In [1]:

```
import numpy as np
```

Preprocessing Graph G

In [2]:

```
Graph_label_class = {}
# format: {'A': ['1', '2'], 'B': ['3', '4']}
Graph_edge = {}
# format: {'1': {'A': ['3', '4'], 'B': ['2']}, '2': {'A': ['1']}...}
Graph_label = {}
# format: {'1': 'A', '2': 'A'...}

testfile = open("test.txt", "r")
for string in testfile:
    # three types of lines
    # 1. "int"
    # 2. "index label\n" -- denotes a vertex's label
    # 3. "index index\n" -- denotes a edge between vertex
    line = string.split(' ')
    if len(line) > 1:
        # case 2 or 3
        first_part = line[0]
        second_part = line[1]
        if second_part[-1] == "\n":
            second_part = second_part[:-1]
        if second_part.isdigit():
            # case 3
            f_l = Graph_label[first_part]
            s_l = Graph_label[second_part]
            if s_l not in Graph_edge[first_part]:
                Graph_edge[first_part][s_l] = []
            Graph_edge[first_part][s_l].append(second_part)
            if f_l not in Graph_edge[second_part]:
                Graph_edge[second_part][f_l] = []
            Graph_edge[second_part][f_l].append(first_part)
        else:
            Graph_label[first_part] = second_part
            template = {}
            Graph_edge[first_part] = {}
            if second_part not in Graph_label_class:
                Graph_label_class[second_part] = []
            Graph_label_class[second_part].append(first_part)

testfile.close()
```

Preprocessing Query q

In [3]:

```
q_edge = {}
q_label = {}

testfile = open("testquery.txt", "r")
for string in testfile:
    line = string.split(' ')
    if len(line) > 1:
        first_part = line[0]
        second_part = line[1]
        if second_part[-1] == "\n":
            second_part = second_part[:-1]
        if second_part.isdigit():
            f_l = q_label[first_part]
            s_l = q_label[second_part]
            if s_l not in q_edge[first_part]:
                q_edge[first_part][s_l] = []
            q_edge[first_part][s_l].append(second_part)
            if f_l not in q_edge[second_part]:
                q_edge[second_part][f_l] = []
            q_edge[second_part][f_l].append(first_part)
        else:
            q_label[first_part] = second_part
            template = {}
            q_edge[first_part] = {}

testfile.close()
```

BuildDAG(q, G)

find the root: prefer the root to have a small number of candidates in G and to have a large degree for better pruning

In [10]:

```
array = []
for i in q_label.keys():
    count = 0
    for j in q_edge[i].values():
        count += len(j)
    array.append(len(Graph_label_class[q_label[i]]) / count)

start_key = list(q_label.keys())[np.argmin(array)] # root key for q_D
tail_key = [] # root key for q_D_i
```

create DAG tree

In [11]:

```
from itertools import chain

def deep_copy(lib):
    ret = {}
    for i in lib.keys():
        ret[i] = {}
        for j in lib[i].keys():
            ret[i][j] = lib[i][j].copy()
    return ret

#pop and push for array
def q_pop(queue):
    temp = queue[0]
    queue.pop(0)
    return temp

def q_push(queue, element_list, bag):
    for element in element_list:
        if element in bag:
            # haven't added in queue before, truely is child
            queue += element
            bag.remove(element)
        else:
            print(element)
            element_list.remove(element)
    return

def degree(elem, library):
    return len(list(chain.from_iterable(list(library[elem].values()))))
```

In [26]:

```
q_D = deep_copy(q_edge)
q_D_i = deep_copy(q_edge)

root_queue = [start_key]
child_queue = []
root_id = 0

bag = list(q_label.keys()).copy()
# bag: ['1', '2, '3', '4'], used to make sure things added in bag before can't be ac

while len(root_queue) + len(child_queue) > 0:
    current = root_queue[root_id]
    groups = q_D[current]
    # groups: {'A': ['1'], 'C': ['3', '2'], 'D': ['4']}
    for groups_1 in range(len(groups)):
        label = list(groups.keys())[groups_1]
        # label: 'A'
```

```
        # label: 'A'

        # delete relationships within layers
        for vertex in groups[label]:
            # vertex: '3'
            if vertex not in root_queue:
                # this is a child, but possibly already in child_queue by other root
                if vertex not in child_queue:
                    child_queue += [vertex]
#                 print(current, vertex, q_label[current])
                q_D[vertex][q_label[current]].remove(current)
                q_D_i[current][q_label[vertex]].remove(vertex)

    root_id += 1
    if root_id == len(root_queue):
        # in one layer
        for elem_1 in child_queue:
            for elem_2 in child_queue:
                label1 = q_label[elem_1]
                label2 = q_label[elem_2]

                if label1 == label2:
                    # within a label
                    degree1 = degree(elem_1, q_edge)
                    degree2 = degree(elem_2, q_edge)
                    if degree1 < degree2 or (degree1 == degree2 and elem_1 < elem_2)
                        if label2 in q_D[elem_1] and elem_2 in q_D[elem_1][label2]:
                            q_D[elem_1][label2].remove(elem_2)
                            q_D_i[elem_2][label1].remove(elem_1)
                else:
                    # within groups
                    freq_label1 = len(Graph_label_class[label1])
                    freq_label2 = len(Graph_label_class[label2])
                    if freq_label1 > freq_label2 or (freq_label1 == freq_label2 and
                        if label2 in q_D[elem_1] and elem_2 in q_D[elem_1][label2]:
                            q_D[elem_1][label2].remove(elem_2)
                            q_D_i[elem_2][label1].remove(elem_1)

        tail_key = root_queue
        root_queue = child_queue
        child_queue = []
        root_id = 0
```

In [ ]:

```
```

BuildCS

initial CS: LG(v) = Lq(u) and degG(v) ≥ degq (u)

In [30]:

```
CS = q_label.copy()
for key in CS:
    label = CS[key]
    CS[key] = []
    base_deg = degree(key, q_edge)
    for G_key in Graph_label_class[label]:
        if degree(G_key, Graph_edge) >= base_deg:
            CS[key].append(G_key)
```

define the refine function

In [31]:

```
def refine(qD, key, cs):
    queue = key.copy()
    marked = []
    while len(queue) > 0:
        node = q_pop(queue)
        if node in marked:
            continue
        marked.append(node)
        node_connected = list(chain.from_iterable(list(qD[node].values())))
        for node2 in node_connected:
            label = q_label[node2]
            if node2 not in marked:
                queue += [node2]
            for candidate in cs[node]:
                c_neighbor = Graph_edge[candidate][label]
                count = 0
                for cs_filter in cs[node2]:
                    if cs_filter in c_neighbor:
                        count += 1
                        break
                if count == 0:
                    cs[node].remove(candidate)
```

refine initial CS three round, each round: q_D_i, q_D

In [32]:

```
for i in range(3):
    refine(q_D_i, tail_key, CS)
    refine(q_D, [start_key], CS)
```

DAG

assign weight to all node's candidates

In [ ]:

In [ ]:

In [ ]: