Bonus Homework: Graph-Matching via Map-Reduce

Zifeng Wang

Question 1

(a)Read a graph from file

```
def read_graph(sc, graph):
    """
    Read a file containing pairs representing the edges of a graph and save
    it inside an rdd.

        Inputs are:
          -sc: SparkContext instance
          -graph: path of the file containing pairs representing the edges of a graph

        The return value is
          - an rdd saves the graph
    """
    return sc.textFile(graph)\
             .map(eval)
```

Here `textFile()` reads a file to rdd, `map()` turns every element of rdd from string to tuple

(b)Computes the degree of each node

```
def compute_degree(graph):
    """
    Compute the degree of rdd graph and save it as (node, degree) pair

        Inputs are:
          -graph: rdd containing nodes pairs representing the edges of a graph

        The return value is
          - an rdd saves the (node, degree) pairs
    """
    return graph.flatMap(lambda (u, v) : [(u, 1),(v, 1)])\
                .reduceByKey(add)
```

Here `flatMap()` maps an edge to 2 nodes with count 1 (because one edge count as both nodes' degree). `reduceByKey()` adds up the degree of each node.

(c)Save the rdd in a file

```
def save_rdd(rdd, save_path):
    """
    Save rdd as text file, each line is an element in RDD

        Inputs are:
            -rdd: rdd to be saved
            -save_path: the path where rdd is saved
    """
    rdd.saveAsTextFile(save_path)
```

Here `saveAsTextFile()` saves the rdd into the path we specified.

Also we have the main code below:

```python
if args.question == 1:
    graph = read_graph(sc, args.graph1)
    degree = compute_degree(graph)
    print(degree.collect())
    save_rdd(degree, 'question1')
```

We executed the code on the small graph shown in the example, result is below:

```
[wang.zife@c0010 BonusAssignment]$ python GraphMatching.py 1 -g1 small
2019-04-11 00:00:40 WARN  NativeCodeLoader:62 - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
[(2, 4), (4, 3), (1, 2), (3, 2), (5, 3)]
```

Or in the saved file, we get the result below:

```
[wang.zife@c0010 question1]$ cat part*
(2, 4)
(4, 3)
(1, 2)
(3, 2)
(5, 3)
```

Question 2

(a) and (c) are identical to question 1

(b) We compute the WL coloring as below:

```python
def compute_WL(graph):
    """
    Compute WL algorithm of a graph

        Inputs are:
            -graph: rdd containing nodes pairs representing the edges of a graph

        The return value is an rdd contains the final (node, color) pairs
    """
    color = graph.flatMap(lambda (u, v) : [u, v])\
                 .distinct()\
                 .map(lambda u: (u, 1))\
                 .cache()
    while(True):
        color_new = graph.flatMap(lambda (u, v) : [(u, v), (v, u)])\
                         .join(color)\
                         .values()\
                         .mapValues(lambda color : [color])\
                         .reduceByKey(add)\
                         .mapValues(lambda clist : hash(str(sorted(clist)))).cache()
        # reannotate the colors
        color_num = color.values().distinct().count()
        color_new_num = color_new.values().distinct().count()
        if color_num == color_new_num:
            break
        color = color_new

    return color_new
```

Here we explain the code in red comments:

```
def compute_WL(graph):
    """
    Compute WL algorithm of a graph

        Inputs are:
            -graph: rdd containing nodes pairs representing the edges of a
```

```
graph

        The return value is an rdd contains the final (node, color) pairs
    """
    color = graph.flatMap(lambda (u, v) : [u, v])\ # extract the nodes
                     .distinct()\ # remove redundant nodes
                     .map(lambda u: (u, 1))\ # color it to “1”
                     .cache() # save it to cache()
    while(True):
        color_new = graph.flatMap(lambda (u, v) : [(u, v), (v, u)])\
                             # add reverse edge so that every node is counted
     .join(color)\ # join with color so that every node get one neighboring
color
     .values()\ # get the value
     .mapValues(lambda color : [color])\ # map the color to a list
     .reduceByKey(add)\ # concatenate the list so that every node get its
clist
     .mapValues(lambda clist : hash(str(sorted(clist)))).cache()
        # sort the clist and turn it to string (hashable), then hash it to
get new color
        # count the number of colors of previous and this iteration
        color_num = color.values().distinct().count()
        color_new_num = color_new.values().distinct().count()
        # if number of colors remains the same, terminate
         if color_num == color_new_num:
            break
        color = color_new

     return color_new
```

Here is the main part of the code:

```
elif args.question == 2:
     graph = read_graph(sc, args.graph1)
     colored_graph = compute_WL(graph)
     print(colored_graph.collect())
     save_rdd(colored_graph, 'question2')
```

We execute the code on the small graph in Question 1 as well, the result is below:

```
[wang.zife@c0010 BonusAssignment]$ python GraphMatching.py 2 -g1 small
2019-04-11 02:19:48 WARN  NativeCodeLoader:62 - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
[(1, -4583713779659305522), (2, 7640972701872404190), (3, -4583713779659305522), (4, -1705017144523782370), (5, -1705017144523782370)]
```

In the saved file, we get the results below:

```
[wang.zife@c0010 question2]$ cat part-*
(1, -4583713779659305522)
(2, 7640972701872404190)
(3, -4583713779659305522)
(4, -170501714452378237)
(5, -170501714452378237)
```

Question 3

Here is the code below:

```python
def get_color_distribution(color):
    """
    Compute distribution of a color RDD

        Inputs are:
            -color: rdd to of the format (node, color)

        The return value is an rdd contains the sorted number of appearances
        of each color
    """
    return color.mapValues(lambda c: (c, 1))\
                .values()\
                .reduceByKey(add)\
                .sortBy(lambda (c, n): n)\
                .values()
```

This function computes the distribution of colors in a graph. I map each node's color to tuple (color, 1) and get the values only, so I got a (color, 1) pair RDD, and I do reduce so that I get the (color, number of that color) RDD. Then I sort it by its value and got the values only. Finally I got a sorted list of number of appearances of each color, which represents the color distribution of the graph.

```python
def get_num_nodes(graph):
    """
    Compute number of nodes in a graph rdd

        Inputs are:
            -graph: rdd containing nodes pairs representing the edges of a graph

        The return value is the number of nodes in the graph
    """
    return graph.flatMap(lambda (u, v) : [u, v])\
                .distinct()\
                .count()
```

This function computes the number of nodes in a graph. It is straight forward that we map the graph from edges (u, v) to [u, v] so that we get every single nodes (but there is redundancy), and we call `distinct()` and `count()` to get the number of nodes.

```python
def compare_WL(graph1, graph2, save_path):
    """
    Compare if 2 graphs are isomorphic or maybe or not

        Inputs are:
            -graph1: rdd containing nodes pairs representing the edges of the 1st graph
            -graph2: rdd containing nodes pairs representing the edges of the 2nd graph
            -save_path: where to save the mapping if 2 graphs are isomorphic or maybe

        The return value is string shows if 2 graphs are isomorphic or maybe or not
    """
    n1 = get_num_nodes(graph1)
    n2 = get_num_nodes(graph2)
    if n1 != n2:
        return "not isomorphic"
    n = n1
    color1 = compute_WL(graph1).cache()
    color2 = compute_WL(graph2).cache()
    dist1 = get_color_distribution(color1).collect()
    dist2 = get_color_distribution(color2).collect()

    if dist1 == dist2 and len(dist1) == n:
        color1_reverse = color1.map(lambda (u, c) : (c, u))
        color2_reverse = color2.map(lambda (u, c) : (c, u))
        mapping = color1_reverse.join(color2_reverse)\
                              .values()

        save_rdd(mapping, save_path)
        return "isomorphic"
    elif dist1 == dist2 and len(dist1) < n:
        color1_reverse = color1.map(lambda (u, c) : (c, [u]))\
                             .reduceByKey(add)
        color2_reverse = color2.map(lambda (u, c) : (c, [u]))\
                             .reduceByKey(add)
        mapping = color1_reverse.join(color2_reverse)\
                              .values()
        save_rdd(mapping, save_path)
        return "maybe isomorphic"

    else:
        return "not isomorphic"
```

