Entrega de programación distribuida

Cano Jon, González Jorge, Lattes Adrián, Lobato Pablo

21/05/2021

Índice

1 Grafo multifichero: **2**
1.1 Unión de rdds: . . . 2
1.2 Triciclos . . . 2

2 Multiples grafos: **3**

Listings

1 Grafo multifichero: mixed . . . 2
2 Grafo multifichero: tag . . . 2
3 Grafo multifichero: tricycles . . . 2
4 Grafo multifichero: process data . . . 2
5 Multiples grafos: función para distinguir grafos . . . 3
6 Multiples grafos: get distinct edges . . . 3
7 Multiples grafos: nombres de los ficheros en los rdds . . . 3
8 Multiples grafos: tag . . . 3
9 Multiples grafos: construyendo los triciclos . . . 3
10 Multiples grafos: process data . . . 3

1 Grafo multifichero:

En este ejercicio analizamos un grafo formado por multiples archivos y devolvemos todos sus triciclos.

1.1 Unión de rdds:

Primero unimos los rdd:

```
62 def mixed(sc, rdds):
63     for rdd in rdds:
64         rdd.cache()
65     data = sc.union(rdds)
```

1.2 Triciclos

Dado el rdd con los nodos y sus listas de adyacencia considerando solo nodos posteriores definimos la función tag que genera la lista de 'exist' y 'pending' para un nodo:

```
32 def tag(node_adjs): # Función iterativa, no perezosa
33     node = node_adjs[0]
34     adjs = list(node_adjs[1])
35     adjs.sort()
36     result = [((node, x), 'exists') for x in adjs]
37     for i in range(len(adjs)):
38         for j in range(i, len(adjs)):
39             result.append(((adjs[i], adjs[j]), ('pending', node)))
40     return result
```

Generamos las listas de adyacencia y generamos los triciclos:

tricycles

```
43 def tricycles(tags):
44     return tags\
45             .groupByKey()\
46             .filter(lambda x: len(x[1]) > 1 and 'exists' in x[1])\
47             .flatMap(
48                 lambda line: map(
49                     lambda x: (x[1], line[0][0], line[0][1]),
50                     filter(lambda x: not x == 'exists', line[1])
51                     )
52                 )
```

process data

```
55 def process_data(data):
56     edges = get_distict_edges(data)
57     node_adjs = get_node_adjs(edges)
58     tags = node_adjs.flatMap(tag)
59     return tricycles(tags)
```

2 Multiples grafos:

Para analizar cada archivo como un grafo a parte cada arista contiene también el nombre del archivo al que pertenece:

Distinguiendo los grafos

```
tupleMorph = lambda tup, x : (tup[0], (tup[1], x))
def get_distict_edges(rdd):
    return rdd\
        .map(lambda x: tupleMorph(get_edges(x[0]),x[1]))\
        .filter(lambda x: x[1] is not None)\
        .distinct()
def independent(sc, rdds, files):
    assert len(rdds) == len(files)
    tagged_rdds = [rdd.map(lambda x: (x,graph)).cache() for rdd, graph in zip(rdds, files)]
    data = sc.union(tagged_rdds)
    for tricycle in process_data(data).collect():
        print(tricycle[0], tricycle[1])
```

Ahora realizamos un procedimiento análogo al del archivo anterior siempre teniendo en cuenta el archivo al que pertenece cada arista para que los triciclos esten compuestos por tres nodos del mismo grafo.

tag

```
def tag(node_adjs): # Función iterativa, no perezosa
    node = node_adjs[0]
    adjs = list(node_adjs[1])
    adjs.sort()
    result = [((node, x), (graph, 'exists')) for (x, graph) in adjs]
    for i in range(len(adjs)):
        for j in range(i, len(adjs)):
            if adjs[j][1] == adjs[i][1] and not adjs[i][0] == adjs[j][0]: #Los nodos aparecen en el mismo grafo
                result.append(((adjs[i][0], adjs[j][0]), (adjs[i][1], ('pending', node))))
    return result
```

Construyendo los triciclos

```
def findExists(tag_list):
    for tag in tag_list:
        if tag[1] == 'exists':
            return True
    return False

def list2dict(lista):
    result = {}
    for key, value in lista:
        if key not in result:
            result[key] = [value]
        else:
            result[key].append(value)
    return result

def construct_tricycles(line):
    edge, tags = line
    return (edge, list2dict(tags))


def tricycles(tags):
    return tags\
            .groupByKey()\
            .filter(lambda x: len(x[1]) > 1 and findExists(x[1]))\
            .map(construct_tricycles)
```

process data

```
def process_data(data):
    edges = get_distict_edges(data).cache()
    node_adjs = get_node_adjs(edges).cache()
    tags = node_adjs.flatMap(tag).cache()
    return tricycles(tags)
```