In [3]:

```
from itertools import chain,combinations
import time
from collections import defaultdict

filename="data.txt"
sup = 881

with open(filename) as f:
    database = f.readlines()

database = [x.strip() for x in database]

T,freq_sets = [],defaultdict(int)

for i in range(0,len(database)): T.append(database[i].split())

def apriori(L1,sup):
    k = 0
    L = [L1]
    while(len(L[k])>0):
        # self joining
        Ck_plus1 = []
        lk = len(L[k])
        for i in range(lk):
            for j in range(i+1,lk):
                if L[k][i][:k] == L[k][j][:k]:
                    a,b = min(L[k][i][k],L[k][j][k]),max(L[k][i][k],L[k][j][k])
                    Ck_plus1.append(L[k][i][:k]+[a,b])
        # pruning
        pruned_Ck_plus1 = []
        for candidate in Ck_plus1:
            k_sets = list(map(list,set(combinations(set(candidate),k+1)))) # all k s
            flag = 1
            for i in range(len(k_sets)):
                if list(sorted(k_sets[i])) not in L[k]: flag = 0
            if flag: pruned_Ck_plus1.append(candidate)

        Ck_plus1_sup = defaultdict(int)
        for item in T1:
            for item_1 in pruned_Ck_plus1:
                if (set(item_1).issubset(set(item))):
                    Ck_plus1_sup[tuple(item_1)] += 1
        Lk_plus1_new = []
        for items in Ck_plus1_sup:
            if (Ck_plus1_sup[items] >= sup):
                Lk_plus1_new.append(sorted(list(items)))
                freq_sets[tuple(sorted(list(items)))] = Ck_plus1_sup[items]
        L.append(Lk_plus1_new)
        k += 1
    return L

def assoRule(Set, minConf):
    for itemSet in Set:
        subsets = chain.from_iterable(combinations(itemSet,r) for r in range(1,len(i
        itemSetSup = freq_sets[tuple(sorted(list(itemSet)))]
        for s in subsets:
            conf = float(itemSetSup / freq_sets[tuple(sorted(list(s)))])
            if(conf > minConf):
                print("{} => {} at confidence {}".format(set(s), set(itemSet.differe

start=time.time()
C1,L1,n = defaultdict(int),[],len(T)
```

```
for i in range(n):
    for j in range(len(T[i])):
        elem = T[i][j]
        C1[elem] += 1

for elem in C1.keys():
    if C1[elem] >= sup:
        L1.append([elem])
        freq_sets[(elem,)] = C1[elem]

T1 = []
for i in range(n):
    tmp = []
    for j in range(len(T[i])):
        if [T[i][j]] in L1: tmp.append(T[i][j])
    T1.append(tmp)

L = apriori(L1,sup)
print(len(freq_sets))

freq_set = []
for i in range(len(L)):
    for j in range(len(L[i])):
        freq_set.append(set(L[i][j]))
assoRule(freq_set,0.8)

end=time.time()
print("\n Total Time taken is {} seconds".format( end-start))
```

```
159
{'36'} => {'38'} at confidence 0.9502724795640327
{'37'} => {'38'} at confidence 0.9739292364990689
{'110'} => {'38'} at confidence 0.9753042233357194
{'170'} => {'38'} at confidence 0.9780574378831881
{'286'} => {'38'} at confidence 0.9433643279797126
{'39', '36'} => {'38'} at confidence 0.9548355424644085
{'48', '36'} => {'38'} at confidence 0.96045197740113
{'41', '48'} => {'39'} at confidence 0.8168108227988468
{'110', '39'} => {'38'} at confidence 0.9891984081864695
{'39', '170'} => {'38'} at confidence 0.9805730937348227
{'48', '170'} => {'38'} at confidence 0.9877970456005138
{'225', '48'} => {'39'} at confidence 0.8064516129032258
{'110', '48'} => {'38'} at confidence 0.986231884057971
{'48', '39', '36'} => {'38'} at confidence 0.967741935483871
{'38', '41', '48'} => {'39'} at confidence 0.8386689132266217
{'48', '39', '170'} => {'38'} at confidence 0.9892205638474295
{'110', '48', '39'} => {'38'} at confidence 0.9942140790742526

 Total Time taken is 90.23220777511597 seconds
```

In [2]:

```
print(freq_sets)
```

```
defaultdict(<class 'int'>, {('9',): 1372, ('19',): 1005, ('31',): 920, ('32',): 1516
7, ('36',): 2936, ('37',): 1074, ('38',): 15596, ('39',): 50675, ('41',): 14945, ('4
5',): 911, ('48',): 42135, ('49',): 1120, ('60',): 1489, ('65',): 4472, ('78',): 106
0, ('79',): 1600, ('89',): 3837, ('101',): 2237, ('110',): 2794, ('117',): 1026, ('1
23',): 1302, ('147',): 1779, ('161',): 1010, ('170',): 3099, ('175',): 970, ('17
9',): 998, ('185',): 1376, ('201',): 1133, ('225',): 3257, ('237',): 3032, ('242',):
911, ('249',): 1160, ('255',): 1474, ('258',): 987, ('264',): 895, ('270',): 1734,
('271',): 2094, ('286',): 1183, ('301',): 1204, ('310',): 2594, ('338',): 1274, ('41
3',): 1880, ('438',): 1863, ('475',): 2167, ('479',): 926, ('522',): 974, ('533',):
1487, ('548',): 1137, ('589',): 1119, ('592',): 1227, ('604',): 1209, ('677',): 111
0, ('740',): 1181, ('783',): 965, ('824',): 1210, ('956',): 911, ('1004',): 1102,
('1146',): 1426, ('1327',): 1786, ('1393',): 1161, ('2238',): 1715, ('2958',): 904,
('3270',): 950, ('10515',): 882, ('12925',): 1467, ('13041',): 1051, ('14098',): 129
1, ('15832',): 1143, ('16010',): 1316, ('16217',): 1166, ('36', '38'): 2790, ('36',
```

'39'): 2037, ('37', '38'): 1046, ('38', '39'): 10345, ('38', '41'): 3897, ('39', '4
1'): 11414, ('38', '48'): 7944, ('39', '48'): 29142, ('32', '41'): 3196, ('39', '7
9'): 1111, ('36', '48'): 1416, ('41', '48'): 9018, ('48', '79'): 893, ('39', '89'):
2749, ('101', '39'): 1400, ('48', '89'): 2798, ('101', '48'): 1311, ('110', '38'): 2
725, ('110', '39'): 1759, ('147', '39'): 1137, ('147', '48'): 1036, ('170', '38'): 3
031, ('170', '39'): 2059, ('170', '48'): 1557, ('32', '39'): 8455, ('32', '48'): 803
4, ('32', '38'): 2833, ('39', '65'): 2787, ('237', '39'): 1929, ('48', '65'): 2529,
('255', '39'): 1057, ('255', '48'): 1057, ('225', '39'): 2351, ('270', '39'): 1194,
('271', '39'): 1434, ('237', '48'): 1682, ('286', '38'): 1116, ('310', '39'): 1852,
('39', '60'): 983, ('41', '65'): 995, ('413', '48'): 1135, ('39', '438'): 1260, ('43
8', '48'): 1025, ('39', '475'): 1500, ('475', '48'): 1428, ('225', '48'): 1736, ('27
0', '48'): 957, ('310', '48'): 1692, ('110', '48'): 1380, ('39', '533'): 922, ('114
6', '39'): 983, ('271', '48'): 1090, ('1327', '39'): 1156, ('1327', '48'): 968, ('3
9', '413'): 1130, ('2238', '48'): 955, ('2238', '39'): 1287, ('12925', '39'): 938,
('36', '38', '39'): 1945, ('38', '39', '41'): 3051, ('38', '39', '48'): 6102, ('36',
'38', '48'): 1360, ('36', '39', '48'): 1116, ('38', '41', '48'): 2374, ('39', '41',
'48'): 7366, ('39', '48', '89'): 2125, ('101', '39', '48'): 946, ('110', '38', '3
9'): 1740, ('170', '38', '39'): 2019, ('170', '38', '48'): 1538, ('170', '39', '4
8'): 1206, ('32', '39', '41'): 2359, ('32', '41', '48'): 2063, ('32', '39', '48'): 5
402, ('32', '38', '39'): 1840, ('32', '38', '48'): 1646, ('39', '48', '65'): 1797,
('237', '39', '48'): 1244, ('39', '475', '48'): 1092, ('225', '39', '48'): 1400, ('1
10', '38', '48'): 1361, ('110', '39', '48'): 1037, ('310', '39', '48'): 1347, ('36',
'38', '39', '48'): 1080, ('38', '39', '41', '48'): 1991, ('170', '38', '39', '48'):
1193, ('32', '39', '41', '48'): 1646, ('32', '38', '39', '48'): 1236, ('110', '38',
'39', '48'): 1031})