After we have these helper functions, we can get into the part compares 2 graphs, here as the code is long, I explain it in red comments below:

```
def compare_WL(graph1, graph2, save_path):
    """

    Compare if 2 graphs are isomorphic or maybe or not


        Inputs are:
            -graph1: rdd containing nodes pairs representing the edges of
the 1st graph
            -graph2: rdd containing nodes pairs representing the edges of
the 2nd graph
            -save_path: where to save the mapping if 2 graphs are isomorphic
or maybe


        The return value is string shows if 2 graphs are isomorphic or maybe
or not
    """
```

```
    n1 = get_num_nodes(graph1) # get the number of nodes of 1st graph
    n2 = get_num_nodes(graph2) # get the number of nodes of 2nd graph
    if n1 != n2: # if number of nodes not equal, not isomorphic
        return "not isomorphic"
    n = n1
    color1 = compute_WL(graph1).cache() # compute final color of 1st graph
    color2 = compute_WL(graph2).cache() # compute final color of 2nd graph
     dist1 = get_color_distribution(color1).collect()
                                    # compute color distribution of 1st graph
     dist2 = get_color_distribution(color2).collect()
                                    # compute color distribution of 2nd graph
    # if distributions are same and number of colors is n, isomorphic
     if dist1 == dist2 and len(dist1) == n:
         # get the mapping by reverse the (node, color) rdd to (color,
         node) in two graphs and join them, get the value
        color1_reverse = color1.map(lambda (u, c) : (c, u))
        color2_reverse = color2.map(lambda (u, c) : (c, u))
        mapping = color1_reverse.join(color2_reverse)\
                              .values()
         # save the mapping to save_path
        save_rdd(mapping, save_path)
        return "isomorphic"
    # if distributions are same and number of colors is less than n, maybe
isomorphic
     elif dist1 == dist2 and len(dist1) < n:
         # similar to previous one, but as we have multiple nodes have same
         color and we still want to get the mapping, we just concatenate
         them in a list and map the list instead of a single node
        color1_reverse = color1.map(lambda (u, c) : (c, [u]))\
                             .reduceByKey(add)
        color2_reverse = color2.map(lambda (u, c) : (c, [u]))\
                             .reduceByKey(add)
        mapping = color1_reverse.join(color2_reverse)\
                              .values()
        save_rdd(mapping, save_path)
        return "maybe isomorphic"
    # in the last situation, not isomorphic
     else:
        return "not isomorphic"
```

```python
elif args.question == 3 or args.question == 4:
    graph1 = read_graph(sc, args.graph1)
    graph2 = read_graph(sc, args.graph2)
    save_path = 'question' + str(args.question)
    print(compare_WL(graph1, graph2, save_path))
```

The main code is self-explanatory and only calls of previously defined functions. (As question 3 and 4 are basically the same, I merge them together)

Here are some examples I run:
I run the code first on 2 graphs have different # of nodes

```
(1,2)
(1,3)
```

and

```
(1,2)
(1,3)
(1,4)
```

```
[wang.zife@c0011 BonusAssignment]$ python GraphMatching.py 3 -g1 small11 -g2 small12
2019-04-11 11:23:40 WARN  NativeCodeLoader:62 - Unable to load native-hadoop library for your platfo
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
not isomorphic
```

Of course not isomorphic.

Then I tried it on 2 graphs have same # of nodes but not isomorphic

```
(1,2)
(1,3)
(1,4)
```

and

```
(1,2)
(2,3)
(3,4)
```

```
[wang.zife@c0011 BonusAssignment]$ python GraphMatching.py 3 -g1 small11 -g2 small12
2019-04-11 11:28:25 WARN  NativeCodeLoader:62 - Unable to load native-hadoop library for your
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
not isomorphic
```

Then I tried on two 2-regulare graphs, it outputs maybe isomorphic as expected:

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

and

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

```
[wang.zife@c0011 BonusAssignment]$ python GraphMatching.py 3 -g1 small1 -g2 small2
2019-04-11 11:31:33 WARN  NativeCodeLoader:62 - Unable to load native-hadoop library
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(new
maybe isomorphic
```

The mapping is

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

Which means they all have the same color.

Finally ,I tried the graph1 and graph2 in question 4, they are isomorphic, the results are included in the answer of question 4

Question 4

I ran the code in question 3 and changed the input parameters as `-g1 graph1 – g2 graph2`, the result is below:

```
[wang.zife@c0011 BonusAssignment]$ python GraphMatching.py 4 -g1 graph1 -g2 graph2
2019-04-11 11:37:59 WARN  NativeCodeLoader:62 - Unable to load native-hadoop library
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(ne
isomorphic
```

They are isomorphic, the sigma is as below (too many to use screenshot):

(3854346387105314125, 8027265736582163570)
(6397274927410324456, 9163559287318476566)
(8582635754767605027, 4083145163487094257)
(2717441818119289755, 7941851862541315469)
(1088132028126179424, 3644416727655320082)
(6300861295650947930, 2077387379767794092)
(5123161242087570393, 5846697931548679865)
(3588263106106724995, 4614542960620455499)
(4316604587267337634, 8636964411301544600)
(5897189019220884548, 4319385327290251286)
(4350681221943325288, 2021624522864021098)
(1514460133087038438, 4063633071244237856)
(8899141786446288440, 6963267116897104602)
(7316816924358054807, 2913855033873642885)
(5410953421829226610, 6627261209215740248)
(7088339445795422518, 5820107855166824144)
(9099642257401296416, 6439313597705981650)
(5469068452363158971, 8292685810751966305)
(5862982264410331112, 3279571361625530300)
(7012490362931148546, 1887278414952232560)
(4777770899578820056, 4794620529638945414)
(7666984131719972667, 8558352365307787053)
(5804139041885353811, 6432662821937666977)
(6727950973158442314, 4738601670943169348)
(1541669400639725827, 2647017302906598453)
(7883668919892241939, 7547417858295491999)
(1912055109368181415, 4758337764402027221)
(6297586557206000355, 7644822491980350031)
(2023148349147468681, 2857239281565429361)

(1085478628391649285, 8055487026984966729)
(2237074709260449506, 3602710950554235108)
(6330262120194447842, 320912596101208604)
(4897349158767958478, 6495332456401757716)
(1424404510967582439, 6389737778113628025)
(2462053627055943785, 1477695173293147561)
(5255823856136678095, 8633242994215042141)
(8259200155796238495, 5289767561791389007)
(1395135018916976084, 4764956539850341222)
(4283668414727448674, 7582472859628262556)
(9149204513216798142, 6907248258457329048)
(7634047959180083707, 3169615289903656211)
(5900940092572313293, 8064286837185270309)
(3132072333858648453, 3129696783197530935)
(3850595313753885380, 6865798482892246378)
(2491451119746550908, 147278066186141098)
(4009774643309465345, 4264048234935508365)
(7349753096897944023, 3804665382832462139)
(601896430734521358, 5952292609580809044)
(1930244765302577217, 8730455626619897933)
(1118414800931538501, 3672480628482477559)
(8307264822873324720, 3171828324847646238)
(1963180937842466177, 2997512028591545587)
(3978863293512734652, 8877210593812151938)
(5270223218891055639, 8271328776308854779)
(3470998216434581607, 871822485413425927)
(5670383123339691825, 6591018444538823263)
(4243478236578826712, 4697151895634087254)
(3816388558943331944, 6933603127364500922)
(4965896657464149378, 7072832422031830696)
(4428146713659911585, 2494614689948724367)
(5138159372183800292, 5764344998007050350)
(3441728724383975252, 2496603723676712730)
(7548394899981340896, 4484888898921513474)
(6943942864234957646, 766227807125296236)
(6744830201485131116, 4119395928404011842)
(3780525993696446699, 3646373760807307037)
(5837075214425242771, 285857594768438047)
(5827119699163445867, 3615186502719746915)
(6100717993815606923, 7958018393044108633)
(8936110518007762826, 8174005575084307768)
(4254267590183948762, 5184172883759259860)
(3209230150669678927, 2551940815967455523)
(4166012744862240159, 9213391108446832719)

(4800377032860710205, 9081206353840847115)
(2764245169391977098, 5907859800300620144)
(382523749049055401, 5944102565233537897)
(7102610807749798974, 4277935551725168616)
(5502593791805350876, 8215991683762111186)
(6227734944866436592, 7638235716532035806)
(4250992851739001187, 383262228453296079)
(5633311351868434499, 8645048881221159751)
(1505129614069452774, 2389020011916595220)
(6848873784072437489, 1914466575716933789)
(6003979646393508405, 1239497976082002585)
(1078039936561958390, 9205265064419561732)
(7300508110841256032, 822246665821072302)
(7169754947554569561, 5338485878633817799)
(8444718047341255695, 7389062235950336737)
(504924304827273341, 8538166507275899211)
(3456000086338351708, 954431420235057234)
(2727644936089495450, 3089392994428422460)
(5088680402401995733, 147710231987477123)
(6786406776883837043, 5269133965342605925)
(5846103036115642278, 8530040463184628192)
(9029879399046735788, 7608571727063432446)
(6072058621490953173, 3072402657434800291)
(5334690128987556353, 5069445681981699139)
(8728698054620969497, 291674371181635349)
(6693744218379888974, 91259207490365032)
(1752915625181643942, 4606416916529184480)

Question 5

First we need to emphasize that the number of nodes in two graphs should be the same and the number of nodes should be greater than 1. That is, we consider the non-trivial case
Consider a graph $G(V, E)$ that is k-regular. Suppose the initial color we give every node is c^0. For the first iteration, Take $v_i \in V$, we have $clist_{v_i} = [c^0, ..., c^0]$ and $|clist_{v_i}| = k$. So basically every node get exactly the same $clist$ (even if $k = 0$, $clist$ is empty list), thus they will again be the same color, iteration terminates. The final status will always be all nodes get exactly the same color.
The situation is the same for $G'(V', E')$ that is k-regular as well.
So the color distribution of $G(V, E)$ and $G'(V', E')$ will always be the same and number of color will be 1 for both graph, less than number of nodes. So the algorithm will always output "maybe isomorphic".