解题代码

```
#  (1) FP-growth算法将数据存储在一种称为FP树的紧凑数据结构中。由于FP树比其他树更加复杂，因此需要一个类来保存树的每一个节点。首先，定义FP树的节点类。

class FPNode(object):
    """FP树中的节点"""

    def __init__(self, tree, item, count=1):
        self._tree = tree
        self._item = item
        self._count = count
        self._parent = None
        self._children = {}
        self._neighbor = None

    def add(self, child):
        """将给定的fp'孩子'节点添加为该节点的子节点"""

        if not isinstance(child, FPNode):
            raise TypeError("Can only add other FPNodes as children")

        if not child.item in self._children:
            self._children[child.item] = child
            child.parent = self

    def search(self, item):
        """
       检查此节点是否包含给定项的子节点。
         如果是，则返回该节点；否则，返回None
        """
        try:
            return self._children[item]
        except KeyError:
            return None

    def __contains__(self, item):
        return item in self._children

    @property
    def tree(self):
        """出现此节点的树"""
        return self._tree

    @property
    def item(self):
        """此节点中包含的项"""
        return self._item

    @property
    def count(self):
        """与此节点项相关的计数。"""
        return self._count

    def increment(self):
        """增加与此节点项相关联的计数。"""
        if self._count is None:
            raise ValueError("Root nodes have no associated count.")
```

```
        self._count += 1

    @property
    def root(self):
        """如果此节点是树的根，则为true；否则为false"""
        return self._item is None and self._count is None

    @property
    def leaf(self):
        """如果此节点是树中的叶子，则为true；否则为false"""
        return len(self._children) == 0

    @property
    def parent(self):
        """父节点"""
        return self._parent

    @parent.setter
    def parent(self, value):
        if value is not None and not isinstance(value, FPNode):
            raise TypeError("A node must have an FPNode as a parent.")
        if value and value.tree is not self.tree:
            raise ValueError("Cannot have a parent from another tree.")
        self._parent = value

    @property
    def neighbor(self):
        """
       节点的邻居；在树中具有相同值的“右边”
        """
        return self._neighbor

    @neighbor.setter
    def neighbor(self, value):
        if value is not None and not isinstance(value, FPNode):
            raise TypeError("A node must have an FPNode as a neighbor.")
        if value and value.tree is not self.tree:
            raise ValueError("Cannot have a neighbor from another tree.")
        self._neighbor = value

    @property
    def children(self):
        """该节点的子节点"""
        return tuple(self._children.itervalues())

    def inspect(self, depth=0):
        print ('  ' * depth) + repr(self)
        for child in self.children:
            child.inspect(depth + 1)

    def __repr__(self):
        if self.root:
            return "<%s (root)>" % type(self).__name__
        return "<%s %r (%r)>" % (type(self).__name__, self.item, self.count)


# （2） 定义Fptree函数来构建FP树
```

```
from collections import defaultdict, namedtuple
class FPTree(object):
    Route = namedtuple('Route', 'head tail')

    def __init__(self):
        # 树的根节点.
        self._root = FPNode(self, None, None)

        #字典将项目映射到路径的头部和尾部
        # "neighbors" that will hit every node containing that item.
        self._routes = {}

    @property
    def root(self):
        #树的根节点.
        return self._root

    def add(self, transaction):
        #将事务添加到树中
        point = self._root

        for item in transaction:
            next_point = point.search(item)
            if next_point:
                next_point.increment()
            else:
                next_point = FPNode(self, item)
                point.add(next_point)

                self._update_route(next_point)

            point = next_point

    def _update_route(self, point):
        assert self is point.tree

        try:
            route = self._routes[point.item]
            route[1].neighbor = point # route[1] 是尾部
            self._routes[point.item] = self.Route(route[0], point)
        except KeyError:
            # 开始一个新路径
            self._routes[point.item] = self.Route(point, point)

    def items(self):
        """
       为树中表示的每个项生成一个2元组。 元组的第一个元素是项本身，
       第二个元素是一个生成器，它将生成树中属于该项的节点。
        """
        for item in self._routes.keys():
            yield (item, self.nodes(item))

    def nodes(self, item):
        """
        生成包含给定项的节点序列
        """
```

```
        try:
            node = self._routes[item][0]
        except KeyError:
            return

        while node:
            yield node
            node = node.neighbor

    def prefix_paths(self, item):
        """生成以给定项结尾的前缀路径."""

