CSE 523 -Final Project Report

Graph Analytics using Oracle PGX & Green-Marl

Aditi Singh (110285096)

Time-Line Overview:

On schedule	🟩	Off Schedule	🟥	Milestone	🟪

	Date	Topic	Work	Output	Lesson	L
1	2/5/2016	Introduction to the PGX model and Project	Read up on PGX Read the paper	Algorithm outline created		🟩
2	2/12/2016	Label Propagation	Infrastructure setup complete + PGX installed and tested with sample tutorial queries	Algorithm outline created		🟩
3	2/19/2016	Label Propagation	Shared the Google Docs sheets with queries and bug fixes		Use sets in Java program to calculate the number of unique labels : Communities	🟥
4	2/26/2016	Label Propagation	Large graph data testing- SOC live journal data received			🟩
5	3/4/2016	Label Propagation	Fixing errors - some bugs in PGX version recognized			🟩
6	3/11/2016	Label Propagation	1. PGX version upgrade done 2. Facebook data JSON created and used for testing GM code	Python code finished and tested using networkX	By default PGX treats graphs as Directed and the to_undirected() function is required to be run	🟩
7	3/18/2016	Label Propagation	Bugs fixed			🟥
8	3/25/2016	Label Propagation	Python and GM code Tested	Python and Gm code gives correct output for small graph but not Facebook data		🟥
9	4/1/2016	Label Propagation	Python and GM code Finalized	Python NetworkX code and GM code converges and gives correct output	EXERCISE 1 COMPLETE	🟪
10	4/8/2016	Label Propagation	Optimization done and tested on Soc live dataset again	Optimization by using set to store neighbors for accessing map - reduces tbe time to 25% All codes converge to the same output for even the large SOC data set	1. since map is not iterable , store the values in a set. This will give a set of unique keys. 2. previously I was traversing over all nodeID as i and checking for each i if it is a key and if yes then performing the maxkey operation. This is a very slow process. Replace with 1 above or avoid it altogether as mentioned in the cell below	🟩
11	4/15/2016	Comparison of old student's code with my code	1. Facebook data - MATCH 2. SOC live data - MATCH		1. using set is okay, but can be avoided by using the inbuilt n.nbrs set provided. No need to store the values again.	🟪
12	4/22/2016	Meeting update	Newman's Algorithm given	-	-	🟩

		for new algorithm	and studied			🟩
13	4/29/2016	Edge betweeness algorithm	Python code completed for edge detection algorithm		Modularity was not implemented. Just a networkx implementation of edge betweeness	🟩
14	5/6/2016	Edge betweeness algorithm	Single definition Python code written as a first step to create Green-Marl Code. All for loops can be converted as for each to parallelize		Referred to the documentation of how edge betweeness is done in the NetworkX source and then tried to replicate it for GM	🟩
15	5/13/2016	Edge betweeness algorithm	Final Report Submission	Haven't tested the code completely.	The code does not seem to work for graphs wherein multiple edges exist between two nodes.	🟩

Motivation:

We had been given tasks to learn about various parallel graph algorithms and also how to use Green-Marl as a domain specification language to write scalable code easily. Since Oracle PGX is still under development and new to us, we wrote some code in Green-Marl and tested its accuracy by writing the same code in Python as well to match results.

I observed that Python already has a lot of libraries in place for graph algorithms, and they are widely in use. Some of the well-known libraries are NetworkX and IGraph, of which I am using the former to test my Green-Marl code. Since it has been under development for quite some years now, ample support is available for it, which is not the case in PGX. Thus, while writing parallel code on PGX, several errors by made and some were found out to be system bugs.

But the main aim in developing a Domain Specification Language, just for Graph Analytics, Green-Marl, is to make it easier for a first time user to write complex code, by reusing many components which are encountered in everyday graph algorithms. For e.g. Functions and directives like inBFS , G.nbrs , G.nodes, G.edges etc.creates an abstraction to code easily. Moreover, each graph also uses Node Properties and Edge Properties to store properties for each node and edge respectively.

During the implementation phase, I had written some small test code snippets, to test if the errors encountered in the previous versions were fixed in the newer versions of PGX or not, apart from the algorithm implementation code.

Approach:

The timeline overview table above describes how I had distributed the work throughout the semester. I had implemented and submitted the correct "Label Propagation Algorithm" by 1st April, 2016. This was my first task and as I was waiting on my second task, I utilized the time in testing and optimizing the code to get better results. During that time I had also written some sample code to test if some of the bugs encountered in the previous versions of PGX still exist in the newer version.

On 22nd April, 2016, I had been given the task of implementation of the Community Detection algorithm using Girvan Newman Algorithm. This is a relatively more complex algorithm to implement. Several doubts had cropped up during this implementation which will be discussed below.

Label Propagation Algorithm:

Algorithm:

```
def  Algorithm_1:
    initialize labels : for each v in V , l(v)[0]=v
    i=0
    while(!stop_criteria) do:
```

```
    i++
    propogation:
            for each v in V do
                    l(v)[i] = argmax over l sum(over all neighbors of v) {l(v)[i-1]== l}
            end
    end
    return final labeling : l(v)[t] for each v in V,
            where t is the last executed step

def stop_criteria:
    if either l(v)[i]= l(v)[i-1] for each v in V or l(v)[i]= l(v)[i-2] for each v in V
        return True
```

Code:

```
procedure label_propagation(g:graph,ID:N_P <int>(g) ;label: N_P <int>(g)):int
{
        bool stop=false;
        nodeProp <int>(g) label_1;
        nodeProp <int>(g) label_2;
        nodeProp <int>(g) label_next;
        int i=1;
        bool mainStop=false;
        int c;
        bool l1_stop;
        bool l2_stop;

        if( g.numNodes()<=1 ) return 0;

        //let the initial labels be equal to the names of the nodes/vertices

        for(n : g.nodes)
        {
                n.label= n.ID;
                n.label_1=-99;
                n.label_2=-99;
                n.label_next=-99;
         }

         //algo1 starts

        i=0;
        do
        {
                i++;
                mainStop=false;
                l1_stop=true;
                l2_stop=true;
                foreach(n:g.nodes)
                {
                        if((n.label!=n.label_1))
                                l1_stop=false;
                }

                foreach(n:g.nodes)
                {
                        if((n.label!=n.label_2))
                                l2_stop=false;
                }
                if(!(l1_stop || l2_stop))
                {

                        foreach(n:g.nodes)
                        {
```

```
                    map<int,int> neighbours;
                    neighbours.clear();
                    // nodeSet(g) unique_neighbours; // redundant *not needed*
                    // unique_neighbours.clear();    //added

                    foreach(s:n.nbrs)
                    {
                        int temp=s.label;
                        if(neighbours.hasKey(temp))
                            neighbours[temp]+=1;
                        else
                        {
                            neighbours[temp]=1;
                            //unique_neighbours.add(s); //added
                        }
                    }
                    int max_key=-99;
                    int j=0;

                    int max_value= neighbours.getMaxValue();

                    foreach(setElement : n.nbrs)
                    {
                        int temp=setElement.label;
                        if(neighbours[temp] == max_value && max_key<temp)
                            max_key=temp;
                    }

                    if(max_key==-99||max_value==-99)
                        max_key=n.label;
                    n.label_next=max_key;
                }

            foreach(n:g.nodes)
            {
            n.label_2=n.label_1;
            n.label_1=n.label;
            n.label=n.label_next;
            }
        }
        else
            mainStop=true;
        }while(!mainStop && i<100);

        map<int,int> labelSet;

        foreach (n : g.nodes)
        {
            int n_label = n.label;
            labelSet[n_label] = 1;
         }

    return labelSet.size();
}
```

Output:

DATA	# OF NODES	TIME
SOC live data	Size of the community:211882	(took 161127ms)
Facebook Data	Size of the community:376	(took 1574ms)

Errors and Inferences:

SN	Errors	Type of Error	Description	Solution
1	Undirected vs Directed Graph	Algorithm Error	The data set I was using was for a directed graph and I was reading it as an default nx.Graph() instead of nx.DiGraph(). Previously, I was reading the graph as Directed in GM code but Undirected in Python. Thus difference in output.	The Label Propagation algorithm is run on undirected graphs. To maintain consistency, I was converted both to undirected graphs.
2	Deterministic Tie break	Algorithm Error	I was using map.getMaxKey() function which when multiple maximum values are encountered returns a random value. This could not give the same results.	I replaced the map.getMaxKey() function in Green-marl(GM) to a more deterministic approach. The algorithm specified the key value corresponding to the max value had to be returned and if multiple keys existed, return the maximum from those keys.
3	Map not Iterable	GM limitation	1. No function to directly access the key elements of the map collection, so creating an iterator from 0 to max number of nodes and checking whether it is a key or not and then comparing it to a max value of the map for the final label. 2. Next I created a set of all the neighbors and then used these values as keys to access the hashmap. 3. The node.nbrs is an iterator which can be used to access all the neighbors of a particular node .	1. This is very slow and wasteful. 2. Faster than 1 , but repetitive and waste of memory and time to populate and store a set again 3. The node.nbrs is a set already provided by the GM language and can be used to access the keys of the hashmap. I was already using it to find out the neighbors, so I don't have to store it again. (Best method)
4	No Neighbors	Overlook	In my code I was not checking for graphs in which some nodes had no neighbors. So was assigning a default value of 0 when I searched for the max key. This is wrong since the minimum label / ID which the node had was 0. When I ran the code the # communities came to 0.	I replaced the minimum to -99 and then ran it. If nodes existed which had no neighbors, then the labels they were assigned in the next iteration would remain unchanged, else, they would be replaced by the most common neighbor's label.
5	Node Property as key for Hashmap	Bug in previous version. Now resolved.	We cannot access the hash value using node property in the foreach loop. Copy it into a separate variable and then use it to access the hash map or else it gives the spinlock error. eg. neighbours[n.label]=1 will give a spinlock error and therefore use this : int temp=n.label; neighbours[temp]=1; to get rid of the error.	Now we can use this. The new PGX version has resolved this bug.

6	getMaxKey()	Overlook	getMaxKey() gives the max key which has the max value in the map and not the maximum key present in the map. I was erroneously setting my maximum limit of the iterations to limit=neighbours.getMaxKey() .	Resolved this error. Later removed as described in point 2.
7	Label assignment error	Overlook	I was not following the order of assignment and since there is not deterministic assignment result in parallel code, this would result in a different result each time.	In the parallel approach first find all the labels for the graph nodes without an update. Store the labels in a temporary node property and then update the nodes with the temp property values after the whole label propagation step is over.
8	Directed/ Undirected	-	To check if the graph correctly converts from directed to undirected and if indeed , the default format in which PGX reads the graph as directed, I tested for it.	No bug on Undirected and directed exists. Ran the code to find out if the graph was correctly converting from a directed to an undirected graph. It is. The default way PGX reads the graph is Directed. Use the undirect() to convert to undirected graph. Wrote a program which prints the neighbors count for each node to test for correctness. [Program written to prove the correctness]
10	Break and Continue	Bug/ missing feature	No break and continue available. So flags have to be used to break loops	
11	No print in GM	Bug/ missing feature	Can't print anything inside GM code. Has to be passed to the java/PGX shell as return value / edge/node properties.	
12	All available functions associated to node and edge properties.	Documentation	There are many missing information regarding the exhaustive list of all functions which are possible for a graph nodes/edges in GM specs.	I had to look into sample code to figure out how to solve certain problems with a function call. Some documentation exists, but many functions which are being used in the PGX examples are missing from the documentation file.

Girvan Newman Algorithm:

Algorithm:

The algorithm's steps for community detection are summarized below

1. The betweenness of all existing edges in the network is calculated first.
2. The edge with the highest betweenness is removed.
3. The betweenness of all edges affected by the removal is recalculated.
4. Steps 2 and 3 are repeated until no edges remain.

