In [1]:

```
import numpy as np
from itertools import chain
from queue import *
```

Preprocessing

Preprocessing Graph G

important variables: Graph_label_class Graph_edge Graph_label

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'...}
Graph_vt_degree = {}
# format: {'1': 6, '2': 9...}

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()

for node in Graph_edge:
    Graph_vt_degree[node] = len(list(chain.from_iterable(list(Graph_edge[node].valu
```

Preprocessing query q

In [3]:

```
q_edge = {}
q_label = {}
q_vt_degree = {}
# format: {'1': 6, '2': 9...}

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()

for node in q_edge:
    q_vt_degree[node] = len(list(chain.from_iterable(list(q_edge[node].values()))))
```

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 [4]:

```
array = []
for i in q_label.keys():
    degree = q_vt_degree[i]
    Candidates = [x for x in Graph_label_class[q_label[i]] if Graph_vt_degree[x] >=
    array.append(len(Candidates) / degree)
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 [5]:

```
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

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

In [6]:

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

In [7]:

```
# use three ordering, compare order layer->q_label->q_vt_degree
order_layer = {} #format: {vertex: 1}

reference_label = {}
for label in Graph_label_class:
    reference_label[label] = len(Graph_label_class[label])

queue = Queue(maxsize=0)
mark = set()
def BFS_layer(node):
    queue.put(node)
    queue.put('*')
    layer = 0
    mark.add(node)
    while queue.empty() == False:
        current = queue.get()
        if current == '*':
            #the layer ended
            layer += 1
            if queue.empty() == False:
                queue.put('*')
        else:
            order_layer[current] = layer
            for con_vt in list(chain.from_iterable(list(q_edge[current].values()))):
                if con_vt not in mark:
                    queue.put(con_vt)
                    mark.add(con_vt)
```

In [8]:

```
#Run only once!!!
BFS_layer(start_key)
```

In [9]:

```
def points_to(key1, key2):
    # In q_D, return True if key1->key2
    if order_layer[key1] != order_layer[key2]:
        return order_layer[key1] < order_layer[key2]
    if q_label[key1] != q_label[key2]:
        if reference_label[q_label[key1]] != reference_label[q_label[key2]]:
            return reference_label[q_label[key1]] < reference_label[q_label[key2]]
        else:
            return q_label[key1] < q_label[key2]
    if q_vt_degree[key1] != q_vt_degree[key2]:
        return q_vt_degree[key1] > q_vt_degree[key2]
    return key1 > key2
```

In [10]:

```
#Run only once!!!
for start in q_edge:
    check_tail = 0
    for label in q_edge[start]:
        for end in q_edge[start][label]:
            if points_to(start, end):
                q_D_i[start][label].remove(end)
                check_tail += 1
            else:
                q_D[start][label].remove(end)
    if check_tail == 0:
        tail_key.append(start)
```

BuildCS

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

In [21]:

```
CS = {}
for i in q_label.keys():
    degree = q_vt_degree[i]
    Candidates = [x for x in Graph_label_class[q_label[i]] if Graph_vt_degree[x] >=
    CS[i] = Candidates
```

define the refine function

In [12]:

```
def refine(qD, key, cs):
    queue = Queue(maxsize=0)
    for root in key:
        queue.put(root)
    marked = set()
    while queue.empty() == False:
        node = queue.get()
        if node in marked:
            continue
        marked.add(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.put(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 [22]:

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

Backtracking

Use Path-size order; assign weight to all node's candidates:

```
1. topological sort
(1.5) compute which nodes in q has only one parent
2. dynamic programming
```

step 1

In [45]:

```
mark = set()
q_sort_result = []
```

In [46]:

```
def topsort(v):
    mark.add(v)
    for children_label in q_D[v]:
        for child in q_D[v][children_label]:
            if child not in mark:
                topsort(child)
    global q_sort_result
    q_sort_result.append(v)
topsort(start_key)
```

step 1.5

In [28]:

```
q_nodes_with_one_parent = []
do_not_add_again = []
for i in q_D:
    for j in q_D[i]:
        for k in q_D[i][j]:
            if k in do_not_add_again:
                continue
            if k in q_nodes_with_one_parent:
                q_nodes_with_one_parent.remove(k)
                do_not_add_again.append(k)
                continue
            q_nodes_with_one_parent.append(k)
```

step 2

In [49]:

```
def common_member(a, b):
    a_set = set(a)
    b_set = set(b)
    if len(a_set.intersection(b_set)) > 0:
        return(a_set.intersection(b_set))
    else:
        return([])
```

In [61]:

```
weight = {}
W_M = {}
# ('2' -- u in q, '3' -- v in G) = weight
for u in q_sort_result:
    child_q = list(chain.from_iterable(list(q_D[u].values())))
    c_list = common_member(q_nodes_with_one_parent, child_q)
    if len(c_list) == 0:
        u_ = 0
        for v in CS[u]:
            weight[(u,v)] = 1
            u_ += 1
        W_M[u] = u_
    else:
        u_ = 0
        for v in CS[u]:
            u_v = np.inf
            for c in c_list:
                temp = 0
                common = common_member(list(chain.from_iterable(list(Graph_edge[v].v
                for ci in common:
                    temp += weight[(c,ci)]
                if temp < u_v:
                    u_v = temp
            weight[(u,v)] = u_v
            u_ += weight[(u,v)]
        W_M[u] = u_
```

backtracking

In [116]:

```
def Backtrack(M, bag):
    if len(M) == len(q_label):
        print(M)
    elif len(M) == 0:
        for v in CS[start_key]:
            M = set()
            M.add((start_key, v))
            bag.add(v)
            Backtrack(M, bag)
            bag.remove(v)
    else:
        extendable = start_key
        min_weight = np.inf
        for u in q_D_i:
            if any(key[0] == u for key in M):
                continue
            b = list(chain.from_iterable(list(q_D_i[u].values())))
            if all(any(key[0] == key2 for key in M) for key2 in b) and W_M[u] < min_
                min_weight = W_M[u]
                extendable = u
        for v in CS[extendable]:
            if v not in bag:
                M.add((extendable, v))
                bag.add(v)
                Backtrack(M, bag)
                bag.remove(v)
```

In [117]:

```
M = set()
bag = set()
Backtrack(M, bag)
```

```
{('2', '3'), ('3', '5'), ('4', '10'), ('1', '1')}
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: