μια mapToPair να μετατρέπουμε κάθε edge (node1, node2) που βρίσκουμε σε ένα pair (node1, 1). Το 1 ισοδυναμεί με έναν σε αριθμό γείτονα το κόμβου node1.

Άρα κάνοντας στη συνέχεια μια reduceByKey, βρίσκουμε το συνολικό αριθμό γειτόνων για κάθε κόμβο, έχοντας πλέον pairs της μορφής (node1, total-number-of-neighbors).

```
JavaPairRDD<String, Integer> nodesWithNeighborsCount = edges.mapToPair((s) -> {
    return(newTuple2<>(s._1(), 1));
}).reduceByKey((v1, v2) -> v1 + v2);
```

Removing pairs of type (node1, node1) and keeping the distinct values of the joinedEdges

Θυμόμαστε από το προηγούμενο παράδειγμα ότι το joinedEdges RDD περιέχει τα instances ακμών από κόμβους που δεν συνδέονται μεταξύ τους στον γράφο, pairs (node1, node1), ακμές που υπάρχουν ήδη στο γράφο, αλλά και τα reverse αποτελέσματα των προηγούμενων.

Με μια flatMapToPair θα μετατρέψουμε τα pairs μορφής (common-neighbor, (node1, node2)) σε pairs (node1, node2), αγνοώντας αυτά της μορφής (node1, node1).

Στη συνέχεια με την distinct θα κρατήσουμε μόνο τα unique αποτελέσματα. Αυτό που θα έχουμε πετύχει σε αυτό το σημείο, είναι να έχουμε ένα RDD με μοναδικά instances από ακμές (node1, node2) αλλά και τα αντίστοιχα (node2, node1), σημειώνοντας ότι αυτά αφορούν ακμές που είτε υπάρχουν είτε δεν υπάρχουν στο γράφο.

```
JavaPairRDD<String, String> tempResultsPart1 = joinedEdges.flatMapToPair(s -> {
    ArrayList<Tuple2<String, String>> arrayList = newArrayList<>();
        if(Integer.parseInt(s._2()._1()) != Integer.parseInt(s._2()._2())) {
        arrayList.add(s._2());
    }
        return arrayList.iterator();
}).distinct();
```

Removing the already-existing edges and joining with the nodesWithNeighborsCount RDD.

Κάνοντας πρώτα ένα subtract το προηγούμενο αποτέλεσμα με το RDD edges (οι υπάρχουσες ακμές του γράφου), πετυχαίνουμε να έχουμε:

- τις ακμές κόμβων (node1, node2) που **δεν** συνδέονται μεταξύ τους σε μοναδικά instances
- τις reverse ακμές (node2, node1) του προηγούμενου bullet

Ως συνέπεια, μπορούμε πλέον να κάνουμε join το σύνολο αυτών των ακμών με το προηγούμενο RDD που έχουμε φτιάξει, αυτό του συνόλου από pairs (κόμβος, σύνολο-γειτόνων).

Αυτό μας επιστρέφει ένα RDD από απο pairs που περιέχει το (node1, (node2, total-neighbors-of-node1)) και το (node2, (node1, total-neighbors-of-node1)) των non-existing ακμών.

```
JavaPairRDD<String, Tuple2<String, Integer>> tempResultPart2 =
            tempResultsPart1.subtract(edges).join(nodesWithNeighborsCount);
```

Finding the sum of neighbors for nodes of the non-existing edges

Με μια mapToPair μπορούμε πλέον να υπολογίσουμε το άθροιμα των γειτόνων για τους κόμβους των ακμών που ψάχνουμε.

Κάθε pair μορφής (node1, (node2, total-neighbors-of-node1)), όπου node1 < node2, το μετατρέπουμε σε pair (node1 ↔ node2, total-neighbors-of-node1).

Από την άλλη, όταν βρίσκουμε node2 > node1 (την reverse edge), φτιάχνουμε pair μορφής (node1 ↔ node2, total-neighbors-of-node2).

Έτσι, με μια reduce-by-key, φτιάχνουμε τελικά ένα RDD με pairs της παρακάτω μορφής:

- (node1 ↔ node2, sum-of-their-neighbors-count)

```
JavaPairRDD<String, Integer> totalNeighbors = tempResultPart2.mapToPair(s -> {
if(Integer.parseInt(s._1()) < Integer.parseInt(s._2()._1())) {
    return newTuple2<>(s._1() + " <-> " + s._2()._1(), s._2()._2());
 } else {
    return newTuple2<>(s._2()._1() + " <-> " + s._1(), s._2()._2());
 }
}).reduceByKey((v1, v2) -> v1 + v2);
```

Finding the common neighbors and calculating the final result

Για τον αριθμητή της μετρικής του jaccard coefficient χρειαζόμαστε το πλήθος των κοινών γειτόνων για κάθε unconnected ακμή. Οπότε με τη συνάρτηση common-neighbors που περιγράψαμε παραπάνω, παίρνουμε ένα σύνολο από pairs μορφής (node1 node2, common-neighbors-count) οπού τα node1 και node2 αφορούν κάθε unconnected ακμή που ψάχνουμε.

Σε αυτό το σημείο λοιπόν έχουμε 2 RDDS με τις unconnected ακμές, όπου το πρώτο περιέχει pairs με το άθροισμα των γειτόνων των δυο κόμβων κάθε unconnected ακμής και το δεύτερο περιέχει το πλήθος των κοινών γειτόνων κάθε unconnected ακμής.

Κάνοντας τα 2 RDDs join, έχουμε πλέον ενωμένη τη πληροφορία που χρειαζόμαστε για να υπολογίσουμε το jaccard coefficient. Έχουμε δηλαδή pairs της μορφής:

(node1 ↔ node2,(<number-of-common-neighbors>, <sum-of-the-nodes-neighbors>))

Με ένα mapValues υπολογίζουμε το jaccard coefficient για κάθε ακμή, κάνοντας τη πράξη:

⇒ *Έστω ότι A = <number-of-common-neighbors> και B = <sum-of-the-nodes-neighbors>*

⇒ *A / (B - A)*

```
JavaPairRDD<String, Integer> commonNeighbors =findCommonNeighborsScores(edges, joinedEdges);

JavaPairRDD<String, Tuple2<Integer, Integer>> jaccardCoefficientTempResult =
commonNeighbors.join(totalNeighbors);

JavaPairRDD<String, Double> jaccardCoefficientScores = jaccardCoefficientTempResult.mapValues(v ->
(double) v._1() / (double) (v._2() - v._1()));
```

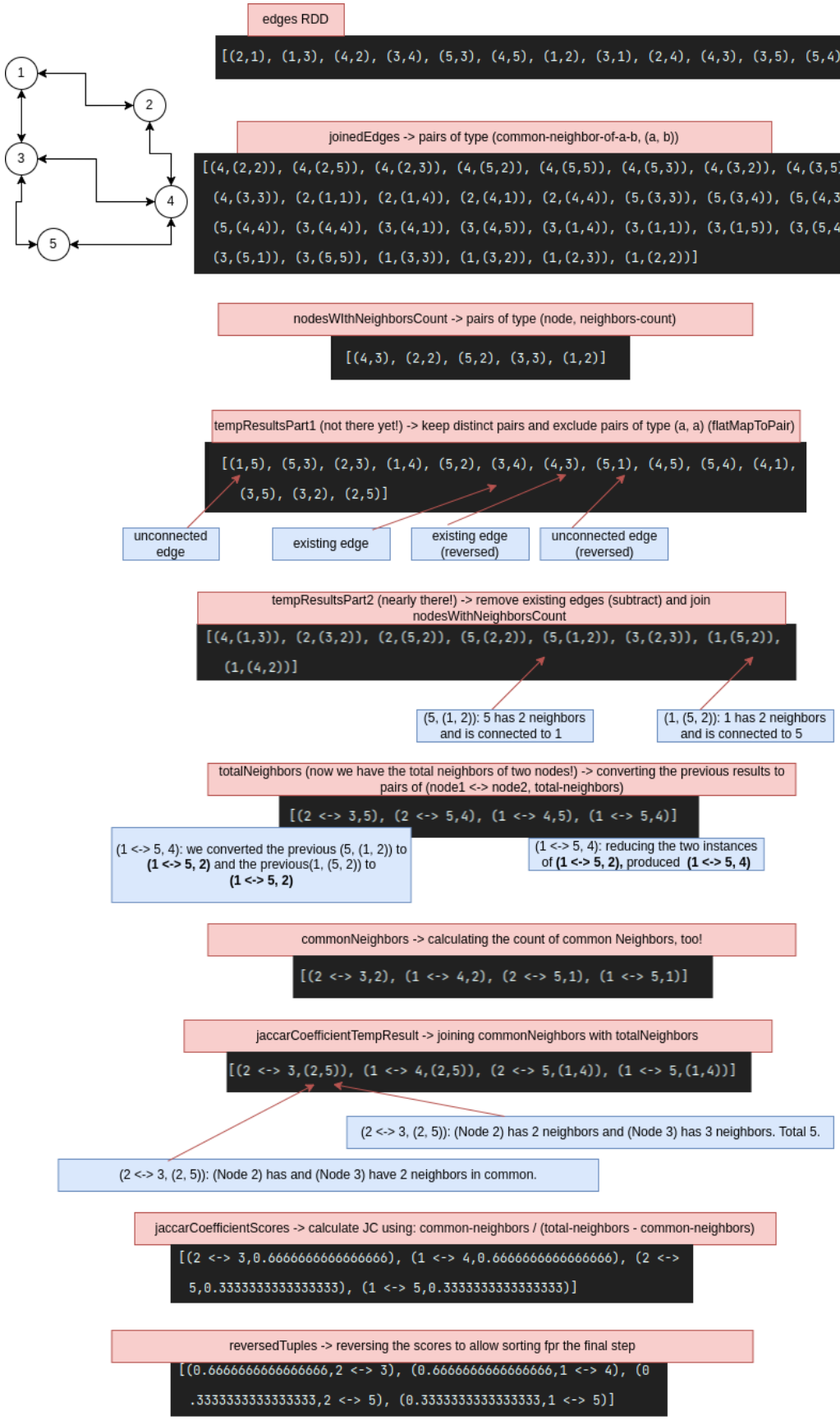
Sorting and returning the results

Κάνουμε και πάλι swap τα tuples του αποτελέσματος ώστε τα pairs να είναι της μορφής: (jaccard-coefficient, node1 ↔ node2) Τέλος, κάνουμε sortByKey σε φθίνουσα σειρά και επιστρέφουμε το αποτέλεσμα.

```
JavaPairRDD<Double, String> reversedTuples = jaccardCoefficientScores.mapToPair(s -> {
    return newTuple2<>(s._2(), s._1());
});

return reversedTuples.sortByKey(false).take(numOfDisplayedScores);
```

Demonstration example given a small graph



3. Adamic Adar

main method

Όπως τα προηγούμενα ερωτήματα. Υπολογίζουμε τα adamic adar scores και τα εκτυπώνουμε σε μορφή **<adamic-adar-score>, node1 ↔ node2** με ακρίβεια 5 δεκαδικών ψηφίων.

adamicAdar method

More of the same…

Όπως ακριβώς στα προηγούμενα ερωτήματα, φτιάχνουμε το **edges** RDD με όλες τις υπάρχουσες ακμές του γράφου, το **nodesWithNeighborSCount** RDD με τους κόμβους και το πλήθος των γειτόνων τους, καθώς και το **joinedEdges** που περιέχει το join αποτέλεσμα του edges με τον εαυτό του.

Calculating all the distinct unconnected edges

Μέσω μιας flatMapToPair φτιάχνουμε pairs μορφής (node1, node2) όπου node1 < node2. Με μια distinct κρατάμε τα μοναδικά instances για κάθε ακμή και μέσω της subtract αφαιρούμε τις ήδη υπάρχουσες ακμές στο γράφο.

Το unconnectedEdgesRaw περιέχει όλες τις unconnected ακμές που ψάχνουμε και μέσω μιας mapToPair φτιάχνουμε το RDD unconnectedEdges που περιέχει pairs μορφής (node1 ↔ node2, -1). Το -1 είναι dummy πληροφορία, αλλά τα pairs αυτής της μορφής θα μας επιτρέψουν το τελικό join που θα εξηγηθεί στο τέλος.

```
JavaPairRDD<String, String> unconnectedEdgesRaw = joinedEdges.flatMapToPair(s -> {
    ArrayList<Tuple2<String, String>> arrayList =newArrayList<>();
        if(Integer.parseInt(s._2()._1()) < Integer.parseInt(s._2()._2())) {
            arrayList.add(s._2());
        }
        return arrayList.iterator();
}).distinct().subtract(edges);

JavaPairRDD<String, Integer> unconnectedEdges = unconnectedEdgesRaw
                .mapToPair(s -> new Tuple2<>(s._1() + " <-> " + s._2(), -1));
```

Collecting adamic adar parameters in one place

Φιλτράρουμε ξανά το joinedEdges και δημιουργούμε ένα νέο RDD filteredJoinedEdges που δεν περιέχει τις ακμές (node1, node1) άλλα ούτε τις reverse ακμές (δηλαδή αυτές που node2 > node1), έχει όμως τόσο unconnected ακμές όσο και κάποιες που υπάρχουν ήδη.

Το filteredJoinedEdges έχει pairs μορφής:

- (common-neighbor-of-node1-and-node2, (node1, node2))

Κάνοντας join το παραπάνω με το RDD nodesWithNeighborsCount (έχει pairs μορφής (node1, count-of-node1-neighbors) για κάθε κόμβο του γραφήματος) παίρνουμε τα pairs:

- *(common-neighbor-of-node1-and-node2, ((node1, node2), count-of-common-neighbor-total-neighbors))*

Στη συνέχεια, με μια mapToPair μπορούμε να αλλάξουμε τη παραπάνω μορφή των pairs σε

- *(node1 ↔ node2, count-of-common-neighbor-total-neighbors)*

Και με μια groupByKey φτάνουμε τελικά σε pairs της μορφής:

- (node1 ↔ node2, [*count-of-common-neighbor-1-neighbors, count-of-common-neighbor-2-total-neighbors, …*])

```
JavaPairRDD<String, Tuple2<String, String>> filteredJoinedEdges = joinedEdges.filter(s -> {
    returnInteger.parseInt(s._2()._1()) < Integer.parseInt(s._2()._2());
});

JavaPairRDD<String, Iterable<Integer>> adamicAdarParameters = filteredJoinedEdges
    .join(nodesWithNeighborsCount)
  .mapToPair(s -> {
        return new Tuple2<>(s._2()._1()._1() + " <-> " + s._2()._1()._2(), s._2()._2());
}).groupByKey();
```

Calculating adamic adar scores

Κάνουμε join το adamicAdarParameters με το RDD unconnectedEdges ώστε να κρατήσουμε μόνο τις unconnected ακμές στο τελικό αποτέλεσμα (θυμίζουμε ότι το unconnected edges είχε όλα τα unique instances των unconnected ακμών σε μορφή (node1 ↔ node2, -1)).

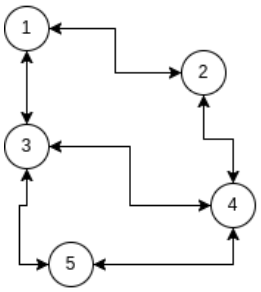
Στο σημείο αυτό μπορούμε να υπολογίσουμε τον τύπο του adamic adar για κάθε unconnected ακμή μέσω μιας mapValues αφού έχουμε όλες τις παραμέτρους που χρειαζόμαστε.

```
JavaPairRDD<String, Double> adamicAdarScores = adamicAdarParameters
.join(unconnectedEdges)
.mapValues(v -> {
        Iterable<Integer> countsOfNeighborsOfCommonNeighbors = v._1();
        doubleadamicAdar = 0;
        for(Integer n : countsOfNeighborsOfCommonNeighbors) {
        adamicAdar += (1 / Math.log10(n));
    }
        return adamicAdar;
});
```

Sorting and returning

Όπως και στα άλλα ερωτήματα, κάνουμε reverse τα tuples του τελικού αποτελέσματος και επιστρέφουμε το αποτέλεσμα ταξινομημένο με τη sortByKey κατά φθίνουσα σειρά.

Demonstration example given a small graph



edges RDD

```
[(2,1), (1,3), (4,2), (3,4), (5,3), (4,5), (1,2), (3,1), (2,4), (4,3), (3,5), (5,4)]
```

joinedEdges -> pairs of type (common-neighbor-of-a-b, (a, b))

```
[(4,(2,2)), (4,(2,5)), (4,(2,3)), (4,(5,2)), (4,(5,5)), (4,(5,3)), (4,(3,2)), (4,(3,5)),
 (4,(3,3)), (2,(1,1)), (2,(1,4)), (2,(4,1)), (2,(4,4)), (5,(3,3)), (5,(3,4)), (5,(4,3)),
 (5,(4,4)), (3,(4,4)), (3,(4,1)), (3,(4,5)), (3,(1,4)), (3,(1,1)), (3,(1,5)), (3,(5,4)),
 (3,(5,1)), (3,(5,5)), (1,(3,3)), (1,(3,2)), (1,(2,3)), (1,(2,2))]
```

nodesWIthNeighborsCount -> pairs of type (node, neighbors-count)

```
[(4,3), (2,2), (5,2), (3,3), (1,2)]
```

unconnectedEdgesRaw -> keeping only the distinct unconnected edges (remove the (a, a), the reversed ones and the existing ones)

```
[(1,5), (2,3), (1,4), (2,5)]
```

unconnectedEdges -> map the previous result to pairs of (node1 <-> node 2, -1) (-1 is dummy value)

```
[(1 <-> 5,-1), (2 <-> 3,-1), (1 <-> 4,-1), (2 <-> 5,-1)]
```

filteredJoinedEdges -> Removing pairs containing (a, a) pairs and the reverse edges (filter)

```
[(4,(2,5)), (4,(2,3)), (4,(3,5)), (2,(1,4)), (5,(3,4)), (3,(4,5)),
 (3,(1,4)), (3,(1,5)), (1,(2,3))]
```

adamicAdarParameters -> join filteredJoinedEdges with nodesWithNeighborsCount (Showing the joining result only!)

```
[(4,((2,5),3)), (4,((2,3),3)), (4,((3,5),3)), (2,((1,4),2)), (5,((3,4),2)), (3,((4,5),3)), (3,((1,4),3)),
 (3,((1,5),3)), (1,((2,3),2))]
```

(4, ((2, 5), 3)): Edge 2-5 has common neighbor the node 4. Node 4 has 3 neighbors.

adamicAdarParameters -> after mapping and grouping

```
[(4 <-> 5,[3]), (2 <-> 3,[3, 2]), (3 <-> 4,[2]), (2 <-> 5,[3]), (1 <-> 4,[2, 3]),
 (1 <-> 5,[3]), (3 <-> 5,[3])]
```

(2 <-> 3, [3, 2]): Edge 2-3 has 2 common neighbors because [3, 2] array has a length of 2.

(2 <-> 3, [3, 2]): Array [3, 2] shows that the first common neighbor has 3 total neighbors itself and the second common neighbor has 2 total neighbors itself.

adamicAdarScores -> first join with the unconnectedEdges RDD to keep the unconnected edges to the final result. Then calculate the adamic adar scores.

```
[(2 <-> 3,5.4178313691767475), (2 <-> 5,2.095903274289385),
 (1 <-> 4,5.4178313691767475), (1 <-> 5,2.095903274289385)]
```

reversedTuples -> Lastly, reverse the previous result to allow sorting