Code:

```
/*greenmarl code for edge betweeness*/

procedure label_propagation(G:graph, ID:N_P <int>(G);BC:nodeProp <int>(G)):int
{
        bool stop = false;
        edgeProp <int>(G) weight;              // weight of each edge (undirected will be 1)
        edgeProp <bool>(G) removed;             //present or removed from graph
        nodeProp <int>(G) original_degree;   //summation of all weights of the
        int modularity = 0;
        int max_ = -1;

        if( G.numNodes() <= 1 ) return 0;        //if no nodes means no communities

        //let the initial labels be equal to the names of the nodes/vertices

        foreach(e : G.edges)
        {
                e.weight = 1;
                e.removed = false;
        }

        foreach(e : G.edges)
        {
                modularity += e.weight;
        }

        modularity = modularity / 2;

        foreach(n : G.nodes)
        {
                //n.label = n.ID;
                n.original_degree = n.degree();
        }

                        //run the girvan newman algo

                        //run GirvanNewman algorithm and find the best community split by maximizing modularity measure
                        //let's find the best split of the graph

                        int BestQ = 0;
                        int Q = 0;
                        nodeProp <float>(G) SCC;
                        //nodeProp <float>(G) BC;

                        int init_comp = 1;
                        int compId = 0;
                        while(mainstop == True)
                        {
                                //communityGirvanNewman Step


                                //https://docs.oracle.com/cd/E56133_01/1.2.0/reference/algorithms/kosaraju.html => code to find the
connected components
                                // Initialize SCCbership
                                G.SCC = -1;

                                N_P<bool> Checked;
                                G.Checked = false;

                                // [Phase 1]
                                // Obtain reverse-post-DFS-order of node sequence.
                                // nodeOrder can be also used here but nodeSeq is faster
                                nodeSeq Seq;
                                for(t:G.nodes) (!t.Checked)
                                {
                                        inDFS(n:G.nodes from t)[!n.Checked]
                                        {} // do nothing at pre-visit
                                        inPost{ // check at post-visit
                                                n.Checked = true;
                                                Seq.pushFront(n);
                                        }
                                }

                                // [Phase 2]
                                // Starting from each node in the sequence
                                //   do BFS on the transposed graph G^.
                                //   and every nodes that are (newly) visited compose one SCC.
                                //
                                for(t:Seq.items)(t.SCC == -1)
                                {
                                        inBFS(n:G^.nodes from t)[n.SCC == -1]
                                        {
                                                n.SCC = compId;
                                        }
                                        compId++;
                                }
                                init_comp = compId;

                                //code completes for compId search
                                //start for betweeness centrality

                                G.BC = 0; // Initialize

                                foreach (s:G.nodes)
                                {
                                        // temporary values per node
                                        nodeProperty<double> sigma;
                                        nodeProperty<double> delta;
                                        G.sigma = 0;
                                        s.sigma = 1;

                                        // BFS order iteration from s
                                        inBFS(v: G.nodes from s) (v != s)
                                        {
                                                // Summing over BFS parents
                                                v.sigma = sum(w:v.upNbrs) { w.sigma };
                                        }
                                        inReverse(v!=s)
                                        {
                                                // Reverse-BFS order iteration to s
                                                v.delta =  // Summing over BFS children
                                                                sum (w:v.downNbrs) { (1+ w.delta) / w.sigma } * v.sigma;

                                                v.BC += v.delta ; // accumulate BC
                                        }
                                }
                        //find the edge with max centrality <find the nodes with max centrality and remove all edges associated with it>

                                max_ =  max(n1 : G.nodes){n1.BC};

                                foreach (n1 : G.nodes)
                                {
                                        if(n1.BC == max_)
                                                {
                                                        //find out the edges corresponding to those edges to delete them.
                                                }
```

```
                                        }
                                        // code ends


                                }
                                return max_;
}
```

Output:

DATA	# OF NODES	TIME
Facebook Data	#nodes = 4096 Max Betweeness Centrality= 7831194 #Components = 1	(took 2286ms)

Errors and Inferences:

SN	Errors	Type of Error	Description	Solution
1	Rewriting the modules in GM already implemented	-	There seems to be no examples for calling a green-Marl function definition inside another function. I have thus opened up and rewritten the whole code again in my code .	Have to test for function calls within another function call.
2	Edge can't be deleted	Possible Bug	In the Girvan Newman algorithm, we need to delete the edge which has the maximum betweeness value. I could not delete the particular edge from the graph and reassign it.	My code thus gives an error, since I am just relabeling the edge property as removed if and when I delete it.
3	Graph with Multiple edges between same two nodes.	-	When I was printing the degree of each node with multiple edges between two nodes, it gave me 0.	I have not tested this properly, but maybe I have to run this algorithm on simple graphs.