In [1]:

```
from itertools import chain, combinations
import time
from collections import defaultdict


class FPNode:
    def __init__(self,ID,ancestor,frequency):
        self.ID = ID   #id of the fpnode class item
        self.descendent = {} #child of the fpnode class item
        self.frequency = frequency #frequency of the fpnode class item
        self.Ancestor = ancestor #parent of the fpnode class item
        self.N_Link = None #link node of the fpnode class

def construct_FP(S_min,DataBase):
    Reference_Table = {}
    for row in DataBase:
        for element in row:
            # if item is not in reference table, we will add it from database
            if(Reference_Table.get(element)==None):
                Reference_Table[element]=DataBase[row]
            # if item is in reference table, we will update it using database
            else:
                Reference_Table[element] = Reference_Table.get(element) + DataBase[r
    #now,we will delete elements having support less than minimum support
    for element in list(Reference_Table):
        if Reference_Table[element] < S_min: del(Reference_Table[element])
   # now we form frequent 1 itemset i.e. recurrent_set
    recurrent_set = set(Reference_Table.keys())
    # if no recurrent set , return none
    n=len(recurrent_set)
    if n == 0: return None, None

    # now create reference table in the form of [element,{frequency,nodelink}]
    for element in Reference_Table:
        Reference_Table[element] = [Reference_Table[element], None]

    # initialize fptree using null set
    FPTREE = FPNode('Null Set',None,1)

    #now,we will again scan the data second time and make the fptree
    for SET,freq in DataBase.items():
        recurrent_items = {}
        for element in SET:
            if element in recurrent_set: recurrent_items[element] = Reference_Table[
        if len(recurrent_items) != 0:
            #now we will order the elements in the decreasing order of frequency cou
            ordered_SET = [v[0] for v in sorted(recurrent_items.items(), key=lambda
            # now we will update the fpTree
            SET,tree=ordered_SET,FPTREE
            n,i=len(SET),0
            while(i<n):
                # if element is not in tree.descendent, we will make a new node for
                if SET[i] not in tree.descendent:
                    tree.descendent[SET[i]] = FPNode(SET[i],tree,freq)
                    if Reference_Table[SET[i]][1] == None: Reference_Table[SET[i]][1
                    else:
                        Test_Node=Reference_Table[SET[i]][1]
                        # we will update the link node here
                        while (Test_Node.N_Link != None):
                            Test_Node = Test_Node.N_Link
                        Test_Node.N_Link = tree.descendent[SET[i]]
                else:
```

```
                    tree.descendent[SET[i]].frequency +=freq
                if i!= n-1: tree=tree.descendent[SET[i]]
                i =i+1
    return FPTREE, Reference_Table

def assoRule(Set,frequency_set,minimum_confidence):
    for i in range(len(Set)):
        subsets = chain.from_iterable(combinations(Set[i],r) for r in range(1,len(Se
        itemSetSup = frequency_set[i]
        for s in subsets:
            conf = float(itemSetSup / frequency_set[Set.index(set(s))])
            if(conf > minimum_confidence):
                print("{} => {} at confidence {}".format(set(s), set(Set[i].differen

def FP_Mine(tree, Reference_Table, S_min, pathprefix, recurrent_set,frequency_set):
    # now we will mine the given fptree
    y=sorted(Reference_Table.items(),key=lambda p: p[1][0])
     # list contain items and their frequency from the reference table
    List = [[x[0],x[1][0]] for x in y]
    for L in List:
        new_recurrentset = pathprefix.copy() # we will copy old prefix path
        new_recurrentset.add(L[0]) # add element in set
        frequency_set.append(L[1]) # add corrosponding frquency in the set
        recurrent_set.append(new_recurrentset)
        #define new fpnode
        FPNode=Reference_Table[L[0]][1]
        cp_base = {}  # conditional_patterns_base
        #we will find prefixpath for the given element
        while FPNode != None:
            path_prefix = []
            LNode=FPNode
            while(LNode.Ancestor !=None):
                path_prefix.append(LNode.ID)
                LNode=LNode.Ancestor
            if len(path_prefix) != 1: cp_base[tuple(path_prefix[1:])] = FPNode.frequ
            FPNode = FPNode.N_Link
            # we will remove element from fptree which have support less than suppor
        c_tree, c_header = construct_FP(S_min,cp_base)
        if c_header != None: FP_Mine(c_tree, c_header, S_min, new_recurrentset, recu

def main(input_file,S_min):
    # now we will open our file
    with open(input_file) as file:
        DataBase = file.readlines()
    elements = [row.strip() for row in DataBase]
    Tns = []
    n=len(elements)

    for i in range(n): Tns.append(elements[i].split())
    Dataset = defaultdict(int)
    for row in Tns: Dataset[tuple(row)] += 1
    recurrent_set,frequency_set = [],[]
    tree, Reference_Table = construct_FP(S_min,Dataset) #tree making
    FP_Mine(tree, Reference_Table, S_min, set([]), recurrent_set,frequency_set)  #mi
    print("Frequent itemsets are:")
    for i in range(len(frequency_set)):
        print(" {} with frequency  {}".format(recurrent_set[i],frequency_set[i]))
    print("\n Total frequent items are {} \n".format(len(recurrent_set)))
    print("Association rules are:")
    assoRule(recurrent_set,frequency_set,0.8)


start= time.time()
main("data.txt",881)
```

```
end=time.time()
print("\n Total Time taken is {} seconds".format( end-start))
```

```
Frequent itemsets are:
 {'10515'} with frequency  882
 {'264'} with frequency  895
 {'2958'} with frequency  904
 {'45'} with frequency  911
 {'242'} with frequency  911
 {'956'} with frequency  911
 {'31'} with frequency  920
 {'479'} with frequency  926
 {'3270'} with frequency  950
 {'783'} with frequency  965
 {'175'} with frequency  970
 {'522'} with frequency  974
 {'258'} with frequency  987
 {'179'} with frequency  998
 {'19'} with frequency  1005
 {'161'} with frequency  1010
 {'117'} with frequency  1026
 {'13041'} with frequency  1051
 {'78'} with frequency  1060
 {'37'} with frequency  1074
 {'38', '37'} with frequency  1046
 {'1004'} with frequency  1102
 {'677'} with frequency  1110
 {'589'} with frequency  1119
 {'49'} with frequency  1120
 {'201'} with frequency  1133
 {'548'} with frequency  1137
 {'15832'} with frequency  1143
 {'249'} with frequency  1160
 {'1393'} with frequency  1161
 {'16217'} with frequency  1166
 {'740'} with frequency  1181
 {'286'} with frequency  1183
 {'38', '286'} with frequency  1116
 {'301'} with frequency  1204
 {'604'} with frequency  1209
 {'824'} with frequency  1210
 {'592'} with frequency  1227
 {'338'} with frequency  1274
 {'14098'} with frequency  1291
 {'123'} with frequency  1302
 {'16010'} with frequency  1316
 {'9'} with frequency  1372
 {'185'} with frequency  1376
 {'1146'} with frequency  1426
 {'1146', '39'} with frequency  983
 {'12925'} with frequency  1467
 {'12925', '39'} with frequency  938
 {'255'} with frequency  1474
 {'255', '48'} with frequency  1057
 {'255', '39'} with frequency  1057
 {'533'} with frequency  1487
 {'39', '533'} with frequency  922
 {'60'} with frequency  1489
 {'60', '39'} with frequency  983
 {'79'} with frequency  1600
 {'79', '48'} with frequency  893
 {'79', '39'} with frequency  1111
 {'2238'} with frequency  1715
 {'2238', '48'} with frequency  955
 {'2238', '39'} with frequency  1287
 {'270'} with frequency  1734
 {'270', '48'} with frequency  957
 {'270', '39'} with frequency  1194
 {'147'} with frequency  1779
```

```
{'147', '48'} with frequency  1036
{'147', '39'} with frequency  1137
{'1327'} with frequency  1786
{'1327', '48'} with frequency  968
{'39', '1327'} with frequency  1156
{'438'} with frequency  1863
{'438', '48'} with frequency  1025
{'438', '39'} with frequency  1260
{'413'} with frequency  1880
{'413', '39'} with frequency  1130
{'413', '48'} with frequency  1135
{'271'} with frequency  2094
{'271', '48'} with frequency  1090
{'39', '271'} with frequency  1434
{'475'} with frequency  2167
{'475', '48'} with frequency  1428
{'475', '39', '48'} with frequency  1092
{'475', '39'} with frequency  1500
{'101'} with frequency  2237
{'48', '101'} with frequency  1311
{'48', '39', '101'} with frequency  946
{'39', '101'} with frequency  1400
{'310'} with frequency  2594
{'310', '48'} with frequency  1692
{'310', '39', '48'} with frequency  1347
{'310', '39'} with frequency  1852
{'110'} with frequency  2794
{'48', '110'} with frequency  1380
{'48', '39', '110'} with frequency  1037
{'48', '39', '38', '110'} with frequency  1031
{'48', '38', '110'} with frequency  1361
{'39', '110'} with frequency  1759
{'39', '38', '110'} with frequency  1740
{'38', '110'} with frequency  2725
{'36'} with frequency  2936
{'36', '48'} with frequency  1416
{'36', '39', '48'} with frequency  1116
{'36', '39', '38', '48'} with frequency  1080
{'36', '38', '48'} with frequency  1360
{'36', '39'} with frequency  2037
{'36', '39', '38'} with frequency  1945
{'36', '38'} with frequency  2790
{'237'} with frequency  3032
{'237', '48'} with frequency  1682
{'237', '39', '48'} with frequency  1244
{'237', '39'} with frequency  1929
{'170'} with frequency  3099
{'170', '48'} with frequency  1557
{'170', '39', '48'} with frequency  1206
{'170', '39', '38', '48'} with frequency  1193
{'170', '38', '48'} with frequency  1538
{'170', '39'} with frequency  2059
{'170', '39', '38'} with frequency  2019
{'170', '38'} with frequency  3031
{'225'} with frequency  3257
{'225', '48'} with frequency  1736
{'225', '39', '48'} with frequency  1400
{'225', '39'} with frequency  2351
{'89'} with frequency  3837
{'39', '89'} with frequency  2749
{'39', '89', '48'} with frequency  2125
{'89', '48'} with frequency  2798
{'65'} with frequency  4472
{'65', '41'} with frequency  995
{'65', '48'} with frequency  2529
{'39', '65', '48'} with frequency  1797
{'39', '65'} with frequency  2787
{'41'} with frequency  14945
{'32', '41'} with frequency  3196
```

```
 {'48', '32', '41'} with frequency  2063
 {'48', '39', '32', '41'} with frequency  1646
 {'39', '32', '41'} with frequency  2359
 {'38', '41'} with frequency  3897
 {'48', '38', '41'} with frequency  2374
 {'48', '38', '39', '41'} with frequency  1991
 {'38', '39', '41'} with frequency  3051
 {'48', '41'} with frequency  9018
 {'48', '39', '41'} with frequency  7366
 {'39', '41'} with frequency  11414
 {'32'} with frequency  15167
 {'38', '32'} with frequency  2833
 {'38', '32', '48'} with frequency  1646
 {'38', '32', '39', '48'} with frequency  1236
 {'38', '32', '39'} with frequency  1840
 {'32', '48'} with frequency  8034
 {'39', '32', '48'} with frequency  5402
 {'39', '32'} with frequency  8455
 {'38'} with frequency  15596
 {'38', '48'} with frequency  7944
 {'38', '39', '48'} with frequency  6102
 {'38', '39'} with frequency  10345
 {'48'} with frequency  42135
 {'39', '48'} with frequency  29142
 {'39'} with frequency  50675

 Total frequent items are 159

Association rules are:
{'37'} => {'38'} at confidence 0.9739292364990689
{'286'} => {'38'} at confidence 0.9433643279797126
{'110', '39', '48'} => {'38'} at confidence 0.9942140790742526
{'110', '48'} => {'38'} at confidence 0.986231884057971
{'39', '110'} => {'38'} at confidence 0.9891984081864695
{'110'} => {'38'} at confidence 0.9753042233357194
{'36', '39', '48'} => {'38'} at confidence 0.967741935483871
{'36', '48'} => {'38'} at confidence 0.96045197740113
{'36', '39'} => {'38'} at confidence 0.9548355424644085
{'36'} => {'38'} at confidence 0.9502724795640327
{'170', '39', '48'} => {'38'} at confidence 0.9892205638474295
{'170', '48'} => {'38'} at confidence 0.9877970456005138
{'170', '39'} => {'38'} at confidence 0.9805730937348227
{'170'} => {'38'} at confidence 0.9780574378831881
{'225', '48'} => {'39'} at confidence 0.8064516129032258
{'41', '38', '48'} => {'39'} at confidence 0.8386689132266217
{'41', '48'} => {'39'} at confidence 0.8168108227988468

 Total Time taken is 1.5916128158569336 seconds
```

