The purpose of the following code is to analyze subgroups of a given free group. Given the generators of the subgroup, it should be able to answer: (1) What is the rank of this subgroup? (2) is it normal in its free group? (3) Given a certain element g of a free group, is g a member of the subgroup? (4) What is the subgroup index? (5) Also it finds the minimal set of generators. The code was done using Jupyter Notebook. It is not the most efficient and may contain bugs, however, it was able to produce correct answers in all the past trials. No part of the code can be run separately, so all of the code must be present in the shell. Function for finding generators contains a bug, but still is able to run.

```
# Defining Graph from the given set of generators
# If H= <aaa, bbb, aB, Ba>  where capital implies inverse, then the input here is
# l=['aaa','bbb','aB','Ba'] and Output is a dictionary with edges as keys and edge types as values.
# For example, if l=['aaa','bbb','aB','Ba'], then the output of graph_generator1(l):
# {('','a'):{'a','b'},('a','aa'):{'a'},('aa',''):{'a'},('','b'):{'b'},('b','bb'):{'b'},
# ('bb',''):{'b'},('B',''):{'a','b'}}.
# Direction of the edge is defined as (incident, terminal). Types of the edges are put into sets.
# vertices in the edges can be any string symbols. The centre vertex (root) is denoted
#  by empty string ''. It will be present in the reduced graph and used for finding generators.
#_____________________________________________________________________________________________
def c(w, i):
    'Given a word from the list of generators and an index that corresponds to some letter in \
    the generator string, returns an edge that corresponds to this index. Function is used in \
    defining an edge.'
    if i!=len(w)-1:
        if w[i].isupper():
            return (w[0:i+1],w[0:i])
        else:
            return (w[0:i],w[0:i+1])
    else:
        if w[i].isupper():
            return ('',w[0:i])
        else:
            return (w[0:i],'')

def graph_generator1(l):
    "Intial graph generating function based on the original input list."
    D=dict()
    for j in range(len(l)):
        w=l[j]
        for i in range(len(w)):
            e=c(w,i)
            if e in D:
                other={c(w,i):{w[i].lower()}.union(D.get(e))}
            else:
                other={c(w,i):{w[i].lower()}}
            D.update(other)
    return D
#_____________________________________________________________________________________________
#
#
def same_incident_pairs(l):
    "Returns list of lists; each list contains two distinct indices; each index correspond to an edge \
    in l; edges in the same list have the same inital vertex."
    d=[]
    for i in range(len(l)):
        first=l[i][0]
        for j in range(len(l)):
```

```
            second=l[j][0]
            if j!=i and second==first:
                d.append([i,j])
    return d
#
def same_terminal_pairs(l):
    "Returns list of lists; each list contains two distinct indices; each index correspond to an edge \
    in l; edges in the same list have the same terminal vertex."
    d=[]
    for i in range(len(l)):
        first=l[i][1]
        for j in range(len(l)):
            second=l[j][1]
            if j!=i and second==first:
                d.append([i,j])
    return d
#
def common_member(a, b):
    a_set = set(a)
    b_set = set(b)
    if (a_set & b_set):
        return True
    else:
        return False
#
def nonred_edges(D):
    "returns a list of adjacent edges that can be reduced."
    l=list(D.keys())
    x=same_terminal_pairs(l)
    x.extend(same_incident_pairs(l))
    L=[]
    for i in range(len(x)):
        y=x[i][0]
        z=x[i][1]
        y_tuple=l[y]
        z_tuple=l[z]
        if D.get(y_tuple)&D.get(z_tuple):
            list_y = [y_tuple]
            list_y.extend([z_tuple])
            L.append(list_y)
    return L
#
def common(l):
    "returns a list with vertex which is in common, index of its location, is '' an uncommon vertex \
    for tuples in l, are any loops present."
    if l[0][0]==l[1][0]: # if the incident vertex is in common.
        return [l[0][0], 0, ''==l[0][1] or ''==l[1][1],l[0][0]==l[0][1] or l[1][0]==l[1][1]]
    else: # if the terminal vertex is in common.
        return [l[0][1], 1, l[0][0]==='' or l[1][0]==='',l[0][0]==l[0][1] or l[1][0]==l[1][1]]
#
def new_edge(l,i):
    "returns new created edge out of Given=[(x1,x2),(x1,x3)] with position of changed vertex.\
    Number i represents the ith graph. If any of x2 or x3 is '' vertex, then the new vertex is \
    of '' type."
    x=common(l) # Type of outcome [common vertex x1, its index, is '' present?, is loop present?]
```

```
    t=['1','2']
    j=(x[1]+1)%2
    if x[2]==True:
        t[j]=''
        if x[3]==True:
            t[x[1]]=''
        else:
            t[x[1]]=x[0]
    else:
        if x[3]==True:
            t[j]=i
            t[x[1]]=i
        else:
            t[j]=i
            t[x[1]]=x[0]
    return (tuple(t),j)
#
def update_edge(t,v,w,i):
    "Given vertices which get unified, an index of algorithm iteration, and an edge tuple \
    where v is assumed to be in t, returns an updated edge."
    if v=='':
        if w in t: # case where {(c,v):{'a'}, (c,w):{'a'}, (v,w):{'b'}} and t=(v,w)
            return ('','')
        else:
            return t
    if w=='': # need to have v-->w:
        if t[0]==t[1]: # case where t=(v,v)
            return ('','')
        else:
            if t[0]==v:
                return ('',t[1])
            else:
                return (t[0],'')
    else: # need to have v-->i
        if t[0]==v:
            if t[1]==w:
                return (i,i)
            else:
                return (i,t[1])
        else:
            if t[0]==w:
                return (i,i)
            else:
                return (t[0],i)
#
def new_graph(i,D):
    "If the list of nonreduced edges is non empty, then returns a new graph dictionary made out of D.\
    Keeps track of overall iteration step."
    nonreduced_edges=nonred_edges(D) # checks for edges which can be reduced in the original graph.
    if len(nonreduced_edges)==0:
        return D
    else:
        # If list of non reduced edges is non empty then define 'given' -
        # as a pair of edges that can be reduced.
        given=nonreduced_edges[0]
```

```
        # Then, define a new edge made from reducing the previous two.
        # Consider all cases of how it can happen.
        # One case is if one of the edges is a loop.
        # Important to keep '' vertex as future reference point in defining generators.
        e = new_edge(given,i)[0]
        # Need to know index of vertex that's different for both edges.
        # Consider all cases of how it can happen.
        index=new_edge(given,i)[1]
        v1=given[0][index] # different vertex from edge 1 in given
        v2=given[1][index] # different vertex from edge 2 in given
        # Define all ways how two vertices in the new edge can be connected.
        edge_types=(D.get(given[0])).union(D.get(given[1]))
        # Define new graph with new edge included.
        new_graph_dict={e:edge_types}
        # Begin adding corrected edges to the new graph.
        for j in range(len(D.keys())): # Unpack old edges.
            edge=list(D.keys())[j]
            if edge==given[0] or edge==given[1]: # Don't include the edges which
            # were chosen to be reduced.
                continue
            # Correctly update the given edge and add it to the new graph.
            # Three cases possible (w/o above).
            elif v1 in edge: # Does not imply that v2 IS NOT in the edge.
                e_new=update_edge(edge,v1,v2,i)
                # For method of updating refer to the update_edge() definition.
                if e_new in new_graph_dict:
                # the updated version of the edge maybe in the graph already.
                    val=(D.get(edge)).union(new_graph_dict.get(e_new)) # Take union of all edge types.
                    other={e_new:val}
                    new_graph_dict.update(other)
                    # The above mainly applies to the case: {('v1','c'):{'a'},('v2','c'):{'b'}}
                else:
                    other={e_new:D.get(edge)}
                    # vertices in the original edge are updated; its values stay the same.
                    new_graph_dict.update(other)
            elif v2 in edge: # repeat of the above.
                e_new=update_edge(edge,v2,v1,i)
                if e_new in new_graph_dict:
                    val=(D.get(edge)).union(new_graph_dict.get(e_new))
                    other={e_new:val}
                    new_graph_dict.update(other)
                else:
                    other={e_new:D.get(edge)}
                    new_graph_dict.update(other)
            else:
            # If v1 or v2 not in the edge, then the edge with its values
            # gets added to a new graph without any changes.
                other={edge:D.get(edge)}
                new_graph_dict.update(other)
    return new_graph_dict
#
def algorithm(l):
    "Takes the set of generators for the subgroup and runs the algorithm. Produces delta:\
    reduced edge graph, an immersion into Rn. Tried generating sets: \
    ['aaB','aaab','Abb','aba','aaaBBA'], ['aaBa','aba','aab','abbb','aB']."
```

```
    D=graph_generator1(l)
    i=0
    while len(nonred_edges(D))>0:
        i+=1
        D=new_graph(i,D)
    return D
# Example: if l=['aaBa','aba','aab','abbb','aB']  then algorithm(l):
# {(2, ''): {'b'},(1, ''): {'a'},('', 'a'): {'a', 'b'},('a', 2): {'a'},(1, 2): {'b'},
# ('a', 1): {'b'}}
# Names of the vertices are unimportant except for the root vertex which is an empty string.
# To draw this reduced edge graph:
# (1) find all distinct types of vertices appearing in the edges
# (2) draw directed edges between them with colour types in the corresponding edge set.
#_____________________________________________________________________________________________
def rank(G):
    'Returns the rank of pi(G,'') of graph Del, where G=algorithm(l).'
    l=list(G.keys())
    vert_degs={}
    for i in range(len(l)):
        edge=l[i]
        v1=edge[0]
        v2=edge[1]
        deg=len(G.get(edge))
        d1=vert_degs.get(v1,0)
        d2=vert_degs.get(v2,0)
        if v1==v2:
            other={v1:d1+2}
        else:
            other={v1:d1+deg, v2:d2+deg}
        vert_degs.update(other)
    E=0.5*sum(list(vert_degs.values()))
    V=len(list(vert_degs.keys()))
    n=1+E-V
    return {'rank of G is':int(n)}
# Example: if H=<'aaBa','aba','aab','abbb','aB'>
# l=['aaBa','aba','aab','abbb','aB']
# G=algorithm(l)
# rank(G)
# >>> {'rank of G is': 4}
#_____________________________________________________________________________________________
# Membership problem
# given tuple with vertex v in G, color type, and position index of v (v,'c',i), find
# the corresponding edge.
def edge_dir(s):
    if s.lower()==s:
        return 0
    else:
        return 1
#
def find_next_edge(t,l):
    'Returns the next edge in the path or False if does not find it'
    for i in range(len(l)):
        if t[0]==l[i][t[2]] and t[1] in G.get(l[i]):
            return l[i]
        else:
```

```
            continue
    return False
#
def check_membership(g,G):
    'Checks whether the given element is a member of the subgroup. If element is a member\
    it returns a tuple with 'is a member' and the path in G that defines this element'
    l=list(G.keys())
    x=['']
    i=0
    t=('', g[0].lower(), edge_dir(g[0]))
    while i<len(g) and find_next_edge(t,l)!=False:
        e=find_next_edge(t,l)
        j=(t[2]+1)%2
        x.append(e[j])
        i+=1
        if i<len(g):
            t=(e[j],g[i].lower(), edge_dir(g[i]))
    if len(x)!=(len(g)+1) or x[len(g)]!='':
        return 'not a member'
    else:
        return ('is a member',x)
# Example:
# l=['aaBa','aba','aab','abbb','aB']
# G=algorithm(l)
# g='aaBBAbA'
# check_membership(g,G)
# >>> ('is a member', ['', 'a', 2, 1, 'a', '', 'a', ''])
#_____________________________________________________________________________________________
# Index
def check_covering(G,n):
    'If l is the original generating set then G=algorithm(l). Returns True if every vertex in G has \
    degree 2n and False otherwise. It should be stated beforehand, for which n Delta-->Rn \
    is an immersion.'
    l=list(G.keys())
    vert_degs={}
    for i in range(len(l)):
        edge=l[i]
        v1=edge[0]
        v2=edge[1]
        deg=len(G.get(edge))
        d1=vert_degs.get(v1,0)
        d2=vert_degs.get(v2,0)
        if v1==v2:
            other={v1:d1+2}
        else:
            other={v1:d1+deg, v2:d2+deg}
        vert_degs.update(other)
    f=list(vert_degs.values())
    for i in range(len(f)):
        if f[i]!= 2*n:
            return False
        else:
            continue
    return (True,len(list(vert_degs.keys())))
#
```

```
def subgroup_index(G,n):
    'Returns the index of the subgroup.'
    if check_covering(G,n)==False:
        return 'Index of the subgroup is infinite'
    else:
        return {'Index of the subgroup is':check_covering(G,n)[1]}
# Example:
# l=['aaBa','aba','aab','abbb','aB']
# G=algorithm(l)
# subgroup_index(G,2)
# >>>'Index of the subgroup is infinite'
#_____________________________________________________________________________________________________
# Normality
def is_normal(G,n):
    "Checks if the subgroup is normal in G."