Appendix :

Python code for Label Propagation:

```
import community
import networkx as nx
import matplotlib.pyplot as plt

def label_prop():
    #G=nx.read_adjlist("facebook_combined.adj",create_using=nx.Graph().to_undirected(),nodetype=int)
    #print "graph loaded"
    #print len(G.nodes())
    G=nx.read_edgelist("facebook_combined.txt", create_using = nx.DiGraph(), nodetype = int)
    #G=nx.read_edgelist("small_graph.edge", create_using = nx.Graph(), nodetype = int)
    G=G.to_undirected()
    print nx.info(G)
    #g=nx.generate_adjlist(G,delimiter=',')
    #print len(g.nodes()), len(g.edges())

    for n in G.nodes():
        print "nodes -",n,"  neighbours -",len(G.neighbors(n))

    for i in G.nodes():
        G.node[i]['label']=i
        G.node[i]['ID']=i
        G.node[i]['l_1']=-99
    G.node[i]['l_2']=-99
    G.node[i]['l_next']=-99
com=set()
for i in G.nodes():
    com.add(G.node[i]['label'])
print "the number of unique labels= ",len(com)


'''
for n,nbrs in G.adjacency_iter():
    for nbr,edict in nbrs.items():
        if nbr==200:
            print n, nbrs, G.node[nbr]['label']
'''
mainStop=False
i=0
while(not mainStop):
    if i==100:
        set_communities=set()
        for n in G.nodes():
            set_communities.add(G.node[n]['label'])
            print G.node[n]['label'],
        print "\n the number of communities after convergence==",len(set_communities)
        return i

    i+=1
    mainStop=False
    l1_stop=True
    l2_stop=True
    for n in G.nodes():
        if(not(G.node[n]['label']==G.node[n]['l_1'])):
            l1_stop=False
    for n in G.nodes():
        if(not(G.node[n]['label']==G.node[n]['l_2'])):
            l2_stop=False

    #print l1_stop, l2_stop
    if( not (l1_stop or l2_stop)):
        #print "in not loop"
        #for n,nbrs in G.adjacency_iter() :
        for n in G.nodes():
            dict={}
            dict.clear()
            for nbr in G.neighbors(n):
            #for nbr,d in nbrs.items():
                temp=G.node[nbr]['label']
                if not dict.has_key(temp):
                    dict.update({temp:1})
                else:
                    dict[temp]+=1
            #print dict
            max_key=-99
            max_freq=-99
            max_list=[]
            #print dict

            for element in dict:
                if max_freq<=dict[element]:
                    max_freq=dict[element]

            for element in dict:
                if dict[element]==max_freq:
                    max_list.append(element)
            if len(max_list)>0:
                max_key=max(max_list)
            #print " iteration ",i,"max list is ",n,max_list
            if(max_key==-99 or max_freq==-99):
                max_key=G.node[n]['label']

            #max_key= max(dict,key=dict.get)
            #print "FINAL",max_key
```

```
                G.node[n]['l_next']=max_key

            for n in G.nodes():
                G.node[n]['l_2']=G.node[n]['l_1']
                G.node[n]['l_1']=G.node[n]['label']
                G.node[n]['label']=G.node[n]['l_next']

        else:
            print "The Community converges"
            mainStop=True
            set_communities=set()
            x=0
            for n in G.nodes():
                set_communities.add(G.node[n]['label'])

            print set_communities

            print "the number of communities after",i," iterations==",len(set_communities)
            for n in G.nodes():
                print G.node[n]['label'],
            return i

def main():
    label_prop()

if __name__ == '__main__':
    main()
```

Python code for Girvan Newman:

```
#!/usr/bin/env python
import networkx as nx
import math
import csv
import random as rand
import sys
_DEBUG_ = True
def main():
    graph_fn = "D:/00-SUNYSBU/00-Courses/2-2016SPRING/CSE523-Project/CODE/finalV2/WeightedEdgeBetweenessGraph.txt"
    G = nx.Graph()  #let's create the graph first
    #buildG(G, graph_fn, ',')
    ##########################################################################################################
    reader = csv.reader(open(graph_fn), delimiter=",")
    for line in reader:
        if len(line) > 2:
            if float(line[2]) != 0.0:
                #line format: u,v,w
                G.add_edge(int(line[0]),int(line[1]),weight=float(line[2]))
        else:
            #line format: u,v
            G.add_edge(int(line[0]),int(line[1]),weight=1.0)

    ##########################################################################################################
    if _DEBUG_:
        print 'G nodes:', G.nodes()
        print 'G no of nodes:', G.number_of_nodes()

    n = G.number_of_nodes()    #|V|
    A = nx.adj_matrix(G)    #adjacenct matrix


    m_ = 0.0    #the weighted version for number of edges
    for i in range(0,n):
        for j in range(0,n):
            m_ += A[i,j]
    m_ = m_/2.0
    if _DEBUG_:
        print "m: %f" % m_

    #calculate the weighted degree for each node
    Orig_deg = {}
```

```
#Orig_deg = UpdateDeg(A, G.nodes())
##########################################################################################################
nodes = G.nodes()
deg_dict = {}
n = len(nodes)  #len(A) ---> some ppl get issues when trying len() on sparse matrixes!
B = A.sum(axis = 1)
for i in range(n):
    deg_dict[nodes[i]] = B[i, 0]
Orig_deg = deg_dict
##########################################################################################################
#run Newman alg
#runGirvanNewman(G, Orig_deg, m_)
###########################################################################################################
#run GirvanNewman algorithm and find the best community split by maximizing modularity measure
#let's find the best split of the graph
BestQ = 0.0
Q = 0.0
while True:
    #CmtyGirvanNewmanStep(G)
    ############################################################

    if _DEBUG_:
        print "Calling CmtyGirvanNewmanStep"
    init_ncomp = nx.number_connected_components(G)    #no of components
    ncomp = init_ncomp
    while ncomp <= init_ncomp:
        bw = nx.edge_betweenness_centrality(G, weight='weight')    #edge betweenness for G
        #find the edge with max centrality
        max_ = max(bw.values())
        #find the edge with the highest centrality and remove all of them if there is more than one!
        for k, v in bw.iteritems():
            if float(v) == max_:
                G.remove_edge(k[0],k[1])    #remove the central edge
        ncomp = nx.number_connected_components(G)    #recalculate the no of components


    ############################################################
    #Q = _GirvanNewmanGetModularity(G, Orig_deg, m_);
    ###########################################################
    deg_ = Orig_deg
    New_A = nx.adj_matrix(G)
    New_deg = {}
    #New_deg = UpdateDeg(New_A, G.nodes())
    ################################################
    nodes = G.nodes()
    deg_dict = {}
    n = len(nodes)  #len(A) ---> some ppl get issues when trying len() on sparse matrixes!
    B = New_A.sum(axis = 1)
    for i in range(n):
        deg_dict[nodes[i]] = B[i, 0]
    New_deg = deg_dict
    ################################################
    #Let's compute the Q
    comps = nx.connected_components(G)    #list of components
    print 'No of communities in decomposed G: %d' % nx.number_connected_components(G)
    Mod = 0    #Modularity of a given partitionning
    for c in comps:
        EWC = 0    #no of edges within a community
        RE = 0    #no of random edges
        for u in c:
            EWC += New_deg[u]
            RE += deg_[u]        #count the probability of a random edge
        Mod += ( float(EWC) - float(RE*RE)/float(2*m_) )
    Mod = Mod/float(2*m_)
    if _DEBUG_:
        print "Modularity: %f" % Mod
    Q = Mod


    ###########################################################
    print "Modularity of decomposed G: %f" % Q
```

```
        if Q > BestQ:
            BestQ = Q
            Bestcomps = nx.connected_components(G)    #Best Split
            print "Components:", Bestcomps
        if G.number_of_edges() == 0:
            break
    if BestQ > 0.0:
        print "Max modularity (Q): %f" % BestQ
        print "Graph communities:", Bestcomps
    else:
        print "Max modularity (Q): %f" % BestQ
    #################################################################################################################


if __name__ == "__main__":
    sys.exit(main())
```