In [ ]:

In [ ]:

2.1 12 / 12

- **2 pts** Partial Correct

✓ **- 0 pts** **NA**

Modification in Apriori algorithm

We have modified the Apriori algorithm to implement the AprioriTID algorithm. The difference between the two algorithms is in the step of counting the support of the candidate frequent item sets. While the Apriori algorithm makes a pass over the dataset for each candidate, AprioriTID algorithm does not use the database for counting the support of candidates after the 1st pass. In AprioriTID algorithm we use a vector containing potentially frequent k-item sets with id = TID. If c is a candidate k-itemset such that the k-1 itemset obtained by removing the last entry and the k-1 itemset obtained by removing the second to last entry both belong to the potential k-1 frequent vector with id t, then c is a k frequent itemset.

Code for counting support in Apriori algorithm:

```
Ck_plus1_sup = defaultdict(int)
    for item in T1:
      for item_1 in pruned_Ck_plus1:
        if (set(item_1).issubset(set(item))):
          Ck_plus1_sup[tuple(item_1)] += 1
```

Code for counting support in AprioriTID algorithm:

```
for t in C[k]:
      C_t = []
      for item in pruned_Ck_plus1:
        if (sorted(list(item[:-1])) in t) and (sorted(list(item[:-2])+[item[-1]]) in t):
          C_t.append(sorted(item))
          C_t_sup[tuple(sorted(item))] += 1
      if (C_t != []):
        Ck_plus1_prime.append(C_t)
```

Therefore, we only make a single pass over the dataset in case of AprioriTID algorithm, resulting in good reduction in the overall space complexity.

One disadvantage of the AprioriTID algorithm is that for small values of k, the potential k frequent vector can be larger than the database itself.

