Group 4 - Final Project

Mining Frequent Subgraphs

Karla Mahmoud Alvarez Paul Jackson, Diaka Kuhufiwa, Harry Fox Jochowitz
CDSC-GW Data Mining
Professor Stefano Nicoletti

NOTE: This project is currently using only two datasets created by us.
Please, see the "Datasets.pdf" contains information about the test datasets.

Please Follow, docs/ and run tests.

Setup Methods:

In [28]:
```
import copy
import gzip
import itertools

# create list of graphs in gzipped directory
def get_graph_database(dir_path):
    dir_path = os.path.expanduser(dir_path)
    graph_database = []
    print("Loading...")

    for path in graph_paths:
        graph_database.append(nxg.LoadEdgelist(nxg.TWOGraph, path, ' ', ?))

    return graph_database

# return all graph where vx in list
def list_graph(graphs):
    graph_list = []

    if graph.GetNodes() == 0:
        print("Error: Graph is empty")
        graph_list.append(N.Graph())
    else:
        for E in graph.Edges():
            if not graph.IsNode(E.GetSrcNid(), E.GetDstNid()):
                graph_list.append(nxd.subgraph(graph, [E.GetSrcNid(), E.GetDstNid()]))

    return graph_list

# write graphs and their sequences / prime divt
def write_to_file(graphs):
    for G in graphs:
        for edge in G.Edges():
            print("%d%d", E.GetSrcNid(), E.GetDstNid())

    for vx in G.Nodes():
        print("%d", vx.GetId(), vx.GetDeg(), G[vx])

# return size if given graphs are the same
def compare_graphs(G, G2):

    if G1.GetNodes() == G2.GetNodes() and G1.GetEdges() == G2.GetEdges():
        return False

    for E in G1.Edges():
        if not G2.IsNode(E.GetSrcNid(), E.GetDstNid()):
            return False

    for E in G2.Edges():
        if not G1.IsNode(E.GetSrcNid(), E.GetDstNid()):
            return True

    return True
```

Node-based Join Growth:

In [29]:
```
# Node-based Join Growth

#FIND:(get_all_node_sequence)
#This function finds all nodes and an label repletion in any of the graphs.
#That is, some of the graphs have more than one node with it.

# return dict with all nodes in a set their support
def get_all_node_sequence(graph_database):
    dict = {}

    for graph in graph_database:
        for E in graph.Edges():
            if E.GetSrcNid() == E:
                if E.GetSrcNid() in dict:
                    dict[E.GetSrcNid()] += 1
                else:
                    dict[E.GetSrcNid()] = 1
    return dict

# return dict with frequent singleton graphs and their sequences
def get_frequent_singleton_graph(dict, minsup):
    dict = {}

    for k in dk:
        if dk[k] >= minsup:
            singleton = snap.TNEGraph.New(1, 0)#create new graph
            singleton.AddNode(k)# add frequent node
            fi[singleton] = dk[k]# create dict sequence + node support

    return fi

# return candidates by joining singletons
def join_singletons(singleton, subgraph):
    # create a list of all nodes in the given subgraph
    l = snap.TCnComV(subgraph).GetSubVec(subgraph)
    cand = []
    for n in l:
        c.AddNode(subgraph.GetId(n))# add subgraph node to subgraph
        c.AddEdge(singleton.GetId(0), subgraph.GetId(n))# add edge between nodes
    return c

#FIND: (subgraph_match)
#The back recursively climes a bigraphic for this,
#which is recursive. I am not a fan recursion so I made the following,
#which might not be as efficient but done the job.

# return size and edge if they are the only non-matching node
def subgraph_match(G1, G2):
    edge_1 = False # non-matching node found
    edge_2 = False # non-matching edge found
    edge = False # edge found
    node = False # non-matching edge

    res = False

    for n in G1.Nodes():
        if not G2.IsNode(n.GetId()):
            if edge_1:
                edge = False
                break
            else:
                edge = G2.GetDeg(n.GetId(), N.GetInDeg())
                edge_1 = True
    if edge_2:
        for n in G2.Nodes():
            if not G1.IsNode(n.GetId()):
                if edge_1:
                    edge = False
                    break
                else:
                    edge = G1.GetDeg(n.GetId(), N.GetInDeg())
                    edge_1 = True
    if edge_1 and edge:
        res = True

    return res, edge

# return true if all k-1 subgraphs of candidate are in fk
def prune_candidates(c, fk):
    all_subs = []

    for k in c.Nodes():
        c_minus = c.GetNbrNodes(k)
        c_minus.AddNode(k)
        l_minus = snap.TCnComV(c_minus.GetConnectedCompVec(snap.IsWeak, 1ull_nullptr))

    for subsets in l_minus:
        if l_minus[subsets].GetNodes() == c.GetNodes():
            for graph in fk:
                if compare_graphs(l_minus[subsets], graph):
                    return True
    return False

# return candidates by performing node-based join
def generate_candidates(G, minsup, subgraph1, NEW, NAME):

    # create new candidate c subgraph
    c1 = snap.TUNGraph.New()
    c2 = snap.TUNGraph.New()
    # add nodes from subgraph1 to subgraph2
    for n in subgraph1.Nodes():
        c1.AddNode(subgraph1.GetId(n))
        c2.AddNode(subgraph1.GetId(n))

    c1 = snap.ConvertSubGraph(type(subgraph1), subgraph1)# copy subgraph1
    c1.AddNode(NAME, NEW)# add new candidate node
    c2 = snap.ConvertSubGraph(type(subgraph1), subgraph1)

    c = snap.ConvertSubGraph(type(c1), c1)# copy candidate c
    c.AddEdge(NAME, NEW,)# add edge between non-matching nodes of subgraph1 and subgraph2

    return c1, c2

#FIND:(generate_candidates)
#This method is incomplete i.e. no optimization/pruning of candidates

# return candidates list by joining graphs in fk
def generate_candidates(fk):
    candidates = []

    for i in range(0, len(fk)):
        ant = fk[i]

        for j in range(i+1, len(fk)):
            con = fk[j]

            # if graphs are from fk, simply join them
            if k == 1:
                c = join_singletons(ant, con)
                candidates.append(c)
            # if graphs are from fk where k > 1,
            else:
                # join if they match have only one uncommon node
                match, node, node2 = subgraph_match(ant, con)
                if match:
                    for n2 in con.GetNodes(node2), node2:
                        if n2 == node:
                            if prune_candidates(c, fk):
                                candidates.append(c)
    return candidates

# generate frequent k+1 edge graphs dict by counting c in g
def generate_fk(graph, c, minsup):
    fk = {}
    supports = 0
    support_in_subgraph = False

    for candidate in c:
        for graph in g:
            # check if graph contains all candidate's nodes
            for n in candidate.Nodes():
                if not graph.IsNode(n.GetId):
                    support_in_subgraph = False
                    break
            # check if graph contains all candidate's edges
            if support_in_subgraph:
                for E in candidate.Edges():
                    if not graph.IsEdge(E.GetSrcNid(), E.GetDstNid()):
                        support_in_subgraph = False
                        break
            # increment support
            if support_in_subgraph:
                supports += 1
            # reset for next graph
            support_in_subgraph = True

        if supports >= minsup:
            fk[candidate] = supports
        supports = 0 # reset supports for next candidate

    return fk

#FIND: (update_fk)
#This method is not optimized because it is not necessary.
#But it is used because if there varies in the graphs,
#it will update the frequent graphs.

# return the latest updated subgraphs and in fed from fk
def update_fk(c, fk):
    for k in fk:
        if c == k:
            fk[k] = c

        if fk[k].IsEqual(c):
            fk[k] = fk[k]

    return fk

# update dict with frequent subgraphs and support
def node_based_join_growth(g, minsup):

    # k is 1 and k-1 candidates
    k = 1 # k is 1 for the first run
    c = {} # c is frequent singleton graphs c
    fk = get_frequent_singleton_graph(dk, minsup)# # frequent 1 subgraphs

    # print("Frequent singleton graphs", fk)

    fmk = {} # k-1 frequent subgraphs
    c = {} # candidates from fk

    # Apriori Algorithm
    while fk:
        c = generate_candidates(list(fk.keys()), k+1)
        fk = generate_fk(list(c), g, minsup)

        # add c to more frequent subgraphs
        for k in fk:
            break
        fmk = update_fk(fk, fmk)# append all frequent subgraphs

        k = k + 1

    return fmk
```

Edge-Based Join Growth:

In [30]:
```
# Edge-based Join Growth

#FIND:(get_all_edge_sequence)
#This function finds all edges in any sequence.
#That is, the edge should matched and double-matched are considered the same.

# return dict with all edges in graph database and their sequence
def get_all_edge_sequence(graph_database):
    dict = {}

    for graph in graph_database:
        for E in graph.Edges():
            if E.GetSrcNid() <= E.GetDstNid():
                edge = (E.GetSrcNid(), E.GetDstNid())
                if edge in dict:
                    dict[edge] += 1
                elif reverse_edge(edge) in dk:
                    dict[reverse_edge(edge)] += 1
                else:
                    dict[edge] = 1
    return dk

# return all edges that have at least minsup support
def get_frequent_singleton_graph_EB(dk, minsup):
    dict = {}

    for k in dk:
        if dk[k] >= minsup:
            singleton = snap.TNEGraph.New(1, 0)#create new graph
            singleton.AddNode(k[0])
            singleton.AddNode(k[1])
            singleton.AddEdge(k[0], k[1])
            fi[singleton] = dk[k]

    return fi

#FIND: (subgraph_match_EB)
#The back recursively climes a bigraphic for this,
#which is recursive. I am not a fan recursion so I made the following,
#which might not be as efficient but done the job.

# return size and edge if they are the only non-matching node
def subgraph_match_EB(G1, edge):
    edge_1 = False # non-matching node found
    edge_2 = False # non-matching edge

    for n in G1.Nodes():
        if not G1.IsNode(n.GetId(), n.GetDeg(0)):
            if edge_1:
                edge_1 = False
                break
            else:
                edge = n.GetDeg(0)
                edge_1 = True
    if edge_1:
        if not G1.IsNode(edge[0]) and not G1.IsNode(edge[1]):
            edge = False
            edge = edge[0], edge[1]
        elif not G1.IsNode(edge[0]):
            edge = edge[0]
        elif not G1.IsNode(edge[1]):
            edge = edge[1]
    return False, False, False

# return candidates by performing node-based join
def join_subgraphs_EB(singleton, subgraph1, node, NAME):

    c = snap.ConvertSubGraph(type(subgraph1), subgraph1)

    if node in c.Nodes:
        c.AddNode(NAME)
        c.AddEdge(node[0], node[1])

    return c

# return true if all k-1 subgraphs of candidate are in fk
def prune_candidates_EB(c, fk):
    all_subs = []

    for k in c.Nodes():
        c_minus = c.GetNbrNodes(k)

    # generate all node subsets
    l_minus = snap.TCnComV(c_minus.GetConnectedCompVec(snap.IsWeak, 1ull_nullptr))

    for subsets in l_minus:
        if l_minus[subsets].GetNodes() == c.GetNodes():
            for graph in fk:
                if compare_graphs(l_minus[subsets], graph):
                    return True
    return False

# return candidates list by joining graphs in fk
def generate_candidates_EB(fk, k):
    candidates = []

    for i in range(0, len(fk)):
        ant = fk[i]

        for j in range(i+1, len(fk)):
            con = fk[j]

            # if graphs are from fk where k > 1,
            match, node, node2 = subgraph_match_EB(ant, con)
            if match:
                for n2 in con.GetNodes(node2), node2:
                    if n2 == node:
                        if prune_candidates_EB(c, fk):
                            candidates.append(c)
    return candidates

# generate frequent k+1 edge graphs dict by counting c in g
def generate_fk_EB(graph, c, minsup):
    fk = {}
    supports = 0
    support_in_subgraph = False

    for candidate in c:
        for graph in g:
            # check if graph contains all candidate's nodes
            for n in candidate.Nodes():
                if not graph.IsNode(n.GetId):
                    support_in_subgraph = False
                    break
            # check if graph contains all candidate's edges
            if support_in_subgraph:
                for E in candidate.Edges():
                    if not graph.IsEdge(E.GetSrcNid(), E.GetDstNid()):
                        support_in_subgraph = False
                        break
            # increment support
            if support_in_subgraph:
                supports += 1
            # reset for next graph
            support_in_subgraph = True

        if supports >= minsup:
            fk[candidate] = supports
        supports = 0 # reset supports for next candidate

    return fk

#FIND: (update_fk)
#This method is not optimized because it is not necessary.
#But it is used because if there varies in the graphs,
#it will update the frequent graphs.

# return the latest updated subgraphs and in fed from fk
def update_fk(c, fk):
    for k in fk:
        if c == k:
            fk[k] = c

        if fk[k].IsEqual(c):
            fk[k] = fk[k]

    return fk

# update dict with frequent subgraphs and support
def edge_based_join_growth(g, minsup):

    # k is 1 and k-1 candidates - (candidates - frequent) c
    k = 1 # k is 1 for the first run
    c = {} # c is frequent singleton graphs c
    fk = get_frequent_singleton_graph_EB(dk, minsup)

    fmk = {} # k-1 frequent subgraphs
    c = {} # c candidates from fk

    # Apriori Algorithm
    while fk:
        c = generate_candidates_EB(list(fk.keys), k+1)
        fk = generate_fk_EB(list(c), g, minsup)

        # add c to more frequent subgraphs
        for k in fk:
            break
        fmk = update_fk(fk, fmk)# append all frequent subgraphs

        k = k + 1

    return fmk
```

Main:

The following call is used to set parameters (i.e. minimum support and input graphs).

In [31]:
```
# minsup = minimum support
minsup = 1
# g = graph_database
g = get_graph_database('./datasets/test-graphs.txt')

# The following call uses node-based join growth to mine frequent subgraphs and displays the final result.

In [32]: # fmk = frequent subgraphs (with Node-based Join Growth)
fmk = node_based_join_growth(g, minsup)

# Print Results
print("Number of graphs:")
print("Total", len(fmk))

Graphs: (1)1, Edges(0): 0
Graphs: (2)1, Edges(1): 0
Graphs: (3)1, Edges(2): 0
Graphs: (4)1, Edges(3): 0
Graphs: (5)1, Edges(4): 0
Graphs: (6)1, Edges(5): 0
Graphs: (7)1, Edges(6): 0
Graphs: (8)1, Edges(7): 0
Graphs: (9)1, Edges(8): 0
Graphs: (10)1, Edges(9): 0
Graphs:
```