Student name: Thien Toan Tran
Student ID: A1808080

Assignment 3 Exercise 1

In [22]:

```
from itertools import combinations
import random
import os
from datetime import datetime
import time
from multiprocessing import Pool
```

In [23]:

```
dataPath = './data/'
fileDataPath = []
fileDataNamesOnly = []
for a in os.listdir(dataPath):
    if a.endswith('.dat'):
        fileDataPath.append(dataPath + a)
        fileDataNamesOnly.append(a.split('.')[0])

fileDataPath = sorted(fileDataPath)
fileDataNamesOnly = sorted(fileDataNamesOnly)
fileDataPath
```

Out[23]:

```
['./data/T10I4D100K.dat',
 './data/T40I10D100K.dat',
 './data/chess.dat',
 './data/connect.dat',
 './data/mushroom.dat',
 './data/pumsb.dat',
 './data/pumsb_star.dat']
```

In [24]:

```
dataLineCounts = dict()
for path in fileDataPath:
    dataLineCounts[path] = sum(1 for line in open(path))
dataLineCounts
```

Out[24]:

```
{'./data/T10I4D100K.dat': 100000,
 './data/T40I10D100K.dat': 100000,
 './data/chess.dat': 3196,
 './data/connect.dat': 67557,
 './data/mushroom.dat': 8124,
 './data/pumsb.dat': 49046,
 './data/pumsb_star.dat': 49046}
```

Exercise 1: Frequent Itemsets

1.1. Implement the simple, randomized algorithm given in 6.4.1

Suppose need to randomly select n out of m baskets in the entire file, where $\frac{n}{m} = p$

My random algorithm will be implemented as follow:

1. select random subset of entire dataset
   - For each of basket in data file, a random float in the range [0.0, 1.0) will be generated. It will represent the selection probability for this basket.
   - If the probability $\geq 1 - p$, then this basket will be selected for further process
2. Full itemset is initialized as a Python dictionary, where key is an itemset, value is its count on the subset
3. For each basket data, generate a set of itemset and update the full itemset
   - since the singleton and pairs are likely to dominate, only those 2 types are generated
   - for each generated itemset,
     - if it exists in the full itemset dictionary, then its value is increased by 1
     - otherwise, create new record using itself as key and value is 1
4. count itemset over the subset
5. sort itemsets and get most frequent itemset in the subset
   - lower the confident to ps, since the original data has expected confident s and we only process a portion p of the data
   - lower the confident more, like $0.9ps$, if there is enough memory. By doing that, the result is more likely to achieve the confident s in the whole dataset
   - equivalently change to least support count $0.9psm$

In [25]:

```python
def randomSelect(m,p):
    """
    Randomly select ids based on given probability

    :param m: dataset's record
    :param p: probability
    :return: list of selected ids
    """
    result = []
    for i in range(m):
        if random.random().__ge__(1-p):
            result.append(i)
    return result
```

Test random select result

In [26]:

```
list_m = [100,500,1000,10000]
list_p = [0.5,0.8,0.3]
for m in list_m:
    for p in list_p:
        a = randomSelect(m, p)
        expect = m*p
        selected = len(a)
        pc = (expect- selected)/expect
        print(f'm={m}, p={p}: expected: {expect}, result: {selected}, delta = {pc
    print()

```

```
m=100, p=0.5: expected: 50.0, result: 51, delta = -2.0%
m=100, p=0.8: expected: 80.0, result: 71, delta = 11.25%
m=100, p=0.3: expected: 30.0, result: 31, delta = -3.3333333333333335%

m=500, p=0.5: expected: 250.0, result: 268, delta = -7.199999999999999%
m=500, p=0.8: expected: 400.0, result: 395, delta = 1.25%
m=500, p=0.3: expected: 150.0, result: 154, delta = -2.666666666666667%

m=1000, p=0.5: expected: 500.0, result: 486, delta = 2.800000000000000
3%
m=1000, p=0.8: expected: 800.0, result: 790, delta = 1.25%
m=1000, p=0.3: expected: 300.0, result: 302, delta = -0.666666666666666
7%

m=10000, p=0.5: expected: 5000.0, result: 4984, delta = 0.32%
m=10000, p=0.8: expected: 8000.0, result: 7947, delta = 0.6625%
m=10000, p=0.3: expected: 3000.0, result: 3017, delta = -0.566666666666
6667%
```

In [27]:

```
# def testBasketIsSorted(basket: str):
#     item = basket.split('\\s')
#     i = 0
#     while i < len(item)-2:
#         if item[i]>= item[i+1]: return False
#     return True
```

In [28]:

```python
def generateItemset(basketItem:[int]):
    """
    Generate singleton and pairs from basket's item id

    :param basketItem: array of basket's item id
    :return: set of Itmeset
    """
    result = set()
    for i in range(1, 4):
        tmpset = combinations(basketItem,i)
        for j in tmpset: result.add(j)

    # i = 0
    # while i < len(basketItem):
    #     result.add((basketItem[i]))
    #     j = i+1
    #     while j < len(basketItem):
    #         result.add((basketItem[i],basketItem[j]))
    #         j+=1
    #     i+=1
    return result
```

In [29]:

```python
def updateItemsetCount(fullItemset: dict, basketItemset):
    """
    Update basket's itemset into full itemset
    if an itemset is already in the full itemset, increase its count by 1 in full
    otherwise, add to the full itemset by creating new record of itself in full i

    :param fullItemset: full itemset
    :param basketItemset: basket itemset
    :return:
    """
    for itemset in basketItemset:
        if itemset in fullItemset:
            # id = fullItemset.index(itemset)
            fullItemset[itemset] +=1
        else:
            fullItemset[itemset] = 1
```

In [30]:

```python
def convertLine(line:str):
    """
    Convert line into list of int

    :param line: basket data
    :return: list of item's id
    """
    id_str = line.split()
    return list(map(int, id_str))
```

In [31]:

```
def processFileLine(line):
    ids = convertLine(line)
    return generateItemset(ids)
```

In [32]:

```
def filterResult(itemset:dict, leastSupportCount)-> dict:
    """
    Filter input itemset for only accepting ones with frequent greater or equal g
    Input itemset must be sorted for correct result

    :param itemset: collected itemsets and their frequent
    :param leastSupportCount: given threshold
    :return: itemsets that satisfy the condition
    """
    result = dict()
    for key in itemset.keys():
        if itemset[key] < leastSupportCount: break
        result[key] = itemset[key]
    return result
```

In [33]:

```
def processLimitedPassAlgo(filePath, m, p, s, resultLogPath, random, rangeId):
    """
    Modified version for run both randomize and SON

    :param filePath:
    :param m:
    :param p: selecting ratior for randomize, 1/c for SON
    :param s:
    :param resultLogPath:
    :param random: True for run randomized , False for run SON algo
    :param rangeId: only need when run SON
    :return: limited-pass frequent itemset
    """

    numberBasket = m

    leastSupportCount = 0
    intro = ''

    processLineId = []
    if random:
        processLineId = randomSelect(numberBasket, p)
        leastSupportCount = int(0.9*p*s*numberBasket)
        intro = f'processing {filePath}, numberBasket={numberBasket}, numberSelec
        intro += f', leastSupportCount={leastSupportCount}'
    else:
        processLineId = rangeId
        leastSupportCount = int(s*numberBasket)
        intro = f'processing {filePath}[{processLineId[0]}:{processLineId[-1]}]'
    fullItemset = dict()

    if random: print(intro)

    timeStart = datetime.now()
    currentLine = 0
    with open(filePath, 'r') as file:
        for line in file:
            line = line.strip()
            if (currentLine in processLineId) and (len(line)>0):
                updateItemsetCount(fullItemset, processFileLine(line))

            currentLine += 1
    # for itemset in fullItemset:
    #     itemset.conf = itemset.count/numberBasket

    frequentItemset = dict(sorted(fullItemset.items(),key=lambda item: item[1], r
    if random:
        frequentItemset = filterResult(frequentItemset,leastSupportCount)
    timeStop = datetime.now()

    end = f'finish in {str(timeStop - timeStart)}'
    if random: print(end)

    with open(resultLogPath,'w') as file:
        file.write(f'{intro}\n{end}\n')
        for key in frequentItemset.keys():
            file.write(f'{key}->{frequentItemset[key]}\n')

    return frequentItemset
```

In [34]:

```
def createDirectory(path):
    """
    Build folder based on given path
    Support nested folder, ie. './a/b/c',
    """
    path = path.split(os.sep)
    currentDir = '.'
    for i in path:
        currentDir += os.sep + i
        try:
            os.mkdir(currentDir)
        except:
            # this folder is existed
            continue
p=.05
s=.5
dataDir = './data/'
resultDir = f'{dataDir}/p{p}s{s}/'
createDirectory(resultDir)

fileDataPath = []
for a in os.listdir(dataDir):
    if a.endswith('.dat'): fileDataPath.append(dataDir + a)
fileDataPath = sorted(fileDataPath)
```

Run randomised algo as requested in 1.3

Dataset for running need to be found in folder "data"
Result will be saved in folder with format "data/p{s}s{s}", ex. "data/p0.05s0.5/"

In [14]:

```
for path in fileDataPath:
    m = sum(1 for line in open(path))
    outputPath = path.replace('.dat','.resultv2')\
                         .replace(dataDir,resultDir)
    processLimitedPassAlgo(path, m, p, s, outputPath, True, [])

```

```
processing ./data/T10I4D100K.dat, numberBasket=100000, numberSelecting=
5120, leastSupportCount=2250
finish in 0:00:06.740719
processing ./data/T40I10D100K.dat, numberBasket=100000, numberSelecting
=4973, leastSupportCount=2250
finish in 0:01:43.280514
processing ./data/chess.dat, numberBasket=3196, numberSelecting=153, le
astSupportCount=71
finish in 0:00:00.583342
processing ./data/connect.dat, numberBasket=67557, numberSelecting=344
7, leastSupportCount=1520
finish in 0:00:31.512795
processing ./data/mushroom.dat, numberBasket=8124, numberSelecting=398,
leastSupportCount=182
finish in 0:00:00.295855
processing ./data/pumsb.dat, numberBasket=49046, numberSelecting=2468,
leastSupportCount=1103
finish in 0:02:57.801484
processing ./data/pumsb_star.dat, numberBasket=49046, numberSelecting=2
393, leastSupportCount=1103
finish in 0:00:52.036848
```

In [ ]:

```

```

1.2. Implement SON algorithm

SON algorithm is implemented as follow phases: **Phase 1:**

1. Divide full dataset into c chunks
    - modify RandomAlgo to take a range of line Id instead of randomly generating
    - equivalently $p = \frac{1}{c}$
2. For each of data chunk:
    - handled by a parallel 'process'
    - each 'process' execute same process as Random Algorithm
    - store outputs into files

Phase 2:

1. Collect and summarize frequent itemset from output files
2. Sort itemsets and get most frequent itemset
    - expected confident is s
    - equivalently change to least support count sm

In [ ]:

```
1
```

Implementation for phase 1

I chose $multiprocessing.Pool$ for implementing paralle processing.
$Pool$ will be initialize with a specific amount of subprocess, ie. 5 in my implementation. Those subprocess is automatically mapped with given function and given arguments set, then will be executed independently from the main process

In [35]:

```
def buildDataForPool(path, m, p, s, outputPath):
    """
    Build list of arguments list for pool's subprocess

    :param filePath: path to dataset
    :param m:
    :param p:
    :param s:
    :param outputPath: output path for chunk's result
    :return: list of arguments list for pool's subprocess
    """

    data =[]
    mOVerC = int(m/c)
    maxCounter = c if m%c==0 else c+1

    for counter in range(maxCounter):
        # each item must strictly follow the argument order from
        # processLimitedPassAlgo(filePath, m, p, s, resultLogPath, random, rangeI
        data.append([path, m, p, s, outputPath+str(counter), False
                        , [x for x in range(mOVerC*counter, min(m, mOVerC*(counte
                    ])
    return data


```

In [36]:

```
def parallelProcess(processInput):
    """
    Execute processLimitedPassAlgo with given argument list
    """

#     processLimitedPassAlgo(filePath, m, p, s, resultLogPath, random, rangeId):
    processLimitedPassAlgo(processInput[0],processInput[1],processInput[2],proces
                           ,processInput[4],processInput[5],processInput[6])


```

In [17]:

```
pool = Pool(5)
c = 100
resultDir = f'{dataDir}SONc{c}s{s}/'
createDirectory(resultDir)
```

Implementation for phase 2:

In [18]:

```
def collectItemsetFromChunks(chunkResultList):
    """
    Combine full itemset result from all created results

    :param chunkResultList: list of chunk's result files, ie. chess.result.sonXX
    :return: full itemset
    """

    result = dict()
    for filePath in chunkResultList:
        # print(f'executing collectItemsetFromChunks: {filePath}')
        with open(filePath,'r') as file:
            # skip 2 info lines
            next(file)
            next(file)

            for line in file:
                key, value = line.split('->')
                value = int(value)
                if key in result: result[key] += int(value)
                else: result[key] = int(value)
    return result

```

In [19]:

```
def collectItemsetFromDataset(datasetName, resultDir):
    """
    SON phase 2's functionalities
    """

    timeStart = datetime.now()
    m = dataLineCounts[f'{dataDir}{datasetName}.dat']
    chunkResult = datasetName + '.result.son'

    # scan resultDir for related chunk result
    chunkResultList = [resultDir + x for x in os.listdir(resultDir) if chunkResul


    fullItemset = collectItemsetFromChunks(chunkResultList)
    fullItemset = dict(sorted(fullItemset.items(),key=lambda item: item[1], rever
    timeStop = datetime.now()

    print(f'finish collect from {chunkResult} in {str(timeStop - timeStart)}')

    with open(f'{resultDir}{datasetName}.final','w') as file:
        for key in fullItemset.keys():
            if fullItemset[key] < s*m: break
            file.write(f'{key}->{fullItemset[key]}\n')
    # return fullItemset
```

In [ ]:

```

```

Run SON algorithm as request in 1.3

In [20]:

```
%%time

# run SON's phase 1

for path in fileDataPath:
    timeStart = datetime.now()
    m = dataLineCounts[path]
    outputPath = path.replace('.dat','.result.son')\
                     .replace(dataDir,resultDir)

    parallelData = buildDataForPool(path, m, 1/c, s, outputPath)
    pool.map(parallelProcess, parallelData)
    timeStop = datetime.now()
    print(f'finsih processing {path} in {str(timeStop-timeStart)}')
```

```
finsih processing ./data/T10I4D100K.dat in 0:01:12.780084
finsih processing ./data/T40I10D100K.dat in 0:15:20.015748
finsih processing ./data/chess.dat in 0:00:06.403641
finsih processing ./data/connect.dat in 0:04:33.407238
finsih processing ./data/mushroom.dat in 0:00:03.636486
finsih processing ./data/pumsb.dat in 0:23:17.710465
finsih processing ./data/pumsb_star.dat in 0:07:42.315305
CPU times: user 195 ms, sys: 24.8 ms, total: 219 ms
Wall time: 52min 16s
```

In [21]:

```
%%time

# run SON's phase 2

for filename in fileDataNamesOnly:
    collectItemsetFromDataset(filename,resultDir)

```

```
finish collect from T10I4D100K.result.son in 0:00:26.140894
finish collect from T40I10D100K.result.son in 0:17:00.326199
finish collect from chess.result.son in 0:00:01.601326
finish collect from connect.result.son in 0:00:07.835119
finish collect from mushroom.result.son in 0:00:01.119917
finish collect from pumsb.result.son in 0:07:01.125634
finish collect from pumsb_star.result.son in 0:03:28.021837
CPU times: user 27min 28s, sys: 34.8 s, total: 28min 3s
Wall time: 28min 12s
```

In [ ]:

```

```

1.3. Report the outcomes

a3e13_result.zip containts the outcomes of procssing following datasets:

- T10I4D100K.dat
- T40I10D100K.dat

- chess.dat
- connect.dat
- mushroom.dat
- pumsb.dat
- pumsb_star.dat

Folder includes:

- folder **SONc100s0.5** containts only final results of SON's algo running with 100 chunks and s=0.5
- folder **p0.05s0.5** containts result of randomised algo running with p=0.05 and s=0.5

1.4. Experiment with different sample sizes in the simple randomized algo-rithm such as 1, 2, 5, 10% and compare your results

In [41]:

```
p_list = [.01,.02,.1,.2]

for p in p_list:
    s=.5
    dataDir = './data/'
    resultDir = f'{dataDir}/p{p}s{s}/'
    createDirectory(resultDir)

    for path in fileDataPath:
        m = sum(1 for line in open(path))
        outputPath = path.replace('.dat','.resultv2')\
                         .replace(dataDir,resultDir)
        processLimitedPassAlgo(path, m, p, s, outputPath, True, [])
    print()
```

```
processing ./data/T10I4D100K.dat, numberBasket=100000, numberSelecting
=998, leastSupportCount=450
finish in 0:00:01.349564
processing ./data/T40I10D100K.dat, numberBasket=100000, numberSelectin
g=994, leastSupportCount=450
finish in 0:00:13.346378
processing ./data/chess.dat, numberBasket=3196, numberSelecting=31, le
astSupportCount=14
finish in 0:00:00.139196
processing ./data/connect.dat, numberBasket=67557, numberSelecting=71
2, leastSupportCount=304
finish in 0:00:05.447991
processing ./data/mushroom.dat, numberBasket=8124, numberSelecting=10
5, leastSupportCount=36
finish in 0:00:00.094768
processing ./data/pumsb.dat, numberBasket=49046, numberSelecting=499,
leastSupportCount=220
finish in 0:00:31.899542
processing ./data/pumsb_star.dat, numberBasket=49046, numberSelecting=
```

In [ ]:

```

```

In [ ]:

```
1
```

T10I4D100K.dat and T40I10D100K.dat

Randomised algo's results and SON algo's result both show empty itemset. We can conclude that itemsets in those dataset are fregmented, therefore they dont have enough repeatation to achieve expected confident s

chess.dat's top 5 frequent item

p=1% s=0.5	p=2% s=0.5	p=5% s=0.5	p=10% s=0.5	p=20% s=0.5	SON c=100 s=0.5
(48, 58)->31	(58,)->66	(52, 58)->153	(58,)->310	(58,)->625	(58,)->3130
(40, 48, 58)->31	(60,)->66	(58,)->153	(52, 58)->309	(29,)->624	(60,)->3127
(52, 60)->31	(58, 60)->66	(52,)->153	(52,)->309	(52, 58)->624	(58, 60)->3126
(40, 48, 60)->31	(29, 58, 60)->65	(40,)->152	(29,)->308	(29, 58)->624	(52,)->3120
(40,)->31	(52, 60)->65	(40, 52, 58)->152	(29, 58)->308	(52,)->624	(52, 58)->3119

Consider as SON has the most accurate (process 100% dataset instead of picking random parts)

- p=20%'s result containts 3 correct frequent itemset
- p=10%'s result containts 2 correct frequent itemset
- p=5%'s result containts 3 correct frequent itemset
- p=2%'s result containts 1 correct frequent itemset
- p=1%'s result containts 0 correct frequent itemset

The incorrect itemset appear in other results are called *false negative*
The higher value of p, the lesser false negative we get.

The frequent of itemset seems to be more accurate with higher p.

- SON result indicates singleton (58) repeats 3130 times
- p=20%'s indicates singleton (58) repeats 625 times -> 625*5 = 3125 times over the whole dataset: extremely closed to SON's result
- p=10%'s indicates singleton (58) repeats 310 times -> 310*10 = 3100 times over the whole dataset: closed
- p=5%'s indicates singleton (58) repeats 153 times -> 153*20 = 3060 times over the whole dataset: the differences is ~100
- p=2%'s indicates singleton (58) repeats 66 times -> 66*50=3300 times over the whole dataset: the differences is ~170
- p=1%'s indicates singleton (58) repeats 31 times -> 31*100=3100 times over the whole dataset: really closed, but (58) is only the 19th among most frequent itemset, while it actually the 1st one in final

Similar analyze could be done based other dataset's results