CONCLUSION

Time taken by Apriori algorithm = 90.23s

Time taken by AprioriTID algorithm = 131.45s

In [1]:

```
from itertools import chain,combinations
import time
from collections import defaultdict

filename="data.txt"
sup = 881

with open(filename) as f:
    database = f.readlines()

database = [x.strip() for x in database]

T,freq_sets = [],defaultdict(int)

for i in range(0,len(database)): T.append(database[i].split())

def aprioriTID(L1,sup,C1_prime):
    k = 0
    L = [L1]
    C = [C1_prime]
    while(len(L[k])>0):
        # self joining
        Ck_plus1 = []
        lk = len(L[k])
        for i in range(lk):
            for j in range(i+1,lk):
                if L[k][i][:k] == L[k][j][:k]:
                    a,b = min(L[k][i][k],L[k][j][k]),max(L[k][i][k],L[k][j][k])
                    Ck_plus1.append(L[k][i][:k]+[a,b])
        # pruning
        pruned_Ck_plus1 = []
        for candidate in Ck_plus1:
            k_sets = list(map(list,set(combinations(set(candidate),k+1)))) # all k s
            flag = 1
            for i in range(len(k_sets)):
                if list(sorted(k_sets[i])) not in L[k]: flag = 0
            if flag: pruned_Ck_plus1.append(candidate)
        Ck_plus1_prime,C_t_sup = [],defaultdict(int)
        for t in C[k]:
            C_t = []
            for item in pruned_Ck_plus1:
                if (sorted(list(item[:-1])) in t) and (sorted(list(item[:-2])+[item[
                    C_t.append(sorted(item))
                    C_t_sup[tuple(sorted(item))] += 1
            if (C_t != []):
                Ck_plus1_prime.append(C_t)
        Lk_plus1_new = []
        for items in pruned_Ck_plus1:
            if (C_t_sup[tuple(sorted(items))] >= sup):
                freq_sets[tuple(sorted(items))] = C_t_sup[tuple(sorted(items))]
                Lk_plus1_new.append(items)
        L.append(Lk_plus1_new)
        C.append(Ck_plus1_prime)
        k += 1
    return L

def assoRule(Set, minConf):
    for itemSet in Set:
        subsets = chain.from_iterable(combinations(itemSet, r) for r in range(1, len
        itemSetSup = freq_sets[tuple(sorted(list(itemSet)))]
        for s in subsets:
            confidence = float(itemSetSup / freq_sets[tuple(sorted(list(s)))])
```

```
            if(confidence > minConf):
                print("{} => {} at confidence {}".format(set(s), set(itemSet.differe

start=time.time()
C1,L1,n,C1_prime = defaultdict(int),[],len(T),[]
for i in range(n):
    for j in range(len(T[i])):
        elem = T[i][j]
        C1[elem] += 1
C1_prime = [[[col] for col in row] for row in T]
for elem in C1.keys():
    if C1[elem] >= sup:
        L1.append([elem])
        freq_sets[(elem,)] = C1[elem]

T1 = []
for i in range(n):
    tmp = []
    for j in range(len(T[i])):
        if [T[i][j]] in L1: tmp.append(T[i][j])
    T1.append(tmp)

L = aprioriTID(L1,sup,C1_prime)
print(len(freq_sets))

freq_set = []
for i in range(len(L)):
    for j in range(len(L[i])):
        freq_set.append(set(L[i][j]))
assoRule(freq_set,0.8)

end=time.time()
print("\n Total Time taken is {} seconds".format( end-start))
```

```
159
{'36'} => {'38'} at confidence 0.9502724795640327
{'37'} => {'38'} at confidence 0.9739292364990689
{'110'} => {'38'} at confidence 0.9753042233357194
{'170'} => {'38'} at confidence 0.9780574378831881
{'286'} => {'38'} at confidence 0.9433643279797126
{'39', '36'} => {'38'} at confidence 0.9548355424644085
{'48', '36'} => {'38'} at confidence 0.96045197740113
{'110', '39'} => {'38'} at confidence 0.9891984081864695
{'110', '48'} => {'38'} at confidence 0.986231884057971
{'170', '39'} => {'38'} at confidence 0.9805730937348227
{'170', '48'} => {'38'} at confidence 0.9877970456005138
{'41', '48'} => {'39'} at confidence 0.8168108227988468
{'225', '48'} => {'39'} at confidence 0.8064516129032258
{'39', '36', '48'} => {'38'} at confidence 0.967741935483871
{'41', '38', '48'} => {'39'} at confidence 0.8386689132266217
{'110', '39', '48'} => {'38'} at confidence 0.9942140790742526
{'170', '48', '39'} => {'38'} at confidence 0.9892205638474295

 Total Time taken is 131.4547417163849 seconds
```

In [2]:

```
print(freq_sets)
```

```
defaultdict(<class 'int'>, {('9',): 1372, ('19',): 1005, ('31',): 920, ('32',): 1516
7, ('36',): 2936, ('37',): 1074, ('38',): 15596, ('39',): 50675, ('41',): 14945, ('4
5',): 911, ('48',): 42135, ('49',): 1120, ('60',): 1489, ('65',): 4472, ('78',): 106
0, ('79',): 1600, ('89',): 3837, ('101',): 2237, ('110',): 2794, ('117',): 1026, ('1
23',): 1302, ('147',): 1779, ('161',): 1010, ('170',): 3099, ('175',): 970, ('17
9',): 998, ('185',): 1376, ('201',): 1133, ('225',): 3257, ('237',): 3032, ('242',):
911, ('249',): 1160, ('255',): 1474, ('258',): 987, ('264',): 895, ('270',): 1734,
('271',): 2094, ('286',): 1183, ('301',): 1204, ('310',): 2594, ('338',): 1274, ('41
```

```
3',): 1880, ('438',): 1863, ('475',): 2167, ('479',): 926, ('522',): 974, ('533',):
1487, ('548',): 1137, ('589',): 1119, ('592',): 1227, ('604',): 1209, ('677',): 111
0, ('740',): 1181, ('783',): 965, ('824',): 1210, ('956',): 911, ('1004',): 1102,
('1146',): 1426, ('1327',): 1786, ('1393',): 1161, ('2238',): 1715, ('2958',): 904,
('3270',): 950, ('10515',): 882, ('12925',): 1467, ('13041',): 1051, ('14098',): 129
1, ('15832',): 1143, ('16010',): 1316, ('16217',): 1166, ('32', '38'): 2833, ('32',
'39'): 8455, ('32', '41'): 3196, ('32', '48'): 8034, ('36', '38'): 2790, ('36', '3
9'): 2037, ('36', '48'): 1416, ('37', '38'): 1046, ('38', '39'): 10345, ('38', '4
1'): 3897, ('38', '48'): 7944, ('110', '38'): 2725, ('170', '38'): 3031, ('286', '3
8'): 1116, ('39', '41'): 11414, ('39', '48'): 29142, ('39', '60'): 983, ('39', '6
5'): 2787, ('39', '79'): 1111, ('39', '89'): 2749, ('101', '39'): 1400, ('110', '3
9'): 1759, ('147', '39'): 1137, ('170', '39'): 2059, ('225', '39'): 2351, ('237', '3
9'): 1929, ('255', '39'): 1057, ('270', '39'): 1194, ('271', '39'): 1434, ('310', '3
9'): 1852, ('39', '413'): 1130, ('39', '438'): 1260, ('39', '475'): 1500, ('39', '53
3'): 922, ('1146', '39'): 983, ('1327', '39'): 1156, ('2238', '39'): 1287, ('12925',
'39'): 938, ('41', '48'): 9018, ('41', '65'): 995, ('48', '65'): 2529, ('48', '79'):
893, ('48', '89'): 2798, ('101', '48'): 1311, ('110', '48'): 1380, ('147', '48'): 10
36, ('170', '48'): 1557, ('225', '48'): 1736, ('237', '48'): 1682, ('255', '48'): 10
57, ('270', '48'): 957, ('271', '48'): 1090, ('310', '48'): 1692, ('413', '48'): 113
5, ('438', '48'): 1025, ('475', '48'): 1428, ('1327', '48'): 968, ('2238', '48'): 95
5, ('32', '38', '39'): 1840, ('32', '38', '48'): 1646, ('32', '39', '41'): 2359, ('3
2', '39', '48'): 5402, ('32', '41', '48'): 2063, ('36', '38', '39'): 1945, ('36', '3
8', '48'): 1360, ('36', '39', '48'): 1116, ('38', '39', '41'): 3051, ('38', '39', '4
8'): 6102, ('38', '41', '48'): 2374, ('110', '38', '39'): 1740, ('110', '38', '48'):
1361, ('170', '38', '39'): 2019, ('170', '38', '48'): 1538, ('39', '41', '48'): 736
6, ('39', '48', '65'): 1797, ('39', '48', '89'): 2125, ('39', '475', '48'): 1092,
('101', '39', '48'): 946, ('110', '39', '48'): 1037, ('170', '39', '48'): 1206, ('22
5', '39', '48'): 1400, ('237', '39', '48'): 1244, ('310', '39', '48'): 1347, ('32',
'38', '39', '48'): 1236, ('32', '39', '41', '48'): 1646, ('36', '38', '39', '48'): 1
080, ('38', '39', '41', '48'): 1991, ('110', '38', '39', '48'): 1031, ('170', '38',
'39', '48'): 1193})
```

Modification in FPtree algorithm

We have modified the FPtree algorithm to implement the projected database FPtree algorithm. The difference between the two algorithm is that in the case of FPtree algorithm we are using the complete database to construct the FPtree of our algorithm but in the case of projected database FPtree algorithm we are partitioning the database into several projected databases and then for each projected database we construct and mining the corresponding FPtree. It is because disk size of a computer is fixed and for large data it is unrealistic that FPtree algorithm will work in such a case.

x-Projected database: x-Projected database contains frequent items such that items with frequency less than x are not included and item x is also not included.

Thus, it leads to much better space optimization.

Code for projected database:

```
DBS=[]
RTable=sorted(Reference_Table.items(),key=lambda p: p[1][0], reverse=True)
for i in range(len(RTable)):
    element1 =RTable[i][0]
    PDB={}
    y=DataBase.copy()
    y.clear()
    y=DataBase.copy()
    for Set,freq in y.items():
        if element1 in Set:
            PDB[Set]=freq
            del(DataBase[Set])
    DBS.append(PDB)
```

CONCLUSION

Time taken by FPtree algorithm: 1.59s

Time taken by projected database FPtree algorithm:7.89s

In [1]:

```
from itertools import chain, combinations
import time
from collections import defaultdict


class FPNode:
    def __init__(self,ID,ancestor,frequency):
        self.ID = ID   #id of the fpnode class item
        self.descendent = {} #child of the fpnode class item
        self.frequency = frequency #frequency of the fpnode class item
        self.Ancestor = ancestor #parent of the fpnode class item
        self.N_Link = None #link node of the fpnode class

def projected_Construct_FP(S_min,DataBase):
    Reference_Table = {}
    for row in DataBase:
        for element in row:
            # if item is not in reference table, we will add it from database
            if(Reference_Table.get(element)==None):
                Reference_Table[element]=DataBase[row]
            # if item is in reference table, we will update it using database
            else:
                Reference_Table[element] = Reference_Table.get(element) + DataBase[r
    #now,we will delete elements having support less than minimum support
    for element in list(Reference_Table):
        if Reference_Table[element] < S_min: del(Reference_Table[element])
   # now we form frequent 1 itemset i.e. recurrent_set
    recurrent_set = set(Reference_Table.keys())
    # if no recurrent set , return none
    n=len(recurrent_set)
    if n == 0: return None, None

    # now create reference table in the form of [element,{frequency,nodelink}]
    for element in Reference_Table:
        Reference_Table[element] = [Reference_Table[element], None]


   ###################################################################333#################
    # now we will break database into projected database
    DBS=[]
    RTable=sorted(Reference_Table.items(),key=lambda p: p[1][0],reverse=True)
    for i in range(len(RTable)):
        element1 =RTable[i][0]
        PDB={}
        y=DataBase.copy()
        y.clear()
        y=DataBase.copy()
        for Set,freq in y.items():
            if element1 in Set:
                PDB[Set]=freq
                del(DataBase[Set])
        DBS.append(PDB)



    FTREES=[]
    RT=[]
    for i in range(len(DBS)):
        database=DBS[i]
        R_Table=Reference_Table
        # initialize fptree using null set
        FPTREE = FPNode('Null Set',None,1)
```

```
        #now,we will again scan the data second time and make the fptree
        for SET,freq in database.items():
            recurrent_items = {}
            for element in SET:
                if element in recurrent_set: recurrent_items[element] = R_Table[elem
            if len(recurrent_items) != 0:
                #now we will order the elements in the decreasing order of frequency
                ordered_SET = [v[0] for v in sorted(recurrent_items.items(), key=lam
                # now we will update the fpTree
                SET,tree=ordered_SET,FPTREE
                n,i=len(SET),0
                while(i<n):
                    # if element is not in tree.descendent, we will make a new node
                    if SET[i] not in tree.descendent:
                        tree.descendent[SET[i]] = FPNode(SET[i],tree,freq)
                        if R_Table[SET[i]][1] == None: R_Table[SET[i]][1] = tree.des
                        else:
                            Test_Node=R_Table[SET[i]][1]
                            # we will update the link node here
                            while (Test_Node.N_Link != None):
                                Test_Node = Test_Node.N_Link
                            Test_Node.N_Link = tree.descendent[SET[i]]
                    else:
                         tree.descendent[SET[i]].frequency +=freq
                    if i!= n-1: tree=tree.descendent[SET[i]]
                    i =i+1
        FTREES.append(FPTREE)
        RT.append(R_Table)
    ###############################################################################
    return FTREES, RT

def assoRule(Set,frequency_set,minimum_confidence):
    for i in range(len(Set)):
        subsets = chain.from_iterable(combinations(Set[i],r) for r in range(1,len(Se
        itemSetSup = frequency_set[i]
        for s in subsets:
            conf = float(itemSetSup / frequency_set[Set.index(set(s))])
            if(conf > minimum_confidence):
                print("{} => {} at confidence {}".format(set(s), set(Set[i].differen

def FP_Mine(tree, Reference_Table, S_min, pathprefix, recurrent_set,frequency_set):
    # now we will mine the given fptree
    y=sorted(Reference_Table.items(),key=lambda p: p[1][0])
     # list contain items and their frequency from the reference table
    List = [[x[0],x[1][0]] for x in y]
    for L in List:
        new_recurrentset = pathprefix.copy() # we will copy old prefix path
        new_recurrentset.add(L[0]) # add element in set
        if (new_recurrentset not in recurrent_set):
            frequency_set.append(L[1]) # add corrosponding frquency in the set
            recurrent_set.append(new_recurrentset)
        #define new fpnode
        FPNode=Reference_Table[L[0]][1]
        cp_base = {}  # conditional_patterns_base
        #we will find prefixpath for the given element
        while FPNode != None:
            path_prefix = []
            LNode=FPNode
            while(LNode.Ancestor !=None):
                path_prefix.append(LNode.ID)
                LNode=LNode.Ancestor
            if len(path_prefix) != 1: cp_base[tuple(path_prefix[1:])] = FPNode.frequ
            FPNode = FPNode.N_Link
            # we will remove element from fptree which have support less than suppor
```

```
        c_tree, c_header = projected_Construct_FP(S_min,cp_base)
        if c_header != None: FP_Mine(c_tree, c_header[0], S_min, new_recurrentset, r

def main(input_file,S_min):
    # now we will open our file
    with open(input_file) as file:
        DataBase = file.readlines()
    elements = [row.strip() for row in DataBase]
    Tns = []
    n=len(elements)

    for i in range(n): Tns.append(elements[i].split())
    Dataset = defaultdict(int)
    for row in Tns: Dataset[tuple(row)] += 1
    recurrent_set,frequency_set = [],[]
    trees, Reference_Table = projected_Construct_FP(S_min,Dataset) #tree making
    for i in range(len(trees)):
        FP_Mine(trees[i], Reference_Table[i], S_min, set([]), recurrent_set,frequenc
    print("Frequent itemsets are:")
    for i in range(len(frequency_set)):
        print(" {} with frequency  {}".format(recurrent_set[i],frequency_set[i]))
    print("\n Total frequent items are {} \n".format(len(recurrent_set)))
    print("Association rules are:")
