商业决策平台的智能数据管理系统分析报告

卢羿舟
统计与数据科学系

更新：May 19, 2025

目录

1 项目背景

在当今快速发展和竞争激烈的商业环境中，企业对于精准、高效的决策能力的需求日益增长。为了应对这一挑战，构建一个强大的商业决策平台显得尤为重要。本项目的核心目标是打造该平台底层的智能数据管理模块，旨在对关键业务数据进行高效处理和分析，从而为上层决策提供坚实的数据支撑。

本项目需要为一家新兴数据驱动商业决策平台设计底层数据处理模块。该平台存储的主要数据包括：

1. 营销任务：储存任务的名称，紧急度，影响力
2. 表示营销任务间前置依赖的有向图
3. 表示客户关系的加权有向图
4. 商品数据：包括商品名称、价格、热度

其中所有数据类型均为数字或英文。基于这些数据，项目需要实现以下分析模块：

1. 营销任务优先调度模块
2. 客户网络与影响力传播分析模块
3. 商品数据检索模块

具体而言，**营销任务优先调度模块**维护一个存储所有营销任务的结构。该结构需要有基础的插入、删除、更改营销任务的功能，并支持执行优先级最高的任务、查看优先级最高的前 k 个任务等高级操作，其中，一个任务的优先级定义为其紧急度和影响力的乘积。同时，需要考虑营销任务之间的前置依赖关系。在第3.1节详细讨论了该模块的设计与实现。**客户网络与影响力传播分析模块**基于存储客户之间关系的加权有向图进行分析。需要评估每一位客户的重要性，并寻找每一位客户能影响到的所有其他客户，同时，当客户之间的影响路径过长或者影响力过小的时候，忽略两客户之间的影响。在第3.2节详细讨论了该模块的设计与实现。**商品数据检索模块**需要实现一个外存高效的数据结构来实现商品数据的增删改查，同时实现支持通过商品名进行前缀搜索。在第3.3节详细讨论了该模块的设计与实现。

除此之外，本报告在第2节介绍了项目源代码结构，在第4节讨论了由于时间关系其他因素导致该项目不完善的地方。对于附录，本报告在附录A中演示了运行 `app.py` 的方法以及运行结果，在附录B中展示了运行单元测试的方法以及项目每一个部分的单元测试代码，其中包含了具体的测试用例和测试方法。最后在附录C对项目开发过程中使用的 LLM 代码工具进行了详细的说明。

2 源代码文件树说明[1]

```
/
├── docs/
│   └── Project Guidelines.pdf ..... 该项目的要求说明
├── src/
│   ├── __init__.py
│   ├── data_structure/
│   │   ├── __init__.py
│   │   ├── b_plus_tree.py ..... B+ 树
│   │   ├── customer_graph.py ..... 有向赋权图
│   │   ├── dependency_graph.py ..... 有向无环图
│   │   ├── updatable_max_heap.py ..... 支持快速查找的优先队列
│   │   └── trie.py ..... Trie 树
│   ├── model/
│   │   ├── __init__.py
│   │   ├── marketing_task.py ..... MarketingTask 类
│   │   └── product.py ..... Product 类
│   ├── module/
│   │   ├── __init__.py
│   │   ├── commodity_retrieval.py ..... 商品数据检索模块
│   │   ├── customer_network_analysis.py ..... 客户网络与影响力传播分析模块
│   │   └── marketing_task_schedule.py ..... 营销任务优先调度模块
│   └── utils.py ..... 一些可视化函数
├── tests/
│   ├── __init__.py
│   ├── test_b_plus_tree.py ..... B+ 树测试代码
│   ├── test_customer_graph.py ..... 有向赋权图测试代码
│   ├── test_dependency_graph.py ..... 有向无环图测试代码
│   ├── test_dfs_influence.py ..... 影响力传播算法测试代码
│   ├── test_marketing.py ..... 营销任务调度测试代码
│   ├── test_pagerank.py ..... PageRank 算法测试代码
│   ├── test_product.py ..... Product 类测试代码
│   ├── text_trie.py ..... Trie 树测试代码
│   ├── test_updatable_max_heap.py ..... 支持快速查找的优先队列测试代码
│   └── test_weighted_degree_centrality.py ..... 度中心性算法测试代码
└── app.py ..... 系统入口，Demo 程序
```

[1] 这里省略了 Python 运行时产生的缓存文件。

3 设计思路与功能分析

该部分包括了对项目需要实现的三个模块的设计思路、实现方法、重要算法的时间与空间复杂度分析等内容。

3.1 营销任务优先调度模块

3.1.1 模块功能拆解与分析

该模块需要实现一个储存所有营销任务的数据结构，该数据结构支持对营销数据的增、删、改，同时需要根据营销任务的具体性质，自动计算其优先级。优先级的计算是明确的，定义为 紧急度 × 影响力。随后利用计算得到的优先级，考虑任务之间的相互依赖后优先执行优先级最高的任务，同时能展示任务优先级最高的 k 个任务[2]。

该模块最重要的任务是根据任务的优先级进行排列，执行优先度最高的任务与查看优先级最高的 k 个任务的要求最为符合堆的特点，所以该模块需要维护的一个核心数据结构为最大堆，堆顶始终储存优先级最高的营销任务。对于查看优先级最高的 k 个任务，由于并未提前指定 k，所以无法直接使用一个控制大小为 k 的堆。对于这个功能的实现，有两个方案

1. 每次调用该功能时，将最大堆连续弹出 k 个节点，储存后再插入恢复；
2. 每次调用该功能时，复制一份最大堆，然后进行破坏性弹出。

考虑在实际生产环境中，可能存在大量营销任务，复制一份最大堆需要的空间复杂度过高，因此在本模块的实现中最终选择了第一个方案。

为了考虑任务之间的前置依赖关系，我们需要实现一个有向图，图中的每个节点均为一个营销任务，边的指向代表了任务之间的依赖关系，具体而言，我们规定

$$\text{前置任务} \longrightarrow \text{后置任务}$$

为有向图的存储规则。由于任务之间存在依赖关系，为了更好的管理任务状态，我们考虑为任务附加一个属性 `_status`，用来表示任务当前的状态。我们将任务的状态分成三类：`Pending`，`Ready` 和 `Complete`，分别代表三种状态：该任务存在前置依赖，该任务不存在前置依赖与该任务已完成。通过维护任务依赖关系有向图来区分 `Pending` 和 `Ready` 的任务，同时考虑仅将状态为 `Ready` 的任务加入可以执行的最大堆。这样就实现了同时考虑任务优先级和前置依赖关系的营销任务规划[3]。

最后，考虑任务列表的增、删、改。注意到这三个操作都有可能直接或者间接的影响任务之间的依赖关系和优先级，其中修改任务可能会修改任务的紧急度和影响力，进而影响任务优先级，而增和删可能破坏原有的依赖结构，让原本处于 `Ready` 状态的任务重新回到 `Pending`，所以为了让最大堆能够高效地适应这些变化，我们需要让这个最大堆拥有可以直接查找的特性。考虑到这种增、删、改的操作可能是频繁的，在堆中额外维护一个查找字典是一个好的选择，尽管这样做会增加需要的存储空间，但是相比于每次查找都遍历堆来说，这种额外开销是可以接受的。因此，为了让任务能够存储在字典中，我们为定义了 `_task_id`，这个属性是任务被创建时候定义的，且在营销任务整个生命周期都不能改变，相比

[2] 由于项目要求并未说明当考虑任务相互依赖关系后，在展示任务优先级最高的 k 个任务时是否考虑任务前置依赖，由于这两种需求的实现方式类似，所以本项目在实现时仅实现了一种可能，展示的是考虑任务前置依赖下的 k 个优先级最高的任务，只有不存在前置依赖的任务才会被展示。对于不考虑任务优先级的示任务优先级最高的 k 个任务，实现原理与方法类似，故不再过多陈述。

[3] 由于时间原因，暂时实现了该较为简单的算法，在第4节中还讨论了另一个更加完善的版本。

之下任务的名称是可以被改变的。生成 _task_id 的规则为“时间戳-UUID”。考虑合并时间戳与 UUID 来作为 ID 是因为我们既希望每一个任务的 ID 独一无二，又希望它是有序的，好管理的。我们可以直接在所有的数据结构中均使用 ID 来进行统一管理。

3.1.2 功能实现

为了实现上述的功能，在该模块中，总共实现了四个类，分别为

- MarketingTask
- TaskManager
- UpdatableMaxHeap
- TaskDependencyGraph

其中 MarketingTask 表示每个任务对象，TaskManager 为储存营销任务并进行规划的类，这个类中主要维护以下两个数据结构：UpdatableMaxHeap 和 TaskDependencyGraph。他们分别用来实现根据优先级进行排列的优先队列和表示任务之间依赖关系的有向图。下面具体讲解每一个类的实现方式。

MarketingTask 类

在实现这个类之前，首先定义了三个常量

- TASK_STATUS_PENDING
- TASK_STATUS_READY
- TASK_STATUS_COMPLETED

分别表示不同的任务状态。考虑到这个类会被反复、大量初始化，为了节约内存，使用了 __slots__ 命令。在这个类中总共有六个属性，分别为

- _task_id
- _name
- _status
- urgency
- influence
- _priority

通过 @property 和 setter 来控制对这些私有变量的访问和修改。其中 _priority 并不直接初始化，而是在 urgency 和 influence 初始化和改动时自动计算并修改。这一设计让这个类本身变得相对更加安全，拒绝了可能会导致错误的改动，也实现了调用的方便，不再需要重新写 get 函数来实现对私有变量的访问。同时，MarketingTask 类还包含两个私有方法

- _update_details()[4]：更新任务的紧急程度或者影响力，并更新优先级，如果某个参数为 None，则不更新该属性，如果更新成功，则返回 True，如果没有更新，则返回 False
- _set_status()：设置任务的状态

前者负责更新该任务的可修改描述，如紧急度、影响力、名称等，后者负责设置任务的状态。MarketingTask 类不直接暴露这些方法，是因为我们只希望在 TaskManager 内部修改 MarketingTask 的状态。

[4]注意，由于美观性以及空间因素，在本分析报告中所有出现的函数都默认不填写参数名和参数类型，详细的参数说明可以参考源代码注释。

`TaskDependencyGraph` 类

为了实现高效查找节点和依赖关系，在 `TaskDependencyGraph` 类我们使用了两个字典，分别储正向依赖关系以及反向依赖关系，以及一个集合用于存储任务 ID。其中正向依赖关系指的是从某个任务节点指向其他后续任务的映射，而反向依赖关系指的是某个任务节点被哪些任务前置影响的映射。这个类中包含了常规的图应有的方法，包括

- `add_task()`：向图中添加一个任务节点
- `remove_task()`：从图中移除一个任务节点及其所有相关依赖, 并返回这个节点 id
- `add_dependency()`：添加依赖关系：prerequisite_id -> dependent_id，在添加前会进行环路检测
- `remove_dependency()`：移除依赖关系：prerequisite_id -> dependent_id
- `get_dependents()`：获取直接依赖于 task_id 的任务集合
- `get_prerequisites()`：获取 task_id 的前置任务集合
- `has_task()`：检查任务是否存在于图中
- `get_all_tasks()`：返回图中所有任务的列表

分别用于节点的增加、删除，边的增加、删除，获取正向依赖关系，获取反向依赖关系，判断节点是否在图中以及获取图中所有的任务列表。值得注意的是，在这个模块背景中，依赖关系应该是一个无环图，否则环路上的任务将永远也无法满足前置条件。因此在添加边的时候，需要检测添加这条边后会不会导致环路出现。这个检测的原理即通过深度优先搜索寻找是否存在一条待添加边的反向通路，倘若存在，则不允许添加这条边，否则会导致图中出现环。这个深度优先搜索是通过一个私有方法 `_has_path()` 实现的。这个方法通过列表模拟栈来实现了一个迭代式的深度优先搜索，可以根据搜索结果来确定是否可以在图中添加一条前置关系。另一个值得注意的是当使用 `remove_task()` 这个方法时，除了该任务，同时会删除所有和这个任务相关的边，包括前置和后置边。

`UpdatableMaxHeap` 类

这个数据结构本质是一个利用以列表实现的最小堆模拟的最大堆（存入负优先级），同时为了支持高效的删除和查找操作，这个类额外维护了一个位置映射字典，它用于将堆中存放的任务 ID 映射到该任务在列表中的索引，以此来实现对堆的高效查找和删除。这个类存在一系列最小堆的私有方法，用于实现查找父节点、左右子节点和节点交换、上浮和下沉。值得注意的是，由于我们额外添加了一个位置映射字典，所以在常规最小堆的 `_swap()` 的基础上，额外添加一步对位置映射的修改。由于上浮和下沉操作完全依赖交换操作，所以仅需要改动这里一处即可。

对于这个数据结构的公共接口，和常规最小堆的公共接口也没有太大的区别，只是需要在插入和删除的时候，增添一步对位置映射的更新即可。除此之外额外添加了一个 `update_priority()` 方法，该方法搜索给定的任务 ID，然后更新该任务在堆中的优先级，然后重新上浮或者下沉该任务节点。下面展示这个数据结构的公共接口列表。

- `insert()`：向堆中插入一个新任务，如果任务已存在，则更新优先级
- `peek_max()`：查看堆顶元素, 不移除
- `extract_max()`：移除并返回堆顶元素
- `is_empty()`：检查堆是否为空
- `update_priority()`：更新堆中已存在任务的优先级
- `delete()`：从堆中删除指定任务 ID 的元素
- `get_heap_size()`：返回堆中元素的数量

`TaskManager` 类

这个类是该模块的核心类，用于实现所有营销任务优先调度模块需要实现的功能。它维护了多个数据结构：

- `_tasks`：字典，任务 ID 到任务对象的映射
- `_task_graph`：`TaskDependencyGraph` 类，储存任务之间的依赖关系
- `_ready_queue`：`UpdatableMaxHeap` 类，储存所有状态为 Ready 的任务的优先队列
- `_in_degree`：字典，状态为 Pending 的任务 ID 到其前置依赖数量的映射

这个类通过入度字典监控所有的任务状态，当任务状态发生变化时（如出现任务和任务关系的增删改），按照相应要求更新 Ready 队列，以保证所有处于 Ready 队列中的任务均为 Ready 状态。同时提供了一个执行任务的方法，即从 Ready 队列中选择优先度最高的任务取出，并更新任务状态为完成，随后考虑该任务的后继任务，同步更新入度字典和 Ready 队列。此外，创建新任务的时候，会用 `_generate_task_id()` 为任务赋予一个独一无二的任务 ID，组成为“时间戳-UUID”。下面展示这个类的公共接口

- `add_task()`：往任务管理器中添加一个任务
- `add_dependency()`：添加一个任务间的依赖关系，如果添加的依赖会导致后继任务不再 ready，则需要将其从 ready 队列中移除
- `remove_dependency()`：移除一个已经存在的依赖关系，如果移除成功且导致后继任务的状态变为 ready，则更新 ready 队列
- `mark_task_as_completed()`：将指定任务标记为已完成，同时更新其后续任务的入度字典
- `execute_next_highest_priority_task()`：从 ready 队列中提取优先级最高的任务，并执行
- `get_top_k_ready_tasks()`：查看当前 ready 队列中的优先级最高的 k 个任务
- `update_task_info()`：更新现有的任务信息，如果更新的信息会导致优先级的变化，那么如果这个任务处于 ready 状态，会改变其在 ready 队列中的位置
- `delete_task()`：从系统中彻底删除一个任务及其所有相关依赖，包括 1. 任务列表中的任务 2. 前置依赖图中的节点和依赖的边 3. ready 队列中的任务（如果存在）4. 更新该任务的后置任务的入度字典，并考虑是否将失去前置条件的任务加入 ready 队列 5. 把该任务从入度字典中删除

这个类实现时需要注意的细节为，需要在所有可能改变依赖关系和优先级顺序的操作下监控任务状态，其余需要注意的细节都已经被抽象到之前定义的数据结构中完成了。

3.1.3 时间和空间复杂度分析

在这一节中，我们将分析该模块使用到的数据结构以及重要算法的时间和空间复杂度。要分析的数据结构包括

- `TaskDependencyGraph` 类
- `UpdatableMaxHeap` 类

由于 `TaskManager` 类的操作均为依赖以上两个数据结构接口的逻辑判断，所以在此不做更多的分析。

首先对 `TaskDependencyGraph` 类进行分析。考察数据结构的整体空间复杂度，有

- `self.nodes`，空间复杂度为 $O(V)$
- `self.adj`，空间复杂度为 $O(V+E)$
- `self.rev_adj`，空间复杂度为 $O(V+E)$

所以数据结构整体的空间复杂度为 $O(V+E)$。

表 1: TaskDependencyGraph 方法复杂度分析

方法	时间复杂度	空间复杂度
`add_task()`	$O^*(1)$	$O(1)$
`remove_task()`	$O^*(V)$	$O(V)$
`_has_path()`	$O^*(E+V)$	$O(V)$
`add_dependency()`	$O^*(E+V)$	$O(V)$
`remove_dependency()`	$O^*(1)$	$O(1)$
`get_dependents()`	$O^*(V)$	$O(V)$
`get_prerequisites()`	$O^*(V)$	$O(V)$
`has_task()`	$O^*(1)$	$O(1)$
`get_all_tasks()`	$O^*(V)$	$O(V)$

下面考虑不同方法的时间和空间复杂度。

- `add_task()`：向字典中添加一个新元素，时间和空间复杂度均为 $O(1)$，其中时间复杂度为摊销时间复杂度[5]。
- `remove_task()`：由于需要遍历所有节点，且涉及创建列表副本，所以时间复杂度为 $O^*(V)$。而由于创建了前置和后置依赖的副本，所以空间复杂度为 $O(V)$。
- `_has_path()`：注意到这是一个标准的 DFS，所以时间复杂度为 $O^*(V+E)$，而空间复杂度为 $O^(V)$。
- `add_dependency()`：该方法的主要时间和空间复杂度来自于 `_has_path()`，所以这个方法的时间复杂度为 $O^*(V+E)$，空间复杂度为 $O^(V)$。
- `remove_dependency()`：这个方法主要为字典查找和删除，所以时间复杂度和空间复杂度分别为 $O^*(1)$ 和 $O(1)$。
- `get_dependents()`：尽管查找步骤的时间复杂度为 $O(1)$，但是由于胃了防止可变对象被外部修改，返回的是结果的副本，因此时间复杂度为 $O^*(V)$，空间复杂度为 $O(V)$。
- `get_prerequisites()`：同上，时间复杂度为 $O^*(V)$，空间复杂度为 $O(V)$。
- `has_task()`：时间复杂度为 $O^*(1)$，空间复杂度为 $O(1)$。
- `get_all_tasks()`：时间复杂度为 $O^*(V)$，空间复杂度为 $O(V)$。

以上分析的结果总结可以查看表1。下面我们继续对 `UpdatableMaxHeap` 类的分析。显然，该数据结构本身的空间复杂度为 $O(N)$。

- `insert()`：无论是插入还是更新，都涉及堆的上浮与下沉操作，加上位置映射字典的更新，时间复杂度为 $O^*(\log N)$，空间复杂度为 $O(1)$。
- `peek_max()`：查看堆顶，时间和空间复杂度均为 $O(1)$。
- `extract_max()`：从堆顶取出节点，涉及位置映射字典的更新与下沉操作，时间复杂度为 $O^*(\log N)$，空间复杂度为 $O(1)$。
- `is_empty()`：时间和空间复杂度均为 $O(1)$。
- `update_priority()`：涉及查找与更新堆，并更新位置映射字典，所以时间复杂度为 $O^*(\log N)$，空间复杂度为 $O(1)$。
- `delete()`：类似地，同样以上浮和下沉操作为主，时间复杂度为 $O^*(\log N)$，空间复杂度为 $O(1)$。
- `get_heap_size()`：时间和空间复杂度均为 $O(1)$。

对 `UpdatableMaxHeap` 类的分析总结可以查看表2。

[5]在接下来，我们使用 $O^*(\cdot)$ 来标记摊销时间复杂度。

表 2: UpdatableMaxHeap 方法复杂度分析

方法	时间复杂度	空间复杂度
`insert()`	$O^*(\log N)$	$O(1)$
`peek_max()`	$O(1)$	$O(1)$
`extract_max()`	$O^*(\log N)$	$O(1)$
`is_empty()`	$O(1)$	$O(1)$
`update_priority()`	$O^*(\log N)$	$O(1)$
`delete()`	$O^*(\log N)$	$O(1)$
`get_heap_size()`	$O(1)$	$O(1)$

3.2 客户网络与影响力传播分析模块

3.2.1 模块功能拆解与分析

这个模块需要实现对客户关系网络的分析。首先我们定义所有客户与相互的影响关系被储存在一个有向图（这里的图可以有环）中，每个节点均表示一个客户。他们之间的影响关系如下

$$\text{客户}A \xrightarrow{0.8} \text{客户}B$$

代表客户 A 可以直接影响客户 B，而边权代表客户 A 对客户 B 的影响程度，这个数值的取值范围为了分析方便被限制在了 $(0, 1]$，数值越大代表影响程度越深，从而依靠这种关系来构建整个有向图。在这个模块中总共有两个分析任务：

1. 客户重要性评价
2. 寻找每一位客户所能影响到的所有客户

这两个任务都是图数据结构中十分经典的任务。对于客户重要性评价，本质上是对有向图中的节点重要性进行评价。这样的评价指标有许多。在这个项目中，我们选择了两个比较经典的算法：加权度中心性和 PageRank。下面简要介绍。

要衡量一个图中节点的重要性，一个很直观的方法就是看他的入边和出边有多少。这类似于评价一个交通枢纽的重要程度，它能够连接起越多的路，一般而言就越重要。这个评价指标就被称为加权度中心性（**W**eighted **D**egree **C**entrality），加权度中心性越高，代表节点的重要性越高。在有向赋权图中，这个指标进一步变为加权入度中心性（WIDC）和加权出度中心性（WODC），这是为了区分不同边的性质和不同的边权，具体计算方式如下

$$WIDC(o) = \sum_{u \in in(o)} w(u, o), \quad WODC(o) = \sum_{u \in out(o)} w(o, u),$$

其中 $in(o)$ 和 $out(o)$ 分别代表节点 o 的所有入射节点和出射节点。

然而，度中心性存在一个很显然的问题：它没办法衡量相距超过 1 的点之间的影响。在我们的例子中，客户 A 如果能通过客户 B 而对 C 有影响，那么客户 A 的重要程度也应该能传导到对客户 C 的重要程度的评价上，而度中心性显然无法衡量这一影响。因此为了给管理者提供更加全面的决策参考，我们引入了 PageRank 算法 [Page et al., 1999] 来衡量节点重要性。PageRank 最开始被用于评价网页重要性，其核心思想基于这样一种直观的理念：一个网页的重要性取决于指向它的其他网页的数量和质量。简单来说，PageRank 算法模拟了一个在互联网上随机冲浪的用户。用户从一个随机的网页开始，然后不断地点击页面上的链接。一个网页被访问的概率越高，那么这个网页就被认为越重要。具体来说，PageRank 算

Algorithm 1 适用于有向赋权图的 PageRank 算法

Require: 有向赋权图 $G = (V, E, W)$；阻尼因子 d；最大迭代次数 N_{max}；收敛阈值 ϵ。
Ensure: 各节点 $u \in V$ 的 PageRank 得分 $PR[u]$。

若 V 为空, 则返回空结果。
预处理: 计算图中各节点的入链信息 $InMap[u]$ (包含源节点 v 及权重 W_{vu}) 及加权出度 $W_{out}[v]$。
初始化: 对所有节点 $u \in V$, $PR[u] \leftarrow 1/|V|$。
for $iteration = 1$ **to** N_{max} **do**
    $PR_{old} \leftarrow PR$
    $S_{dangling} \leftarrow \sum_{v \in V \text{ s.t. } W_{out}[v]=0} PR_{old}[v]$ ▷ 计算悬挂节点的 PageRank 总和
    for 每个节点 $u \in V$ **do**
        $P_{links} \leftarrow 0$
        if $InMap[u]$ 非空 **then**
            for 每个 $(v, W_{vu}) \in InMap[u]$ **do**
                if $W_{out}[v] > 0$ **then**
                    $P_{links} \leftarrow P_{links} + \frac{PR_{old}[v] \cdot W_{vu}}{W_{out}[v]}$
                end if
            end for
        end if
        $PR[u] \leftarrow \frac{1-d}{|V|} + d \cdot P_{links} + d \cdot \frac{S_{dangling}}{|V|}$
    end for
    if $\sum_{u \in V} |PR[u] - PR_{old}[u]| < \epsilon$ **then**
        break ▷ 已收敛
    end if
end for
return PR

法将互联网看作一个有向图，其中网页是节点，网页之间的链接是边。每个网页的 PageRank 值（PR 值）是通过迭代计算得到的。一个网页的 PR 值由所有指向它的网页的 PR 值贡献而来。指向它的网页越重要（PR 值越高），它获得的贡献也就越多。同时，一个网页会将自己的 PR 值平均分配给它链接出去的所有网页。为了解决一些特殊情况，比如没有出链的“终止点”网页（dangling nodes）或者互相链接但不链接到外部的“陷阱”网页（spider traps），PageRank 算法引入了阻尼因子 (damping factor)，通常设为 0.85。这个阻尼因子代表了用户在任何一步都有一定的概率 (1 − 阻尼因子) 会随机跳转到互联网上的任意一个网页，而不是继续点击当前页面的链接。这确保了所有网页的 PR 值最终会收敛到一个稳定的值。适用于有向赋权图的 PageRank 算法伪代码详见算法1。在这个项目中，我们实现了加权出度中心性，加权出度中心性和 PageRank 算法。

第二个任务是寻找每一个客户所能影响到的所有客户。倘若不考虑相距太远和距离和过小的影响权重，一个最直接的方式是从图中每一个节点出发，进行深度优先搜索，然后将所有途径的点都加入一个列表，这个列表即起始节点能影响到的所有节点。为了综合考虑两个节点的距离和权重对影响程度的影响，拓展两个节点之间的影响程度的定义从相邻节点的边权到路径影响力。具体而言，对于节点 u 和节点 v，设这两个节点之间的路径集合为 $\mathcal{L}$，则这两点之间的路径影响力定义为

$$IF(u, v) = \max_{l \in \mathcal{L}} \prod_{o, o' \in l} w(o, o'),$$

即为路径途径所有边的边权乘积中的最大值。由于之前限制了边权的取值范围为 $(0, 1]$，所以最终的路径影响力取值范围也应该为 $(0, 1]$，且影响力值随着路径延长单调递减，我们可以很容易设置一个截断阈值

来获得在一定范围和影响力内的客户影响力集合。倘若不设置截断阈值，算法就退化为不考虑距离和边权的深度优先搜索。

3.2.2 功能实现

为了实现对网络的分析，首先需要实现一个允许环路的有向赋权图。这一部分在真实开发环境中可以和前一模块的有向图继承自同一个基类，但是为了代码的清晰和易于理解，这里单独实现了一个 `CustomerGraph` 类。`CustomerGraph` 类通过字典实现的邻接列表来储存图，这个数据结构实现了以下公共接口：

- `add_customer()`：向图中添加一个新客户（节点）如果客户已存在，则不执行任何操作
- `get_all_customers()`：返回图中所有客户名称的列表
- `add_influence()`：在两个客户之间添加一条有向的影响力关系（边）如果'from_customer' 或 'to_customer' 不在图中，会先将它们添加进来
- `get_direct_influencees()`：获取一个客户直接影响的所有其他客户及其对应的影响力权重
- `get_influence_weight()`：获取从 from_customer 到 to_customer 的直接影响力权重

基于这个数据结构，我们实现了以下四个算法：

- `calculate_weighted_out_degree_centrality()`
- `calculate_weighted_in_degree_centrality()`
- `calculate_pagerank()`
- `analyze_all_customer_influence_nodes()`

其中关于度中心性的算法只需要简单迭代调用 `CustomerGraph` 类的公共接口即可。计算 PageRank 的算法依赖一个辅助函数 `_preprocess_for_pagerank()`，对图数据构建计算 PageRank 需要的入链字典和加权出度节点。随后按照算法1中给出的流程不断迭代直至收敛即可得到 PageRank 值。

`analyze_all_customer_influence_nodes()` 的实现相对较为复杂，它依赖两个辅助函数。首先是一个深度优先搜索函数 `_dfs_explore_influence()`，它给定起始点和一个起始点的相邻节点，通过用列表模拟的栈进行迭代深度优先搜索。同时它还接受一个最小路径影响力的阈值，当路径影响力小余该阈值时剪枝。第二个辅助函数是 `_calculate_influenced_nodes_for_single_customer()`，这个函数利用前一个辅助函数计算从某一个节点出发，在达到阈值前能够遍历到的节点。最后通过对所所有节点调用该函数，即可得到所有的能够影响到的节点。注意，由于在这个模块中并不要求图中无环，所以需要识别环路，当出现环路时要及时剪枝，防止无意义迭代甚至死循环（当最小影响力阈值设置成 0 的时候）。`_dfs_explore_influence()` 通过检查新进入栈的节点是否与起始节点相同来避免这个问题。

3.2.3 时间和空间复杂度分析

在这一节中，我们将分析该模块使用到的数据结构以及重要算法的时间和空间复杂度。要分析的数据结构为 `CustomerGraph` 类，同时也将分析以下算法的时间和空间复杂度：

- `calculate_weighted_out_degree_centrality()`
- `calculate_weighted_in_degree_centrality()`
- `calculate_pagerank()`
- `analyze_all_customer_influence_nodes()`

表 3: CustomerGraph 方法复杂度分析

方法	时间复杂度	空间复杂度
`add_customer()`	$O^*(1)$	$O(1)$
`get_all_customers()`	$O^*(V)$	$O(V)$
`add_influence()`	$O^*(1)$	$O(1)$
`get_direct_influencees()`	$O^*(V)$	$O(V)$
`get_influence_weight()`	$O^*(1)$	$O(1)$

首先分析 `CustomerGraph` 类，该数据结构只使用了一个字典来实现邻接列表存储的有向图，所以数据结构本身的空间复杂度为 $O(V+E)$。下面分析公共接口的时间和空间复杂度。

- `add_customer()`：往字典中添加一条键值对，所以时间复杂度和空间复杂度分别为 $O^*(1)$ 和 $O(1)$。
- `get_all_customers()`：返回客户名称的列表，存在转化为 list 的操作，所以时间复杂度为 $O^*(V)$，空间复杂度为 $O(V)$。
- `add_influence()`：时间复杂度和空间复杂度分别为 $O^*(1)$ 和 $O(1)$。
- `get_direct_influencees()`：为查找与复制的操作，所以时间复杂度为 $O^*(V)$，空间复杂度为 $O(V)$。
- `get_influence_weight()`：涉及字典查找，时间复杂度和空间复杂度分别为 $O^*(1)$ 和 $O(1)$。

对 `CustomerGraph` 类的分析总结可以查看表3。下面分析四个函数的时间和空间复杂度。

对于函数 `calculate_weighted_out_degree_centrality()`：

1. 调用 `get_all_customers()`，时间复杂度为 $O^*(V)$，空间复杂度为 $O(V)$。
2. 对于每一个出边，累加其边权，由于总共有 E 条边，所以时间复杂度为 $O(E)$.
3. 最终返回每个节点的出度中心性字典，空间复杂度为 $O(V)$.

所以 `calculate_weighted_out_degree_centrality()` 的时间复杂度为 $O^*(V+E)$，空间复杂度为 $O(V)$。

对于函数 `calculate_weighted_in_degree_centrality()`：

1. 调用 `get_all_customers()`，时间复杂度为 $O^*(V)$，空间复杂度为 $O(V)$。
2. 初始化度中心性字典，时间复杂度为 $O^*(V)$，空间复杂度为 $O(V)$.
3. 对于每一个入边，累加其边权，由于总共有 E 条边，所以时间复杂度为 $O(E)$.
4. 最终返回每个节点的入度中心性字典，空间复杂度为 $O(V)$.

所以 `calculate_weighted_in_degree_centrality()` 的时间复杂度为 $O^*(V+E)$，空间复杂度为 $O(V)$。

对于函数 `calculate_pagerank()`：

1. 构建入链字典和加权出度，时间复杂度为 $O^*(V+E)$，空间复杂度为 $O(V+E)$.
2. 初始化 PR 值，时间复杂度为 $O^*(V)$，空间复杂度为 $O(V)$.
3. 进入主循环，若最大迭代次数为 K 次，每次迭代时间复杂度为 $O(V+E)$

所以 `calculate_pagerank()` 的时间复杂度为 $O^*(V+E)$，空间复杂度为 $O(V+E)$。

对于函数 `analyze_all_customer_influence_nodes()`，发现其时间复杂度依赖图的拓扑结构，在最坏情况下为遍历完全图中所有的节点对的所有路径，该时间复杂度超过 $O(V!)$，不是一个有意义的时间复杂度上界，而空间复杂度由 DFS 主导，所以空间复杂度为 $O(V^2)$。

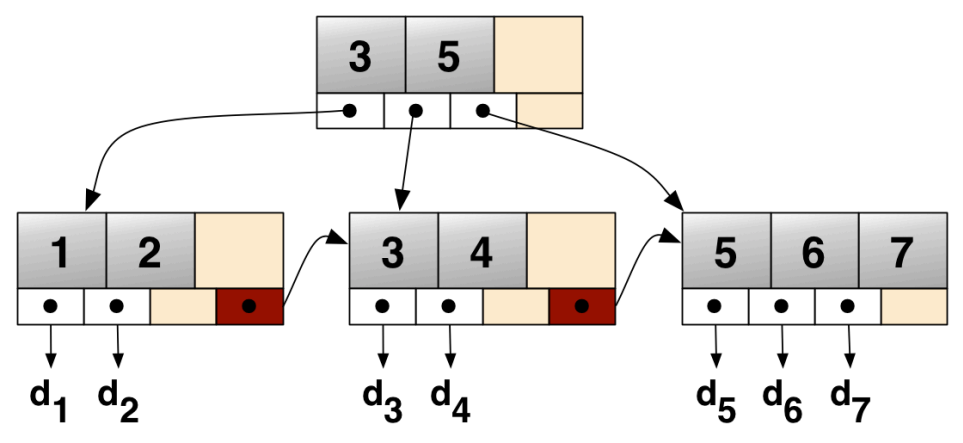

图 1: B+ 树的基本结构

3.3 商品数据检索模块

3.3.1 模块功能拆解与分析

在这个模块中，我们需要维护一个管理所有商品的结构，需要实现的功能如下：

- 高效插入、删除、更改商品信息和搜索
- 搜索任意价格范围内的所有商品
- 根据商品名称前缀进行前缀搜索

同时，还需要考虑由于商品数目过多，无法直接读取到内存进行处理。也就是说，我们需要考虑一种外存高效的数据结构来实现。即需要尽可能减少读取硬盘的操作，因为相比于读取内存，读取硬盘的速度过于慢了，这也是为什么二叉搜索树的性能无法令人满意。在这种情况下，B+ 树因为其高扇出与叶节点内存储的值可以在硬盘中连续存储的特点体现出优异的性能（见图1）。B+ 树与二叉树最主要的区别为，不同于二叉树每个节点最多有两个子节点，B+ 树的每个节点可以有多个子节点，且通过借用节点、聚合节点以及分裂节点，将每个节点的子节点数量控制在一定范围内。同时 B+ 树的节点分为内部节点和叶节点，内部节点仅储存子节点的指针，只有叶子节点才存储键值对。这使得叶子结点中的键值对在硬盘中是连续存储的，在读取时可以通过直接读取一整个叶子节点来实现搜索，从而大大提升效率。同时，由于其键之间存储是天然有序的，加上其相邻叶节点以双向链表的方式存储，所以也可以高效支持范围查询。因此本模块优先考虑通过 B+ 树来作为主要数据结构存储商品数据。

对于根据商品名称前缀进行前缀搜索任务，可以由 Trie 树实现。Trie 是一种天然支持前缀搜索的树，其每一个节点都代表字符串中的某一位字符（如图2所示），因此如果想要实现前缀搜索，只需要搜索到前缀末尾的那个字符对应的节点，然后遍历即可。同时，由于我们可以假定商品的名称不会过于长，且 Trie 树搜索匹配次数依赖字符串长度，所以尽管普通 Trie 树并不能高效处理外存查找，但是也是可以接受的实现方案。在第4节中，我们还讨论了一些外存高效的用于前缀匹配的数据结构。

在确定了使用的数据结构后，考虑该模块的设计。首先确认的是，我们应该允许用户对商品名称的修改，所以为了能够正常索引，如之前 `MarketingTask` 类一样，我们需要为每个商品设计独一无二的 ID。在这里考虑和此前相同的“时间戳-UUID”方案。为了和 `MarketingTask` 类的 ID 区分，我们额外为商品类的 ID 前缀增加了“PROD-”。其次，我们需要确认模块需要支持哪些查找方式。首先针对商品名的查找可以通过 Trie 树实现。其次需要支持对价格的查询，包括对价格的精确查询和对价格的范围查询。最后，作

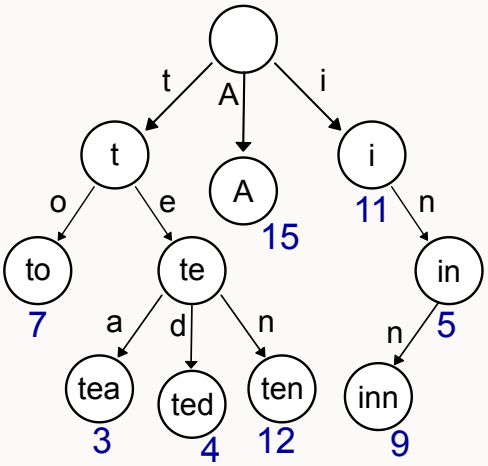

图 2: 一个保存了 8 个键的 trie 结构，”A”, ”to”, ”tea”, ”ted”, ”ten”, ”i”, ”in”, ”inn”

为商品唯一的不变属性，我们还希望能够直接通过 ID 查询。为了同时支持这些查询方式，同时让存储最优，我们借用了数据库中主键和次键的概念，定义主键为商品 ID，次键为商品价格和商品名称。对于主键，我们采用一个 B+ 树来储存“键-对象”键值对，而对于次键，我们仅存储“商品名-商品 ID”和“价格-商品 ID”键值对[6]。

值得注意的是，完整的 B+ 树实现较为复杂。B+ 树本身就存在检查叶节点和内部节点的上溢和下溢，并相应分裂、借用邻居键、合并等操作，有大量需要实现的细节。不仅如此，对于完整的 B+ 树，还需要考虑节点与磁盘块的对应（Node-to-Disk-Block Mapping）、节点对象的序列化与反序列化、缓冲池管理、并发控制和恢复等。出于学习目的，以及时间限制，该模块的所有操作均在内存内直接完成，仅仅完整体现了 B+ 树的核心思想和方法。

3.3.2 功能实现

为了实现上述的功能，在该模块中，总共实现了以下类

- `BPlusTreeNode`
- `BaseBPlusTree`
- `BPlusTreeProducts`
- `BPlusTreeID`
- `TrieNode`
- `ProductPrefixTrie`
- `Product`
- `ProductManager`

其中，`BPlusTreeNode` 为 B+ 树的节点类，定义了叶节点和内部节点的不同行为，且为不同的节点提供了键数量上溢和下溢的检查（根节点除外）和是否可以给兄弟节点借用键的检查。`BaseBPlusTree` 是一个 B+ 树的基类，它定义了 B+ 树的一些基本行为，包括节点分裂、节点借用、节点合并、对上溢和下溢的处理等。`BPlusTreeProducts` 和 `BPlusTreeID` 继承了这个基类，同时根据不同的存储目的实现了不同的查找、删除、插入逻辑。具体而言，`BPlusTreeProducts` 中每个值包含的是一个列表，列表中存储的是

[6]注意，由于本模块仅实现原理，并未实现真正的外存操作，所以这样的设计对该模块本身的效率提升不大，但是这种设计思路在真实环境中有许多好处。

相同价格的商品 ID，BPlusTreeID 中每个值是一个 Product 对象，所以他们在具体的插入查找删除逻辑上存在一些差异，且 BPlusTreeProducts 还额外支持了查找一定价格范围内所有商品 ID。

TrieNode 定义了 Trie 树的节点，每个节点代表一个字符，并且额外存在一个布尔变量用来标记该节点是否为一个字段的末尾，如果布尔变量为真，那么该节点存储一个包含所有同名商品的 ID 的集合。基于这个类，ProductPrefixTrie 实现了存储商品名的 Trie 树，不仅实现了正常插入、前缀匹配逻辑，还实现了一个基于回溯的节点删除逻辑，与标准的 Trie 树相比，这样优化了存储空间，减少了冗余节点数量。

最后是 Product 和 ProductManager 类。与 MarketingTask 类类似，Product 类通过 @property 和 setter 来规定对其属性的访问和修改操作。ProductManager 基于前一节的分析，实现了通过两个 B+ 树和一个 Trie 树的高效增删改查和前缀匹配操作。

该模块内容较多，且有大量类似重复逻辑，所以在此选择较为重要的部分讲解，具体实现过程请参考源代码。下面以 BaseBPlusTree 和 BPlusTreeID 为例讲解 B+ 树的实现。首先简要介绍 BPlusTreeNode

BPlusTreeNode

该类用于表示 B+ 树中的内部节点和叶节点。对于内部节点和叶节点具有以下通用属性

- order
- is_leaf
- parent
- keys

分别用来表示 B+ 树的阶（用于计算节点键数的上下界），是否为叶节点，父节点的指针和键列表。对于内部节点，有属性

- chileren

用于表示子节点的索引，而对于叶子节点有属性

- values
- next_leaf
- prev_leaf

分别用来存储每个键对应的值，指向左兄弟的指针和指向右兄弟的指针用于快速查找。同时实现了以下方法：

- is_overflow()：检查节点的键数量是否超过上限
- min_keys_for_node()：返回此节点（如果非根）应包含的最小键数
- is_deficient()：检查节点是否下溢
- can_lend_key()：检查节点是否有富余的键可以借给兄弟节点（即键数量 > 最小允许数量）不考虑根节点，因为根节点不参与这种借用
- get_num_keys()：返回当前节点中的键数量
- get_num_children()：返回当前内部节点的子节点数量

下面首先讲解 BPlusTreeID，将一些细节操作抽象到基类之后的具体增删查操作的实现[7]。

search()：精确查找具有指定 ID 的商品

[7]对于 B+ 树的改操作，本质上是查找后删除，然后重新插入新的节点，所以被整合进了 ProductManager 类

首先通过辅助函数 `_find_leaf_node()` 找到键所在的叶节点，然后在叶节点中遍历查找具体的对象实现查找

`insert()`：向 B+ 树中插入一个商品

首先通过 `_find_leaf_node()` 找到键应该存入的叶节点，然后使用 `_insert_into_leaf()` 将值插入该节点，随后检查当前节点是否出现上溢，如果上溢，利用辅助函数 `_split_leaf()` 实现上溢处理。

`_insert_into_leaf()`：辅助函数：将商品插入到指定的叶节点中

由于叶节点中键的存放是有序的，所以通过二分查找寻找键应该插入的位置后直接插入

`delete()`：从 B+ 树中删除一个具有指定 product_id 的商品

首先通过 `_find_leaf_node()` 找到键应该所在的叶节点，然后检查该节点内是否存在该键，如果存在，则直接从键列表和值列表中均弹出该键值对，最后检查删除该键后节点是否出现下溢。需要注意的是，对于根节点需要额外判断，因为 B+ 树允许根节点为叶节点时不设置下溢阈值.

可以看到，`BPlusTreeID` 的实现较为简单，是因为 `BaseBPlusTree` 隐藏了大量的底层细节，下面一一讲解

`_find_leaf_node()`：根据输入的键，找到这个键对应的叶子结点

从根开始，查找输入键在每一个内部节点应该进入的下一个位置，直到到达叶子结点

`_split_leaf()`：辅助函数：分裂一个已满的叶节点

创建一个新的叶子结点，然后从原节点差不多中间的位置裂开，将右边的键值对赋予新的叶子节点，然后在原节点中删除，随后更新节点之间的链表指针，最后以右边节点的第一个键作为上推到父节点的键，使用 `_insert_into_parent()` 上推到父亲节点。

`_insert_into_parent()`：辅助函数：在父节点中插入一个键和指向新右子节点的指针

获取当前节点的父节点，如果父节点为 None，说明当前节点为根节点，对于这种情况，需要创建一个新的以属性为内部节点的根节点，否则，找到在父节点孩子列表中的插入位置，同步更新父节点的键列表和值列表，最后检查父节点是否因为这次插入溢出，如果溢出，使用 `_split_internal_node()` 处理。

`_split_internal_node()`：辅助函数：分裂一个已满的内部节点

类似 `_split_leaf()` 的实现逻辑，只是需要注意对于内部节点，键只表示左右界，而不像叶节点的一一对应，所以需要注意分裂位置和修改位置的选取。当实现分裂后，重新调用 `_insert_into_parent()` 将分裂得到的新键上插到父节点。

`_handle_leaf_node_underflow()`：处理叶节点下溢。尝试从兄弟节点借用，否则进行合并

根据当前节点的位置以及左右兄弟是否可以借出键（借出后不会导致下溢），考虑以下四种操作：和左兄弟借用键、和右兄弟借用键、和左兄弟合并、和右兄弟合并。如果发生了合并，需要更新父节点中的键，导致父节点中的键变少，所以检查父节点是否发生下溢，调用 `_handle_internal_node_underflow()` 处理。如果父节点是根节点，也需要调用上述函数，在该函数中处理根节点的各种情况。

`_handle_internal_node_underflow()`：处理内部节点下溢。尝试从兄弟节点借用，否则进行合并

表 4: BPlusTreeProducts 方法复杂度分析 (N 为商品条目总数, m 为 B+ 树阶数)

方法	时间复杂度	空间复杂度
`insert()`	$O^*(m \log_m N)$	$O(m)$
`search_exact()`	$O^*(\log_m N \cdot \log m)$	$O(1)$
`search_range()`	$O^*(\log_m N \cdot \log m)$	$O(1)$
`delete()`	$O^*(m \log_m N)$	$O(m)$

同样的借用、合并逻辑，只是需要额外对根节点进行判断，当前节点是根节点且没有键了，那么需要降低树高，将唯一一个孩子节点上提作为新的根节点（如果根节点作为内部节点没有键了，说明它只有一个孩子）。如果不是根节点，同样的，如果发生了合并，需要检查其父节点是否出现下溢，并递归调用该函数。

关于 Trie 树的实现，较为简单，所以在此不做更多的分析。

3.3.3 时间和空间复杂度分析

本节会对该模块中使用的 B+ 树和 Trie 树的基本操作进行时间和空间复杂度进行分析，其他操作均为这些操作的叠加，所以不另外做分析。对于 B+ 树，如上节，选取 `BPlusTreeProducts` 为例子作为分析，两 B+ 树性质类似，对于另一个 B+ 树不做单独分析。

首先考虑 `BPlusTreeProducts`。设该 B+ 树的 order 为 m，存储的商品数量为 N：

- `insert()`：对于插入，类似二叉搜索树的插入，只是每个节点延伸的子节点数量为 $O(m)$，所以单独插入的时间复杂度为 $O^*(\log m \cdot \log_m N)$，为每一步的二分查找乘以树高，但是如果发生了节点分裂，就会变成 $O^*(m \log_m N)$，所以最终的时间复杂度为 $O^*(m \log_m N)$。而空间复杂度考虑分裂节点时创建的新节点，为 $O(m)$。
- `search_exact()`：对于搜索，其主导时间复杂度还是查找叶节点，所以时间复杂度为 $O^*(\log m \cdot \log_m N)$，空间复杂度与查找到的节点数量有关，这里假设每次查找返回的商品数量为 $O(1)$，即为空间复杂度。
- `search_range()`：对于范围搜索，时间复杂度依赖其范围有多大，假设范围横跨的节点数量为 $O(1)$（范围不太大），那么范围搜索的时间复杂度仍然为 $O^*(\log m \cdot \log_m N)$，空间复杂度为 $O(1)$[8]。
- `delete()`：这一步的主导复杂度为处理下溢，所以时间复杂度为 $O^*(m \log_m N)$，空间复杂度为 $O(m)$。

分析的总结可以参考表4。下面分析 `ProductPrefixTrie` 的时间和空间复杂度，设 L 为输入的商品名或前缀长度

- `insert()`：插入每一步对商品的一个字符拓展一个节点，所以时间复杂度为 $O(L)$，空间复杂度为 $O(L)$
- `get_product_ids_with_prefix()`：考虑该前缀下的商品数量不太多，所以搜索不会覆盖大部分的 Trie 树，因此主要复杂度仍然在搜索前缀上，时间和空间复杂度均为 $O(L)$。
- `delete()`：搜索和回溯清理的时间和空间复杂度均为 $O(L)$，所以总的时间复杂度和空间复杂度为 $O(L)$

[8]在真实场景中，可能横跨节点极大，这里只是简化分析，如果横跨节点数量很大，那么需要在时间复杂度中加入这一项，以及相应的添加节点的时间成本。同样的，对于下文的 Trie 树，对于复杂度严重依赖查找到的数量的操作，我也做了类似的简化。

对 `ProductPrefixTrie` 分析的总结可以查看表5。

4 改进思路

这一节主要涵盖项目可以改进的算法和系统设计设计方向。由于时间因素，许多在系统成型后的想法无法推倒重来，所以总结如下。下面每一个子章节均考虑的是对于具体任务的提升思路。对于系统整体，一个比较重要的提升方向是，当前系统内的数据结构均不是线程安全的，这意味着直接拿到并发场景下可能会出现错误，导致系统内存储的数据偏离其规则。虽然当前系统已经进行了相当充分的类型检查、数值检查等操作，但是这也只能发现问题，而不能解决问题，且由于没有操作记录，不能回滚操作，所以未来可以让系统适配多线程场景，且需要设置操作记录，以便当出现问题时会滚数据。

4.1 营销任务优先调度模块

在考虑营销任务前置依赖的优先调度时，该模块目前仅仅考虑在不存在前置依赖的任务中进行优先度排序，从中选择优先度最高的执行。然而，在实际环境中，可能存在某个前置任务虽然不紧急，但是其后置任务十分紧急的情况，而当前的算法无法捕捉这种后继任务的重要性。后续可以考虑的优化方向为，由于任何一个有向无环图可以通过若干条不相交的路径进行路径覆盖，因此考虑将依赖图拆分成多条路径，对于每一条路径进行整体的优先级排序。这一步需要定义路径优先级，目前的想法是可以定义一个衰减因子 γ[9]，则路径中某个节点 o 对路径优先级的影响力可以计算为

$$IF(o) = \gamma^k \text{priority}(o),$$

其中 k 为路径起点到节点 o 的步数，衰减因子可以由任务的时效性决定，时效性越强的任务，γ 越接近于1。通过对路径优先级进行排序，来决定优先执行哪一条路径的第一个任务。当有任务被完成时，重新计算路径优先级。该想法由于时间原因，无法在项目中实现，故在此陈述。

4.2 客户网络与影响力传播分析模块

对于客户重要性评价任务，本项目只选取了三个指标进行计算。如果要用于真实环境的商业分析，也许可以考虑更多指标，然后对这些指标进行统计分析，综合筛选最为重要的客户。

对于计算客户的影响范围任务，目前采用的方案是定义了一个最小影响力阈值，对路径影响力小余阈值的路径进行剪枝处理。然而在实际环境中，这样的方式存在一个较大的问题：对于不同大小和复杂程度的图，合理的最小影响力阈值是不同的。当前需要指定的阈值在真实场景中可能会严重偏离目标，所以需要自适应调整该阈值的逻辑，或者更换全新的，本身就考虑了图结构的指标来实现面对不同规模和复杂度的图，都能较为准确的反映目标。

最后，当前的算法设计假设了一开始存在一个图，随后对图进行分析，即假设数据是 offline 的，所以每次图更新都需要重新跑一次算法。然而这样的算力开销在数据量较大的时候会比较大。假设真实系统的数据是在线（online）获取的，当前的分析模块可以通过记录当前的遍历结果，结合新增的影响力关系直接计算新的影响网络下的指标，这样虽然会增加内存开销，但是避免了每次更新都完整遍历一遍整个影响力图，是一个潜在的优化方向。

[9]本质是一个惩罚项，越靠后的高优先级任务需要提前完成的代价越大。

表 5: ProductPrefixTrie 方法复杂度分析 (L 为输入的名称或前缀长度)

方法	时间复杂度	空间复杂度
`insert()`	$O(L)$	$O(L)$
`get_product_ids_with_prefix()()`	$O(L)$	$O(L)$
`delete()`	$O(L)$	$O(L)$

4.3 商品数据检索模块

本模块由于其实现难度较大，有需要可以改进的地方。本质上是需要实现一个关系型数据库来对大量数据进行高效的存储。同时，完整的 B+ 树也是一个可以完善的重点，具体需要实现的内容在前文已经讨论过了，所以不再过多赘述。关于前缀匹配，当前的 Trie 树的实现方案虽然可以通过一些方式实现对外存的读取，在外存中往往会消耗大量时间，这里存在许多可以改进的方向。一个比较简单的思路是将 B+ 树和 Trie 树结合起来，将 Trie 树作为 B+ 树的一个值存储起来，这样可以在一次读取中实现对部分 Trie 树的加载，然后利用 Trie 树在内存中的高效性实现高效的前缀匹配。关于在外存中进行前缀匹配，也有许多其他的方案，如 Generalized Suffix Tree（GST）等。

5 总结

本报告详细阐述了为新兴数据驱动商业决策平台设计的底层智能数据管理系统。该系统核心围绕三大模块构建：营销任务优先调度、客户网络与影响力传播分析以及商品数据检索，旨在为上层决策提供坚实的数据支撑。

在设计与实现过程中，针对营销任务优先调度模块，通过结合任务的紧急度与影响力定义优先级，并利用可更新最大堆（UpdatableMaxHeap）和任务依赖有向图（TaskDependencyGraph）实现了高效的任务管理、状态监控（Pending, Ready, Complete）以及考虑前置依赖的任务执行与展示。进一步，报告对该模块所用数据结构及算法的时间和空间复杂度进行了分析。

对于客户网络与影响力传播分析模块，系统基于客户间的加权有向图，采用了加权度中心性（包括 WIDC 和 WODC）与 PageRank 算法来评估客户的重要性。同时，通过深度优先搜索结合路径影响力阈值，实现了对客户影响范围的分析。此模块同样包含了相关数据结构如 CustomerGraph 及算法的复杂度分析。

在商品数据检索模块方面，为了应对大量商品数据并实现高效的外存操作，系统采用了 B+ 树作为主要数据结构来管理商品信息（包括 ID、价格、热度等），支持高效的增删改查及价格范围搜索。同时，利用 Trie 树实现了商品名称的前缀搜索功能。这一部分也对 B+ 树和 Trie 树的操作复杂度进行了分析。

此外，报告还探讨了各模块未来的改进思路。例如，营销任务调度模块可以引入路径优先级的概念，综合考量整个任务链的紧急性；客户影响力分析模块可以探索自适应阈值调整或引入更多维度的客户重要性评价指标，并考虑在线数据更新的算法优化；商品数据检索模块则可以进一步完善 B+ 树的实现，使其更贴近真实外存操作，并探索如 GST 等更优的前缀匹配方案。

综上所述，本项目成功设计并分析了一个多模块的智能数据管理系统，为商业决策平台提供了关键的数据处理与分析能力。通过对各模块功能、实现及复杂度的详细阐述，以及对未来改进方向的思考，本报告为该系统的进一步开发和优化奠定了基础。

参考文献

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking : Bringing order to the web. In *The Web Conference*, 1999. URL https://api.semanticscholar.org/CorpusID:1508503.

A 系统测试 Demo

该项目在 `app.py` 中提供了一些 Demo，用以演示系统的功能，可以在项目根目录下运行

```
python app.py
```

来查看运行结果。运行结果也在此展示。

```
    开始演示数据驱动商业决策平台 - 底层数据处理模块...

========== 客户网络分析演示 ==========
客户图构建完毕。

--- 客户重要性 (模拟 PageRank) ---
  David: 0.2711
  Bob: 0.2672
  Eve: 0.2649
  Charlie: 0.1668
  Alice: 0.0300

--- 客户重要性 (加权出度) ---
  Alice: 1.70
  Bob: 1.60
  David: 0.90
  Charlie: 0.50
  Eve: 0.20

--- 客户影响力传播 (模拟, 无限制) ---
  客户 'Alice' 能影响到 (阈值 0.1): {'Charlie', 'David', 'Eve', 'Bob'}
  客户 'Bob' 能影响到 (阈值 0.1): {'Charlie', 'David', 'Eve'}

========== 商品目录演示 ==========
商品添加完毕。

--- 按ID查找商品 (P1) ---
  Product(id='PROD-20250515211921800467-a87dbb17282e4a09b2dd9336117e6146', name='智能手机X', price=799.99, heat=95.00)

--- 价格在 [100.00, 300.00] 的商品 ---
  Product(id='PROD-20250515211921800616-22672751af4e48dcb6925db7bb206e16', name='无线耳机Air', price=159.00, heat=90.00)
  Product(id='PROD-20250515211921800604-f2d4445c3aee4119a80bd6c057827ce9', name='智能手表S2', price=249.50, heat=88.00)

--- 价格在 [1000.00, 2000.00] 的商品 ---
  Product(id='PROD-20250515211921800585-867a6ec209c044c7bd482d663e0c83a7', name='笔记本电脑Pro', price=1299.00, heat=92.00)

--- 按前缀 '智能' 推荐 (Top 2 热度) ---
  Product(id='PROD-20250515211921800657-927e3040384040469a3fbb5ca2e3c29c', name='智能手机X Plus', price=999.99, heat=98.00)
  Product(id='PROD-20250515211921800467-a87dbb17282e4a09b2dd9336117e6146', name='智能手机X', price=799.99, heat=95.00)

--- 按前缀 '笔记' 推荐 (Top 1 热度) ---
  Product(id='PROD-20250515211921800585-867a6ec209c044c7bd482d663e0c83a7', name='笔记本电脑Pro', price=1299.00, heat=92.00)

--- 更新商品 'PROD-20250515211921800585-867a6ec209c044c7bd482d663e0c83a7' (笔记本电脑Pro) ---

--- 更新后 ---
  Product(id='PROD-20250515211921800585-867a6ec209c044c7bd482d663e0c83a7', name='笔记本电脑Pro', price=1299.00, heat=92.00)

--- 更新后，价格在 [1300.00, 1500.00] 的商品 ---
  (无结果)

--- 删除商品 'PROD-20250515211921800616-22672751af4e48dcb6925db7bb206e16' (无线耳机Air) ---

--- 删除后查找 (应为None或无结果) ---
  None

--- 删除后，按前缀 '无线' 推荐 ---
  (无结果)

========== 营销任务优先级调度演示 ==========
初始任务添加完毕。

--- 初始Top 3就绪任务 ---
  MarketingTask(id='20250515211921800894-073726c7886e415ababc0f13b520f2c5', name='B:社交媒体预热', urgency=8.00, influence=8.00, priority=64.00, status='READY')
  MarketingTask(id='20250515211921800930-430bcfa745734042aa846aca1d5b7f95', name='E:网红直播推广', urgency=6.00, influence=9.00, priority=54.00, status='READY')
  MarketingTask(id='20250515211921800874-2fd4c57464b9456fa0331d860a5e2314', name='A:新品发布会策划', urgency=5.00, influence=10.00, priority=50.00, status='READY')

--- 添加依赖关系 ---
依赖关系添加完毕。

--- 添加依赖后Top 5就绪任务 ---
  MarketingTask(id='20250515211921800874-2fd4c57464b9456fa0331d860a5e2314', name='A:新品发布会策划', urgency=5.00, influence=10.00, priority=50.00, status='READY')
  MarketingTask(id='20250515211921800920-223d9f2123e54809bcb219e24c84bcb2', name='D:渠道合作洽谈', urgency=7.00, influence=6.00, priority=42.00, status='READY')
  MarketingTask(id='20250515211921800909-108dae6f0abf4f5cbf2d66c0eb2bf13b', name='C:广告素材设计', urgency=3.00, influence=10.00, priority=30.00, status='READY')
  MarketingTask(id='20250515211921800939-ba6ffe181876473baa7aad1b15a26d04', name='F:官网信息更新', urgency=4.00, influence=7.00, priority=28.00, status='READY')
```

```
--- 按优先级顺序执行任务 ---

第 1 次执行:
  已执行: A:新品发布会策划 (ID: 20250515211921800874-2fd4c57464b9456fa0331d860a5e2314, Prio: 50.0)

---   当前Top 3就绪任务 ---
  MarketingTask(id='20250515211921800920-223d9f2123e54809bcb219e24c84bcb2', name='D:渠道合作洽谈', urgency=7.00, influence=6.00, priority=42.00, status='READY')
  MarketingTask(id='20250515211921800909-108dae6f0abf4f5cbf2d66c0eb2bf13b', name='C:广告素材设计', urgency=3.00, influence=10.00, priority=30.00, status='READY')
  MarketingTask(id='20250515211921800939-ba6ffe181876473baa7aad1b15a26d04', name='F:官网信息更新', urgency=4.00, influence=7.00, priority=28.00, status='READY')

第 2 次执行:
  已执行: D:渠道合作洽谈 (ID: 20250515211921800920-223d9f2123e54809bcb219e24c84bcb2, Prio: 42.0)

---   当前Top 3就绪任务 ---
  MarketingTask(id='20250515211921800909-108dae6f0abf4f5cbf2d66c0eb2bf13b', name='C:广告素材设计', urgency=3.00, influence=10.00, priority=30.00, status='READY')
  MarketingTask(id='20250515211921800939-ba6ffe181876473baa7aad1b15a26d04', name='F:官网信息更新', urgency=4.00, influence=7.00, priority=28.00, status='READY')

第 3 次执行:
  已执行: C:广告素材设计 (ID: 20250515211921800909-108dae6f0abf4f5cbf2d66c0eb2bf13b, Prio: 30.0)

---   当前Top 3就绪任务 ---
  MarketingTask(id='20250515211921800894-073726c7886e415ababc0f13b520f2c5', name='B:社交媒体预热', urgency=8.00, influence=8.00, priority=64.00, status='READY')
  MarketingTask(id='20250515211921800939-ba6ffe181876473baa7aad1b15a26d04', name='F:官网信息更新', urgency=4.00, influence=7.00, priority=28.00, status='READY')

第 4 次执行:
  已执行: B:社交媒体预热 (ID: 20250515211921800894-073726c7886e415ababc0f13b520f2c5, Prio: 64.0)

---   当前Top 3就绪任务 ---
  MarketingTask(id='20250515211921800930-430bcfa745734042aa846aca1d5b7f95', name='E:网红直播推广', urgency=6.00, influence=9.00, priority=54.00, status='READY')
  MarketingTask(id='20250515211921800939-ba6ffe181876473baa7aad1b15a26d04', name='F:官网信息更新', urgency=4.00, influence=7.00, priority=28.00, status='READY')

第 5 次执行:
  已执行: E:网红直播推广 (ID: 20250515211921800930-430bcfa745734042aa846aca1d5b7f95, Prio: 54.0)

---   当前Top 3就绪任务 ---
  MarketingTask(id='20250515211921800939-ba6ffe181876473baa7aad1b15a26d04', name='F:官网信息更新', urgency=4.00, influence=7.00, priority=28.00, status='READY')

第 6 次执行:
  已执行: F:官网信息更新 (ID: 20250515211921800939-ba6ffe181876473baa7aad1b15a26d04, Prio: 28.0)

---   当前Top 3就绪任务 ---
  (无结果)

--- 所有预期任务执行完毕 ---

--- 所有任务最终状态 ---
  MarketingTask(id='20250515211921800874-2fd4c57464b9456fa0331d860a5e2314', name='A:新品发布会策划', urgency=5.00, influence=10.00, priority=50.00, status='COMPLETED')
  MarketingTask(id='20250515211921800894-073726c7886e415ababc0f13b520f2c5', name='B:社交媒体预热', urgency=8.00, influence=8.00, priority=64.00, status='COMPLETED')
  MarketingTask(id='20250515211921800909-108dae6f0abf4f5cbf2d66c0eb2bf13b', name='C:广告素材设计', urgency=3.00, influence=10.00, priority=30.00, status='COMPLETED')
  MarketingTask(id='20250515211921800920-223d9f2123e54809bcb219e24c84bcb2', name='D:渠道合作洽谈', urgency=7.00, influence=6.00, priority=42.00, status='COMPLETED')
  MarketingTask(id='20250515211921800930-430bcfa745734042aa846aca1d5b7f95', name='E:网红直播推广', urgency=6.00, influence=9.00, priority=54.00, status='COMPLETED')
  MarketingTask(id='20250515211921800939-ba6ffe181876473baa7aad1b15a26d04', name='F:官网信息更新', urgency=4.00, influence=7.00, priority=28.00, status='COMPLETED')

============================
所有演示场景执行完毕。
```

B 单元测试代码

该项目代码通过了本节提供的全部的 150 个单元测试，充分体现了代码的正确性。可以在项目根目录下运行命令

```
python -m unittest
```

来验证单元测试。

B.1 营销任务优先调度模块

这一部分包括营销任务优先调度模块在实现时用到的所有单元测试代码。

B.1.1 test_dependency_graph.py

Listing 1: test_dfs_influence.py

```python
import unittest
from src.data_structure.dependency_graph import TaskDependencyGraph

class TestTaskDependencyGraph(unittest.TestCase):

    def setUp(self):
        """在每个测试方法运行前初始化一个空的依赖图。"""
        self.graph = TaskDependencyGraph()

    # --- 测试 add_task ---
    def test_add_task_new(self):
        """测试添加一个新任务节点。"""
        self.assertTrue(self.graph.add_task("T1"))
        self.assertIn("T1", self.graph.nodes)
        self.assertIn("T1", self.graph.adj)
        self.assertIn("T1", self.graph.rev_adj)
        self.assertEqual(self.graph.adj["T1"], set())
        self.assertEqual(self.graph.rev_adj["T1"], set())
        self.assertEqual(len(self.graph.nodes), 1)

    def test_add_task_existing(self):
        """测试添加一个已存在的任务节点。"""
        self.graph.add_task("T1")
        self.assertFalse(self.graph.add_task("T1")) # 再次添加应返回 False
        self.assertEqual(len(self.graph.nodes), 1) # 节点数量不应改变

    # --- 测试 remove_task ---
    def test_remove_task_non_existent(self):
        """测试移除一个不存在的任务节点。"""
        with self.assertRaisesRegex(IndexError, "节点 T_non_existent 不存在"):
            self.graph.remove_task("T_non_existent")

    def test_remove_task_isolated(self):
        """测试移除一个孤立的任务节点。"""
        self.graph.add_task("T1")
        removed_id = self.graph.remove_task("T1")
        self.assertEqual(removed_id, "T1")
        self.assertNotIn("T1", self.graph.nodes)
        self.assertNotIn("T1", self.graph.adj)
        self.assertNotIn("T1", self.graph.rev_adj)
        self.assertTrue(not self.graph.nodes) # 集合应为空

    def test_remove_task_with_dependencies(self):
        """测试移除有依赖关系的任务节点。"""
        self.graph.add_task("T1")
        self.graph.add_task("T2")
        self.graph.add_task("T3")
        self.graph.add_task("T4")
        self.graph.add_dependency("T1", "T2") # T1 -> T2
        self.graph.add_dependency("T3", "T2") # T3 -> T2
        self.graph.add_dependency("T2", "T4") # T2 -> T4

        # 移除 T2
        removed_id = self.graph.remove_task("T2")
        self.assertEqual(removed_id, "T2")
```

```
        self.assertNotIn("T2", self.graph.nodes)
        self.assertNotIn("T2", self.graph.adj)
        self.assertNotIn("T2", self.graph.rev_adj)

        # 检查其他节点的依赖是否正确更新
        self.assertNotIn("T2", self.graph.adj.get("T1", set()))
        self.assertNotIn("T2", self.graph.adj.get("T3", set()))
        self.assertNotIn("T2", self.graph.rev_adj.get("T4", set()))

        # 确保其他节点和依赖还在
        self.assertTrue(self.graph.has_task("T1"))
        self.assertTrue(self.graph.has_task("T3"))
        self.assertTrue(self.graph.has_task("T4"))
        self.assertEqual(self.graph.adj["T1"], set()) # T1 不再指向 T2
        self.assertEqual(self.graph.rev_adj["T4"], set()) # T4 不再依赖 T2

    # --- 测试 add_dependency ---
    def test_add_dependency_valid(self):
        """测试添加有效的依赖关系。"""
        self.graph.add_task("T1")
        self.graph.add_task("T2")
        self.assertTrue(self.graph.add_dependency("T1", "T2"))
        self.assertIn("T2", self.graph.adj["T1"])
        self.assertIn("T1", self.graph.rev_adj["T2"])

    def test_add_dependency_node_non_existent(self):
        """测试添加依赖时节点不存在。"""
        self.graph.add_task("T1")
        self.assertFalse(self.graph.add_dependency("T1", "T_non")) # T_non 不存在
        self.assertFalse(self.graph.add_dependency("T_non", "T1")) # T_non 不存在
        self.assertEqual(self.graph.adj["T1"], set()) # 确认没有添加成功
        self.assertEqual(self.graph.rev_adj["T1"], set())

    def test_add_dependency_self_loop(self):
        """测试添加自环依赖。"""
        self.graph.add_task("T1")
        self.assertFalse(self.graph.add_dependency("T1", "T1")) # 不应允许自环
        self.assertEqual(self.graph.adj["T1"], set())
        self.assertEqual(self.graph.rev_adj["T1"], set())

    def test_add_dependency_existing(self):
        """测试添加已存在的依赖。"""
        self.graph.add_task("T1")
        self.graph.add_task("T2")
        self.graph.add_dependency("T1", "T2")
        # 再次添加应返回 True，但集合大小不变
        self.assertTrue(self.graph.add_dependency("T1", "T2"))
        self.assertEqual(len(self.graph.adj["T1"]), 1)
        self.assertEqual(len(self.graph.rev_adj["T2"]), 1)

    def test_add_dependency_creates_cycle_simple(self):
        """测试添加依赖导致简单环路。"""
        self.graph.add_task("T1")
        self.graph.add_task("T2")
        self.graph.add_dependency("T1", "T2")
        self.assertFalse(self.graph.add_dependency("T2", "T1")) # 添加 T2->T1 会形成环路
```

```
        # 确认 T2->T1 未被添加
        self.assertNotIn("T1", self.graph.adj["T2"])
        self.assertNotIn("T2", self.graph.rev_adj["T1"])

    def test_add_dependency_creates_cycle_longer(self):
        """测试添加依赖导致长环路。"""
        self.graph.add_task("T1")
        self.graph.add_task("T2")
        self.graph.add_task("T3")
        self.graph.add_task("T4")
        self.graph.add_dependency("T1", "T2")
        self.graph.add_dependency("T2", "T3")
        self.graph.add_dependency("T3", "T4")
        self.assertFalse(self.graph.add_dependency("T4", "T1")) # 添加 T4->T1 会形成 T1->T2->T3->T4->T1 环路
        self.assertNotIn("T1", self.graph.adj["T4"])
        self.assertNotIn("T4", self.graph.rev_adj["T1"])

    def test_add_dependency_no_cycle(self):
        """测试添加不会导致环路的依赖。"""
        self.graph.add_task("T1")
        self.graph.add_task("T2")
        self.graph.add_task("T3")
        self.graph.add_task("T4")
        self.graph.add_dependency("T1", "T2")
        self.graph.add_dependency("T1", "T3")
        self.graph.add_dependency("T2", "T4")
        self.graph.add_dependency("T3", "T4")
        # 添加 T1 -> T4 不会形成环路
        self.assertTrue(self.graph.add_dependency("T1", "T4"))
        self.assertIn("T4", self.graph.adj["T1"])
        self.assertIn("T1", self.graph.rev_adj["T4"])

    # --- 测试 remove_dependency ---
    def test_remove_dependency_valid(self):
        """测试移除存在的依赖关系。"""
        self.graph.add_task("T1")
        self.graph.add_task("T2")
        self.graph.add_dependency("T1", "T2")
        self.assertTrue(self.graph.remove_dependency("T1", "T2"))
        self.assertNotIn("T2", self.graph.adj["T1"])
        self.assertNotIn("T1", self.graph.rev_adj["T2"])

    def test_remove_dependency_non_existent_edge(self):
        """测试移除不存在的依赖关系（节点存在）。"""
        self.graph.add_task("T1")
        self.graph.add_task("T2")
        self.assertFalse(self.graph.remove_dependency("T1", "T2")) # T1->T2 不存在

    def test_remove_dependency_non_existent_node(self):
        """测试移除依赖时节点不存在。"""
        self.graph.add_task("T1")
        self.assertFalse(self.graph.remove_dependency("T1", "T_non"))
        self.assertFalse(self.graph.remove_dependency("T_non", "T1"))

    # --- 测试查询方法 ---
```

```python
    def test_get_dependents_and_prerequisites(self):
        """测试 get_dependents 和 get_prerequisites 方法。"""
        self.graph.add_task("T1")
        self.graph.add_task("T2")
        self.graph.add_task("T3")
        self.graph.add_dependency("T1", "T2")
        self.graph.add_dependency("T1", "T3")
        self.graph.add_dependency("T2", "T3")

        self.assertEqual(self.graph.get_dependents("T1"), {"T2", "T3"})
        self.assertEqual(self.graph.get_dependents("T2"), {"T3"})
        self.assertEqual(self.graph.get_dependents("T3"), set())

        self.assertEqual(self.graph.get_prerequisites("T1"), set())
        self.assertEqual(self.graph.get_prerequisites("T2"), {"T1"})
        self.assertEqual(self.graph.get_prerequisites("T3"), {"T1", "T2"})

        # 测试不存在的节点
        self.assertEqual(self.graph.get_dependents("T_non"), set())
        self.assertEqual(self.graph.get_prerequisites("T_non"), set())

    def test_get_methods_return_copy(self):
        """测试 get_dependents/prerequisites 返回的是副本。"""
        self.graph.add_task("T1")
        self.graph.add_task("T2")
        self.graph.add_dependency("T1", "T2")

        dependents = self.graph.get_dependents("T1")
        dependents.add("T_fake")
        self.assertEqual(self.graph.get_dependents("T1"), {"T2"}) # 原图不应被修改

        prereqs = self.graph.get_prerequisites("T2")
        prereqs.add("T_fake")
        self.assertEqual(self.graph.get_prerequisites("T2"), {"T1"}) # 原图不应被修改


    def test_has_task(self):
        """测试 has_task 方法。"""
        self.assertFalse(self.graph.has_task("T1"))
        self.graph.add_task("T1")
        self.assertTrue(self.graph.has_task("T1"))
        self.graph.remove_task("T1")
        self.assertFalse(self.graph.has_task("T1"))

    def test_get_all_tasks(self):
        """测试 get_all_tasks 方法。"""
        self.assertEqual(self.graph.get_all_tasks(), [])
        self.graph.add_task("T1")
        self.graph.add_task("T3")
        self.graph.add_task("T2")
        # 返回列表，顺序不保证，所以比较集合
        self.assertEqual(set(self.graph.get_all_tasks()), {"T1", "T2", "T3"})
        self.assertEqual(len(self.graph.get_all_tasks()), 3)
        self.graph.remove_task("T2")
        self.assertEqual(set(self.graph.get_all_tasks()), {"T1", "T3"})