        def collect_path(node):
            path = []
            while node and not node.root:
                path.append(node)
                node = node.parent
            path.reverse()
            return path

        return (collect_path(node) for node in self.nodes(item))

    def inspect(self):
        print( 'Tree:')
        self.root.inspect(1)

        print()
        print ('Routes:')
        for item, nodes in self.items():
            print ('  %r' % item)
            for node in nodes:
                print( '    %r' % node)


#  (3) 定义conditional_tree_from_paths()，从给定的前缀路径构建条件FP树。前缀路径是介于所查找元素项与树根节点之间的所有内容。

def conditional_tree_from_paths(paths):
    """从给定的前缀路径构建条件FP树."""
    tree = FPTree()
    condition_item = None
    items = set()

    #将路径中的节点导入新树。只有叶子节点的计数才重要;剩下的计数将根据叶子节点的计数进行重构。
    for path in paths:
        if condition_item is None:
            condition_item = path[-1].item

        point = tree.root
        for node in path:
            next_point = point.search(node.item)
            if not next_point:
                # Add a new node to the tree.
                items.add(node.item)
                count = node.count if node.item == condition_item else 0
                next_point = FPNode(tree, node.item, count)
                point.add(next_point)
```

```
                tree._update_route(next_point)
            point = next_point

    assert condition_item is not None

    # Calculate the counts of the non-leaf nodes.
    for path in tree.prefix_paths(condition_item):
        count = path[-1].count
        for node in reversed(path[:-1]):
            node._count += count

    return tree


#  (4) 从构建的条件FP树中寻找频繁项集。
#
# 在给定的事务中使用FP-growth查找频繁项集。该函数返回一个生成器，而不是快速填充的项列表。 “事务”参数可以是项的任何可迭代项。 “minimum_support”应该是一个整数，它指定要

def find_frequent_itemsets(transactions, minimum_support, include_support=False):
    items = defaultdict(lambda: 0) #从项到其支持度的映射

    #加载传入的事务并计算单个项的支持度
    for transaction in transactions:
        for item in transaction:
            items[item] += 1

    #从项支持字典中删除不常见的项。
    items = dict((item, supports) for item, supports in items.items()
        if supports >= minimum_support)

    #建立FP-tree。 在任何事务可以被添加到树之前，他们必须被剥夺不常出现的项，并且剩余的项必须按照频率的降序排序。
    def clean_transaction(transaction):
        transaction = filter(lambda v: v in items, transaction)
        transaction_list = list(transaction)  # 为了防止变量在其他部分调用，这里引入临时变量transaction_list
        transaction_list.sort(key=lambda v: items[v], reverse=True)
        return transaction_list

    master = FPTree()
    for transaction in map(clean_transaction, transactions):
        master.add(transaction)
    def find_with_suffix(tree, suffix):
        for item, nodes in tree.items():
            supports = sum(nn.count for nn in nodes)
            if supports >= minimum_support and item not in suffix:
                # 新赢家
                found_set = [item] + suffix
                yield (found_set, supports) if include_support else found_set

                # #从项支持字典中删除不常见的项。
                cond_tree = conditional_tree_from_paths(tree.prefix_paths(item))
                for s in find_with_suffix(cond_tree, found_set):
                    yield s # pass along the good news to our caller
    # 搜索频繁的项目集，并产生我们找到的结果。
    for itemset in find_with_suffix(master, []):
        yield itemset
```

```
# 调用FP增长树算法，实现寻找频繁项集

def load_data(filename):
    data=list()
    with open(filename,'r',encoding='utf-8') as f:
        for line in f.readlines():
            linestr=line.strip()
            linestrlist=linestr.split(",")
            data.append(linestrlist)
    return data


dataset = load_data('data/apriori.txt')  # 读取数据

#*************** BEIGN ******************
frequent_itemsets = find_frequent_itemsets(dataset, minimum_support=56, include_support=True)
#**************** END ********************

print(type(frequent_itemsets))  # print type

result = []
for itemset, supports in frequent_itemsets:    # 将generator结果存入list
    result.append((itemset, supports))

result = sorted(result, key=lambda i: i[0])   # 排序后输出
for itemset, supports in result:
    print(str(itemset) + ' ' + str(float(supports/len(dataset))))
print(len(result))  # 输出频繁项集的个数
```