    if check_covering(G,n)==False:
        return 'G is not normal in Fn'
    else:
        vert_loops={}
        l=list(G.keys())
        for i in range(len(l)):
            edge=l[i]
            v1=edge[0]
            v2=edge[1]
            a=0
            b=0
            if v1==v2:
                for j in range(len(l)):
                    if l[j]==(v1,v1):
                        a+=1
                    else:
                        continue
                other={v1:a}
                vert_loops.update(other)
            else:
                for j in range(len(l)):
                    if l[j]==(v1,v1):
                        a+=1
                    elif l[j]==(v2,v2):
                        b+=1
                    else:
                        continue
                other={v1:a, v2:b}
                vert_loops.update(other)
        f=list(vert_loops.values())
        for i in range(len(f)):
            if f[i]!= f[0]:
                return 'G is not normal in Fn'
            else:
                continue
        return 'G is normal in Fn'
# l=['aaaa','bbbb','ab','ba','aabb']
# G=algorithm(l)
# is_normal(G,2)
# >>> 'G is normal in Fn'
```

```
#_____________________________________________________________________________________________
# Finding generators
def neighbours(v,G):
    'returns neighbours in a graph G which is defined as above.'
    nbhd=[]
    l=list(G.keys())
    for i in range(len(l)):
        if v in l[i]:
            if l[i][0]==v:
                nbhd.append(l[i][1])
            else:
                nbhd.append(l[i][0])
        else:
            continue
    return nbhd
#
def new_nghbrs(v,G,l):
    'Given finds all neigbrs of v and returns a list of vertices not in l.'
    k=neighbours(v,G)
    e=list(set(k)-set(l))
    return e
#
def new_vertices(G,l):
    'G is the original graph, l is the list of keys from the current tree dictionary T.'
    i=0
    new_vertices=[]
    x=[]
    while i in range(len(l)) and len(new_vertices)==0:
        new_vertices+=new_nghbrs(l[i],G,l)
        x.append(l[i])
        i+=1
    if len(new_vertices)==0:
        return False
    else:
        return (new_vertices,x[len(x)-1])
#
def addition(t,T,G):
    'takes new vertices connected to x, defines path for each, puts it into T.'
    n_v=t[0]
    x=t[1]
    path_to_x=T.get(x)
    for i in range(len(n_v)):
        w=n_v[i]
        if (x,w) in G.keys():
            e=(list(G.get((x,w)))[0])
            other={w:path_to_x+e}
            T.update(other)
        else:
            e=list(G.get((w,x)))[0].capitalize()
            other={w:path_to_x+e}
            T.update(other)
    return T
#
def maximal_tree(G):
    'Returns a new type of graph: a dictionary with keys as vertices and corresponding values are the pat
```

```
    paths are defined as strings. If returns {'':"#"}, then maximal tree consists of one vertex only.\
    Returns the maximal tree of a graph G where for some list l generators => g=algorithm(l).'
    T={'':'#'}
    start=neighbours('',G)
    if start==['']:
        return T
    else:
        for i in range(len(start)):
            x=start[i]
            if (x,'') in G.keys():
                e=(list(G.get((x,'')))[0]).capitalize()
                other={x:''+e}
                T.update(other)
            else:
                e=list(G.get(('',x)))[0]
                other={x:''+e}
                T.update(other)
    l=list(T.keys())
    while new_vertices(G,l)!=False:
        n_v=new_vertices(G,l)
        l=list(addition(n_v,T,G).keys())
    return T
# define function that finds the degree of a vertex (not '') in a maximal tree.
def deg_in_max(v,T):
    'Finds the degree of a vertex (not '') in a maximal tree.'
    path=T.get(v)
    vertices=list(T.keys())
    deg=1
    for i in range(len(vertices)):
        w=vertices[i]
        p_w=T.get(w)
        if w!='' and p_w[:len(p_w)-1]==path:
            deg+=1
        else:
            continue
    return deg
# define function that finds the degree of a vertex (not '') in the original graph G.
def deg_in_G(v,G):
    'Finds the degree of a vertex (not '') in the original graph G.'
    edges=list(G.keys())
    deg=0
    for i in range(len(edges)):
        if v in edges[i]:
            deg+=len(G.get(edges[i]))
        else:
            continue
    return deg
# define inverse function for a path string
# define function which finds all neighbours (except for '') of a vertex v in a maximal tree
def edges_in_tree(v,T):
    'Returns list of lists with neighbours of v in T except for '' vertex and correspodning edges with v.