```

```
    # --- 测试 _has_path (间接通过 add_dependency, 但也可以补充直接测试) ---
    def test_has_path_direct(self):
        """直接测试 _has_path: 直接路径。"""
        self.graph.add_task("A"); self.graph.add_task("B")
        self.graph.add_dependency("A", "B")
        self.assertTrue(self.graph._has_path("A", "B"))
        self.assertFalse(self.graph._has_path("B", "A"))

    def test_has_path_indirect(self):
        """直接测试 _has_path: 间接路径。"""
        self.graph.add_task("A"); self.graph.add_task("B"); self.graph.add_task("C")
        self.graph.add_dependency("A", "B")
        self.graph.add_dependency("B", "C")
        self.assertTrue(self.graph._has_path("A", "C"))
        self.assertFalse(self.graph._has_path("C", "A"))

    def test_has_path_no_path(self):
        """直接测试 _has_path: 不存在路径。"""
        self.graph.add_task("A"); self.graph.add_task("B"); self.graph.add_task("C")
        self.graph.add_dependency("A", "B")
        self.assertFalse(self.graph._has_path("A", "C"))
        self.assertFalse(self.graph._has_path("C", "A"))

    def test_has_path_with_cycle(self):
        """直接测试 _has_path: 在有环图中查找路径。"""
        # Setup graph nodes
        self.graph.add_task("A"); self.graph.add_task("B"); self.graph.add_task("C")
        self.graph.add_task("D")

        # Manually create the cyclic structure for testing _has_path
        # Bypassing add_dependency's cycle check for this specific test case setup
        self.graph.adj["A"].add("B")
        self.graph.rev_adj["B"].add("A")

        self.graph.adj["B"].add("C")
        self.graph.rev_adj["C"].add("B")

        self.graph.adj["C"].add("A") # Manually add the edge that forms the cycle
        self.graph.rev_adj["A"].add("C")

        self.graph.adj["C"].add("D") # Add the exit edge
        self.graph.rev_adj["D"].add("C")

        # Now the graph truly contains the cycle A->B->C->A and edge C->D

        # --- Assertions on the cyclic graph ---

        # Can we reach D from A? (A -> B -> C -> D) - YES
        self.assertTrue(self.graph._has_path("A", "D"), "应能找到路径 A->D")

        # Can we reach A from A (via cycle)?
        self.assertTrue(self.graph._has_path("A", "A"), "_has_path(node, node) 应返回 True")

        # Can we reach A from B? (B -> C -> A) - YES
        self.assertTrue(self.graph._has_path("B", "A"), "应能找到路径 B->A") # 现在这个断言应
    该通过了
```

```
278
279         # Can we reach A from D? (D has no outgoing edges) - NO
280         self.assertFalse(self.graph._has_path("D", "A"), "不应找到路径 D->A")
281
282
283 if __name__ == '__main__':
284     unittest.main()
```

B.1.2 test_updatable_max_heap.py

Listing 2: test_dfs_influence.py

```
1  import unittest
2  from src.data_structure.updatable_max_heap import UpdatableMaxHeap
3
4  class TestUpdatableMaxHeap(unittest.TestCase):
5
6      def setUp(self):
7          """在每个测试方法运行前初始化一个空的堆。"""
8          self.heap = UpdatableMaxHeap()
9
10     def test_initialization(self):
11         """测试堆的初始状态。"""
12         self.assertTrue(self.heap.is_empty())
13         self.assertEqual(self.heap.get_heap_size(), 0)
14         self.assertIsNone(self.heap.peek_max())
15         self.assertIsNone(self.heap.extract_max())
16         self.assertEqual(self.heap._heap, [])
17         self.assertEqual(self.heap._position_map, {})
18
19     def test_insert_single_element(self):
20         """测试插入单个元素。"""
21         self.heap.insert("task1", 50)
22         self.assertFalse(self.heap.is_empty())
23         self.assertEqual(self.heap.get_heap_size(), 1)
24         self.assertIn("task1", self.heap)
25         self.assertIn("task1", self.heap._position_map)
26         self.assertEqual(self.heap._position_map["task1"], 0)
27         self.assertEqual(self.heap._heap[0], (-50.0, "task1")) # 内部存储负优先级
28         priority, task_id = self.heap.peek_max()
29         self.assertEqual(task_id, "task1")
30         self.assertAlmostEqual(priority, 50.0)
31
32     def test_insert_multiple_elements_order(self):
33         """测试插入多个元素后，extract_max 是否按优先级降序返回。"""
34         tasks = {"task1": 50, "task2": 30, "task3": 70, "task4": 40, "task5": 70}
35         for task_id, priority in tasks.items():
36             self.heap.insert(task_id, priority)
37
38         self.assertEqual(self.heap.get_heap_size(), 5)
39
40         extracted_order = []
41         while not self.heap.is_empty():
42             priority, task_id = self.heap.extract_max()
43             extracted_order.append((priority, task_id))
```

```
        # 预期顺序：优先级高的在前，优先级相同的不保证顺序
        self.assertEqual(extracted_order[0][0], 70) # task3 或 task5
        self.assertEqual(extracted_order[1][0], 70) # task3 或 task5
        self.assertEqual(extracted_order[2][0], 50) # task1
        self.assertEqual(extracted_order[3][0], 40) # task4
        self.assertEqual(extracted_order[4][0], 30) # task2

        # 确保 ID 都在里面
        extracted_ids = {item[1] for item in extracted_order}
        self.assertEqual(extracted_ids, set(tasks.keys()))

        # 提取后应为空
        self.assertTrue(self.heap.is_empty())
        self.assertEqual(self.heap._position_map, {})

    def test_insert_duplicate_task_id_updates_priority(self):
        """测试插入已存在的 task_id 时，行为应等同于更新优先级。"""
        self.heap.insert("taskA", 50)
        self.heap.insert("taskB", 70)

        # 插入 taskA，但使用新优先级
        self.heap.insert("taskA", 80)

        self.assertEqual(self.heap.get_heap_size(), 2) # 大小应不变
        priority, task_id = self.heap.peek_max()
        self.assertEqual(task_id, "taskA") # taskA 现在优先级最高
        self.assertAlmostEqual(priority, 80.0)

        # 再次插入 taskA，优先级降低
        self.heap.insert("taskA", 60)
        self.assertEqual(self.heap.get_heap_size(), 2)
        priority, task_id = self.heap.peek_max()
        self.assertEqual(task_id, "taskB") # taskB 现在优先级最高
        self.assertAlmostEqual(priority, 70.0)

    def test_peek_max_on_empty_heap(self):
        """测试在空堆上调用 peek_max。"""
        self.assertIsNone(self.heap.peek_max())

    def test_peek_max_does_not_modify(self):
        """测试 peek_max 不会修改堆。"""
        self.heap.insert("task1", 50)
        self.heap.insert("task2", 70)
        initial_heap = list(self.heap._heap)
        initial_map = dict(self.heap._position_map)
        initial_size = self.heap.get_heap_size()

        priority, task_id = self.heap.peek_max()
        self.assertEqual(task_id, "task2")
        self.assertAlmostEqual(priority, 70.0)

        # 验证堆未被修改
        self.assertEqual(self.heap._heap, initial_heap)
        self.assertEqual(self.heap._position_map, initial_map)
        self.assertEqual(self.heap.get_heap_size(), initial_size)
```

```
100
101     def test_extract_max_on_empty_heap(self):
102         """测试在空堆上调用 extract_max。"""
103         self.assertIsNone(self.heap.extract_max())
104
105     def test_extract_max_single_element(self):
106         """测试从只有一个元素的堆中提取。"""
107         self.heap.insert("task1", 50)
108         priority, task_id = self.heap.extract_max()
109         self.assertEqual(task_id, "task1")
110         self.assertAlmostEqual(priority, 50.0)
111         self.assertTrue(self.heap.is_empty())
112         self.assertEqual(self.heap._position_map, {})
113
114     def test_contains(self):
115         """测试 __contains__ (__in__) 的行为。"""
116         self.assertFalse("task1" in self.heap)
117         self.heap.insert("task1", 50)
118         self.assertTrue("task1" in self.heap)
119         self.assertFalse("task2" in self.heap)
120         self.heap.insert("task2", 70)
121         self.assertTrue("task2" in self.heap)
122         self.heap.extract_max() # 提取 task2
123         self.assertTrue("task1" in self.heap)
124         self.assertFalse("task2" in self.heap)
125         self.heap.delete("task1")
126         self.assertFalse("task1" in self.heap)
127
128     def test_update_priority_non_existent(self):
129         """测试更新不存在的任务时抛出 KeyError。"""
130         with self.assertRaisesRegex(KeyError, "任务 'task_non_existent' 不在堆中"):
131             self.heap.update_priority("task_non_existent", 100)
132
133     def test_update_priority_increase(self):
134         """测试增加任务优先级（可能触发上浮）。"""
135         self.heap.insert("task1", 50)
136         self.heap.insert("task2", 30)
137         self.heap.insert("task3", 70) # Max
138         self.heap.insert("task4", 40)
139
140         self.heap.update_priority("task4", 90) # task4 优先级变为最高
141         self.assertEqual(self.heap.get_heap_size(), 4)
142         priority, task_id = self.heap.peek_max()
143         self.assertEqual(task_id, "task4")
144         self.assertAlmostEqual(priority, 90.0)
145         # 可以进一步检查内部堆结构或通过extract_max验证顺序
146
147     def test_update_priority_decrease(self):
148         """测试降低任务优先级（可能触发下沉）。"""
149         self.heap.insert("task1", 50)
150         self.heap.insert("task2", 80) # Max
151         self.heap.insert("task3", 70)
152         self.heap.insert("task4", 40)
153
154         self.heap.update_priority("task2", 20) # task2 优先级降到最低
155         self.assertEqual(self.heap.get_heap_size(), 4)
```

```
        priority, task_id = self.heap.peek_max()
        self.assertEqual(task_id, "task3") # task3 现在应为最高
        self.assertAlmostEqual(priority, 70.0)
        # 可以进一步检查内部堆结构或通过extract_max验证顺序

    def test_update_priority_no_change(self):
        """测试更新为相同优先级（堆结构应不变）。"""
        self.heap.insert("task1", 50)
        self.heap.insert("task2", 70)
        initial_heap = list(self.heap._heap)
        initial_map = dict(self.heap._position_map)

        self.heap.update_priority("task1", 50)

        self.assertEqual(self.heap._heap, initial_heap)
        self.assertEqual(self.heap._position_map, initial_map)

    def test_delete_non_existent(self):
        """测试删除不存在的任务时抛出 KeyError。"""
        with self.assertRaisesRegex(KeyError, "任务 'task_non_existent' 不在堆中"):
            self.heap.delete("task_non_existent")

    def test_delete_root(self):
        """测试删除堆顶元素。"""
        self.heap.insert("task1", 50)
        self.heap.insert("task2", 30)
        self.heap.insert("task3", 70) # Max
        self.heap.insert("task4", 40)

        self.heap.delete("task3") # 删除当前最大值
        self.assertEqual(self.heap.get_heap_size(), 3)
        self.assertNotIn("task3", self.heap)
        priority, task_id = self.heap.peek_max()
        self.assertEqual(task_id, "task1") # task1 应该成为新的最大值
        self.assertAlmostEqual(priority, 50.0)
        # 验证堆性质是否保持 (可以通过多次extract_max检查顺序)
        items = []
        while not self.heap.is_empty(): items.append(self.heap.extract_max())
        self.assertEqual([p for p,i in items], [50.0, 40.0, 30.0])


    def test_delete_leaf(self):
        """测试删除一个叶子节点元素。"""
        # 构建一个堆，使某个元素确定在叶子层
        # 例如: insert 70, 50, 30, 40, 20
        self.heap.insert("t70", 70)
        self.heap.insert("t50", 50)
        self.heap.insert("t30", 30)
        self.heap.insert("t40", 40)
        self.heap.insert("t20", 20) # t20 可能是叶子节点

        # 假设我们知道 't20' 或 't30' 或 't40' 在叶子层（取决于具体插入顺序和堆实现）
        # 为了确定性，我们先检查一个叶子节点
        # 内部堆可能是 [(-70,t70), (-50,t50), (-30,t30), (-40,t40), (-20,t20)]
        # 或者           [(-70,t70), (-50,t50), (-30,t30), (-40,t40), (-20,t20)] 或其他变种
        # 索引:            0         1         2         3         4
```

```
        # 最后一个元素 t20 在索引 4，其父为 (4-1)//2 = 1。它肯定是叶子。
        # 元素 t40 在索引 3，其父为 (3-1)//2 = 1。它也肯定是叶子。
        # 元素 t30 在索引 2，其父为 (2-1)//2 = 0。它也肯定是叶子。

        leaf_to_delete = "t30" # 尝试删除叶子 t30

        self.assertIn(leaf_to_delete, self.heap)
        self.heap.delete(leaf_to_delete)

        self.assertEqual(self.heap.get_heap_size(), 4)
        self.assertNotIn(leaf_to_delete, self.heap)

        # 验证堆性质
        items = []
        while not self.heap.is_empty(): items.append(self.heap.extract_max())
        self.assertEqual([p for p,i in items], [70.0, 50.0, 40.0, 20.0])

    def test_delete_internal_node(self):
        """测试删除一个内部节点元素。"""
        self.heap.insert("t70", 70)
        self.heap.insert("t50", 50) # 将成为内部节点
        self.heap.insert("t30", 30)
        self.heap.insert("t40", 40)
        self.heap.insert("t20", 20)
        self.heap.insert("t60", 60) # t50 或 t60 可能是内部节点

        # 假设 't50' 是内部节点 (index 1)
        internal_to_delete = "t50"

        self.assertIn(internal_to_delete, self.heap)
        self.heap.delete(internal_to_delete)

        self.assertEqual(self.heap.get_heap_size(), 5)
        self.assertNotIn(internal_to_delete, self.heap)

        # 验证堆性质
        items = []
        while not self.heap.is_empty(): items.append(self.heap.extract_max())
        # 预期顺序: 70, 60, 40, 30, 20
        self.assertEqual([p for p,i in items], [70.0, 60.0, 40.0, 30.0, 20.0])

    def test_delete_only_element(self):
        """测试删除堆中唯一的元素。"""
        self.heap.insert("task1", 100)
        self.heap.delete("task1")
        self.assertTrue(self.heap.is_empty())
        self.assertEqual(self.heap._position_map, {})

    def test_delete_all_elements(self):
        """测试通过 delete 删除所有元素。"""
        tasks = {"task1": 50, "task2": 30, "task3": 70}
        for task_id, priority in tasks.items():
            self.heap.insert(task_id, priority)

        self.heap.delete("task1")
        self.heap.delete("task3")
```

```
        self.heap.delete("task2")

        self.assertTrue(self.heap.is_empty())
        self.assertEqual(self.heap._position_map, {})

    def test_mixed_operations(self):
        """测试混合插入、更新、删除、提取操作。"""
        self.heap.insert("A", 10)
        self.heap.insert("B", 20)
        self.assertTrue("A" in self.heap)
        self.assertTrue("B" in self.heap)
        self.assertEqual(self.heap.peek_max(), (20.0, "B"))

        self.heap.insert("C", 5)
        self.heap.update_priority("A", 30) # A 变为最大
        self.assertEqual(self.heap.peek_max(), (30.0, "A"))

        self.heap.delete("B") # 删除原来的最大值 B
        self.assertNotIn("B", self.heap)
        self.assertEqual(self.heap.get_heap_size(), 2)
        self.assertEqual(self.heap.peek_max(), (30.0, "A")) # A 仍然最大

        extracted_priority, extracted_id = self.heap.extract_max() # 提取 A
        self.assertEqual(extracted_id, "A")
        self.assertAlmostEqual(extracted_priority, 30.0)

        self.assertEqual(self.heap.peek_max(), (5.0, "C")) # 只剩 C

        self.heap.insert("D", 15)
        self.assertEqual(self.heap.peek_max(), (15.0, "D"))

        self.heap.update_priority("C", 25) # C 现在最大
        self.assertEqual(self.heap.peek_max(), (25.0, "C"))

        # 清空
        self.heap.extract_max()
        self.heap.extract_max()
        self.assertTrue(self.heap.is_empty())

if __name__ == '__main__':
    unittest.main()
```

B.1.3 test_marketing.py

Listing 3: test_dfs_influence.py

```
import unittest

from src.model.marketing_task import *
from src.module.marketing_task_schedule import *

class TestMarketingTaskManager(unittest.TestCase):

    def setUp(self):
        """在每个测试方法运行前初始化一个空的 TaskManager。"""
```

```
        self.tm = TaskManager()
        # 为了可预测的ID，我们可以mock _generate_task_id，或者接受其随机性
        # 这里我们暂时接受其随机性，测试主要关注逻辑
        self.id_counter = 1 # 用于辅助生成可预测的ID（如果需要）

    def _add_sample_task(self, urgency, influence, name=None, expect_success=True) -> str | None:
        """辅助方法添加任务，并处理可能的None返回值"""
        # 如果要使用可预测ID进行测试：
        # task_id_to_use = f"T{self.id_counter}"
        # self.tm._tasks[task_id_to_use] = MarketingTask(...) # 手动插入以控制ID
        # self.tm._task_graph.add_task(task_id_to_use)
        # self.tm._in_degree[task_id_to_use] = 0
        # ... 手动加入ready_queue ...
        # self.id_counter += 1
        # return task_id_to_use
        # 但这会绕过 add_task 的逻辑，所以我们还是调用 add_task
        task_id = self.tm.add_task(urgency, influence, name)
        if expect_success:
            self.assertIsNotNone(task_id, f"添加任务 (u:{urgency}, i:{influence}, n:{name}) 时期望成功，但返回 None")
            if task_id: # 确保 task_id 不是 None 才进行后续断言
                self.assertIn(task_id, self.tm._tasks)
        else:
            self.assertIsNone(task_id, f"添加任务 (u:{urgency}, i:{influence}, n:{name}) 时期望失败，但返回 task_id")
        return task_id

    # --- 测试 MarketingTask 本身 (确保与 TaskManager 交互的基础是正确的) ---
    def test_marketing_task_creation_and_priority(self):
        task = MarketingTask("id1", 10, 5, "Task One")
        self.assertEqual(task.task_id, "id1")
        self.assertEqual(task.name, "Task One")
        self.assertEqual(task.urgency, 10)
        self.assertEqual(task.influence, 5)
        self.assertEqual(task.priority, 50) # 10 * 5
        self.assertEqual(task.status, TASK_STATUS_PENDING)

    def test_marketing_task_update_details(self):
        task = MarketingTask("id1", 10, 5, "Task One")
        task.urgency = 20 # 通过setter更新，应重新计算优先级
        self.assertEqual(task.priority, 100) # 20 * 5
        task.influence = 2
        self.assertEqual(task.priority, 40) # 20 * 2

        # 测试 update_details 方法 (你的实现中这个方法似乎和直接用setter效果重叠了)
        # 如果 MarketingTask.priority 是直接通过 _recalculate_priority 在setter中更新的，
        # 那么 MarketingTask.update_details 中的 self.priority = self._calculate_priority() 可能是多余的
        # 但如果 update_details 是主要的更新接口，则测试它：
        task._update_details(urgency=5, influence=5)
        self.assertEqual(task.priority, 25)


    # --- 测试 TaskManager ---
    def test_add_task(self):
```

```
        """测试添加新任务。"""
        tid1 = self._add_sample_task(10, 5, "Task Alpha")
        self.assertIn(tid1, self.tm._ready_queue) # 新任务应在就绪队列
        self.assertEqual(self.tm._tasks[tid1].status, TASK_STATUS_READY)
        self.assertEqual(self.tm._in_degree[tid1], 0)

        tid2 = self._add_sample_task(20, 2, "Task Beta")
        self.assertIn(tid2, self.tm._ready_queue)
        self.assertEqual(self.tm._ready_queue.get_heap_size(), 2)

        # 测试添加无效参数的任务 (MarketingTask的__init__会校验)
        invalid_tid = self.tm.add_task(0, 5, "Invalid Urgency") # urgency <= 0 会失败
        self.assertIsNone(invalid_tid)
        invalid_tid_2 = self.tm.add_task(5, -1, "Invalid Influence") # influence <=0 会失败
        self.assertIsNone(invalid_tid_2)
        self.assertEqual(self.tm._ready_queue.get_heap_size(), 2) # 确保无效添加未影响队列

    def test_add_dependency_simple_and_status_change(self):
        """测试添加依赖，以及后继任务状态和就绪队列的变化。"""
        t1 = self._add_sample_task(10, 1, "T1") # P=10
        t2 = self._add_sample_task(5, 1, "T2")  # P=5

        # T1 和 T2 初始都应该是 READY 且在队列中
        self.assertEqual(self.tm._tasks[t1].status, TASK_STATUS_READY)
        self.assertEqual(self.tm._tasks[t2].status, TASK_STATUS_READY)
        self.assertTrue(t1 in self.tm._ready_queue)
        self.assertTrue(t2 in self.tm._ready_queue)
        self.assertEqual(self.tm._in_degree[t2], 0)

        # 添加依赖 T1 -> T2 (T1 未完成)
        self.assertTrue(self.tm.add_dependency(t1, t2))
        self.assertEqual(self.tm._in_degree[t2], 1) # T2 入度增加
        self.assertEqual(self.tm._tasks[t2].status, TASK_STATUS_PENDING) # T2 状态变为 PENDING
        self.assertFalse(t2 in self.tm._ready_queue) # T2 从就绪队列移除
        self.assertTrue(t1 in self.tm._ready_queue) # T1 仍在队列中

    def test_add_dependency_prerequisite_already_completed(self):
        """测试添加依赖时，如果前置任务已完成，不影响后继任务的就绪状态。"""
        t1 = self._add_sample_task(10, 1, "T1")
        t2 = self._add_sample_task(5, 1, "T2")

        # 先完成 T1
        self.tm._tasks[t1]._set_status(TASK_STATUS_COMPLETED)
        # 模拟外部完成，注意这不会自动更新T1的后继，因为还没建立依赖
        # 如果要严格，应该通过 TaskManager 的完成方法来做，但这里是为了测试 add_dependency

        # 添加依赖 T1 -> T2 (T1 已完成)
        self.assertTrue(self.tm.add_dependency(t1, t2))
        self.assertEqual(self.tm._in_degree[t2], 0) # T2 入度不应增加，因为T1已完成
        self.assertEqual(self.tm._tasks[t2].status, TASK_STATUS_READY) # T2 状态应保持 READY
        self.assertTrue(t2 in self.tm._ready_queue) # T2 应仍在就绪队列

    def test_add_dependency_creates_cycle(self):
        """测试添加依赖时形成环路。"""
        t1 = self._add_sample_task(1,1, "T1")
        t2 = self._add_sample_task(1,1, "T2")
```

```
        self.assertTrue(self.tm.add_dependency(t1, t2))
        self.assertFalse(self.tm.add_dependency(t2, t1)) # 应形成环路并失败

    def test_add_duplicate_active_dependency(self):
        """测试重复添加一个已存在的、前置未完成的依赖。"""
        t1 = self._add_sample_task(1,1, "T1")
        t2 = self._add_sample_task(1,1, "T2")
        self.assertTrue(self.tm.add_dependency(t1, t2)) # 第一次添加
        self.assertEqual(self.tm._in_degree[t2], 1)
        self.assertEqual(self.tm._tasks[t2].status, TASK_STATUS_PENDING)
        self.assertFalse(t2 in self.tm._ready_queue)

        # 重复添加同一个活跃依赖
        self.assertTrue(self.tm.add_dependency(t1, t2))
        # in_degree 不应再次增加，状态不应改变
        self.assertEqual(self.tm._in_degree[t2], 1, "重复添加活跃依赖不应增加in_degree")
        self.assertEqual(self.tm._tasks[t2].status, TASK_STATUS_PENDING)
        self.assertFalse(t2 in self.tm._ready_queue)

    def test_remove_dependency_makes_task_ready(self):
        """测试移除依赖后，任务变为就绪状态。"""
        t1 = self._add_sample_task(1,1,"T1")
        t2 = self._add_sample_task(1,1,"T2")
        t3 = self._add_sample_task(1,1,"T3")

        self.tm.add_dependency(t1, t3) # T3 依赖 T1
        self.tm.add_dependency(t2, t3) # T3 依赖 T2

        self.assertEqual(self.tm._in_degree[t3], 2)
        self.assertEqual(self.tm._tasks[t3].status, TASK_STATUS_PENDING)
        self.assertFalse(t3 in self.tm._ready_queue)

        # 移除 T1 -> T3，但 T2 -> T3 仍然存在 (假设 T1, T2 都未完成)
        self.assertTrue(self.tm.remove_dependency(t1, t3))
        self.assertEqual(self.tm._in_degree[t3], 1)
        self.assertEqual(self.tm._tasks[t3].status, TASK_STATUS_PENDING)
        self.assertFalse(t3 in self.tm._ready_queue)

        # 移除 T2 -> T3，现在 T3 应该就绪 (假设 T2 未完成)
        self.assertTrue(self.tm.remove_dependency(t2, t3))
        self.assertEqual(self.tm._in_degree[t3], 0)
        self.assertEqual(self.tm._tasks[t3].status, TASK_STATUS_READY)
        self.assertTrue(t3 in self.tm._ready_queue)

    def test_mark_task_as_completed(self):
        """测试标记任务完成及其对后续任务的影响。"""
        t1 = self._add_sample_task(10,1,"T1") # P=10
        t2 = self._add_sample_task(5,1,"T2")  # P=5
        t3 = self._add_sample_task(20,1,"T3") # P=20

        self.tm.add_dependency(t1, t2) # T1 -> T2
        self.tm.add_dependency(t1, t3) # T1 -> T3

        # 初始状态: T1 READY, T2 PENDING (in=1), T3 PENDING (in=1)
        self.assertEqual(self.tm._tasks[t1].status, TASK_STATUS_READY)
        self.assertEqual(self.tm._tasks[t2].status, TASK_STATUS_PENDING)
```

```
        self.assertEqual(self.tm._tasks[t3].status, TASK_STATUS_PENDING)
        self.assertTrue(t1 in self.tm._ready_queue)
        self.assertFalse(t2 in self.tm._ready_queue)
        self.assertFalse(t3 in self.tm._ready_queue)

        # 准备标记 T1 完成。为了模拟真实场景，T1 应该是从 ready_queue 中取出的。
        # 但这里我们直接测试 mark_task_as_completed。
        # 我们需要确保 T1 的状态是 READY，这在 _add_sample_task 中已保证（如果无依赖）。
        self.assertTrue(self.tm.mark_task_as_completed(t1))

        # 验证 T1 的状态
        self.assertEqual(self.tm._tasks[t1].status, TASK_STATUS_COMPLETED)

        # **关键点：mark_task_as_completed 不负责将 t1 从 ready_queue 中移除。**
        # **execute_next_highest_priority_task 才负责移除。**
        # 所以，如果 ready_queue 的 delete 不是由 mark_task_as_completed 触发，
        # 那么 t1 可能仍然在队列中，或者它的存在与否取决于 extract_max 的行为。
        # 但如果一个任务 COMPLETED 了，它理论上不应该再被视为“就绪”去执行。
        # 这是一个需要厘清的设计点：mark_task_as_completed 是否应该假设任务已被从队列取出？
        # 当前 TaskManager.mark_task_as_completed 检查 task_to_complete.status != TASK_STATUS_READY 时会返回False。
        # 所以，调用 mark_task_as_completed 前，任务必须是 READY。
        # 该方法本身不从 ready_queue 移除。

        # 让我们检查 _ready_queue 的行为。
        # 如果我们期望 mark_task_as_completed 不改变 t1 在队列中的存在性（只是改变状态）：
        # self.assertTrue(t1 in self.tm._ready_queue) # 这会是 True
        # 但这可能不是我们想要的，一个COMPLETED的任务不应该在READY队列。

        # 更好的做法是，测试 "execute_next_highest_priority_task" 时，
        # 再验证任务是否从队列中移除。
        # 对于 mark_task_as_completed，我们主要关注其对 *后继任务* 的影响。

        # T2 和 T3 应该变为 READY 并加入队列
        self.assertEqual(self.tm._in_degree[t2], 0)
        self.assertEqual(self.tm._tasks[t2].status, TASK_STATUS_READY)
        self.assertTrue(t2 in self.tm._ready_queue)

        self.assertEqual(self.tm._in_degree[t3], 0)
        self.assertEqual(self.tm._tasks[t3].status, TASK_STATUS_READY)
        self.assertTrue(t3 in self.tm._ready_queue)

        # 尝试标记一个 PENDING 任务为完成，应该失败
        t4 = self._add_sample_task(1,1,"T4")
        self.tm.add_dependency(t2, t4) # T2 -> T4, T2 此时是 READY
                                       # 所以 add_dependency(t2,t4) 时，由于 t2 未完成，t4 入度为1，状态 PENDING
        self.assertFalse(self.tm.mark_task_as_completed(t4), "不应能标记 PENDING 任务为完成")

        # 尝试标记一个已经是 COMPLETED 的任务 (t1)
        self.assertTrue(self.tm.mark_task_as_completed(t1), "标记已完成的任务应返回 True (幂等)")


    def test_execute_next_highest_priority_task(self):
        """测试执行最高优先级任务。"""
```

```
        t1 = self._add_sample_task(10, 5, "T1_High") # P=50
        t2 = self._add_sample_task(20, 1, "T2_Mid")  # P=20
        t3 = self._add_sample_task(5, 1, "T3_Low")   # P=5

        executed_task = self.tm.execute_next_highest_priority_task()
        self.assertIsNotNone(executed_task)
        self.assertEqual(executed_task.task_id, t1)
        self.assertEqual(self.tm._tasks[t1].status, TASK_STATUS_COMPLETED)
        self.assertFalse(t1 in self.tm._ready_queue)

        executed_task = self.tm.execute_next_highest_priority_task()
        self.assertEqual(executed_task.task_id, t2)

        executed_task = self.tm.execute_next_highest_priority_task()
        self.assertEqual(executed_task.task_id, t3)

        self.assertIsNone(self.tm.execute_next_highest_priority_task(), "队列为空时应返回None"
    )

    def test_get_top_k_ready_tasks(self):
        """测试查看最高优先级的 k 个任务。"""
        t1 = self._add_sample_task(10, 5, "T1_50") # P=50
        t2 = self._add_sample_task(20, 4, "T2_80") # P=80
        t3 = self._add_sample_task(5, 10, "T3_50") # P=50
        t4 = self._add_sample_task(60, 1, "T4_60") # P=60
        t5 = self._add_sample_task(1, 1, "T5_1")   # P=1

        # 当前就绪队列中应有5个任务

        # 查看 top 3
        top_3 = self.tm.get_top_k_ready_tasks(3)
        self.assertEqual(len(top_3), 3)
        # 优先级顺序应为 T2(80), T4(60), T1(50) 或 T3(50)
        self.assertEqual(top_3[0].task_id, t2)
        self.assertEqual(top_3[1].task_id, t4)
        self.assertIn(top_3[2].task_id, {t1, t3}) # 第三个可能是 T1 或 T3

        # 确保原队列未被永久改变
        self.assertEqual(self.tm._ready_queue.get_heap_size(), 5)
        priority, task_id = self.tm._ready_queue.peek_max() # 最高优先级应仍是 T2
        self.assertEqual(task_id, t2)
        self.assertAlmostEqual(priority, 80.0)

        # 查看 top 10 (队列中只有5个)
        top_10 = self.tm.get_top_k_ready_tasks(10)
        self.assertEqual(len(top_10), 5)
        self.assertEqual(top_10[0].task_id, t2) # 顺序应该保持

        # 测试 k <= 0
        with self.assertRaisesRegex(ValueError, "k必须是正整数"):
            self.tm.get_top_k_ready_tasks(0)
        with self.assertRaisesRegex(ValueError, "k必须是正整数"):
            self.tm.get_top_k_ready_tasks(-1)


    def test_update_task_info_priority_and_queue(self):
```

```
        """测试更新任务信息导致优先级变化和在就绪队列中的调整。"""
        t1 = self._add_sample_task(10, 1, "T1") # P=10
        t2 = self._add_sample_task(50, 1, "T2") # P=50, Max

        self.assertEqual(self.tm._ready_queue.peek_max()[1], t2)

        # 提高 T1 的优先级，使其超过 T2
        self.assertTrue(self.tm.update_task_info(t1, new_urgency=60)) # New P(T1) = 60 * 1 = 60
        self.assertEqual(self.tm._tasks[t1].priority, 60.0)
        self.assertEqual(self.tm._ready_queue.peek_max()[1], t1) # T1 现在是最高

        # 降低 T1 的优先级，使其低于 T2
        self.assertTrue(self.tm.update_task_info(t1, new_urgency=4)) # New P(T1) = 4 * 1 = 4
        self.assertEqual(self.tm._tasks[t1].priority, 4.0)
        self.assertEqual(self.tm._ready_queue.peek_max()[1], t2) # T2 重新成为最高

    def test_update_task_info_for_pending_task(self):
        """测试更新 PENDING 状态任务的优先级（不应影响就绪队列）。"""
        t1 = self._add_sample_task(10, 1, "T1")
        t2 = self._add_sample_task(5, 1, "T2")
        self.tm.add_dependency(t1, t2) # T2 变为 PENDING

        self.assertFalse(t2 in self.tm._ready_queue)
        old_t2_priority = self.tm._tasks[t2].priority

        # 更新 T2 的优先级
        self.assertTrue(self.tm.update_task_info(t2, new_urgency=100)) # P(T2) 现在很高
        self.assertNotEqual(self.tm._tasks[t2].priority, old_t2_priority)
        self.assertFalse(t2 in self.tm._ready_queue, "PENDING 任务更新优先级后不应进入就绪队列")


    def test_delete_task_simple(self):
        """测试删除任务。"""
        t1 = self._add_sample_task(10, 1, "T1")
        t2 = self._add_sample_task(5, 1, "T2")

        self.assertTrue(self.tm.delete_task(t1))
        self.assertNotIn(t1, self.tm._tasks)
        self.assertNotIn(t1, self.tm._ready_queue)
        self.assertNotIn(t1, self.tm._in_degree)
        self.assertFalse(self.tm._task_graph.has_task(t1))
        self.assertEqual(self.tm._ready_queue.get_heap_size(), 1) # 只剩 T2

        self.assertFalse(self.tm.delete_task("non_existent_task"))

    def test_delete_task_with_dependencies_updates_dependents(self):
        """测试删除任务时，其后继任务的入度和就绪状态被正确更新。"""
        t1 = self._add_sample_task(1,1,"T1_Prereq")
        t2 = self._add_sample_task(1,1,"T2_Dependent")
        t3 = self._add_sample_task(1,1,"T3_AlsoDependent")
        t4 = self._add_sample_task(1,1,"T4_AnotherPrereq")

        self.tm.add_dependency(t1, t2) # T1 -> T2
        self.tm.add_dependency(t1, t3) # T1 -> T3
```

```
        self.tm.add_dependency(t4, t2) # T4 -> T2

        # T2 依赖 T1, T4. T3 依赖 T1.
        # 初始: T1, T4 READY. T2, T3 PENDING.
        # in_degree: T2=2, T3=1
        self.assertEqual(self.tm._in_degree[t2], 2)
        self.assertEqual(self.tm._tasks[t2].status, TASK_STATUS_PENDING)
        self.assertEqual(self.tm._in_degree[t3], 1)
        self.assertEqual(self.tm._tasks[t3].status, TASK_STATUS_PENDING)

        # 删除 T1
        self.assertTrue(self.tm.delete_task(t1))

        # T2 现在只依赖 T4 (因为 T1 被删除, T1->T2 依赖消失)
        self.assertEqual(self.tm._in_degree[t2], 1, "T2的入度应因T1删除而减少")
        self.assertEqual(self.tm._tasks[t2].status, TASK_STATUS_PENDING, "T2仍应PENDING因为它还依赖T4")

        # T3 现在应该变为 READY (因为它唯一的依赖 T1 被删除)
        self.assertEqual(self.tm._in_degree[t3], 0, "T3的入度应因T1删除而变为0")
        self.assertEqual(self.tm._tasks[t3].status, TASK_STATUS_READY, "T3应变为READY")
        self.assertTrue(t3 in self.tm._ready_queue)

        # 再次删除 T4
        self.assertTrue(self.tm.delete_task(t4))
        # T2 现在应该变为 READY (因为它唯一的依赖 T4 被删除)
        self.assertEqual(self.tm._in_degree[t2], 0, "T2的入度应因T4删除而变为0")
        self.assertEqual(self.tm._tasks[t2].status, TASK_STATUS_READY, "T2应变为READY")
        self.assertTrue(t2 in self.tm._ready_queue)


if __name__ == '__main__':
    # 为了让这个测试文件能独立运行, 你需要确保以下类可以被导入:
    # MarketingTask, TaskManager, TASK_STATUS_* (来自你的 marketing_task_schedule.py)
    # UpdatableMaxHeap (来自你的 updatable_max_heap.py)
    # TaskDependencyGraph (来自你的 dependency_graph.py)
    # 这通常意味着你的 src 目录在 PYTHONPATH 中, 或者你从项目根目录运行测试。
    unittest.main()
```

