Direct Hashing and Pruning

December 11, 2018

```
In [27]: #!/usr/bin/env python3
         # -*- coding: utf-8 -*-
         """
         Created December 2018

         @author: ebecerra
         """

         # Emanuel Becerra Soto
         # MD01 CIC IPN

         # Algorithm Direct Hashing and Pruning

         ######### Libraries #########

         import itertools

         ######### Functions #########

         # Flattens a list
         def flatten(l):
             return list( itertools.chain.from_iterable( l ) )

         # Finds the itemset of a database
         def single_items(database):
             item_union = set()
             candidates_1k = []
             for transaction in database:
                 x = set(transaction)
                 item_union = item_union | x
             for item in item_union:
                 candidates_1k.append([item])
             return candidates_1k

         # Check for a correct value of the minimu support
         def check_min_sup( data_base, min_sup ):
             n_tran = len(data_base)
```

```
    if isinstance(min_sup, float) and min_sup <= 1 and min_sup >= 0:
        cut_off = min_sup * n_tran
    elif isinstance(min_sup, int) and min_sup >= 1:
        cut_off = min_sup
    else:
        exit('Error: min_sup must be between 0.0 and 1.0 or a positive integer')
    return cut_off

# Performs the join operation over candidate item sets
def join(pattern1, pattern2):
    k = len(pattern1)
    candidate_k1 = []
    for idx in range(k):
        if pattern1[idx] == pattern2[idx] and idx + 1 < k:
            candidate_k1.append(pattern1[idx])
        # The less than equal comparission preserves the
        # lexicographic order
        elif pattern1[idx] < pattern2[idx] and idx + 1 == k:
            candidate_k1.append(pattern1[idx])
            candidate_k1.append(pattern2[idx])
        else:
            candidate_k1 = []
            break
    return candidate_k1

# Generates candidate using a hash table
def gen_candidates( freqs, hash_table, cut_off_int ):
    table_size = len(hash_table)
    Ck=[]
    all_2_groups_freq = list( itertools.combinations( freqs, 2) )
    for group in all_2_groups_freq:
        group = list(group)
        cki = join( group[0], group[1] )
        idx = hash( str(cki) ) % table_size
        if hash_table[ idx ] >= cut_off_int:
            Ck.append( cki )
    return Ck

# Extracts the frequent itemsets of 1 element
def part_1( data_base, min_sup, items, table_size ):
    cut_off = check_min_sup( data_base, min_sup )
    ######  DHP ######
    size_2_subsets = list( itertools.combinations( flatten(items), 2) )
    # Initialization of hash table
    hash_table_2 = [0] * table_size
    # Initialization of dictonary of counts
    COUNTS = {}
    for item in items:
        COUNTS[ str(item) ] = { 'count': 0, 'pattern': item }
    for transaction in data_base:
        for item in items:
            if set(item) <= set(transaction):
                COUNTS[ str(item) ]['count'] += 1
        for subset_size_2 in size_2_subsets:
            subset_size_2 = list(subset_size_2)
            if set(subset_size_2) <= set(transaction):
                # hashing the subset of size 2
                # and then taking the modulo to see its position on the table
                idx = hash(str(subset_size_2)) % table_size
                # Adding 1 to the hash table
                hash_table_2[ idx ] += 1
    frequent_size_1 = []
    for key in COUNTS:
        if COUNTS[key]['count'] >= cut_off:
            frequent_size_1.append( COUNTS[key] )
    return { 'freq': frequent_size_1, 'hash_table': hash_table_2 }

# Extracts the frequent itemsets of k element
def part_2( data_base, min_sup, items, freqs_1k, hash_table_2 ):
    table_size = len(hash_table_2)
    cut_off = check_min_sup( data_base, min_sup )
    RESULTS = {}
    k = 3
    Ck = ['Initialization']
    freqs = freqs_1k
    hash_table = hash_table_2
    while Ck: # Candidate Set is not Empty
        Ck = gen_candidates( freqs, hash_table, cut_off )
        # Initialization of the count dictionary
        COUNTS = {}
        # Initialization of the COUNTS dictionary
        for cki in Ck:
            COUNTS[ str(cki) ] = { 'count': 0, 'pattern': cki }
        hash_table_k = [0] * table_size
        size_k_subsets = list( itertools.combinations( freqs, k) )
        for transaction in data_base:
            # Counting
            for candidate in Ck:
                if set( candidate ) <= set(transaction):
                    COUNTS[ str(candidate) ]['count'] += 1
            # Hash Table building
            for subset_size_k in size_k_subsets:
                subset_size_k = flatten(list(subset_size_k))
                if set(subset_size_k) <= set(transaction):
                    # hashing the subset of size k
                    # and then taking the modulo to see its position on the table
                idx = hash( str(subset_size_k) ) % table_size
                # Adding 1 to the hash table
                hash_table_k[ idx ] += 1
    # Getting the frequent patterns
    frequent_detail = []
    for key in COUNTS:
        if COUNTS[key]['count'] >= cut_off:
            frequent_detail.append( COUNTS[key] )
    if frequent_detail:
        RESULTS[ k-1 ] = frequent_detail
    # Increasing for the next iteration
    k += 1
    # Generating freq for next iteration
    freqs = []
    for freq_i in frequent_detail:
        freqs.append( freq_i['pattern'] )
return RESULTS

######### Main Program #########

# The transactions must be in a lexicographic order

D = [
 [1,2,5],
 [2,4],
 [2,3],
 [1,2,4],
 [1,3],
 [2,3],
 [1,3],
 [1,2,3,5],
 [1,2,3],
 ]

def main( data_base, min_sup, table_size ):
items = single_items(data_base)
part_1_out = part_1(data_base, min_sup, items, table_size)
freqs_1k = []
for freq_i in part_1_out['freq']:
    freqs_1k.append( freq_i['pattern'] )
results = part_2( data_base, min_sup, items,
                 freqs_1k, part_1_out['hash_table'] )
results[1] = part_1_out['freq']
return results

main( D, 2/9, 30 )
```

```
Out[27]: {1: [{'count': 6, 'pattern': [1]},
 {'count': 7, 'pattern': [2]},
 {'count': 6, 'pattern': [3]},
 {'count': 2, 'pattern': [4]},
 {'count': 2, 'pattern': [5]}],
2: [{'count': 4, 'pattern': [1, 2]},
 {'count': 4, 'pattern': [1, 3]},
 {'count': 2, 'pattern': [1, 5]},
 {'count': 4, 'pattern': [2, 3]},
 {'count': 2, 'pattern': [2, 4]},
 {'count': 2, 'pattern': [2, 5]}],
3: [{'count': 2, 'pattern': [1, 2, 5]}]}
```