    assoRule(recurrent_set,frequency_set,0.8)


start= time.time()
main("data.txt",881)
end=time.time()
print("\n Total Time taken is {} seconds".format( end-start))
```

```
Frequent itemsets are:
 {'10515'} with frequency  882
 {'264'} with frequency  895
 {'2958'} with frequency  904
 {'45'} with frequency  911
 {'242'} with frequency  911
 {'956'} with frequency  911
 {'31'} with frequency  920
 {'479'} with frequency  926
 {'3270'} with frequency  950
 {'783'} with frequency  965
 {'175'} with frequency  970
 {'522'} with frequency  974
 {'258'} with frequency  987
 {'179'} with frequency  998
 {'19'} with frequency  1005
 {'161'} with frequency  1010
 {'117'} with frequency  1026
 {'13041'} with frequency  1051
 {'78'} with frequency  1060
 {'37'} with frequency  1074
 {'38', '37'} with frequency  1046
 {'1004'} with frequency  1102
 {'677'} with frequency  1110
 {'589'} with frequency  1119
 {'49'} with frequency  1120
 {'201'} with frequency  1133
 {'548'} with frequency  1137
 {'15832'} with frequency  1143
 {'249'} with frequency  1160
 {'1393'} with frequency  1161
 {'16217'} with frequency  1166
 {'740'} with frequency  1181
 {'286'} with frequency  1183
 {'38', '286'} with frequency  1116
```

```
{'301'} with frequency  1204
{'604'} with frequency  1209
{'824'} with frequency  1210
{'592'} with frequency  1227
{'338'} with frequency  1274
{'14098'} with frequency  1291
{'123'} with frequency  1302
{'16010'} with frequency  1316
{'9'} with frequency  1372
{'185'} with frequency  1376
{'1146'} with frequency  1426
{'39', '1146'} with frequency  983
{'12925'} with frequency  1467
{'39', '12925'} with frequency  938
{'255'} with frequency  1474
{'48', '255'} with frequency  1057
{'39', '255'} with frequency  1057
{'533'} with frequency  1487
{'533', '39'} with frequency  922
{'60'} with frequency  1489
{'39', '60'} with frequency  983
{'79'} with frequency  1600
{'79', '48'} with frequency  893
{'39', '79'} with frequency  1111
{'2238'} with frequency  1715
{'48', '2238'} with frequency  955
{'39', '2238'} with frequency  1287
{'270'} with frequency  1734
{'270', '48'} with frequency  957
{'270', '39'} with frequency  1194
{'147'} with frequency  1779
{'147', '48'} with frequency  1036
{'147', '39'} with frequency  1137
{'1327'} with frequency  1786
{'1327', '48'} with frequency  968
{'39', '1327'} with frequency  1156
{'438'} with frequency  1863
{'438', '48'} with frequency  1025
{'438', '39'} with frequency  1260
{'413'} with frequency  1880
{'39', '413'} with frequency  1130
{'48', '413'} with frequency  1135
{'271'} with frequency  2094
{'271', '48'} with frequency  1090
{'271', '39'} with frequency  1434
{'475'} with frequency  2167
{'475', '48'} with frequency  1428
{'475', '39', '48'} with frequency  1092
{'475', '39'} with frequency  1500
{'101'} with frequency  2237
{'101', '48'} with frequency  1311
{'39', '101', '48'} with frequency  946
{'39', '101'} with frequency  1400
{'310'} with frequency  2594
{'310', '48'} with frequency  1692
{'39', '310', '48'} with frequency  1347
{'39', '310'} with frequency  1852
{'110'} with frequency  2794
{'110', '48'} with frequency  1380
{'39', '110', '48'} with frequency  1037
{'38', '39', '110', '48'} with frequency  1031
{'38', '110', '48'} with frequency  1361
{'39', '110'} with frequency  1759
{'38', '39', '110'} with frequency  1740
{'38', '110'} with frequency  2725
{'36'} with frequency  2936
{'48', '36'} with frequency  1416
{'39', '48', '36'} with frequency  1116
{'38', '39', '48', '36'} with frequency  1080
```

```
{'38', '48', '36'} with frequency  1360
{'39', '36'} with frequency  2037
{'38', '39', '36'} with frequency  1945
{'38', '36'} with frequency  2790
{'237'} with frequency  3032
{'237', '48'} with frequency  1682
{'39', '237', '48'} with frequency  1244
{'39', '237'} with frequency  1929
{'170'} with frequency  3099
{'170', '48'} with frequency  1557
{'170', '39', '48'} with frequency  1206
{'170', '39', '48', '38'} with frequency  1193
{'170', '48', '38'} with frequency  1538
{'170', '39'} with frequency  2059
{'170', '39', '38'} with frequency  2019
{'170', '38'} with frequency  3031
{'225'} with frequency  3257
{'225', '48'} with frequency  1736
{'39', '225', '48'} with frequency  1400
{'39', '225'} with frequency  2351
{'89'} with frequency  3837
{'89', '39'} with frequency  2749
{'89', '48', '39'} with frequency  2125
{'89', '48'} with frequency  2798
{'65'} with frequency  4472
{'65', '41'} with frequency  995
{'65', '48'} with frequency  2529
{'39', '65', '48'} with frequency  1797
{'39', '65'} with frequency  2787
{'41'} with frequency  14945
{'32', '41'} with frequency  3196
{'32', '48', '41'} with frequency  2063
{'32', '39', '48', '41'} with frequency  1646
{'32', '39', '41'} with frequency  2359
{'38', '41'} with frequency  3897
{'38', '48', '41'} with frequency  2374
{'38', '39', '48', '41'} with frequency  1991
{'38', '39', '41'} with frequency  3051
{'48', '41'} with frequency  9018
{'39', '48', '41'} with frequency  7366
{'39', '41'} with frequency  11414
{'32'} with frequency  15167
{'38', '32'} with frequency  2833
{'38', '32', '48'} with frequency  1646
{'38', '32', '39', '48'} with frequency  1236
{'38', '32', '39'} with frequency  1840
{'32', '48'} with frequency  8034
{'32', '39', '48'} with frequency  5402
{'32', '39'} with frequency  8455
{'38'} with frequency  15596
{'38', '48'} with frequency  7944
{'38', '39', '48'} with frequency  6102
{'38', '39'} with frequency  10345
{'48'} with frequency  42135
{'39', '48'} with frequency  29142
{'39'} with frequency  50675

 Total frequent items are 159

Association rules are:
{'37'} => {'38'} at confidence 0.9739292364990689
{'286'} => {'38'} at confidence 0.9433643279797126
{'39', '110', '48'} => {'38'} at confidence 0.9942140790742526
{'110', '48'} => {'38'} at confidence 0.986231884057971
{'39', '110'} => {'38'} at confidence 0.9891984081864695
{'110'} => {'38'} at confidence 0.9753042233357194
{'39', '48', '36'} => {'38'} at confidence 0.967741935483871
{'48', '36'} => {'38'} at confidence 0.96045197740113
{'39', '36'} => {'38'} at confidence 0.9548355424644085
```

```
{'36'} => {'38'} at confidence 0.9502724795640327
{'170', '39', '48'} => {'38'} at confidence 0.9892205638474295
{'170', '48'} => {'38'} at confidence 0.9877970456005138
{'170', '39'} => {'38'} at confidence 0.9805730937348227
{'170'} => {'38'} at confidence 0.9780574378831881
{'225', '48'} => {'39'} at confidence 0.8064516129032258
{'38', '48', '41'} => {'39'} at confidence 0.8386689132266217
{'48', '41'} => {'39'} at confidence 0.8168108227988468

 Total Time taken is 7.8900182247161865 seconds
```

In [ ]:

2.2 **2 / 3**

✓ **- 1 pts** Partial Correct

- 0 pts NA