B.2 客户网络与影响力传播分析模块

这一部分包括客户网络与影响力传播分析模块在实现时用到的所有单元测试代码。

B.2.1 test_customer_graph.py

Listing 4: test_dfs_influence.py

```
import unittest
from src.data_structure.customer_graph import CustomerGraph

class TestCustomerGraph(unittest.TestCase):

    def setUp(self):
```

```
        """
        在每个测试方法运行前调用，用于设置测试环境。
        """
        self.graph = CustomerGraph()

    def test_initialization(self):
        """测试图是否被正确初始化为空"""
        self.assertEqual(len(self.graph._adj_list), 0)
        self.assertEqual(self.graph.get_all_customers(), [])

    def test_add_customer(self):
        """测试添加客户节点"""
        self.graph.add_customer("Alice")
        self.assertIn("Alice", self.graph._adj_list)
        self.assertEqual(self.graph._adj_list["Alice"], {})
        self.assertEqual(self.graph.get_all_customers(), ["Alice"])

        self.graph.add_customer("Bob")
        self.assertIn("Bob", self.graph._adj_list)
        self.assertEqual(set(self.graph.get_all_customers()), {"Alice", "Bob"})

        # 测试重复添加客户
        initial_alice_edges = self.graph._adj_list["Alice"].copy()
        self.graph.add_customer("Alice") # 不应有任何改变或错误
        self.assertEqual(len(self.graph._adj_list), 2, "重复添加客户后，客户总数不应改变")
        self.assertEqual(self.graph._adj_list["Alice"], initial_alice_edges, "重复添加客户 '
Alice' 后，其出边字典应保持不变")


    def test_add_customer_invalid_type(self):
        """测试添加无效类型的客户名称（非字符串）"""
        # 准确匹配 CustomerGraph 中定义的错误信息
        with self.assertRaisesRegex(TypeError, r"错误：客户名称 '123' 必须是字符串，但收到了<
class 'int'>"):
            self.graph.add_customer(123)
        with self.assertRaisesRegex(TypeError, r"错误：客户名称 'None' 必须是字符串，但收到了<
class 'NoneType'>"):
            self.graph.add_customer(None)
        with self.assertRaisesRegex(TypeError, r"错误：客户名称 '\['ListCustomer'\]' 必须是字
符串，但收到了<class 'list'>"):
            self.graph.add_customer(["ListCustomer"])
        self.assertEqual(len(self.graph._adj_list), 0, "添加无效类型客户后，图应仍为空")

    def test_get_all_customers(self):
        """测试获取所有客户"""
        self.assertEqual(self.graph.get_all_customers(), [])
        self.graph.add_customer("Alice")
        self.graph.add_customer("Bob")
        customers = self.graph.get_all_customers()
        self.assertEqual(set(customers), {"Alice", "Bob"})
        self.assertEqual(len(customers), 2)

    def test_add_influence_valid(self):
        """测试添加有效的影响力关系"""
        self.graph.add_influence("Alice", "Bob", 0.8)
        self.assertIn("Alice", self.graph._adj_list)
```

```
        self.assertIn("Bob", self.graph._adj_list)
        self.assertIn("Bob", self.graph._adj_list["Alice"])
        self.assertEqual(self.graph._adj_list["Alice"]["Bob"], 0.8)

        self.graph.add_influence("Alice", "Bob", 0.9) # 更新权重
        self.assertEqual(self.graph._adj_list["Alice"]["Bob"], 0.9)

        self.graph.add_influence("Alice", "Charlie", 0.0) # 权重为0是允许的
        self.assertIn("Charlie", self.graph._adj_list)
        self.assertEqual(self.graph._adj_list["Alice"]["Charlie"], 0.0)

    def test_add_influence_invalid_customer_names_type(self):
        """测试 add_influence 时客户名称类型无效 (使用组合错误消息)"""
        # from_customer 无效
        with self.assertRaisesRegex(TypeError, r"错误: 客户名称 '123' 以及 'Bob' 必须是字符串, 但收到了<class 'int'>以及<class 'str'>"):
            self.graph.add_influence(123, "Bob", 0.5)

        # to_customer 无效
        with self.assertRaisesRegex(TypeError, r"错误: 客户名称 'Alice' 以及 '\['Charlie'\]' 必须是字符串, 但收到了<class 'str'>以及<class 'list'>"):
            self.graph.add_influence("Alice", ["Charlie"], 0.5)

        # 两者都无效
        with self.assertRaisesRegex(TypeError, r"错误: 客户名称 '123' 以及 '456' 必须是字符串, 但收到了<class 'int'>以及<class 'int'>"):
            self.graph.add_influence(123, 456, 0.5)

        self.assertEqual(len(self.graph._adj_list), 0, "发生类型错误后, 不应添加任何客户或边")

    def test_add_influence_invalid_weight_type(self):
        """测试 add_influence 时权重类型无效"""
        with self.assertRaisesRegex(TypeError, r"错误: 影响力权重 'invalid_weight' 必须是数字, 但收到了<class 'str'>"):
            self.graph.add_influence("Alice", "Bob", "invalid_weight")
        # 客户可能已被 add_customer 添加 (因为类型检查在 add_customer 之后), 但边不应添加
        # 根据 CustomerGraph 实现, 类型检查优先于 add_customer 调用
        self.assertEqual(len(self.graph._adj_list),0,"客户不应被添加如果参数类型检查失败")


    def test_add_influence_negative_weight(self):
        """测试 add_influence 时权重为负数"""
        with self.assertRaisesRegex(ValueError, r"警告: 影响力权重 '-0.5' 不允许为负数"):
            self.graph.add_influence("Alice", "Bob", -0.5)


    def test_get_direct_influencees(self):
        """测试获取直接影响的客户"""
        self.graph.add_influence("Alice", "Bob", 0.8)
        self.graph.add_influence("Alice", "Charlie", 0.6)
        self.graph.add_influence("Bob", "Charlie", 0.9)

        alice_influences = self.graph.get_direct_influencees("Alice")
        self.assertEqual(alice_influences, {"Bob": 0.8, "Charlie": 0.6})
        # 确保返回的是副本
        alice_influences["David"] = 1.0
```

```
        self.assertEqual(self.graph.get_direct_influencees("Alice"), {"Bob": 0.8, "Charlie": 0.6})


        bob_influences = self.graph.get_direct_influencees("Bob")
        self.assertEqual(bob_influences, {"Charlie": 0.9})

        self.graph.add_customer("David") # David 没有影响任何人
        david_influences = self.graph.get_direct_influencees("David")
        self.assertEqual(david_influences, {})

        non_existent_influences = self.graph.get_direct_influencees("Zoe") # Zoe 不存在
        self.assertEqual(non_existent_influences, {})

    def test_get_direct_influencees_invalid_type(self):
        """测试 get_direct_influencees 时客户名称类型无效"""
        with self.assertRaisesRegex(TypeError, r"错误: 客户名称 '123' 必须是字符串，但收到了<class 'int'>"):
            self.graph.get_direct_influencees(123)

    def test_get_influence_weight(self):
        """测试获取特定的影响力权重"""
        self.graph.add_influence("Alice", "Bob", 0.8)
        self.graph.add_influence("Alice", "Charlie", 0.0)

        self.assertEqual(self.graph.get_influence_weight("Alice", "Bob"), 0.8)
        self.assertEqual(self.graph.get_influence_weight("Alice", "Charlie"), 0.0)
        self.assertIsNone(self.graph.get_influence_weight("Alice", "David")) # David 未被 Alice 影响
        self.assertIsNone(self.graph.get_influence_weight("Bob", "Alice"))   # Bob 未影响 Alice
        self.assertIsNone(self.graph.get_influence_weight("Zoe", "Alice"))   # Zoe 不存在
        self.assertIsNone(self.graph.get_influence_weight("Alice", "Zoe"))   # Zoe 不存在

    def test_get_influence_weight_invalid_customer_names_type(self):
        """测试 get_influence_weight 时客户名称类型无效 (组合错误消息)"""
        with self.assertRaisesRegex(TypeError, r"错误: 客户名称 '123' 以及 'Bob' 必须是字符串，但收到了<class 'int'>以及<class 'str'>"):
            self.graph.get_influence_weight(123, "Bob")
        with self.assertRaisesRegex(TypeError, r"错误: 客户名称 'Alice' 以及 '\['Eve'\]' 必须是字符串，但收到了<class 'str'>以及<class 'list'>"):
            self.graph.get_influence_weight("Alice", ["Eve"])


if __name__ == '__main__':
    unittest.main()
```

B.2.2 test_dfs_influence.py

Listing 5: test_dfs_influence.py

```
import unittest
from src.data_structure.customer_graph import CustomerGraph
from src.module.customer_network_analysis import analyze_all_customer_influence_nodes

```

```python
class TestInfluenceSpreadAnalysis(unittest.TestCase):

    def setUp(self):
        """
        在每个测试方法运行前调用，用于设置测试环境。
        """
        self.graph = CustomerGraph()

    def test_analyze_all_param_validation_invalid_graph(self):
        """测试 analyze_all_customer_influence_nodes 函数传入无效的图类型"""
        with self.assertRaisesRegex(TypeError, "输入图必须是 CustomerGraph 对象"):
            analyze_all_customer_influence_nodes("not_a_graph", 0.5)

    def test_analyze_all_param_validation_invalid_min_influence(self):
        """测试 analyze_all_customer_influence_nodes 函数传入无效的 min_influence"""
        self.graph.add_customer("A")
        with self.assertRaisesRegex(ValueError, "min_influence 必须属于0和1之间"):
            analyze_all_customer_influence_nodes(self.graph, -0.1)
        with self.assertRaisesRegex(ValueError, "min_influence 必须属于0和1之间"):
            analyze_all_customer_influence_nodes(self.graph, 1.1)
        # 测试有效的边界值 (0, 1] - 你的代码实现意味着 min_influence=1 是有效的
        # 如果 min_influence == 1 是有效的:
        try:
            analyze_all_customer_influence_nodes(self.graph, 1.0) # 不应抛出异常
            analyze_all_customer_influence_nodes(self.graph, 0.0001) # 不应抛出异常
        except ValueError:
            self.fail("有效的 min_influence (例如 0.0001 或 1.0) 意外地引发了 ValueError")


    def test_empty_graph(self):
        """测试空图的情况"""
        result = analyze_all_customer_influence_nodes(self.graph, 0.1)
        self.assertEqual(result, {})

    def test_single_node_graph(self):
        """测试只包含一个节点的图"""
        self.graph.add_customer("A")
        result = analyze_all_customer_influence_nodes(self.graph, 0.1)
        self.assertEqual(result, {"A": set()})

    def test_two_nodes_no_influence_below_threshold(self):
        """测试 A->B，但影响力低于阈值的情况"""
        self.graph.add_influence("A", "B", 0.05)
        result = analyze_all_customer_influence_nodes(self.graph, 0.1)
        expected = {
            "A": set(),
            "B": set()
        }
        self.assertEqual(result, expected)

    def test_two_nodes_influence_at_threshold(self):
        """测试 A->B，影响力恰好等于阈值的情况"""
        self.graph.add_influence("A", "B", 0.1)
        result = analyze_all_customer_influence_nodes(self.graph, 0.1)
        expected = {
            "A": {"B"},
```

```
            "B": set()
        }
        self.assertEqual(result, expected)

    def test_two_nodes_influence_above_threshold(self):
        """测试 A->B, 影响力高于阈值的情况"""
        self.graph.add_influence("A", "B", 0.2)
        result = analyze_all_customer_influence_nodes(self.graph, 0.1)
        expected = {
            "A": {"B"},
            "B": set()
        }
        self.assertEqual(result, expected)

    def test_chain_influence_A_B_C(self):
        """测试链式影响 A->B->C 在不同阈值下的情况"""
        self.graph.add_influence("A", "B", 0.5)
        self.graph.add_influence("B", "C", 0.4) # A->B->C 的路径影响力 = 0.5 * 0.4 = 0.2
        self.graph.add_customer("D") # 未连接的节点

        # 情况1: min_influence = 0.3 (A 无法影响到 C, 因为 0.2 < 0.3)
        result1 = analyze_all_customer_influence_nodes(self.graph, 0.3)
        expected1 = {
            "A": {"B"},           # A->B (0.5) 符合条件
            "B": {"C"},           # B->C (0.4) 符合条件
            "C": set(),
            "D": set()
        }
        self.assertEqual(result1, expected1)

        # 情况2: min_influence = 0.1 (A 可以影响到 C)
        result2 = analyze_all_customer_influence_nodes(self.graph, 0.1)
        expected2 = {
            "A": {"B", "C"},     # A->B (0.5), A->B->C (0.2)
            "B": {"C"},          # B->C (0.4)
            "C": set(),
            "D": set()
        }
        self.assertEqual(result2, expected2)

        # 情况3: min_influence = 0.45 (A 无法影响到 C, B 无法影响到 C)
        result3 = analyze_all_customer_influence_nodes(self.graph, 0.45)
        expected3 = {
            "A": {"B"},         # A->B (0.5)
            "B": set(),         # B->C (0.4) < 0.45
            "C": set(),
            "D": set()
        }
        self.assertEqual(result3, expected3)

    def test_branch_and_merge(self):
        """测试分支与合并路径 A->B, A->C, B->D, C->D"""
        self.graph.add_influence("A", "B", 0.6)
        self.graph.add_influence("A", "C", 0.3)
        self.graph.add_influence("B", "D", 0.5) # A->B->D 路径影响力 = 0.6 * 0.5 = 0.3
        self.graph.add_influence("C", "D", 0.9) # A->C->D 路径影响力 = 0.3 * 0.9 = 0.27
```

```
117
118         # min_influence = 0.25
119         result = analyze_all_customer_influence_nodes(self.graph, 0.25)
120         expected = {
121             "A": {"B", "C", "D"}, # B (0.6), C (0.3), D 可通过 A->B->D (0.3) 和 A->C->D (0.27)
    到达
122             "B": {"D"},           # D (0.5)
123             "C": {"D"},           # D (0.9)
124             "D": set()
125         }
126         self.assertEqual(result, expected)
127
128         # min_influence = 0.28 (A->C->D 这条路径被剪枝)
129         result_stricter = analyze_all_customer_influence_nodes(self.graph, 0.28)
130         expected_stricter = {
131             "A": {"B", "C", "D"}, # B (0.6), C (0.3), D 只能通过 A->B->D (0.3) 到达
132             "B": {"D"},           # D (0.5)
133             "C": {"D"},           # D (0.9)
134             "D": set()
135         }
136         self.assertEqual(result_stricter, expected_stricter)
137
138
139     def test_cycle_handling(self):
140         """测试包含环路的图 A->B, B->A"""
141         self.graph.add_influence("A", "B", 0.8)
142         self.graph.add_influence("B", "A", 0.7) # A->B->A 路径影响力 = 0.8 * 0.7 = 0.56
143         self.graph.add_influence("A", "C", 0.2) # A->C 路径影响力 = 0.2
144
145         # min_influence = 0.1
146         result = analyze_all_customer_influence_nodes(self.graph, 0.1)
147         # A 能影响: B (0.8), C (0.2)
148         # B 能影响: A (0.7)
149         # A 通过 A->B->A (0.56) 也能影响到 A, 但起始节点不计入其自身的影响集合。
150         expected = {
151             "A": {"B", "C"},
152             "B": {"A", "C"},
153             "C": set()
154         }
155         self.assertEqual(result, expected)
156
157     def test_cycle_with_exit_path(self):
158         """测试包含环路并有出口的路径 A->B, B->C, C->B (B-C形成环), B->D"""
159         self.graph.add_influence("A", "B", 0.9)
160         self.graph.add_influence("B", "C", 0.8) # A->B->C 路径影响力 = 0.72
161         self.graph.add_influence("C", "B", 0.7) # A->B->C->B 路径影响力 = 0.72 * 0.7 = 0.504
162         self.graph.add_influence("B", "D", 0.6) # A->B->D 路径影响力 = 0.9 * 0.6 = 0.54
163                                                 # A->B->C->B->D 路径影响力 = 0.504 * 0.6 = 0.3024
164
165         # min_influence = 0.3
166         result = analyze_all_customer_influence_nodes(self.graph, 0.3)
167         # A 能影响 B (0.9)
168         # A 能影响 C (通过 A->B->C = 0.72)
169         # A 能影响 D (通过 A->B->D = 0.54 和 A->B->C->B->D = 0.3024)
170         # B 能影响 C (0.8)
171         # B 能影响 D (0.6)
```

```
        # C 能影响 B (0.7)
        # C 能影响 D (通过 C->B->D = 0.7 * 0.6 = 0.42)
        expected = {
            "A": {"B", "C", "D"},
            "B": {"C", "D"},
            "C": {"B", "D"},
            "D": set()
        }
        self.assertEqual(result, expected)

    def test_self_loop(self):
        """测试包含自环的图 A->A"""
        self.graph.add_influence("A", "A", 0.9) # 自环
        self.graph.add_influence("A", "B", 0.5)

        result = analyze_all_customer_influence_nodes(self.graph, 0.1)
        # A 能影响 B (0.5)。根据题目“其他客户”，A 不把自己算作被影响者。
        # 通过 A->A->B (0.9*0.5 = 0.45) 也是 A 影响 B 的一条路径。
        expected = {
            "A": {"B"},
            "B": set()
        }
        self.assertEqual(result, expected)

    def test_zero_weight_edge_pruning(self):
        """测试当路径中某条边权重为0时，路径被剪枝的情况"""
        self.graph.add_influence("A", "B", 1.0)
        self.graph.add_influence("B", "C", 0.0) # 这条边使得 A->B->C 路径影响力为 0
        self.graph.add_influence("C", "D", 1.0)
        self.graph.add_influence("A", "E", 0.5)

        # min_influence 必须为正数 (例如，0.01，根据你代码中的校验)
        result = analyze_all_customer_influence_nodes(self.graph, 0.01)
        # A 能影响 B (1.0)
        # A 能影响 E (0.5)
        # A 不能通过 B 影响到 C 或 D，因为 A->B->C 的路径影响力是 0。
        # B 不能影响 C 或 D。
        # C 能影响 D (1.0)
        expected = {
            "A": {"B", "E"},
            "B": set(),
            "C": {"D"},
            "D": set(),
            "E": set()
        }
        self.assertEqual(result, expected)

if __name__ == '__main__':
    unittest.main()
```

B.2.3 test_pagerank.py

Listing 6: test_dfs_influence.py

```
import unittest
```

```python
import math # 用于 isclose 比较浮点数总和

from src.data_structure.customer_graph import CustomerGraph
from src.module.customer_network_analysis import (
    _preprocess_for_pagerank,
    calculate_pagerank
)


class TestPageRankAnalysis(unittest.TestCase):

    def setUp(self):
        self.graph = CustomerGraph()
        # 常用的 PageRank 参数
        self.damping_factor = 0.85
        self.max_iterations = 100 # 使用一个合理的迭代次数进行测试
        self.epsilon = 1e-7 # 一个较小的 epsilon

    def test_pagerank_parameter_validation(self):
        """测试 calculate_pagerank 函数的参数校验"""
        with self.assertRaisesRegex(TypeError, "输入必须是一个 CustomerGraph 对象"):
            calculate_pagerank("not_a_graph")

        g = CustomerGraph()
        g.add_customer("A")

        with self.assertRaisesRegex(ValueError, "阻尼因子必须在 0 和 1 之间"):
            calculate_pagerank(g, damping_factor=1.5)
        with self.assertRaisesRegex(ValueError, "阻尼因子必须在 0 和 1 之间"):
            calculate_pagerank(g, damping_factor=0)
        with self.assertRaisesRegex(ValueError, "阻尼因子必须在 0 和 1 之间"):
            calculate_pagerank(g, damping_factor=-0.1)

        with self.assertRaisesRegex(ValueError, "最大迭代次数必须大于 0"):
            calculate_pagerank(g, max_iterations=0)
        with self.assertRaisesRegex(ValueError, "最大迭代次数必须大于 0"):
            calculate_pagerank(g, max_iterations=-5)

        with self.assertRaisesRegex(ValueError, "Epsilon 必须大于 0"):
            calculate_pagerank(g, epsilon=0)
        with self.assertRaisesRegex(ValueError, "Epsilon 必须大于 0"):
            calculate_pagerank(g, epsilon=-0.001)

    def test_pagerank_empty_graph(self):
        """测试空图的 PageRank"""
        self.assertEqual(calculate_pagerank(self.graph), {})

    def test_pagerank_single_node_graph(self):
        """测试单个节点的图的 PageRank"""
        self.graph.add_customer("A")
        expected_pagerank = {"A": 1.0 / 1.0} # 初始为 1/N, 迭代后应保持
        # 对于单个节点，没有入链，没有悬挂节点贡献给自己，只有 (1-d)/N
        # 实际PageRank公式 (1-d)/N + d*0 + d*PR(A)/N (如果悬挂节点是这样处理的)
        # 如果悬挂节点PR(A)被分配给它自己(1/N * PR(A))，则PR(A) = (1-d)/N + d*PR(A)/N
        # PR(A)*(1-d/N) = (1-d)/N => PR(A) = (1-d)/(N-d)
        # 但对于单节点 N=1，通常其 PageRank 就是 1.0 (或 1/N)。
```

```
        # 我们的实现中，单节点出度为0，是悬挂节点。
        # PR(A) = (1-d)/1 + d * (PR(A)/1) => PR(A) = 1-d + d*PR(A) => PR(A)(1-d) = 1-d => PR(A)=1

        # 让我们跟踪代码逻辑：
        # pagerank_scores = {"A": 1.0}
        # dangling_nodes_pagerank_sum = 1.0
        # pagerank_from_dangling_nodes = d * (1.0 / 1.0) = d
        # pagerank_from_teleport = (1.0 - d) / 1.0 = 1.0 - d
        # pagerank_from_incoming_links = 0.0
        # new_pagerank_scores["A"] = (1.0 - d) + 0 + d = 1.0
        # 它会立即收敛到 1.0

        result = calculate_pagerank(self.graph,
                                    damping_factor=self.damping_factor,
                                    max_iterations=self.max_iterations,
                                    epsilon=self.epsilon)
        self.assertIn("A", result)
        self.assertAlmostEqual(result["A"], 1.0, places=7)
        self.assertTrue(math.isclose(sum(result.values()), 1.0, rel_tol=1e-5))


    def test_pagerank_two_nodes_symmetric_link(self):
        """测试两个节点双向连接的 PageRank (A<->B)"""
        self.graph.add_influence("A", "B", 1.0)
        self.graph.add_influence("B", "A", 1.0)
        # 预期：A 和 B 的 PageRank 应该相等，且总和为1 (即各0.5)
        result = calculate_pagerank(self.graph,
                                    damping_factor=self.damping_factor,
                                    max_iterations=self.max_iterations,
                                    epsilon=self.epsilon)
        self.assertIn("A", result)
        self.assertIn("B", result)
        self.assertAlmostEqual(result["A"], 0.5, places=5) # 精度可能会有影响
        self.assertAlmostEqual(result["B"], 0.5, places=5)
        self.assertTrue(math.isclose(sum(result.values()), 1.0, rel_tol=1e-5))

    def test_pagerank_three_nodes_loop(self):
        """测试三个节点的简单循环 (A->B, B->C, C->A)"""
        self.graph.add_influence("A", "B", 1.0)
        self.graph.add_influence("B", "C", 1.0)
        self.graph.add_influence("C", "A", 1.0)
        # 预期：A, B, C 的 PageRank 应该相等，且总和为1 (即各约1/3)
        result = calculate_pagerank(self.graph,
                                    damping_factor=self.damping_factor,
                                    max_iterations=self.max_iterations,
                                    epsilon=self.epsilon)
        self.assertIn("A", result)
        self.assertIn("B", result)
        self.assertIn("C", result)
        expected_score = 1.0 / 3.0
        self.assertAlmostEqual(result["A"], expected_score, places=5)
        self.assertAlmostEqual(result["B"], expected_score, places=5)
        self.assertAlmostEqual(result["C"], expected_score, places=5)
        self.assertTrue(math.isclose(sum(result.values()), 1.0, rel_tol=1e-5))

```

```
    def test_pagerank_dangling_node_simple(self):
        """测试包含一个简单悬挂节点的 PageRank (A->B, A->C, B是悬挂节点)"""
        self.graph.add_influence("A", "B", 1.0) # B 是悬挂节点 (无出链)
        self.graph.add_influence("A", "C", 1.0)
        # A 指向 B 和 C。B 是悬挂节点。
        # C 也可能是悬挂节点，如果它没有出链。让我们让 C 指回 A 来使其不悬挂。
        self.graph.add_influence("C", "A", 1.0)

        # 节点: A, B, C
        # 初始 PR: A=1/3, B=1/3, C=1/3
        # WOD: A=2, B=0, C=1

        # 迭代1 (大致思路):
        # Dangling PR sum (from B) = PR(B) = 1/3
        # PR_dangling_contrib_per_node = d * ( (1/3) / 3 ) = d/9
        # For A: (1-d)/3 + d * (PR(C)*1/WOD(C)) + PR_dangling_contrib_per_node
        #        = (1-d)/3 + d * ( (1/3)*1/1 ) + d/9 = (1-d)/3 + d/3 + d/9 = 1/3 + d/9
        # For B (no in-links other than its own dangling contribution):
        #        = (1-d)/3 + d * (0) + PR_dangling_contrib_per_node
        #        = (1-d)/3 + d/9
        # For C: (1-d)/3 + d * (PR(A)*1/WOD(A)) + PR_dangling_contrib_per_node
        #        = (1-d)/3 + d * ( (1/3)*1/2 ) + d/9 = (1-d)/3 + d/6 + d/9
        # A 和 C 应该比 B 有更高的PR。

        result = calculate_pagerank(self.graph,
                                    damping_factor=self.damping_factor,
                                    max_iterations=self.max_iterations,
                                    epsilon=self.epsilon)

        self.assertIn("A", result)
        self.assertIn("B", result)
        self.assertIn("C", result)
        self.assertTrue(result["A"] > result["B"] or math.isclose(result["A"], result["B"],
    rel_tol=1e-5)) # A should be >= B
        self.assertAlmostEqual(result["C"], result["B"], places=5, msg="PR(C) and PR(B) should
     be equal in this graph")
        self.assertTrue(math.isclose(sum(result.values()), 1.0, rel_tol=1e-5),
                        f"Sum of PageRanks is not 1.0: {sum(result.values())}")

    def test_pagerank_node_with_only_zero_weight_out_edge(self):
        """测试节点的唯一出边权重为0 (应视为悬挂节点)"""
        self.graph.add_influence("A", "B", 0.0) # A 的 WOD 是 0, 所以 A 是悬挂节点
        self.graph.add_influence("C", "A", 1.0) # C 指向 A
        # 节点: A, B, C
        # WOD: A=0, B=0, C=1
        # A 和 B 都是悬挂节点

        result = calculate_pagerank(self.graph,
                                    damping_factor=self.damping_factor,
                                    max_iterations=self.max_iterations,
                                    epsilon=self.epsilon)
        self.assertIn("A", result)
        self.assertIn("B", result)
        self.assertIn("C", result)
        # 预期 C 的 PageRank 会比较高，因为它有实际的出链并且接收了随机跳转部分，
        # A 和 B 的 PageRank 主要来自随机跳转和悬挂节点的分配。
```

```
        # A 应该比 B 高，因为它有来自 C 的入链(即使 A 自身是悬挂的)。
        self.assertTrue(result["A"] > result["B"] or math.isclose(result["A"], result["B"])) # A比B高或相近
        self.assertTrue(result["C"] > result["B"] or math.isclose(result["C"], result["B"])) # C比B高或相近
        self.assertTrue(math.isclose(sum(result.values()), 1.0, rel_tol=1e-5))


    def test_pagerank_complex_graph_relative_ranks(self):
        """测试一个稍复杂图的相对 PageRank 顺序"""
        self.graph.add_influence("A", "B", 1.0)
        self.graph.add_influence("A", "C", 1.0)
        self.graph.add_influence("B", "D", 1.0)
        self.graph.add_influence("C", "D", 1.0) # D 被两个节点指向
        self.graph.add_influence("E", "A", 1.0) # E 指向 A (A 的重要性来源之一)
        self.graph.add_influence("D", "E", 0.5) # D 也指回 E 形成一个小循环
        # F 是孤立的，但通过随机跳转获得 PR
        self.graph.add_customer("F")

        # 预期 D 应该有较高的 PageRank
        # A 和 E 也会因为互相指向及 E 指向 A 而有一定 PageRank
        # B 和 C 作为中间节点
        # F 作为孤立节点，PR 应该最低（仅来自随机跳转和悬挂贡献）
        result = calculate_pagerank(self.graph,
                                    damping_factor=self.damping_factor,
                                    max_iterations=200, # 增加迭代次数确保收敛
                                    epsilon=self.epsilon)

        self.assertTrue(all(score >= 0 for score in result.values()))
        self.assertTrue(math.isclose(sum(result.values()), 1.0, rel_tol=1e-4),
                        f"Sum of PageRanks is not 1.0: {sum(result.values())}")

        # 难以精确预测，但可以做一些相对比较
        # print("\nComplex Graph PR:", {k: round(v,4) for k,v in result.items()}) # 用于调试
        if "D" in result and "F" in result:
             self.assertTrue(result["D"] > result["F"], "PR(D) should be > PR(F)")
        if "A" in result and "B" in result:
             self.assertTrue(result["A"] > result["B"] or result["A"] > result["C"], "PR(A) should be relatively high")


    def test_preprocess_function_directly(self):
        """(可选) 直接测试 _preprocess_for_pagerank 的基本功能"""
        self.graph.add_influence("A", "B", 0.5)
        self.graph.add_influence("A", "C", 1.0)
        self.graph.add_influence("X", "A", 0.8)
        self.graph.add_customer("D") # Isolated

        incoming_links_map, weighted_out_degrees = _preprocess_for_pagerank(self.graph)

        expected_incoming_A = {"X": 0.8}
        expected_incoming_B = {"A": 0.5}
        expected_incoming_C = {"A": 1.0}
        expected_incoming_D = {}
        expected_incoming_X = {}

```

```
        self.assertEqual(incoming_links_map.get("A", {}), expected_incoming_A)
        self.assertEqual(incoming_links_map.get("B", {}), expected_incoming_B)
        self.assertEqual(incoming_links_map.get("C", {}), expected_incoming_C)
        self.assertEqual(incoming_links_map.get("D", {}), expected_incoming_D)
        self.assertEqual(incoming_links_map.get("X", {}), expected_incoming_X)
        self.assertEqual(set(incoming_links_map.keys()), {"A", "B", "C", "X", "D"})

        expected_wod_A = 0.5 + 1.0
        expected_wod_X = 0.8
        expected_wod_B = 0.0
        expected_wod_C = 0.0
        expected_wod_D = 0.0

        self.assertAlmostEqual(weighted_out_degrees.get("A", 0.0), expected_wod_A)
        self.assertAlmostEqual(weighted_out_degrees.get("X", 0.0), expected_wod_X)
        self.assertAlmostEqual(weighted_out_degrees.get("B", 0.0), expected_wod_B)
        self.assertAlmostEqual(weighted_out_degrees.get("C", 0.0), expected_wod_C)
        self.assertAlmostEqual(weighted_out_degrees.get("D", 0.0), expected_wod_D)
        self.assertEqual(set(weighted_out_degrees.keys()), {"A", "B", "C", "X", "D"})


if __name__ == '__main__':
    unittest.main()
```

B.2.4 test_weighted_degree_centrality.py

Listing 7: test_dfs_influence.py

```
import unittest
from src.data_structure.customer_graph import CustomerGraph
from src.module.customer_network_analysis import (
    calculate_weighted_out_degree_centrality,
    calculate_weighted_in_degree_centrality
)

class TestCustomerGraphAnalysis(unittest.TestCase):

    def setUp(self):
        """
        在每个测试方法运行前调用，用于设置测试环境。
        """
        self.graph = CustomerGraph()

    def test_calculate_weighted_out_degree_centrality_empty_graph(self):
        """测试空图的加权出度中心性"""
        expected_centrality = {}
        self.assertEqual(calculate_weighted_out_degree_centrality(self.graph),
    expected_centrality)

    def test_calculate_weighted_in_degree_centrality_empty_graph(self):
        """测试空图的加权入度中心性"""
        expected_centrality = {}
        self.assertEqual(calculate_weighted_in_degree_centrality(self.graph),
    expected_centrality)

```

```python
    def test_calculate_weighted_out_degree_centrality_simple_graph(self):
        """测试简单图的加权出度中心性"""
        self.graph.add_influence("A", "B", 0.5)
        self.graph.add_influence("A", "C", 1)
        self.graph.add_influence("B", "C", 0.8)
        self.graph.add_customer("D") # 孤立节点

        expected_centrality = {
            "A": 0.5 + 1,
            "B": 0.8,
            "C": 0.0,
            "D": 0.0
        }
        # 使用 assertAlmostEqual 比较浮点数，或者确保精度一致
        result = calculate_weighted_out_degree_centrality(self.graph)
        for node, score in expected_centrality.items():
            self.assertAlmostEqual(result.get(node, 0.0), score, places=7)
        self.assertEqual(set(result.keys()), set(expected_centrality.keys()))


    def test_calculate_weighted_in_degree_centrality_simple_graph(self):
        """测试简单图的加权入度中心性"""
        self.graph.add_influence("A", "B", 0.5)
        self.graph.add_influence("A", "C", 1.0)
        self.graph.add_influence("B", "C", 0.8)
        self.graph.add_customer("D") # 孤立节点

        expected_centrality = {
            "A": 0.0,
            "B": 0.5,
            "C": 1.0 + 0.8, # 2.3
            "D": 0.0
        }
        result = calculate_weighted_in_degree_centrality(self.graph)
        for node, score in expected_centrality.items():
            self.assertAlmostEqual(result.get(node, 0.0), score, places=7)
        self.assertEqual(set(result.keys()), set(expected_centrality.keys()))

    def test_centrality_with_zero_weight_edges(self):
        """测试包含零权重边的图的中心性"""
        self.graph.add_influence("A", "B", 0.0)
        self.graph.add_influence("A", "C", 1.0)
        self.graph.add_influence("C", "A", 0.5)

        # 加权出度
        expected_out_centrality = {
            "A": 0.0 + 1.0, # 1.0
            "B": 0.0,
            "C": 0.5
        }
        result_out = calculate_weighted_out_degree_centrality(self.graph)
        for node, score in expected_out_centrality.items():
            self.assertAlmostEqual(result_out.get(node, 0.0), score, places=7)
        self.assertEqual(set(result_out.keys()), set(expected_out_centrality.keys()))

        # 加权入度
```

```
        expected_in_centrality = {
            "A": 0.5,
            "B": 0.0,
            "C": 1.0
        }
        result_in = calculate_weighted_in_degree_centrality(self.graph)
        for node, score in expected_in_centrality.items():
            self.assertAlmostEqual(result_in.get(node, 0.0), score, places=7)
        self.assertEqual(set(result_in.keys()), set(expected_in_centrality.keys()))


    def test_calculate_weighted_out_degree_centrality_graph_with_no_edges(self):
        """测试只有节点没有边的图的加权出度中心性"""
        self.graph.add_customer("A")
        self.graph.add_customer("B")
        expected_centrality = {"A": 0.0, "B": 0.0}
        self.assertEqual(calculate_weighted_out_degree_centrality(self.graph),
    expected_centrality)

    def test_calculate_weighted_in_degree_centrality_graph_with_no_edges(self):
        """测试只有节点没有边的图的加权入度中心性"""
        self.graph.add_customer("A")
        self.graph.add_customer("B")
        expected_centrality = {"A": 0.0, "B": 0.0}
        self.assertEqual(calculate_weighted_in_degree_centrality(self.graph),
    expected_centrality)

    def test_out_degree_centrality_type_error(self):
        """测试加权出度中心性函数对无效输入的类型错误"""
        with self.assertRaisesRegex(TypeError, "输入必须是一个 CustomerGraph 对象"):
            calculate_weighted_out_degree_centrality("not_a_graph_object")
        with self.assertRaisesRegex(TypeError, "输入必须是一个 CustomerGraph 对象"):
            calculate_weighted_out_degree_centrality([1, 2, 3])

    def test_in_degree_centrality_type_error(self):
        """测试加权入度中心性函数对无效输入的类型错误"""
        with self.assertRaisesRegex(TypeError, "输入必须是一个 CustomerGraph 对象"):
            calculate_weighted_in_degree_centrality(None)
        with self.assertRaisesRegex(TypeError, "输入必须是一个 CustomerGraph 对象"):
            calculate_weighted_in_degree_centrality({"key": "value"})

    def test_centrality_complex_graph(self):
        """测试一个更复杂的图的中心性计算"""
        self.graph.add_influence("Alice", "Bob", 0.8)
        self.graph.add_influence("Alice", "Charlie", 0.6)
        self.graph.add_influence("Bob", "Charlie", 0.9)
        self.graph.add_influence("Bob", "David", 0.7)
        self.graph.add_influence("Charlie", "David", 0.5)
        self.graph.add_influence("Eve", "Alice", 0.4)
        self.graph.add_customer("Frank")

        # 加权出度
        expected_out = {
            "Alice": 0.8 + 0.6,   # 1.4
            "Bob": 0.9 + 0.7,     # 1.6
            "Charlie": 0.5,
```

```
            "David": 0.0,
            "Eve": 0.4,
            "Frank": 0.0
        }
        result_out = calculate_weighted_out_degree_centrality(self.graph)
        for node, score in expected_out.items():
            self.assertAlmostEqual(result_out.get(node, 0.0), score, places=7, msg=f"Out-degree for {node}")
        self.assertEqual(set(result_out.keys()), set(expected_out.keys()))

        # 加权入度
        expected_in = {
            "Alice": 0.4,
            "Bob": 0.8,
            "Charlie": 0.6 + 0.9, # 1.5
            "David": 0.7 + 0.5,   # 1.2
            "Eve": 0.0,
            "Frank": 0.0
        }
        result_in = calculate_weighted_in_degree_centrality(self.graph)
        for node, score in expected_in.items():
            self.assertAlmostEqual(result_in.get(node, 0.0), score, places=7, msg=f"In-degree for {node}")
        self.assertEqual(set(result_in.keys()), set(expected_in.keys()))

if __name__ == '__main__':
    unittest.main()
```

B.3 商品数据检索模块

B.3.1 test_b_plus_tree.py

Listing 8: test_b_plus_tree.py

```
import unittest

from src.data_structure.b_plus_tree import *
from src.module.commodity_retrieval import *

# --- 测试 BPlusTreeProducts ---
class TestBPlusTreeProducts(unittest.TestCase):
    def setUp(self):
        # 创建一些不同阶的树进行测试
        self.tree_order3 = BPlusTreeProducts(order=3)
        self.tree_order4 = BPlusTreeProducts(order=4)
        self.tree_order2 = BPlusTreeProducts(order=2) # 最小阶

        # 模拟一些商品ID
        self.p1 = Product("prod1", price=10.0)
        self.p2 = Product("prod2", price=20.0)
        self.p3 = Product("prod3", price=10.0) # 相同价格
        self.p4 = Product("prod4", price=30.0)
        self.p5 = Product("prod5", price=5.0)
        self.p6 = Product("prod6", price=15.0)
```

```
        self.p7 = Product("prod7", price=25.0)
        self.p8 = Product("prod8", price=20.0) # 相同价格

    def _insert_products(self, tree, products_with_prices):
        for price, product_id_obj in products_with_prices:
            tree.insert(price, product_id_obj) # BPlusTreeProducts 插入的是 (price,
    product_id_object)

    def test_01_initialization(self):
        self.assertIsNotNone(self.tree_order3.root)
        self.assertTrue(self.tree_order3.root.is_leaf)
        self.assertEqual(self.tree_order3.order, 3)
        with self.assertRaises(ValueError):
            BPlusTreeProducts(order=1)
        with self.assertRaises(ValueError):
            BPlusTreeNode(order=1)

    def test_02_simple_insert_and_search_exact(self):
        tree = self.tree_order3
        tree.insert(10.0, self.p1)
        tree.insert(20.0, self.p2)

        self.assertEqual(len(tree.search_exact(10.0)), 1)
        self.assertIn(self.p1, tree.search_exact(10.0))
        self.assertEqual(len(tree.search_exact(20.0)), 1)
        self.assertIn(self.p2, tree.search_exact(20.0))
        self.assertEqual(len(tree.search_exact(15.0)), 0) # 不存在的价格

    def test_03_insert_duplicate_prices(self):
        tree = self.tree_order3
        tree.insert(10.0, self.p1.product_id)
        tree.insert(10.0, self.p3.product_id) # 相同价格, 不同ID

        results = tree.search_exact(10.0)
        self.assertEqual(len(results), 2)
        self.assertIn(self.p1.product_id, results)
        self.assertIn(self.p3.product_id, results)

    def test_04_search_exact_with_product_id(self):
        tree = self.tree_order3
        tree.insert(10.0, self.p1.product_id)
        tree.insert(10.0, self.p3.product_id)
        tree.insert(20.0, self.p2.product_id)

        self.assertEqual(tree.search_exact(10.0, self.p1.product_id), [self.p1.product_id])
        self.assertEqual(tree.search_exact(10.0, self.p3.product_id), [self.p3.product_id])
        self.assertEqual(len(tree.search_exact(10.0, "nonexistent_id")), 0)
        self.assertEqual(tree.search_exact(20.0, self.p2.product_id), [self.p2.product_id])
        self.assertEqual(len(tree.search_exact(15.0, self.p1.product_id)), 0)


    def test_05_leaf_split(self):
        tree = BPlusTreeProducts(order=2) # 叶节点最多2个key
        # 插入会导致分裂的序列
        # tree._print_tree_structure(tree.root, 3)
        tree.insert(10.0, self.p1)
```

```
        # tree._print_tree_structure(tree.root, 3)
        tree.insert(20.0, self.p2)
        # tree._print_tree_structure(tree.root, 3)
        self.assertEqual(len(tree.root.keys), 2) # 10, 20
        tree.insert(5.0, self.p5) # 插入5.0, 导致分裂 (5), (10,20) -> 根[10], 叶[5],[10,20]
        # tree._print_tree_structure(tree.root, 3)

        self.assertFalse(tree.root.is_leaf)
        self.assertEqual(len(tree.root.keys), 1)
        self.assertEqual(tree.root.keys[0], 20.0) # 上推的键
        self.assertEqual(len(tree.root.children), 2)

        left_leaf = tree.root.children[0]
        right_leaf = tree.root.children[1]

        self.assertTrue(left_leaf.is_leaf)
        self.assertEqual(left_leaf.keys, [5.0, 10.0])
        self.assertEqual(left_leaf.values[0], [self.p5])

        self.assertTrue(right_leaf.is_leaf)
        self.assertEqual(right_leaf.keys, [20.0]) #分裂逻辑可能导致分裂点不同, 这里基于实现
        # 或者 [10.0], [20.0] 如果分裂点是第一个
        # 检查实现: _split_leaf 中 mid_point_index = math.ceil(len(leaf_to_split.keys) / 2)
        # 对于 [5,10,20] order=2 (max keys=2), 插入20后keys=[5,10,20] (len=3). mid=ceil(3/2)=2
        # new_leaf.keys = keys[2:] = [20]
        # leaf_to_split.keys = keys[:2] = [5,10]
        # key_to_push_up = new_leaf.keys[0] = 20
        # 实际插入顺序是 10, 20, 然后 5.
        # 1. 10 -> root=[10]
        # 2. 20 -> root=[10,20]
        # 3. 5 -> keys=[5,10,20], overflow. mid=ceil(3/2)=2. new_leaf.keys=[20], old_leaf.keys
=[5,10]. push_up=20
        #    root=[20], children: leaf1([5,10]), leaf2([20])
        # self.assertEqual(right_leaf.keys, [10.0, 20.0]) # 之前的理解分裂有点问题, 重新分析
        # 插入10: leaf_node.keys = [10], leaf_node.values = [[p1]]
        # 插入20: leaf_node.keys = [10, 20], leaf_node.values = [[p1], [p2]] (order 2, max 2
keys, not overflow)
        # 插入5: leaf_node.keys = [5, 10, 20], leaf_node.values = [[p5], [p1], [p2]] (overflow
!)
        # mid_point = ceil(3/2) = 2. new_leaf gets keys[2:] ([20]), values[2:] ([[p2]])
        # old_leaf gets keys[:2] ([5,10]), values[:2] ([[p5],[p1]])
        # key_to_push_up = new_leaf.keys[0] = 20
        # new_root.keys = [20], children = [old_leaf, new_leaf]
        self.assertEqual(left_leaf.keys, [5.0, 10.0])
        self.assertIn(self.p5, left_leaf.values[0])
        self.assertIn(self.p1, left_leaf.values[1])
        self.assertEqual(right_leaf.keys, [20.0])
        self.assertIn(self.p2, right_leaf.values[0])
        self.assertEqual(tree.root.keys[0], 20.0) # 分裂后上推的键

        # 验证叶节点链表
        self.assertEqual(left_leaf.next_leaf, right_leaf)
        self.assertEqual(right_leaf.prev_leaf, left_leaf)
        self.assertIsNone(left_leaf.prev_leaf)
        self.assertIsNone(right_leaf.next_leaf)

```

```
    def test_06_internal_node_split(self):
        tree = BPlusTreeProducts(order=2) # 内部节点最多2个key，3个孩子
        # 插入序列触发内部节点分裂
        # 1. 10, 20 -> root(leaf)=[10,20]
        # 2. 5 (leaf split) -> root(internal)=[20], children: L([5,10]), R([20])
        tree.insert(10.0, self.p1) # [[p1]]
        tree.insert(20.0, self.p2) # [[p1],[p2]]
        tree.insert(5.0, self.p5)  # [[p5],[p1]], [[p2]] ->推20
        # 当前: root=[20], child1(leaf)=[5,10], child2(leaf)=[20]

        # 3. 30 -> child2=[20,30]
        tree.insert(30.0, self.p4) # [[p5],[p1]], [[p2],[p4]]
        # 当前: root=[20], child1(leaf)=[5,10], child2(leaf)=[20,30]

        # 4. 25 (leaf split, child2 splits) -> [20,30] + [25] -> [20,25,30]
        #    mid=ceil(3/2)=2. new_leaf_val=[30], old_leaf_val=[20,25]. push_up=30
        #    child2 becomes [20,25], new_child_for_root is [30]
        #    parent (root) was [20]. Insert key 30.
        #    _insert_into_parent(child2, 30, new_child_for_root)
        #    root.keys becomes [20, 30] (插入30)
        #    root.children becomes [child1, child2, new_child_for_root]
        #    No internal split yet, root.keys=[20,30], children=[L(5,10),L(20,25),L(30)]
        tree.insert(25.0, self.p7) # [[p5],[p1]], [[p2],[p7]], [[p4]] -> 推30
        self.assertEqual(tree.root.keys, [20.0, 30.0])
        self.assertEqual(len(tree.root.children), 3)

        # 5. 15 (leaf split, child1 splits) -> [5,10] + [15] -> [5,10,15]
        #    mid=ceil(3/2)=2. new_leaf_val=[15], old_leaf_val=[5,10]. push_up=15
        #    child1 becomes [5,10], new_child_for_root is [15]
        #    parent (root) was [20,30]. Insert key 15.
        #    _insert_into_parent(child1, 15, new_child_for_root)
        #    root.keys becomes [15, 20, 30] (插入15) - overflow for order=2 (max 2 keys)
        #    Internal split: keys=[15,20,30]. mid_key_index = 2//2 = 1. key_to_push_up = keys
    [1] = 20
        #    new_internal_node.keys = keys[2:] = [30]
        #    new_internal_node.children = root.children[2:] (original children for 20 and 30)
        #    node_to_split (old root).keys = keys[:1] = [15]
        #    node_to_split.children = root.children[:2] (original children for <15 and between
     15,20)
        #    New root will be [20]
        tree.insert(15.0, self.p6)
        self.assertFalse(tree.root.is_leaf)
        self.assertEqual(len(tree.root.keys), 1) # Internal split, e.g. 20 pushed up
        self.assertEqual(tree.root.keys[0], 20.0) # 推20
        self.assertEqual(len(tree.root.children), 2) # Two internal nodes as children

        # Verify structure (this depends heavily on the split logic)
        # Root: [20]
        #   L-Internal: [15] children: L(5,10), L(15)
        #   R-Internal: [30] children: L(20,25), L(30)
        left_internal = tree.root.children[0]
        right_internal = tree.root.children[1]
        self.assertFalse(left_internal.is_leaf)
        self.assertEqual(left_internal.keys, [15.0])
        self.assertEqual(len(left_internal.children), 2)
        self.assertTrue(all(c.is_leaf for c in left_internal.children))
```

```
        # check values for 5,10,15
        self.assertEqual(left_internal.children[0].keys, [5.0, 10.0])
        self.assertEqual(left_internal.children[1].keys, [15.0])


        self.assertFalse(right_internal.is_leaf)
        self.assertEqual(right_internal.keys, [30.0])
        self.assertEqual(len(right_internal.children), 2)
        self.assertTrue(all(c.is_leaf for c in right_internal.children))
        self.assertEqual(right_internal.children[0].keys, [20.0, 25.0]) # or [25.0] if p7
    causes split, depends on p2/p7 values
        self.assertEqual(right_internal.children[1].keys, [30.0])

        # Search for all items
        self.assertIn(self.p1, tree.search_exact(10.0))
        self.assertIn(self.p2, tree.search_exact(20.0))
        self.assertIn(self.p4, tree.search_exact(30.0))
        self.assertIn(self.p5, tree.search_exact(5.0))
        self.assertIn(self.p6, tree.search_exact(15.0))
        self.assertIn(self.p7, tree.search_exact(25.0))


    def test_07_search_range(self):
        tree = self.tree_order3
        items = [
            (10.0, self.p1), (20.0, self.p2), (10.0, self.p3),
            (30.0, self.p4), (5.0, self.p5), (15.0, self.p6),
            (25.0, self.p7), (20.0, self.p8)
        ]
        self._insert_products(tree, items)

        # Range [10, 20]
        results = tree.search_range(10.0, 20.0)
        self.assertEqual(len(results), 5) # p1,p3 (10), p6 (15), p2,p8 (20)
        expected_ids_in_range = {self.p1, self.p3, self.p6, self.p2, self.p8}
        self.assertTrue(all(item in expected_ids_in_range for item in results))

        # Range [0, 5]
        results = tree.search_range(0.0, 5.0)
        self.assertEqual(len(results), 1)
        self.assertIn(self.p5, results)

        # Range [28, 100]
        results = tree.search_range(28.0, 100.0)
        self.assertEqual(len(results), 1)
        self.assertIn(self.p4, results) # p4 is 30.0

        # Empty range
        self.assertEqual(len(tree.search_range(100.0, 200.0)), 0)
        self.assertEqual(len(tree.search_range(12.0, 14.0)), 0)

        # Min_price > max_price
        self.assertEqual(len(tree.search_range(20.0, 10.0)), 0)

        # Single point range
        results = tree.search_range(10.0, 10.0)
```

```
        self.assertEqual(len(results), 2)
        self.assertIn(self.p1, results)
        self.assertIn(self.p3, results)

        # Full range
        all_products = [p_obj for _, p_obj in items]
        results = tree.search_range(0.0, 100.0)
        self.assertEqual(len(results), len(all_products))
        self.assertTrue(all(item in results for item in all_products))

    def test_08_delete_simple_leaf_no_underflow(self):
        tree = self.tree_order3 # order=3, min_keys = ceil(3/2) = 2
        tree.insert(10.0, self.p1.product_id)
        tree.insert(20.0, self.p2.product_id)
        tree.insert(30.0, self.p4.product_id) # root=[10,20,30]

        self.assertTrue(tree.delete(20.0, self.p2.product_id))
        self.assertEqual(len(tree.search_exact(20.0)), 0)
        self.assertEqual(tree.root.keys, [10.0, 30.0]) # Keys are now [10,30]
        self.assertEqual(len(tree.root.values[0]), 1) # p1
        self.assertEqual(len(tree.root.values[1]), 1) # p4

        # Delete non-existent
        self.assertFalse(tree.delete(15.0, "any_id"))
        self.assertFalse(tree.delete(10.0, "non_existent_id_for_10"))

    def test_09_delete_from_multiple_items_at_same_key(self):
        tree = self.tree_order3
        tree.insert(10.0, self.p1.product_id)
        tree.insert(10.0, self.p3.product_id)
        tree.insert(20.0, self.p2.product_id)

        self.assertTrue(tree.delete(10.0, self.p1.product_id))
        results_10 = tree.search_exact(10.0)
        self.assertEqual(len(results_10), 1)
        self.assertIn(self.p3.product_id, results_10)
        self.assertEqual(tree.root.keys, [10.0, 20.0]) # Key 10.0 still exists

        self.assertTrue(tree.delete(10.0, self.p3.product_id)) # Delete last item for key 10.0
        self.assertEqual(len(tree.search_exact(10.0)), 0)
        self.assertEqual(tree.root.keys, [20.0]) # Key 10.0 is removed

    def test_10_delete_causing_leaf_underflow_borrow_from_right_sibling(self):
        tree = BPlusTreeProducts(order=2) # min_keys = ceil(2/2) = 1
        # Setup: root=[20], L(leaf)=[10], R(leaf)=[20,30] (after inserting 10,20,30 with order
=2)
        # 10 -> [10]
        # 20 -> [10,20]
        # 30 -> [10,20,30] -> split. mid=ceil(3/2)=2. new_leaf=[30], old_leaf=[10,20], push_up
=30
        # root=[30], L=[10,20], R=[30]
        tree.insert(10.0, self.p1.product_id)
        tree.insert(20.0, self.p2.product_id)
        tree.insert(30.0, self.p4.product_id) # Root: [30], Children: Leaf1([10,20]), Leaf2
([30])

```

```
        # Delete 10.0 (from p1). Leaf1 becomes [20], still fine (1 key >= min_keys 1)
        self.assertTrue(tree.delete(10.0, self.p1.product_id))
        # Leaf1 is now [[p2] at key 20]. Parent key [30] is still valid for separation.
        # Leaf1.keys=[20], Leaf2.keys=[30]
        # Root.keys=[30]
        self.assertEqual(tree.root.children[0].keys, [20.0])
        self.assertEqual(tree.root.children[1].keys, [30.0])
        self.assertEqual(tree.root.keys, [30.0])


        # Now, delete 20.0 (from p2) in Leaf1. Leaf1 becomes empty -> underflow.
        # Leaf1 needs to borrow. Right sibling Leaf2([30]) has 1 key, can lend if min_keys allows.
        # Here, min_keys=1. If a node has > min_keys it can lend. Leaf2 has 1 key == min_keys, CANNOT lend.
        # So, it should merge.
        # _handle_leaf_node_underflow -> try borrow right (fails) -> try borrow left (no left) -> merge right
        # Merging L1 (empty) with L2([30])
        # L1 gets L2's keys/values. L1.keys=[30]
        # Parent (root [30]) loses key at index 0 (the key 30) and child at index 1 (L2)
        # Root.keys=[], Root.children=[L1] -> Root underflow
        # _handle_internal_node_underflow for root: if not keys and len children == 1, new root is child[0]
        # So, L1 ([30]) becomes the new root.
        self.assertTrue(tree.delete(20.0, self.p2.product_id))
        self.assertTrue(tree.root.is_leaf)
        self.assertEqual(tree.root.keys, [30.0])
        self.assertIn(self.p4.product_id, tree.root.values[0])

    def test_11_delete_causing_leaf_underflow_borrow_from_left_sibling(self):
        tree = BPlusTreeProducts(order=2) # min_keys = 1
        # Setup: root=[20], L(leaf)=[10,20], R(leaf)=[30] (after inserting 10,20,30 with order=2, then 5)
        # 10,20,30 -> root=[30], L=[10,20], R=[30]
        # Now insert 5: L=[5,10,20] -> split. mid=2. new_L_child=[20], old_L_child=[5,10], push_up=20
        # Root was [30]. insert 20 -> root.keys=[20,30]
        # Root children: L_L([5,10]), L_R([20]), R_orig([30])
        tree.insert(10.0, self.p1.product_id)
        tree.insert(20.0, self.p2.product_id)
        tree.insert(30.0, self.p4.product_id)
        tree.insert(5.0, self.p5.product_id)
        # Structure: root=[20,30]
        # Children: L1([5,10]), L2([20]), L3([30])

        # Delete 30 (p4) from L3. L3 becomes empty -> underflow.
        # L3 needs to borrow/merge. Left sibling is L2([20]). L2 has 1 key (== min_keys), cannot lend.
        # So L3 merges with L2. L2 takes L3's (empty) stuff.
        # Parent (root) loses key at index 1 (which is 30) and child at index 2 (L3).
        # Root.keys becomes [20]. Root.children becomes [L1, L2]. No root underflow.
        self.assertTrue(tree.delete(30.0, self.p4.product_id))
        self.assertEqual(tree.root.keys, [20.0])
        self.assertEqual(len(tree.root.children), 2)
        self.assertEqual(tree.root.children[0].keys, [5.0, 10.0]) # L1
```

```
        self.assertEqual(tree.root.children[1].keys, [20.0])    # L2 (merged with L3's
    nothingness)
        # Check leaf links
        self.assertEqual(tree.root.children[0].next_leaf, tree.root.children[1])
        self.assertEqual(tree.root.children[1].prev_leaf, tree.root.children[0])

    def test_12_delete_causing_leaf_merge_and_internal_underflow_borrow(self):
        tree = BPlusTreeProducts(order=2) # min_keys_leaf=1,
        items_to_insert = [
            (10.0, Product("p10")), (20.0, Product("p20")), (30.0, Product("p30")),
            (40.0, Product("p40")), (5.0, Product("p05")), (60.0, Product("p60"))
        ]
        for price, prod_obj in items_to_insert:
            tree.insert(price, prod_obj.product_id)


        # tree._print_tree_structure(tree.root, 4)
        self.assertTrue(tree.delete(60.0, "p60"))

        root_node = tree.root
        self.assertFalse(root_node.is_leaf)
        self.assertEqual(root_node.keys, [30.0])
        self.assertEqual(len(root_node.children), 2)

        l_internal = root_node.children[0]
        r_internal = root_node.children[1]

        self.assertFalse(l_internal.is_leaf)
        self.assertEqual(l_internal.keys, [20.0])
        self.assertEqual(len(l_internal.children), 2)

        self.assertFalse(r_internal.is_leaf)
        self.assertEqual(r_internal.keys, [40.0])
        self.assertEqual(len(r_internal.children), 2)

        # 获取叶子节点
        leaf_A = l_internal.children[0] # Keys: [5.0, 10.0]
        leaf_B = l_internal.children[1] # Keys: [20.0]
        leaf_C = r_internal.children[0] # Keys: [30.0]
        leaf_D = r_internal.children[1] # Keys: [40.0]

        self.assertTrue(leaf_A.is_leaf)
        self.assertEqual(leaf_A.keys, [5.0, 10.0])
        # ... 其他叶子节点的键和 is_leaf 检查 ...

        # 检查叶节点链表指针
        self.assertEqual(leaf_A.next_leaf, leaf_B)
        self.assertEqual(leaf_B.prev_leaf, leaf_A)

        self.assertEqual(leaf_B.next_leaf, leaf_C)
        self.assertEqual(leaf_C.prev_leaf, leaf_B)

        self.assertEqual(leaf_C.next_leaf, leaf_D)
        self.assertEqual(leaf_D.prev_leaf, leaf_C)

        self.assertIsNone(leaf_A.prev_leaf)
```

```
        self.assertIsNone(leaf_D.next_leaf)


    def test_13_delete_all_elements_empty_tree(self):
        tree = self.tree_order3
        tree.insert(10.0, self.p1.product_id)
        tree.insert(20.0, self.p2.product_id)

        self.assertTrue(tree.delete(10.0, self.p1.product_id))
        self.assertTrue(tree.delete(20.0, self.p2.product_id))

        self.assertTrue(tree.root.is_leaf)
        self.assertEqual(len(tree.root.keys), 0)
        self.assertEqual(len(tree.root.values), 0)
        self.assertEqual(len(tree.search_range(0, 100)), 0)

        # Try deleting from empty tree
        self.assertFalse(tree.delete(10.0, self.p1.product_id))

    def test_14_delete_root_becomes_leaf_after_merge(self):
        tree = BPlusTreeProducts(order=2) # min_keys=1
        # 10, 20, 30 -> Root(30), L1(10,20), L2(30)
        tree.insert(10.0, self.p1.product_id)
        tree.insert(20.0, self.p2.product_id)
        tree.insert(30.0, self.p4.product_id)

        # Delete 10 (p1). L1 becomes (20). Root(30), L1(20), L2(30). Still valid.
        tree.delete(10.0, self.p1.product_id)
        self.assertEqual(tree.root.keys, [30.0])
        self.assertEqual(tree.root.children[0].keys, [20.0])
        self.assertEqual(tree.root.children[1].keys, [30.0])


        # Delete 20 (p2). L1 becomes empty, underflow. Merges with L2(30).
        # L1 gets [30] (p4). Parent Root(30) loses key 30 and child L2.
        # Root.keys=[], Root.children=[L1]. Root underflow.
        # New root is L1. Tree becomes a single leaf node.
        tree.delete(20.0, self.p2.product_id)
        self.assertTrue(tree.root.is_leaf)
        self.assertEqual(tree.root.keys, [30.0])
        self.assertIn(self.p4.product_id, tree.root.values[0])
        self.assertIsNone(tree.root.parent)

    def test_15_large_number_of_insertions_and_deletions_order3(self):
        tree = BPlusTreeProducts(order=3)
        products = []
        num_items = 100
        for i in range(num_items):
            p_id = f"prod_lg_{i}"
            price = float(i * 10.0 + (i % 7) + 1) # Some variation
            product = Product(p_id, price=price)
            products.append(product)
            tree.insert(price, product.product_id)

        # Verify all inserted
        for prod in products:
```

```
            self.assertIn(prod.product_id, tree.search_exact(prod.price, prod.product_id))

        # Delete half of them
        for i in range(0, num_items, 2):
            prod_to_delete = products[i]
            self.assertTrue(tree.delete(prod_to_delete.price, prod_to_delete.product_id), f"
    Failed to delete {prod_to_delete}")

        # Verify deleted items are gone and others remain
        for i in range(num_items):
            prod = products[i]
            if i % 2 == 0: # Deleted
                self.assertEqual(len(tree.search_exact(prod.price, prod.product_id)), 0, f"{
    prod} should be deleted")
            else: # Kept
                self.assertIn(prod.product_id, tree.search_exact(prod.price, prod.product_id),
     f"{prod} should exist")

        # Delete the rest
        for i in range(1, num_items, 2):
            prod_to_delete = products[i]
            self.assertTrue(tree.delete(prod_to_delete.price, prod_to_delete.product_id),  f"
    Failed to delete remaining {prod_to_delete}")

        self.assertTrue(tree.root.is_leaf)
        self.assertEqual(len(tree.root.keys), 0)

    def test_16_find_leaf_node_logic(self):
        tree = self.tree_order3
        tree.insert(10.0, self.p1)
        tree.insert(20.0, self.p2)
        tree.insert(5.0, self.p5) # order 3, root=[5,10,20] is leaf

        leaf = tree._find_leaf_node(10.0)
        self.assertIn(10.0, leaf.keys)
        leaf = tree._find_leaf_node(7.0) # Should go to the leaf where 7 would be inserted
        self.assertEqual(leaf, tree.root) # Still root as it's a leaf containing 5,10,20

        # After split
        tree.insert(30.0, self.p4) # Keys: 5,10,20,30 -> overflow. mid=ceil(4/2)=2. (keys[0],
    keys[1]) | (keys[2],keys[3])
                                   # Old=[5,10], New=[20,30], push_up=20. Root(I)=[20], L(L)
    =[5,10], R(L)=[20,30]
        self.assertFalse(tree.root.is_leaf)
        leaf_for_7 = tree._find_leaf_node(7.0)
        self.assertEqual(leaf_for_7.keys, [5.0, 10.0])

        leaf_for_15 = tree._find_leaf_node(15.0) # Should go to L(L)
        self.assertEqual(leaf_for_15.keys, [5.0, 10.0]) # bisect_right for 15 in [20] gives
    index 0.
                                                        # So _find_leaf_node(15) returns the
    left child [5,10]

        leaf_for_25 = tree._find_leaf_node(25.0)
        self.assertEqual(leaf_for_25.keys, [20.0, 30.0])
```

```
        leaf_for_35 = tree._find_leaf_node(35.0)
        self.assertEqual(leaf_for_35.keys, [20.0, 30.0])


# --- 测试 BPlusTreeID ---
class TestBPlusTreeID(unittest.TestCase):
    def setUp(self):
        self.tree_order3 = BPlusTreeID(order=3)
        self.tree_order2 = BPlusTreeID(order=2)

        self.prod_obj1 = Product("prod_apple_123", price=1.0)
        self.prod_obj2 = Product("prod_banana_456", price=0.5)
        self.prod_obj3 = Product("prod_cherry_789", price=2.0)
        self.prod_obj4 = Product("prod_apricot_000", price=1.5) # To test string ordering
        self.prod_obj5 = Product("prod_blueberry_111", price=2.5)


    def test_01_initialization(self):
        self.assertIsNotNone(self.tree_order3.root)
        self.assertTrue(self.tree_order3.root.is_leaf)
        self.assertEqual(self.tree_order3.order, 3)
        with self.assertRaises(ValueError):
            BPlusTreeID(order=0)

    def test_02_insert_and_search(self):
        tree = self.tree_order3
        tree.insert(self.prod_obj1)
        tree.insert(self.prod_obj2)

        self.assertEqual(tree.search(self.prod_obj1.product_id), self.prod_obj1)
        self.assertEqual(tree.search(self.prod_obj2.product_id), self.prod_obj2)
        self.assertIsNone(tree.search("non_existent_id"))

        # Insert with non-Product type
        with self.assertRaises(TypeError):
            tree.insert("not_a_product_object")


    def test_03_leaf_split(self):
        tree = self.tree_order2 # Max 2 keys per leaf
        # tree._print_tree_structure(tree.root, 4)
        tree.insert(self.prod_obj1, test=True) # id: prod_apple_123
        # tree._print_tree_structure(tree.root, 4)
        tree.insert(self.prod_obj2, test=True) # id: prod_banana_456
        # Leaf: ["prod_apple_123", "prod_banana_456"]
        # tree._print_tree_structure(tree.root, 4)

        tree.insert(self.prod_obj4, test=True) # id: prod_apricot_000 (comes before apple)
        # tree._print_tree_structure(tree.root, 4)
        # Keys before split in leaf: ["prod_apricot_000", "prod_apple_123", "prod_banana_456"]
     - overflow
        # mid = ceil(3/2) = 2.
        # new_leaf_keys = keys[2:] = ["prod_banana_456"]
        # old_leaf_keys = keys[:2] = ["prod_apricot_000", "prod_apple_123"]
        # key_to_push_up = new_leaf_keys[0] = "prod_banana_456"

```

```
        self.assertFalse(tree.root.is_leaf)
        self.assertEqual(tree.root.keys, ["prod_banana_456"])
        left_leaf = tree.root.children[0]
        right_leaf = tree.root.children[1]

        self.assertEqual(left_leaf.keys, ["prod_apple_123", "prod_apricot_000"])
        self.assertIn(self.prod_obj4, left_leaf.values)
        self.assertIn(self.prod_obj1, left_leaf.values)

        self.assertEqual(right_leaf.keys, ["prod_banana_456"])
        self.assertIn(self.prod_obj2, right_leaf.values)


    def test_04_delete_simple_no_underflow(self):
        tree = self.tree_order3
        tree.insert(self.prod_obj1)
        tree.insert(self.prod_obj2)
        tree.insert(self.prod_obj3)
        # Root (leaf): [apple, banana, cherry]

        self.assertTrue(tree.delete(self.prod_obj2.product_id)) # Delete banana
        self.assertIsNone(tree.search(self.prod_obj2.product_id))
        self.assertEqual(tree.root.keys, [self.prod_obj1.product_id, self.prod_obj3.product_id
    ])
        self.assertEqual(tree.search(self.prod_obj1.product_id), self.prod_obj1)
        self.assertEqual(tree.search(self.prod_obj3.product_id), self.prod_obj3)

        self.assertFalse(tree.delete("non_existent_id"))

    def test_05_delete_causing_underflow_and_merge_root_becomes_leaf(self):
        tree = self.tree_order2 # min_keys = 1
        # Insert obj1, obj2, obj4 (apricot, apple, banana)
        # Root: [banana], L: [apricot, apple], R: [banana]
        tree.insert(self.prod_obj1)
        tree.insert(self.prod_obj2)
        tree.insert(self.prod_obj4)

        # Delete apple (prod_obj1) from L. L becomes [apricot]. Still valid.
        self.assertTrue(tree.delete(self.prod_obj1.product_id))
        self.assertEqual(tree.root.children[0].keys, [self.prod_obj4.product_id]) # Apricot

        # Delete apricot (prod_obj4) from L. L becomes empty -> underflow.
        # L merges with R ([banana]).
        # L gets [banana].
        # Parent root [banana] loses key "banana" and child R.
        # Root.keys=[], Root.children=[L]. Root underflow.
        # New root is L.
        # tree._print_tree_structure(tree.root, 3)
        self.assertTrue(tree.delete(self.prod_obj4.product_id))
        # tree._print_tree_structure(tree.root, 3)
        self.assertTrue(tree.root.is_leaf)
        self.assertEqual(tree.root.keys, [self.prod_obj2.product_id]) # Banana
        self.assertEqual(tree.search(self.prod_obj2.product_id), self.prod_obj2)
        self.assertIsNone(tree.root.parent)

    def test_06_delete_all_elements_empty_tree(self):
```

```
        tree = self.tree_order3
        tree.insert(self.prod_obj1)
        tree.insert(self.prod_obj2)

        self.assertTrue(tree.delete(self.prod_obj1.product_id))
        self.assertTrue(tree.delete(self.prod_obj2.product_id))

        self.assertTrue(tree.root.is_leaf)
        self.assertEqual(len(tree.root.keys), 0)
        self.assertEqual(len(tree.root.values), 0)
        self.assertIsNone(tree.search(self.prod_obj1.product_id))

    def test_07_many_insertions_and_deletions_string_keys(self):
        tree = BPlusTreeID(order=4)
        products = []
        num_items = 150 # Should cause multiple splits and merges

        for i in range(num_items):
            # Create somewhat ordered but not perfectly sequential IDs
            # Pad with zeros to ensure lexicographical sort is intuitive
            p_id = f"prod_id_{i:03d}_{chr(97 + (i % 26))}"
            product = Product(p_id, price=float(i + 1))
            products.append(product)
            tree.insert(product)

        # Verify all inserted
        for prod in products:
            self.assertEqual(tree.search(prod.product_id), prod, f"Failed to find {prod.
    product_id} after insertion batch")

        # Delete items in a shuffled order
        import random
        random.shuffle(products)

        for i in range(num_items // 2):
            prod_to_delete = products.pop()
            self.assertTrue(tree.delete(prod_to_delete.product_id), f"Failed to delete {
    prod_to_delete.product_id}")
            self.assertIsNone(tree.search(prod_to_delete.product_id), f"{prod_to_delete.
    product_id} found after deletion")

        # Verify remaining items are still there
        for prod_kept in products:
            self.assertEqual(tree.search(prod_kept.product_id), prod_kept, f"{prod_kept.
    product_id} missing after partial deletion")
        # tree._print_tree_structure(tree.root, 4)
        # Delete the rest
        while products:
            prod_to_delete = products.pop()
            self.assertTrue(tree.delete(prod_to_delete.product_id), f"Failed to delete
    remaining {prod_to_delete.product_id}")
            # tree._print_tree_structure(tree.root, 4)
        # print(f"[DEBUG] Final tree root: {tree.root}")
        # print(f"[DEBUG] Final tree root keys: {tree.root.keys}")
        # print(f"[DEBUG] Final tree root is_leaf: {tree.root.is_leaf}")
        # if not tree.root.is_leaf:
```

```
        #     print(f"[DEBUG] Final tree root children count: {len(tree.root.children)}")
        #     if tree.root.children:
        #         print(f"[DEBUG] Final tree root first child: {tree.root.children[0]}")
        self.assertTrue(tree.root.is_leaf)
        self.assertEqual(len(tree.root.keys), 0, f"Root keys not empty: {tree.root.keys}")

if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

B.3.2 test_trie.py

Listing 9: test_trie.py

```
import unittest

from src.data_structure.trie import ProductPrefixTrie


class TestProductPrefixTrie(unittest.TestCase):

    def setUp(self):
        self.trie = ProductPrefixTrie()

    def test_initialization(self):
        """测试Trie树初始化是否正确。"""
        self.assertIsNotNone(self.trie.root)
        self.assertFalse(self.trie.root.is_end_of_word)
        self.assertEqual(len(self.trie.root.children), 0)
        self.assertEqual(self.trie.root.product_ids, set())

    def test_insert_and_find_prefix_node_simple(self):
        """测试插入单个词和查找其前缀节点。"""
        self.trie.insert("apple", "P001")

        node_a = self.trie.root.children.get('a')
        self.assertIsNotNone(node_a)
        node_p1 = node_a.children.get('p')
        self.assertIsNotNone(node_p1)
        node_p2 = node_p1.children.get('p')
        self.assertIsNotNone(node_p2)
        node_l = node_p2.children.get('l')
        self.assertIsNotNone(node_l)
        node_e = node_l.children.get('e')
        self.assertIsNotNone(node_e)

        self.assertTrue(node_e.is_end_of_word)
        self.assertEqual(node_e.product_ids, {"P001"})

        # 测试 _find_prefix_node
        self.assertIs(self.trie._find_prefix_node("apple"), node_e)
        self.assertIs(self.trie._find_prefix_node("app"), node_p2)
        self.assertIsNone(self.trie._find_prefix_node("apply"))
        self.assertIsNone(self.trie._find_prefix_node("banana"))
        self.assertIs(self.trie._find_prefix_node(""), self.trie.root) # 空前缀应返回根

```

```
    def test_insert_multiple_words_shared_prefix(self):
        """测试插入多个共享前缀的词。"""
        self.trie.insert("apple", "P001")
        self.trie.insert("apply", "P002")
        self.trie.insert("ape", "P003")

        node_ap = self.trie._find_prefix_node("ap")
        self.assertIsNotNone(node_ap)

        node_apple_e = self.trie._find_prefix_node("apple")
        self.assertTrue(node_apple_e.is_end_of_word)
        self.assertEqual(node_apple_e.product_ids, {"P001"})

        node_apply_y = self.trie._find_prefix_node("apply")
        self.assertTrue(node_apply_y.is_end_of_word)
        self.assertEqual(node_apply_y.product_ids, {"P002"})

        node_ape_e = self.trie._find_prefix_node("ape")
        self.assertTrue(node_ape_e.is_end_of_word)
        self.assertEqual(node_ape_e.product_ids, {"P003"})

        # 确保 'app' 不是单词结尾，但 'apple' 的 'p' 应该是 'ape' 的 'p' 的父节点（或同一节点）
        node_app_p = self.trie._find_prefix_node("app")
        self.assertFalse(node_app_p.is_end_of_word)
        self.assertIn('l', node_app_p.children) # for apple, apply

        node_a = self.trie._find_prefix_node("a")
        self.assertIn('p', node_a.children)


    def test_insert_same_word_multiple_product_ids(self):
        """测试为同一个词插入多个product_id。"""
        self.trie.insert("apple", "P001")
        self.trie.insert("apple", "P002")

        node_apple = self.trie._find_prefix_node("apple")
        self.assertTrue(node_apple.is_end_of_word)
        self.assertEqual(node_apple.product_ids, {"P001", "P002"})

    def test_insert_empty_string_or_id(self):
        """测试插入空字符串名称或空product_id（应被忽略）。"""
        self.trie.insert("", "P001")
        self.assertEqual(len(self.trie.root.children), 0, "插入空名称不应创建子节点")

        self.trie.insert("test", "")
        node_test = self.trie._find_prefix_node("test")
        self.assertIsNone(node_test) # 不标记为节点



    def test_get_product_ids_with_prefix(self):
        """测试 get_product_ids_with_prefix 的各种情况。"""
        self.trie.insert("apple pie", "P001")
        self.trie.insert("apple", "P002")
        self.trie.insert("apple", "P007") # "apple" 关联 P002, P007
```

```
        self.trie.insert("apricot", "P003")
        self.trie.insert("banana", "P004")
        self.trie.insert("bandana", "P005")
        self.trie.insert("orange", "P006")

        self.assertEqual(self.trie.get_product_ids_with_prefix("ap"), {"P001", "P002", "P007", "P003"})
        self.assertEqual(self.trie.get_product_ids_with_prefix("apple"), {"P001", "P002", "P007"})
        self.assertEqual(self.trie.get_product_ids_with_prefix("b"), {"P004", "P005"})
        self.assertEqual(self.trie.get_product_ids_with_prefix("ban"), {"P004", "P005"})
        self.assertEqual(self.trie.get_product_ids_with_prefix("banana"), {"P004"})
        self.assertEqual(self.trie.get_product_ids_with_prefix("ora"), {"P006"})
        self.assertEqual(self.trie.get_product_ids_with_prefix("orange"), {"P006"})
        self.assertEqual(self.trie.get_product_ids_with_prefix("xyz"), set())
        self.assertEqual(self.trie.get_product_ids_with_prefix(""),
                         {"P001", "P002", "P007", "P003", "P004", "P005", "P006"},
                         "空前缀应返回所有ID")

    # --- 测试 Delete 操作 ---
    def test_delete_non_existent_word(self):
        """测试删除一个不存在于Trie中的词。"""
        self.trie.insert("apple", "P001")
        self.assertFalse(self.trie.delete("apply", "P001"))
        self.assertFalse(self.trie.delete("apple", "P002")) # ID 不匹配

    def test_delete_product_id_from_word_with_multiple_ids(self):
        """测试从一个关联多个ID的词中删除一个ID。"""
        self.trie.insert("apple", "P001")
        self.trie.insert("apple", "P002")

        self.assertTrue(self.trie.delete("apple", "P001"))
        node_apple = self.trie._find_prefix_node("apple")
        self.assertTrue(node_apple.is_end_of_word) # 仍然是单词结尾
        self.assertEqual(node_apple.product_ids, {"P002"}) # 只剩下P002
        # 节点不应被清理
        self.assertIn('e', self.trie._find_prefix_node("appl").children)

    def test_delete_last_product_id_makes_not_end_of_word(self):
        """测试删除最后一个关联ID后，节点不再是单词结尾（但可能仍有子节点）。"""
        self.trie.insert("apple", "P001")
        self.trie.insert("apple pie", "P002") # "apple" 是 "apple pie" 的前缀

        self.assertTrue(self.trie.delete("apple", "P001"))
        node_apple = self.trie._find_prefix_node("apple")
        self.assertFalse(node_apple.is_end_of_word, "apple节点不应再是单词结尾")
        self.assertEqual(node_apple.product_ids, set())
        self.assertIn(' ', node_apple.children, "apple节点应仍有指向' '的子节点 (for apple pie)")


    def test_delete_word_causes_node_cleanup_simple(self):
        """测试删除单词导致简单路径上的节点被清理。"""
        self.trie.insert("apples", "P001") # a-p-p-l-e-s
        self.trie.insert("apply", "P002")  # a-p-p-l-y

```

```
        # 删除 "apples"
        self.assertTrue(self.trie.delete("apples", "P001"))

        # 's' 节点及其到 'e' 的连接应该被删除
        node_apple_e = self.trie._find_prefix_node("appl")
        self.assertIsNotNone(node_apple_e)
        self.assertFalse(node_apple_e.is_end_of_word) # apple 不是一个词
        self.assertNotIn('s', node_apple_e.children) # 不应再有 's' 的分支

        # "apply" 应该仍然存在
        self.assertIn('y', self.trie._find_prefix_node("appl").children)
        node_apply = self.trie._find_prefix_node("apply")
        self.assertTrue(node_apply.is_end_of_word)
        self.assertEqual(node_apply.product_ids, {"P002"})


    def test_delete_word_causes_partial_cleanup_shared_prefix(self):
        """测试删除单词，但共享的前缀节点由于其他单词而不被清理。"""
        self.trie.insert("team", "P001")
        self.trie.insert("tea", "P002")

        # 删除 "team"
        self.assertTrue(self.trie.delete("team", "P001"))

        node_tea = self.trie._find_prefix_node("tea")
        self.assertIsNotNone(node_tea)
        self.assertTrue(node_tea.is_end_of_word) # "tea" 仍然是一个词
        self.assertEqual(node_tea.product_ids, {"P002"})
        self.assertNotIn('m', node_tea.children) # 不应再有 'm' 的分支

        # 检查 't' 和 'e' 节点仍然存在
        self.assertIsNotNone(self.trie._find_prefix_node("t"))
        self.assertIsNotNone(self.trie._find_prefix_node("te"))


    def test_delete_all_words_empties_trie_except_root(self):
        """测试删除所有单词后，Trie树只剩根节点。"""
        self.trie.insert("a", "P1")
        self.trie.insert("b", "P2")

        self.assertTrue(self.trie.delete("a", "P1"))
        self.assertTrue(self.trie.delete("b", "P2"))

        self.assertEqual(len(self.trie.root.children), 0)
        self.assertFalse(self.trie.root.is_end_of_word)


    def test_delete_complex_scenario_with_shared_paths_and_ids(self):
        """更复杂的删除场景。"""
        self.trie.insert("cat", "C1")
        self.trie.insert("catalog", "C2")
        self.trie.insert("catch", "C3")
        self.trie.insert("bat", "B1")
        self.trie.insert("cat", "C1_dup") # "cat" 关联 C1, C1_dup

        # 1. 删除 "cat" (C1_dup) -> "cat" 节点仍然是 is_end, product_ids={"C1"}
```

```
        self.assertTrue(self.trie.delete("cat", "C1_dup"))
        node_cat = self.trie._find_prefix_node("cat")
        self.assertTrue(node_cat.is_end_of_word)
        self.assertEqual(node_cat.product_ids, {"C1"})
        self.assertIn('a', node_cat.children) # 指向 "catalog" 的 'a'
        self.assertIn('c', node_cat.children) # 指向 "catch" 的 'c'

        # 2. 删除 "catalog" (C2) -> "alog" 分支被清理，但 "cat" 节点保留 (因为它还是 "cat" 和 "catch" 的前缀)
        self.assertTrue(self.trie.delete("catalog", "C2"))
        node_cat_after_catalog_del = self.trie._find_prefix_node("cat")
        self.assertTrue(node_cat_after_catalog_del.is_end_of_word) # "cat" 仍然是词
        self.assertEqual(node_cat_after_catalog_del.product_ids, {"C1"})
        self.assertNotIn('a', node_cat_after_catalog_del.children) # "catalog" 分支被清理
        self.assertIn('c', node_cat_after_catalog_del.children)    # "catch" 分支还在

        # 3. 删除 "cat" (C1) -> "cat" 节点 is_end=False，但因为 "catch" 仍然保留 "c"->"a"->"t" 路径
        self.assertTrue(self.trie.delete("cat", "C1"))
        node_cat_final = self.trie._find_prefix_node("cat")
        self.assertFalse(node_cat_final.is_end_of_word)
        self.assertEqual(node_cat_final.product_ids, set())
        self.assertIn('c', node_cat_final.children) # "catch" 分支还在

        # 4. 删除 "catch" (C3) -> "c"->"a"->"t"->"c"->"h" 路径被完全清理
        self.assertTrue(self.trie.delete("catch", "C3"))
        # self.assertIsNone(self.trie._find_prefix_node("c").children.get('a'),
        #                   "节点 'ca' 应该已被清理，所以 'c' 不再有孩子 'a'")
        # 或者说，'c' 节点本身可能就被清理了，如果它不是其他词的前缀
        # 在这个例子中，"c" 只作为 "cat", "catalog", "catch" 的起始，如果它们都没了，"c" 节点也应被清理
        self.assertNotIn('c', self.trie.root.children, "'c' 分支应被完全清理")

        # "bat" 应该不受影响
        node_bat = self.trie._find_prefix_node("bat")
        self.assertTrue(node_bat.is_end_of_word)
        self.assertEqual(node_bat.product_ids, {"B1"})


if __name__ == '__main__':
    unittest.main()
```

B.3.3 test_product.py

Listing 10: test_product.py

```
import unittest
from src.model.product import Product

class TestProductClass(unittest.TestCase):

    def test_product_creation_valid(self):
        """测试成功创建一个有效的 Product 对象。"""
        p = Product(product_id="P001", name="Test Laptop", price=1200.50, heat=85.0)
        self.assertEqual(p.product_id, "P001")
```

```
        self.assertEqual(p.name, "Test Laptop")
        self.assertAlmostEqual(p.price, 1200.50)
        self.assertAlmostEqual(p.heat, 85.0)
        self.assertTrue(hasattr(p, '_product_id')) # 检查内部属性是否存在 (因为用了slots)
        self.assertTrue(hasattr(p, '_name'))
        self.assertTrue(hasattr(p, '_price'))
        self.assertTrue(hasattr(p, '_heat'))


    def test_product_creation_invalid_product_id(self):
        """测试创建 Product 时使用无效的 product_id。"""
        with self.assertRaisesRegex(ValueError, "product_id必须是一个非空的字符串"):
            Product(product_id="", name="Test Name", price=10.0, heat=5.0)
        with self.assertRaisesRegex(ValueError, "product_id必须是一个非空的字符串"):
            Product(product_id="   ", name="Test Name", price=10.0, heat=5.0) # 空白字符串
        with self.assertRaisesRegex(ValueError, "product_id必须是一个非空的字符串"):
            Product(product_id=None, name="Test Name", price=10.0, heat=5.0) # None

    def test_product_creation_invalid_name(self):
        """测试创建 Product 时使用无效的 name。"""
        with self.assertRaisesRegex(ValueError, "商品名称必须是一个非空的字符串"):
            Product(product_id="P001", name="", price=10.0, heat=5.0)
        with self.assertRaisesRegex(ValueError, "商品名称必须是一个非空的字符串"):
            Product(product_id="P001", name="  ", price=10.0, heat=5.0)
        # 在你的 Product __init__ 中，name 是通过 self.name = name 赋值的，会调用setter
        # 如果 __init__ 直接用 _name = name，则这里的测试方式需要调整

    def test_product_creation_invalid_price(self):
        """测试创建 Product 时使用无效的 price。"""
        with self.assertRaisesRegex(ValueError, "商品价格必须是一个正数"):
            Product(product_id="P001", name="Test Name", price=0, heat=5.0)
        with self.assertRaisesRegex(ValueError, "商品价格必须是一个正数"):
            Product(product_id="P001", name="Test Name", price=-10.0, heat=5.0)
        with self.assertRaisesRegex(ValueError, "商品价格必须是一个正数"):
            Product(product_id="P001", name="Test Name", price="invalid", heat=5.0) # 类型错误

    def test_product_creation_invalid_heat(self):
        """测试创建 Product 时使用无效的 heat。"""
        with self.assertRaisesRegex(ValueError, "商品热度必须是一个非负数字"):
            Product(product_id="P001", name="Test Name", price=10.0, heat=-5.0)
        with self.assertRaisesRegex(ValueError, "商品热度必须是一个非负数字"):
            Product(product_id="P001", name="Test Name", price=10.0, heat="invalid") # 类型错误

    def test_product_id_is_readonly(self):
        """测试 product_id 属性是否为只读。"""
        p = Product(product_id="P001", name="Test Name", price=10.0, heat=5.0)
        with self.assertRaises(AttributeError):
            p.product_id = "P002" # 尝试修改应失败，因为没有setter

    def test_name_setter_valid(self):
        """测试 name 属性的 setter 功能 (有效值)。"""
        p = Product(product_id="P001", name="Old Name", price=10.0, heat=5.0)
        p.name = "New Name"
        self.assertEqual(p.name, "New Name")
        self.assertEqual(p._name, "New Name") # 检查内部变量是否也被更新
```

```python

    def test_name_setter_invalid(self):
        """测试 name 属性的 setter 功能 (无效值)。"""
        p = Product(product_id="P001", name="Valid Name", price=10.0, heat=5.0)
        with self.assertRaisesRegex(ValueError, "商品名称必须是一个非空的字符串"):
            p.name = ""
        with self.assertRaisesRegex(ValueError, "商品名称必须是一个非空的字符串"):
            p.name = "   "
        self.assertEqual(p.name, "Valid Name") # 确认无效设置后值未改变

    def test_price_setter_valid_and_invalid(self):
        """测试 price 属性的 setter 功能。"""
        p = Product(product_id="P001", name="Test Name", price=10.0, heat=5.0)
        p.price = 20.5
        self.assertAlmostEqual(p.price, 20.5)
        self.assertAlmostEqual(p._price, 20.5)

        with self.assertRaisesRegex(ValueError, "商品价格必须是一个正数"):
            p.price = 0
        with self.assertRaisesRegex(ValueError, "商品价格必须是一个正数"):
            p.price = -5
        self.assertAlmostEqual(p.price, 20.5) # 确认无效设置后值未改变

    def test_heat_setter_valid_and_invalid(self):
        """测试 heat 属性的 setter 功能。"""
        p = Product(product_id="P001", name="Test Name", price=10.0, heat=5.0)
        p.heat = 0 # 热度允许为0
        self.assertAlmostEqual(p.heat, 0.0)
        self.assertAlmostEqual(p._heat, 0.0)

        p.heat = 15.5
        self.assertAlmostEqual(p.heat, 15.5)

        with self.assertRaisesRegex(ValueError, "商品热度必须是一个非负数字"):
            p.heat = -10
        self.assertAlmostEqual(p.heat, 15.5) # 确认无效设置后值未改变

    def test_repr_and_str(self):
        """测试对象的字符串表示。"""
        p = Product(product_id="P001", name="Test Laptop", price=1200.99123, heat=85.505)
        expected_repr = "Product(id='P001', name='Test Laptop', price=1200.99, heat=85.50)"
        self.assertEqual(repr(p), expected_repr)
        self.assertEqual(str(p), expected_repr)

    def test_hash_and_eq(self):
            """测试基于 product_id 的哈希和相等性比较。"""
            p1a = Product(product_id="P001", name="Laptop A", price=100.0, heat=10.0)
            p1b = Product(product_id="P001", name="Laptop B", price=200.0, heat=20.0) # 相同ID
    , 不同其他属性
            p2 = Product(product_id="P002", name="Mouse", price=20.0, heat=5.0)

            self.assertEqual(p1a, p1b) # 应该相等因为 product_id 相同
            self.assertNotEqual(p1a, p2)

            self.assertEqual(hash(p1a), hash(p1b)) # 哈希值也应相同
            self.assertNotEqual(hash(p1a), hash(p2))
```

```
120
121             # 测试与不同类型的对象比较
122             self.assertNotEqual(p1a, "P001")
123             self.assertNotEqual(p1a, None)
124
125             # 测试集合操作
126             s = {p1a, p2}
127             self.assertIn(p1a, s)
128             self.assertIn(p1b, s) # 因为 p1a == p1b，所以 p1b 也被认为在集合中
129             self.assertTrue(len(s) == 2)
130
131
132 if __name__ == '__main__':
133     unittest.main()
```

C LLM 工具使用说明

本项目主要使用 Gemini 2.5 pro (preview) 和 ChatGPT 4o 辅助构建。其中，Gemini 2.5 pro (preview) 用于验证规划和设计是否合理，以及根据设计目标生成所有的单元测试代码[10]，ChatGPT 4o 用于关于 Python 语法等问题，以及项目报告在撰写的过程中 LaTeX 编译相关问题。此外，在 B+ 树的构建中，Gemini 2.5 pro (preview) 在帮忙寻找 BUG 的过程中起到了帮助。此外，`app.py` 中的演示代码也主要由 Gemini 2.5 pro (preview) 生成。

[10] 已经验证过正确性。