数据结构实习报告

面向用户的网络购物平台

目录

一、问题提出

随着生活节奏加快与网络的普及，网上购物已经成为现代人生活的一个重要组成部分。而除了产品本身质量的优劣，网购平台建设的好坏也直接决定了用户最终的选择，因此需要商家对此投入足够的时间与精力进行优化，以便达成更好的销售目标。

通过本人在日常生活中使用“星巴克”“肯德基”等大型商户的网上订购平台，并与一些刚起步的小型公司建设的网络购物平台对比，发现一些平台虽能实现用户基本的增删改查功能，却忽视了对用户进行实时推荐：即根据用户已选择的产品，结合数据库中保存的购买记录，推测用户可能感兴趣的其他商品进行推荐，从而鼓励用户购买更多商品。基于这一需求，我打算设计一款**带有自动推荐功能**的网络购物平台，旨在为用户提供更优的购物体验，并为商家带来更大的收益。

二、背景

“推荐系统”是数据挖掘领域热门的研究话题，也已经有许多思路可以解决该问题，如协同过滤法等。在此，我选择的是挖掘频繁项集来进行推荐。

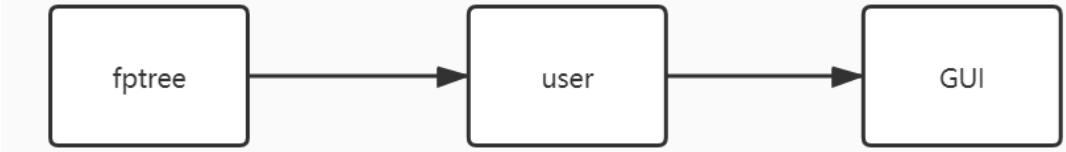
“频繁项集”指的是在购物篮中经常一起出现的商品，最有名的例子莫过于“啤酒与尿布”的故事，而这恰与我们要解决的问题相适应。挖掘频繁项集常用的算法是 Apriori 算法，但该算法需要多遍扫描购

买记录数据库，这意味着大量的 IO 开销，在我们的问题场景下显然是十分低效的。因此，我选择采用频繁模式增长（FP-Growth）算法，该算法巧妙地利用了**树**的数据结构来压缩存储购买记录，只需要对数据库进行两遍扫描，就可以直接在树上挖掘频繁项集。

为了实现一个完整的网络购物平台，除了推荐算法本身之外，还需要设计数据结构来实现菜单、购物篮等的功能，并根据输入输出需要设计相应的数据存储格式。另外，为使得用户与系统的交互更为顺利，我们还设计了用户图形界面(GUI)。这些都将在接下来的设计说明部分进行解释。

三、设计说明

我们采用的是面向对象的设计方法，即将数据结构与相应的功能封装在一个类中。这样的设计方法更有利于大型项目的开发。具体而言，本项目程序分为“fptree”，“user”和“GUI”三个模块组成，三者的依赖关系如下：



3.1 FP-Growth 算法实现

3.1.1 算法思想与效率分析

FP-Growth 算法由韩家炜等学者在 2000 年提出，旨在减少对事务数据库（即客户购买记录）的遍历次数，从而减少挖掘频繁项集的时间开销。该算法采用的是用空间换时间的策略，通过设计一种名为 FP-Tree 的数据结构，辅以项头表来进行树的遍历（图 1），从而实现了对事务数据库的压缩存储，并能在树上直接进行频繁模式的挖掘。

FP-Growth 算法分为“建树”和“在树上挖掘频繁项集”两个步骤。构建 FP-Tree 的基本思想是：先扫描一遍数据库，得到每个商品出现的次数，将其从高到低记录在项头表中。再次扫描数据库，对于每条记录，将记录中的商品按照项头表的次序排列，并在 FP-Tree 中生成分支。如果已经该商品的节点在树中已经存在，则将该节点的频数加一，并以之为根节点继续生成下一个商品的节点；如果不存在，则在根节点为该商品生成新的节点，并将项头表的节点链指针指向它，再以之为根继续生成下一个商品的节点。因此，只需要两次遍历数据库，就可以构建一棵 FP-Tree，之后只需要在该树上挖掘频繁项集。挖掘频繁项集则需要从树的底端开始，构建每一个商品的“条件 FP-Tree”，并在其上递归地进行挖掘。该算法较为复杂，我们在此给出文献中对其完整的算法描述（图 2），感兴趣的读者可以查阅参考文献[1]进行深入的了解。

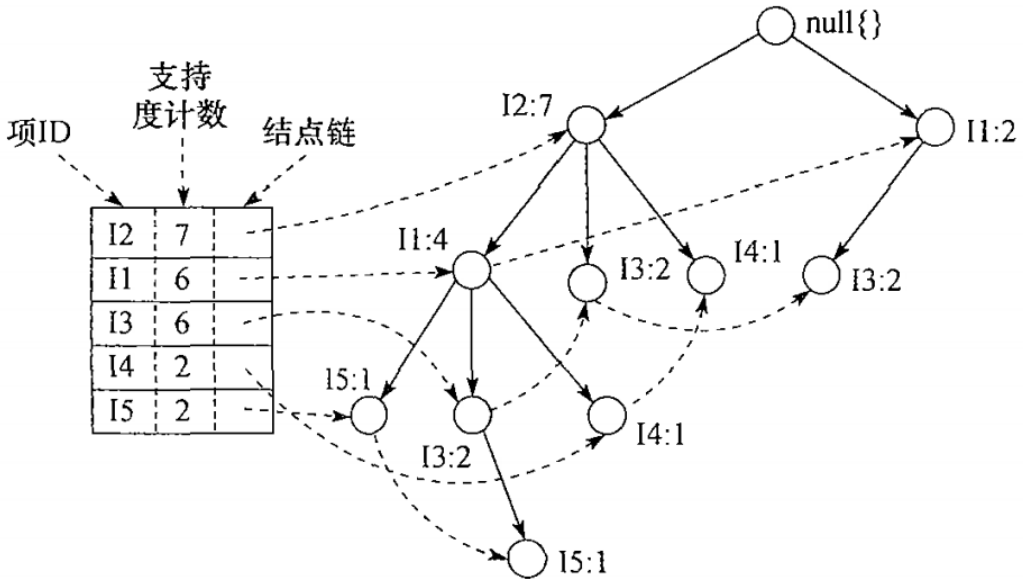


图 1 FP-Tree 的逻辑结构，左为项头表，右为 FP-Tree。[1]

算法：FP-Growth。使用 FP 树，通过模式增长挖掘频繁模式。

输入：

- D：事务数据库。
- *min_ sup*：最小支持度阈值。

输出：频繁模式的完全集。

方法：

1. 按以下步骤构造 FP 树：
   (a)扫描事务数据库 D 一次。收集频繁项的集合 F 和它们的支持度计数。对 F 按支持度计数降序排序，结果为频繁项列表 L。
   (b) 创建 FP 树的根结点，以“null”标记它。对于 D 中每个事务 *Trans*，执行：
   选择 *Trans* 中的频繁项，并按 L 中的次序排序。设 *Trans* 排序后的频繁项列表为 $[p \mid P]$，其中 p 是第一个元素，而 P 是剩余元素的列表。调用 **insert_tree**$([p \mid P], T)$。该过程执行情况如下。如果 T 有子女 N 使得 $N.\ item\text{-}name = p.\ item\text{-}name$，则 N 的计数增加 1；否则，创建一个新结点 N，将其计数设置为 1，链接到它的父结点 T，并且通过结点链结构将其链接到具有相同 *item-name* 的结点。如果 P 非空，则递归地调用 **insert_tree**(P, N)。
2. FP 树的挖掘通过调用 **FP_growth**(*FP_tree*, *null*)实现。该过程实现如下。
   procedure FP_growth(*Tree*, α)
   (1)**if** *Tree* 包含单个路径 P **then**
   (2)**for** 路径 P 中结点的每个组合(记作 β)
   (3)产生模式 $\beta \cup \alpha$，其支持度计数 *support_count* 等于 β 中结点的最小支持度计数；
   (4)**else for** *Tree* 的头表中的每个 a_i {
   (5)产生一个模式 $\beta = a_i \cup \alpha$，其支持度计数 $support\_count = a_i.\ support\_count$；
   (6)构造 β 的条件模式基，然后构造 β 的条件 *FP* 树 $Tree_\beta$；
   (7)**if** $Tree_\beta \neq \varnothing$ **then**
   (8)调用 **FP_growth**($Tree_\beta, \beta$)；}

图 2 FP-Growth 算法描述，摘自[1]

在效率方面，由于该算法较为复杂，很难简单地计算其时间复杂度开销，已有学者对该算法进行分析[2]，求出其时间复杂度大约为O(N2)，这在巨型数据集上依然是一个相当可观的开销，但比传统的Apriori 算法快了一个数量级，主要得益于减少了 IO 次数，代价是增加了空间复杂度，即需要用额外的存储空间来存放 FP-Tree。FP-Tree的大小取决于事务数据库的情况，最坏情况下也能达到 O(N)的量级。

3.1.2 算法物理结构与接口设计

FP-Growth 只提供了算法所需数据结构的逻辑结构设计，其物理结构仍然需要结合算法的特点来进行设计，此外还要考虑应当以什么样的形式给用户提供该算法的接口。

为实现该算法，我们设计了 Node 和 FP-Tree 两个数据结构，前者是后者实现的基础。两者的物理结构设计与 API 如下：

Node 数据成员： * item：str，项目名 * parent：Node*，祖先节点 * support_count：int，支持度计数 * next_sibling：Node*, fptree 中该 item 的下一个同名节点位置 * childrens_dict: 哈希表{key = item, value = Node}，该节点的子女列表，用哈希表存储，键为项目名，值为对应的节点

API：

* get_child(self, child_item):

    """

    返回名为 child_item 的子女节点

    * param child_item: str，子女节点名

    * return: Node，该子女节点/None，不存在指定子女节点

    """

* add_child(self, new_child_item):

    """

    为该节点添加一个名为 new_child_item 的子女节点。

    若已有同名子女，则直接返回子女节点。

    * param new_child_item: str，子女节点名

    * return: Node，该子女节点

    """

Node 节点需要存储指向祖先结点的指针，是因为 FP-Growth 算法在挖掘频繁项集时需要**自底向上遍历树**。存储支持度计数和指向树中下一个同名节点的指针（next_sibling），也是为了挖掘频繁项集。此外，由于这是一棵**多叉树**，且需要通过孩子节点的值来查找指定的孩子节点，故采用**哈希表**存储该节点的孩子节点，这与多叉树常见的“孩子兄弟表示法”等相当不同，这也是由实际应用的具体需求决定的。

Node 节点的 API 是根据 FP-Growth 中的操作需求设计的，我们给出了每个接口的功能与输入输出。

FP-Tree

* 数据成员:

    __header_table：项头表

    transactions：list(list)，交易记录

* 内部方法:

    * __init_header_table(self): 初始化项头表

    * __construct_fptree(self):

    """

    构建 FPTree。

    """

    * __insert_trans(self, root, trans):

    """

    向根节点递归地插入交易记录。

        * param root: Node，根节点

        * param trans: list，一条交易记录

    """

    * __mine_frequent_pattern(self):

    """

    挖掘频繁项集。

```
    """
* 外部接口：
    * fpgrow(self, min_support = 2, min_len = 2, max_len = 9):
    """
    挖掘支持度高于 min_support 的频繁模式。
        * param min_support: int，支持度计数阈值
        * return: dict，挖掘结果，{频繁模式:支持度}
    """
    * getrules(self):
    '''返回频繁模式(dict){频繁模式:支持度}，没有则返回 false'''
    * gettoprules(self, top = 5):
    '''打印支持度最高的 top 条模式，默认 top5；若支持度相同，则长的规则优先'''
    * saverules(self):
    '''将规则保存到数据库，成功返回 True，失败返回错误名'''
```

值得注意的是，我们将项头表封装进了 FP-Tree 中，成为子数据结构。由于项头表的功能与图的邻接表类似，故我们借鉴了**图的邻接表**表示法来设计项头表。该数据结构的 API 分为内部函数与外部函数，其中内部函数大多视为了实现 FP-Growth 算法设计的。

在整个系统运行过程中，FP-Growth 是怎么起到作用的？我们将该方法的实现封装在用户的 pay 和 recommendation 两个方法中。具体而言，pay 方法的执行逻辑是：

用户点击结算按钮，触发该过程： 1.计算价格 2.将此条购买记录写入事务数据库 records.txt 3.调用 fp-tree 更新规则，并将规则与支持度保存在 patterns.txt 文档中 4.若一切执行正常，返回价格（float）；若出现错误，返回 false。

这样就实现了数据库中规则的自动更新。而向用户进行推荐，则是使用 recommendation 方法。该方法将会根据用户购物篮的情况，利用**集合的交运算**匹配规则中支持度最高的一个（即，在有这些商品的购买记录中最频繁出现的一条，例如{“啤酒”，”尿布”}:3, {“啤酒”, “牛奶”}:2，当用户选择了啤酒时，我们将推荐尿布而非牛奶）。如果没有匹配的商品，则向用户随机推荐一个菜单中的商品。

3.2 User 模块的数据结构设计与功能实现

User 模块使用的数据结构设计与原因如下：

购物篮（busket）：哈希表，键为商品名，值为该商品选购个数。

设计的原因：购物篮对于增删改查的要求都很高，而由于需要实时更新总金额，查询每个商品的选购数量变得尤为重要，而哈希表能在 O(1)时间内完成对值的查询，大大加快了该过程的执行效率。

API:

- 查看购物篮中全部商品：遍历哈希表，O(N)
- 清空购物篮中商品：清空哈希表，O(N)
- 向购物篮添加商品：增加记录，O(1)
- 修改商品个数：修改哈希表中的值，涉及查找操作，O(1)
- 删除商品：查找并删除哈希表中的一个键值对。由于 Python 中哈希表的存储空间是事先分配好的，键值对的删除不会引起顺序表中数据大量移动，故开销为 O(1)。

购买记录（records）：哈希表，键为记录 ID，值为该订单下的商品列表，以顺序表的形式存储。

设计的原因：购买记录的查询操作居多，而在现实生活中也一般以哈希表的形式存储在关系型数据库中，故我们也选用哈希表存储。

菜单（menu）：哈希表，键为商品名，值为价格。

设计的原因：菜单并不经常需要增加或删除条目，但对查询效率的要求非常高，故哈希表依然是最合适的存储结构。

API:

- 查找商品价格：哈希表查找，O(1)
- 查询全部商品：遍历哈希表，O(N)

以上所有数据结构的 API 均有恰当的输出。若没有输出，则返回布尔值，以表示操作是否执行成功。此外，我们还设计了**错误类型 Empty**，当执行一些非法操作（如购物篮为空时执行结算操作）时，函数返回该错误对象，而程序中也设计了相应的逻辑来使得该错误能得到正确的处理，从而加强了程序的鲁棒性。

从面向对象的角度来看，我们设计了两个类“菜单类”和“用户类”。其中菜单类即为上述菜单数据结构的设计，用户类的描述与数据成员如下：

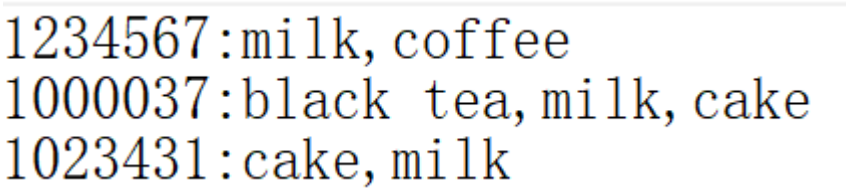
```
Class User
'''
Description:
    用户进入系统时自动生成的对象。
Attributes:
    recordID(str): 订单编号
    busket(dict): 用户购物篮
        {key = 商品名(str), value = 该商品购买个数(int)}
    records(dict): 读入历史购买记录
```

```
        {key = recordID(str), value = transaction:该订单下的商品列表(list)}
    menu(Menu): 菜单
    patterns(list(set)): 频繁模式，每条模式用集合表示，按支持度-长度从高到低排序:
    total(float): 总金额
'''
```

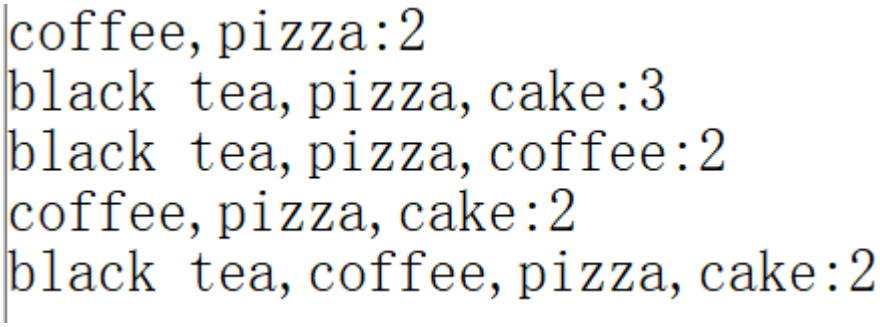
3.3 数据存储设计

为模拟真实网购系统，我们设计了三个 txt 文件模拟数据存储。

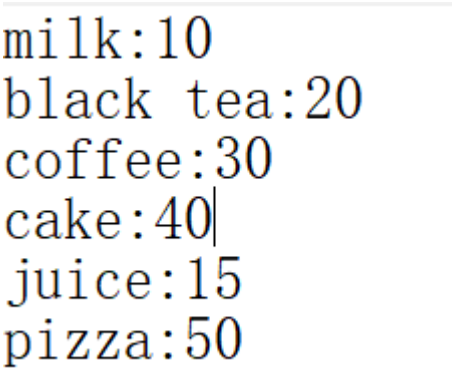
- records.txt：存储购买记录，存储形式与购买记录的数据结构设计一致。该数据在用户进入系统时被读入，并在用户买单后写入一条新记录。



- patterns.txt：存储挖掘出的频繁项集，存储格式为"item1, item2: 在购买记录中出现过的次数"。该数据在用户进入系统时被读入，主要用于根据用户已选商品进行实时推荐，并在用户买单，records.txt 被更新后自动重新挖掘频繁项集进行更新。



- menu.txt：存储商品与价格信息，存储形式与菜单的数据结构设计一致。



3.4 用户图形界面设计

GUI 的设计是一套专门的方法学。本项目中设计的 GUI 仅为更好地展示系统运行效果、突出关键功能设计，因而界面较为简陋，但具备所有的基本功能。在 GUI 的设计中使用了**树形结构**，界面上的按钮、输入框等都是窗口对象的“孩子”。本项目中的 GUI 主要包含两个窗口：主窗口与购物篮窗口。关于 GUI 的细节请见下一节的数据测试。

四、开发工具与运行平台

4.1 开发工具

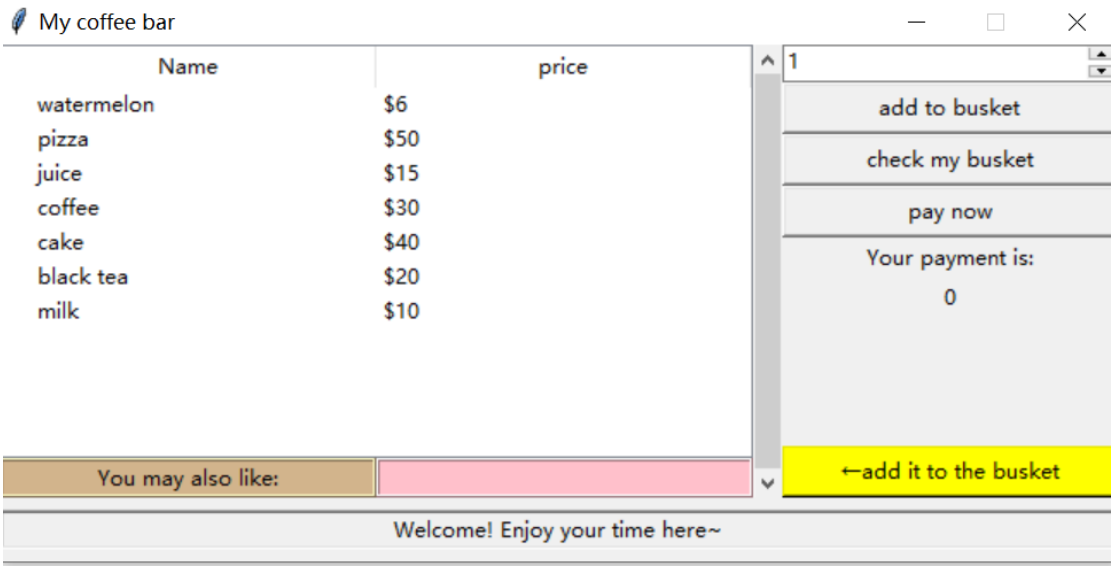
本项目采用 Python3.8 语言，使用 spyder 编译器开发。这是因为 FP-Growth 算法相对复杂，若采用 C++等更为底层的语言会使得算法的描述更加繁琐。此外，由于本项目有许多数据结构使用了哈希表进行存储，而 Python 内置的字典数据类型能够便捷地实现哈希表及其基本操作，故 Python 很适合进行本项目的开发。另外，Python 中关于 GUI 有成熟的库 tkinter，这也方便了项目中 GUI 的开发过程。

4.2 运行环境

本项目所生成的 exe 文件能在 **64 位**的系统上运行。

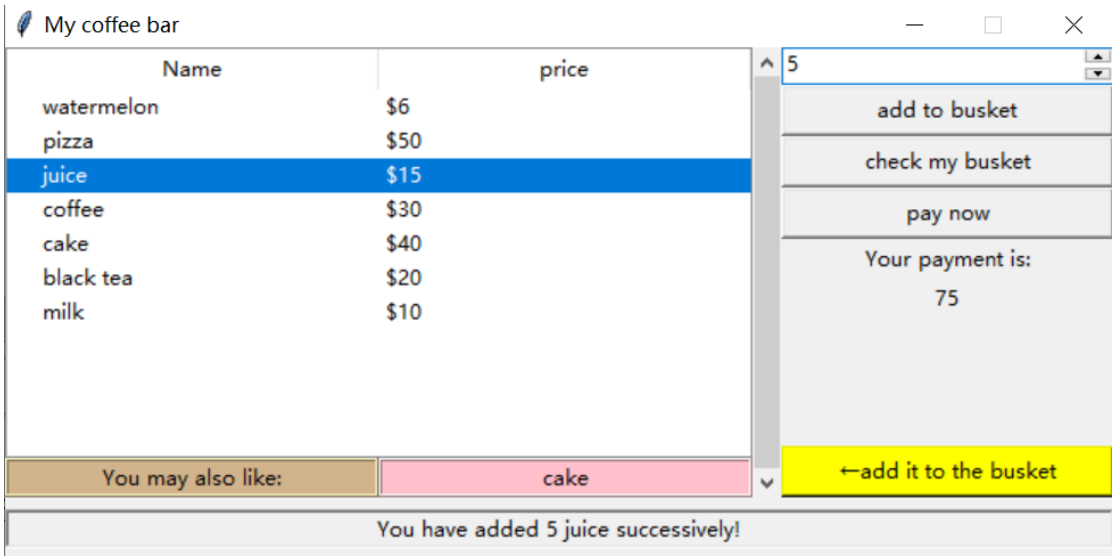
4.3 GUI 测试

本项目最终生成的 GUI 能更好地支持用户交互。进入系统时，主页面如下：

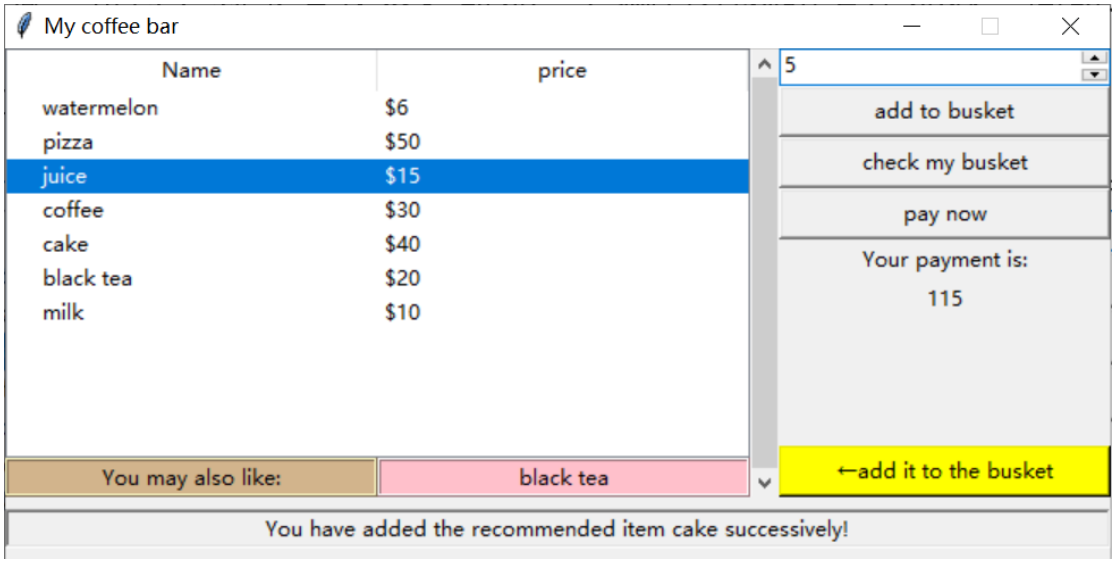


可以看到，菜单以树形结构显示，右侧有相应的功能按钮，下方有状态栏对用户进行提示。图中有颜色的部分是推荐系统的关键。

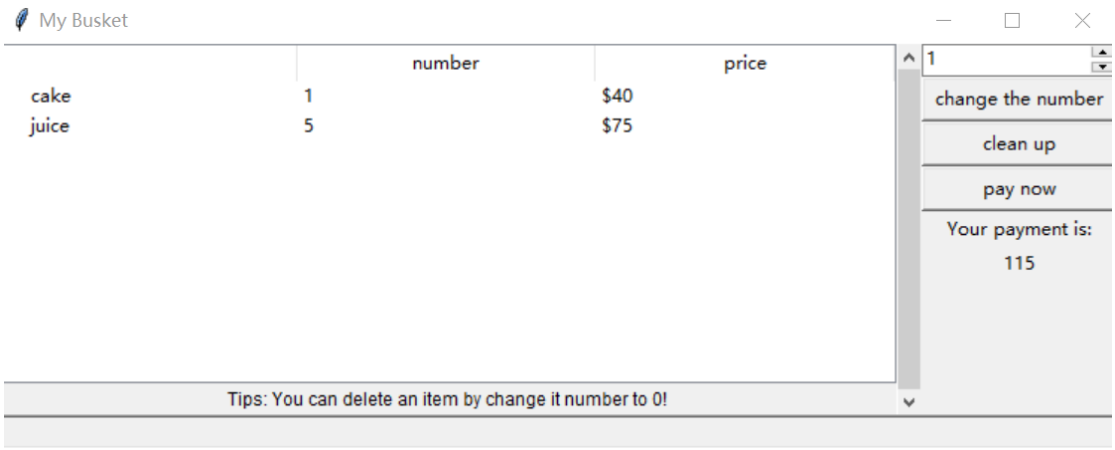
我们选中 juice，修改右上角的数目，然后点击“add to busket“进行测试。可以看到下方状态栏改变，右侧总金额也发生变化，同时推荐栏出现了 cake，用户可以选择右边的黄色按钮将其加入购物车。



如果此时选择右边的黄色按钮，变成如下情况：可以发现状态栏、总金额和推荐商品都相应的发生了改变。

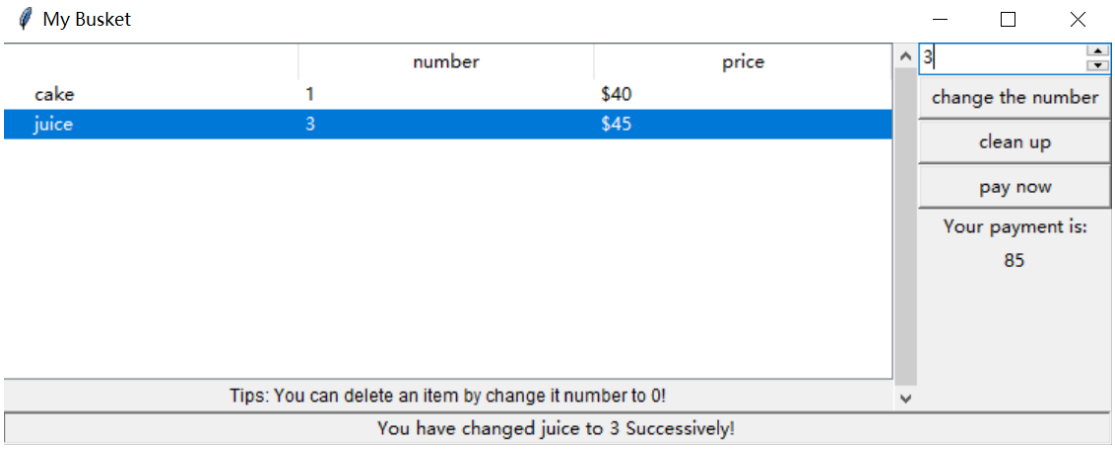


点击右上角的“check my busket”，跳转到购物篮界面。可以看到刚刚选择的两件商品都出现在购物篮中。

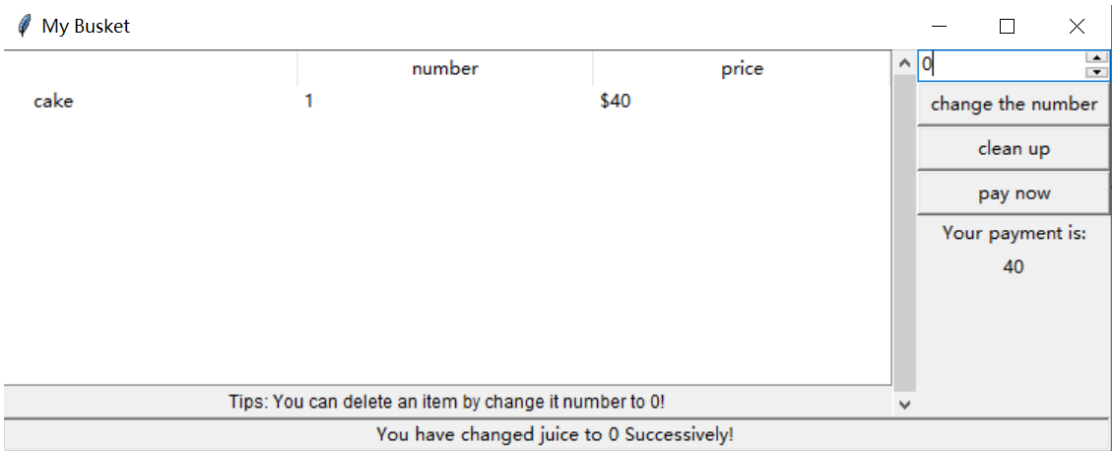


下面的 Tips 提示了用户如何删除一个商品。

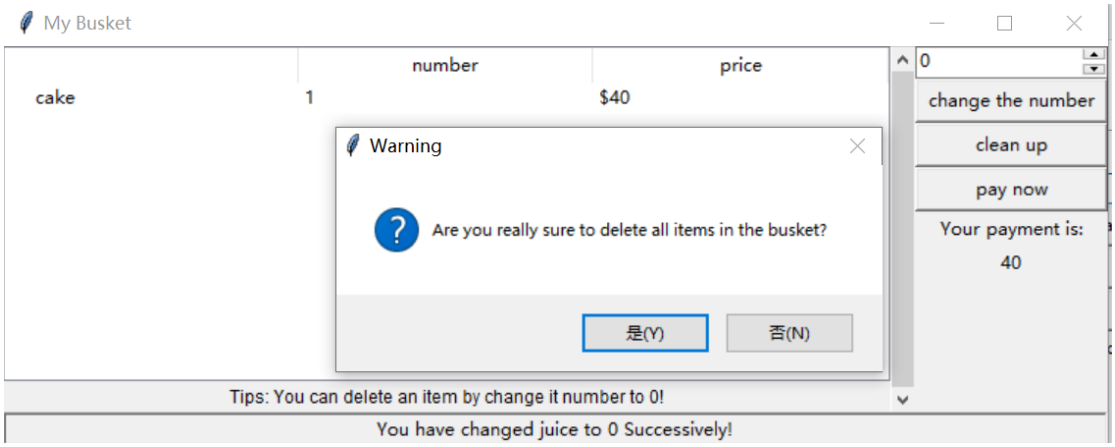
我们尝试着修改商品数量：右上角输入新的数量，点击“change the number”，可以看到状态栏、商品数和总价发生了变化：



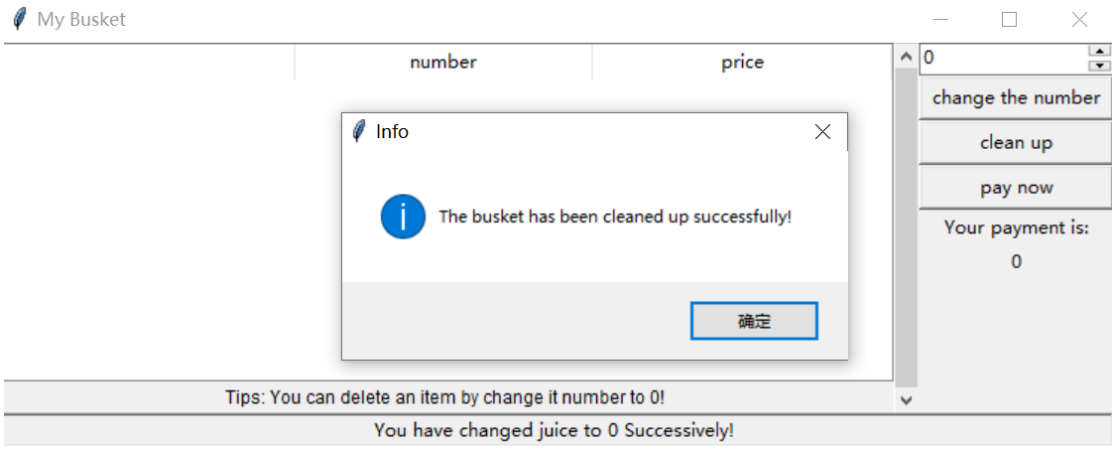
也可以直接将商品数改为 0 进行删除：



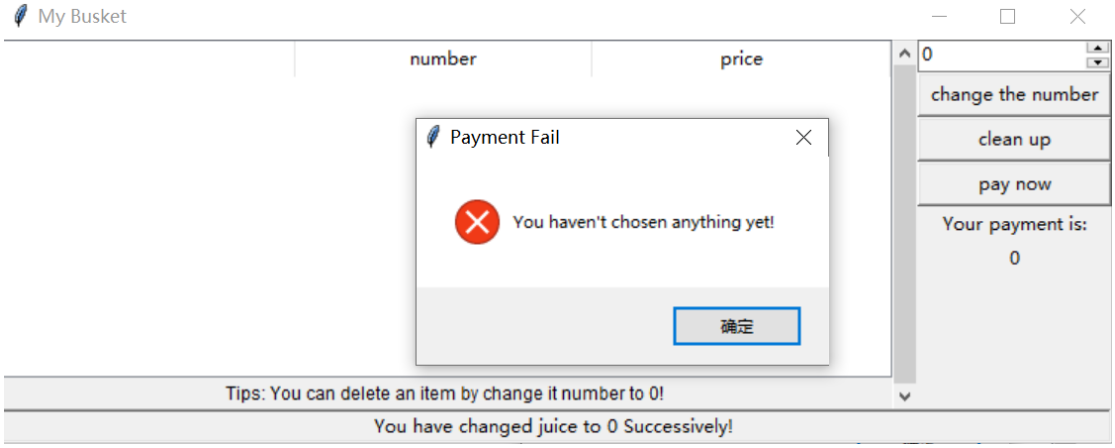
点击“clean up”按钮清空购物篮，首先跳出提示框，询问用户是否确定要清空购物篮：



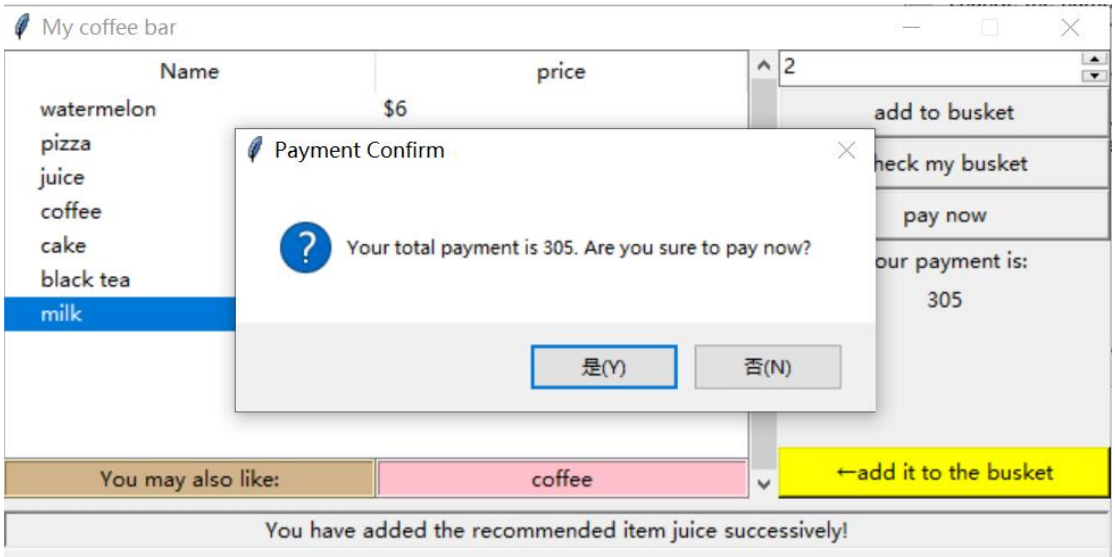
点击“是”，购物篮被清空，总价格归 0：



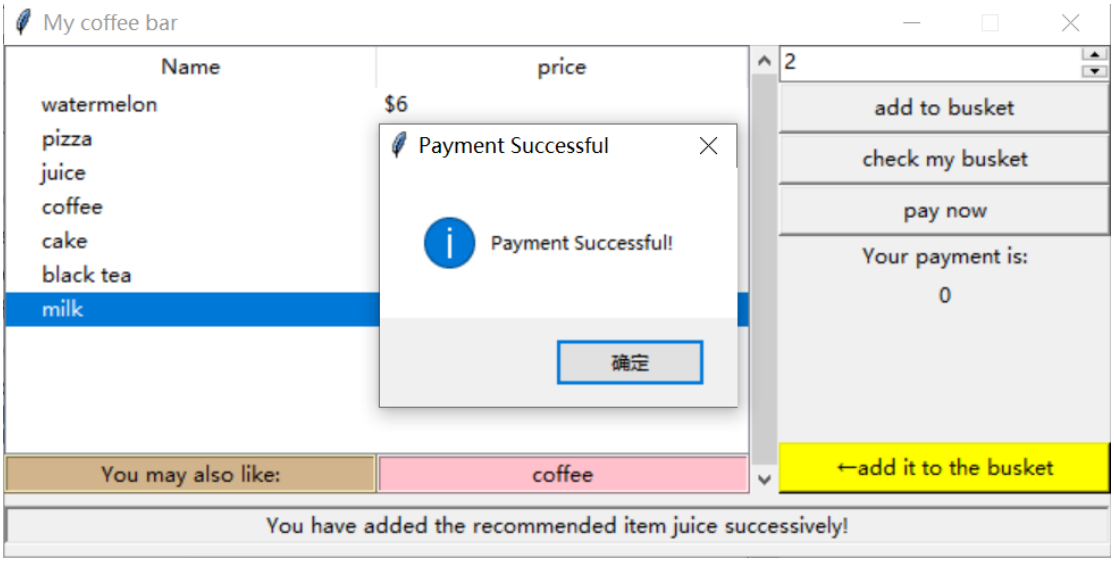
若在此时点击“pay now”尝试付款，系统会提示购物篮为空，不允许该操作:



返回主菜单，重新选取若干商品，再点击“pay now”。此时，跳出提示框，请求客户确认:



点击“是”，跳出新提示框提示用户付款成功，此时购物篮清空、总价格归 0.



查看后台的 records.txt，可以看到该条购买记录已被添加到最后（下图最后一行），其中 id 为系统为用户随机生成的 7 位数字：

```
5210214:cake
5062927:cake,black tea
2292689:juice,cake
7690850:cake,pizza,milk,coffee,black tea,juice
```

此外，我们的 GUI 对于许多非法操作都做了相应处理，使得程序更加健壮。例如，当在购物车没有选择商品时点击“change the number”，下方状态栏会提示用户先选择一个商品：

五、小结

5.1 实验过程的体会

这个大作业我陆陆续续做了很长的时间，并且不断迭代完善，感触非常深刻。总结下来，大概是如下几点：

- 实现了一个自己一直感兴趣的算法。我对数据挖掘很感兴趣，但之前对这些算法只有抽象的概念（也就是逻辑结构），比如 FP-Growth 用了树的结构和很多指针，但并不知道怎么写成代码执行。学完这门课才知道这些数据结构具体是如何实现的，在设计的时候应该根据具体需求有哪些考量等等。当然在实现过程中还遇到了很多问题，我也查阅了很多资料，最后成功实现的时候非常有成就感。
- 对面向对象的编程思想有了深入的了解。一开始我用的是函数式编程，但随着数据结构和 API 的增多，我渐渐地感受到函数式编写的程序非常不好维护，于是改成了面向对象的方法重新写了一遍，真切体会到面向对象的好处。
- 学习了 GUI 编程。其实这个项目中，花费我最多时间的是 GUI 的编写，因为我之前对 GUI 完全没有概念，以为只是一个普通的 Python 包，会调用接口函数就可以了，结果发现 GUI 有一套专门的设计思路，只知道 GUI 是由按钮、输入框组成是不够的，还要知道这些组件是如何组合构成一个完整的窗口。入门阶段非常困难，我看完了 tkinter 的官方文档，也学习了很多现成的 GUI 代码，但自己上手编

写的时候还是一头雾水。直到后来发现了一本好书叫《Think in Tkinter》[3]，专门介绍的就是如何用 Tkinter 的思想编写 GUI，而不是简单的介绍 GUI 有哪些组件。这本书是开源的，但只有英文版，不过因为写的很好也不是很妨碍阅读。读完这本书才真的入门了 GUI 编程，之后写程序就比较顺利了，我想在之前读的别人写的程序也起到了很大作用吧。

- 对 Python 更加熟悉了。

5.2 实验的不足与改进方向

- 最终生成的 exe 文件可移植性不佳：在 32 位系统上无法运行。

- 程序的健壮性仍需进一步完善，对各种非法操作进行检测与控制。

- 可以添加后台系统，加入自动进货等功能，使得该购物平台更加自动化。

- 对 GUI 可以进一步美化。

六、引用文献

[1]数据挖掘：概念与技术（原书第 3 版）[M]. 机械工业出版社, [美]韩家炜(Han. J)等著，范明，孟小峰译, 2012

[2] Kosters W.A., Pijls W., Popova V. (2003) Complexity Analysis of Depth First and FP-Growth Implementations of APRIORI. In: Perner P., Rosenfeld A. (eds) Machine Learning and Data Mining in Pattern Recognition. MLDM 2003. Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence), vol 2734. Springer, Berlin, Heidelberg

[3]Think in Tkinter. (2007, May 4) Retrieved from

http://thinkingtkinter.sourceforge.net/