    path=T.get(v)
    x=path[len(path)-1]
    neighbrs=dict()
    l=list(T.keys())
```

```
    if len(path)==1:
        if (T.get(v)).islower():
            other={('',v):{x}}
            neighbrs.update(other)
        else:
            other={(v,''):{x.lower()}}
            neighbrs.update(other)
        for i in range(len(l)):
            w=l[i]
            p_w=T.get(w)
            y=p_w[len(p_w)-1]
            if p_w[:len(p_w)-1]==path:
                if y.isupper():
                    other={(w,v):{y.lower()}}
                    neighbrs.update(other)
                else:
                    other={(v,w):{y}}
                    neighbrs.update(other)
    else:
        for i in range(len(l)):
            w=l[i]
            p_w=T.get(w)
            y=p_w[len(p_w)-1]
            if path[:len(path)-1]==p_w:
                if x.isupper():
                    other={(v,w):{x.lower()}}
                    neighbrs.update(other)
                else:
                    other={(w,v):{x}}
                    neighbrs.update(other)
            elif p_w[:len(p_w)-1]==path:
                if y.isupper():
                    other={(w,v):{y.lower()}}
                    neighbrs.update(other)
                else:
                    other={(v,w):{y}}
                    neighbrs.update(other)
    return neighbrs
# missing edges function: should find loops as well as extra edges in G
# find set of edges in T that v is part of. find set of edges in G that v is part of.
def edges_in_G(v,G):
    'return the set of edges that v is part of in G.'
    l=list(G.keys())
    edges={}
    for i in range(len(l)):
        if v in l[i]:
            other={l[i]:G.get(l[i])}
            edges.update(other)
    return edges
def missing(v,G):
    edges_in_T=edges_in_tree(v,T)
    edges=edges_in_G(v,G)
    l=list(edges.keys())
    missing={}
    for i in range(len(l)):
```

```
        e=l[i]
        if e not in edges_in_T.keys():
            other={e:edges.get(e)}
            missing.update(other)
        elif e in edges_in_T.keys() and edges_in_T.get(e)!=edges.get(e):
            s=edges.get(e)-edges_in_T.get(e)
            other={e:s}
            missing.update(other)
    return missing
# create a {loop, inverse(loop)} function
def create_a_loop(e,T):
    x=list(e.keys())[0][0]
    y=list(e.keys())[0][1]
    if x==’’:
        path_x=’’
        path_y=T.get(y)
    elif y==’’:
        path_x=T.get(x)
        path_y=’’
    else:
        path_x=T.get(x)
        path_y=T.get(y)
    letter=list(e.values())[0]
    loop=path_x+letter+inverse(path_y)
    return {loop,inverse(loop)}
def inverse(p):
    ’Defines an inverse path for a non-empty string.’
    return (p[::-1]).swapcase()
#_____________//_________________//__________________//_____________//_________________//_____________
# Before running the find_generators algorithm, please, insert the list of original generators below.
# This is done due to a mysterious bug in the code which will appear without running this cap code first.
l=[’aaaa’,’bbbb’,’ab’,’ba’,’aabb’] # <---- insert your original list of generators.
G=algorithm(l)
T=maximal_tree(G)
vertices=list(T.keys())
for i in range(len(vertices)):
    v=vertices[i]
    miss=missing(v,G)
#_____________________________________________________________________________________________________
def find_generators(l):
    ’l is a list of elements which generate a subgroup of Fn. (e.g. l=[abb,BaB,A] s.t. H=<abb,BaB,A>).’
    G=algorithm(l) # defines a graph G which is an immersion into Rn.
    T=maximal_tree(G) # finds the maximal tree of the graph G.
    list_of_loops=[] # collects all the loops in G as path1-edge-inverse(path2)
    vertices=list(T.keys()) # vertices of maximal tree
    for i in range(len(vertices)):
        v=vertices[i]
        miss=missing(v,G)
    for i in range(len(vertices)):
        v=vertices[i]
        if deg_in_max(v,T)!=deg_in_G(v,G):#if degree of v in G and T is the same => v will be part of som
            if v==’’: # then in G vertex ’’ is part of some loop. All other neighbours of ’’ are in T.
                loops=G.get((’’,’’),0)
                if loops!=0:
                    list_of_loops.append(loops)
```

```
            else: # then there exists a vetrex w adjacent to v in G but not in T.
                miss=missing(v,G) # denotes the dictionary of missing edges in T for which v is part of i
                edges=list(miss.keys())
                for j in range(len(edges)):
                    edge=edges[j]
                    colours=list(miss.get(edge))
                    for k in range(len(colours)):
                        c=colours[k]
                        t={edge:c}
                        loop=create_a_loop(t,T)
                        if loop not in list_of_loops:
                             list_of_loops.append(loop)
    generators=[]
    for i in range(len(list_of_loops)):
        a=list(list_of_loops[i])[0]
        generators.append(a)
    return generators
# Example:
# find_generators(l)
# >>> ['BBAb', 'BA', 'Baaa', 'ba', 'AbAb']
# Reminder: Capital letters are equivalent to inverses.
```