高级算法——三维装箱问题实验报告

2022202210046 张沛豪　　2022202210057 郭茜雅

2022 年 12 月

I 基本实现部分

一、问题概述

1.1 原问题

物流公司在流通过程中，需要将打包完毕的箱子装入到一个货车的车厢中，为了提高物流效率，需要将车厢尽量填满，显然，车厢如果能被 100%填满是最优的，但通常认为，车厢能够填满 85%，可认为装箱是比较优化的。

箱子装载问题：设车厢为长方形，其长宽高分别为 L，W，H；共有 n 个箱子，箱子也为长方形，第 i 个箱子的长宽高为 li，wi，hi（n 个箱子的体积总和是要远远大于车厢的体积），做以下假设和要求：

1. 长方形的车厢共有 8 个角，并设靠近驾驶室并位于下端的一个角的坐标为（0,0,0），车厢共 6 个面，其中长的 4 个面，以及靠近驾驶室的面是封闭的，只有一个面是开着的，用于工人搬运箱子；

2. 需要计算出每个箱子在车厢中的坐标，即每个箱子摆放后，其和车厢坐标为（0,0,0）的角相对应的角在车厢中的坐标，并计算车厢的填充率。

1.2 约束条件

1. 所有的参数为整数；

2. 静态装箱，即从 n 个箱子中选取 m 个箱子，并实现 m 个箱子在车厢中的摆放（无需考虑装箱的顺序，即不需要考虑箱子从内向外，从下向上这种在车厢中的装箱顺序）；

3. 所有的箱子全部平放，即箱子的最大面朝下摆放（即一种箱子有两种摆放方式，宽与 x 轴平行和长与 x 轴平行）；

4. 算法时间不做严格要求，只要 1 天内得出结果都可。

5. 稳定约束：每个被装载的箱子必须得到容器底部或者其它箱子的支撑。也就是说，稳定性约束禁止被装载的箱子悬空。

二、算法思想概述

先将相同的小块组成一个合成块，列出所有合成块可能的组成方式，生成一个所有可能的块列表。接着初始化剩余空间，开始时只有整个容器这个剩余空间。在迭代的过程中，从取出栈顶的剩余空间，然后在总的块列表中找出可以放置到此剩余空间的可行块，生成可行块列表。利用块选择算法从可行块列表中选择最优块放入剩余空间内。未被填满的空间则划分为新的剩余空间放入剩余空间的栈中。若无可行块，则放弃该剩余空间，并尝试将未使用

空间中可重新利用的部分并入堆栈中相应的剩余空间。当堆栈为空时迭代结束。以上就是整个基础启发式算法的过程，而其中的放置方案、块选择算法和剩余空间分割方法都与此框架无关。

三、算法具体内容

3.1 基本定义

（1）箱子 Box：在本算法中箱子只有一条边可以竖直摆放，即总是最大的面朝下。这样一种箱子会得到两种摆放方式，但我们在输入箱子数据的时仍认为是一种箱子，不同的摆放方式会在生成简单块时进行考虑。我们用 lx、ly、lz 来表示箱子三边的长度，不同数据的箱子用 type 来进行区分。Box 定义如下：

（2）剩余空间 Space，也利用三边长度 lx、ly、lz 来表示，额外在加一个参考点坐标 x，y，z。

（3）块结构 Block：Block 既可以表示简单块也可以表示复合块，它有一个 require 向量指出其包含的所有的物品的数量。按照本实验算法对块的定义，当考虑稳定性约束时，由于块中有空隙，块的顶部有一部分可能由于失去支撑而不能继续放置其他块，顶部可放置矩形记录了块的顶部可以继续放置其他块的矩形区域。这里，块顶部可放置矩形以左后上角为参考点，块结构的域 ax、ay 表示其长宽。另外，尽管本实验算法约束了复合块生成的方式，但是复合块的数目可能产生“组合爆炸”，所以我们限制了块的最大复合次数，Block 结构中的 times 域用于记录复合次数。

（4）PackingState 表示部分放置方案，包含已生成放置、剩余空间堆栈、剩余物品向量以及已装载物品总体积和最终总体积的评估值。

3.2 复合块的概念与生成

3.2.1 简单块

简单块是由同一朝向的同种类型的物品堆叠而成的,物品和物品之间没有空隙,堆叠的结果必须恰好形成一个长方体。图 3-1 展示了两个简单块的例子，其中 nx、ny、nz 表示在每个维度上的物品数。nx*ny*nz 则是简单块所需要的总物品数。

简单块的生成算法枚举所有合法的组合(nx，ny，nz)，包括一种箱子的两种摆放方式，并将其对应的简单块加入块表。其中合法组合应满足两点约束:所包含的物品数目小于对应可用物品数目，块大小应小于容器大小。

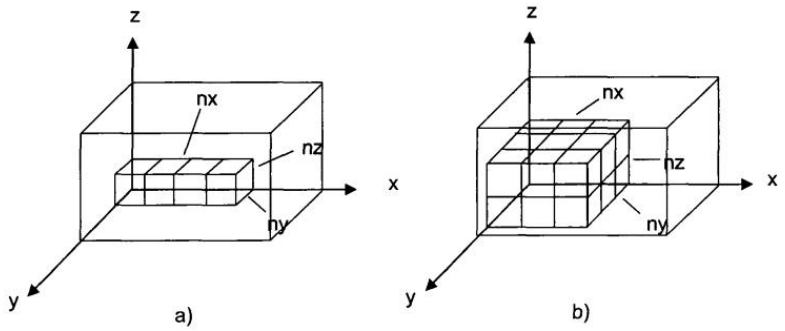


图 3-1 简单块

3.2.2 复合块

复合块是通过不断复合简单块而得到的，我们定义复合块如下：

（1）简单块是最基本的复合块。

（2）有两个复合块 a，b。可以按三种方式进行复合得到复合块 c:按 x 轴方向复合，按 y 轴方向复合，按 z 轴方向复合。c 是包含 a、b 的最小长方体。

图 3-2 展示了三种复合方式，其中的虚线描绘的是新复合块的范围。

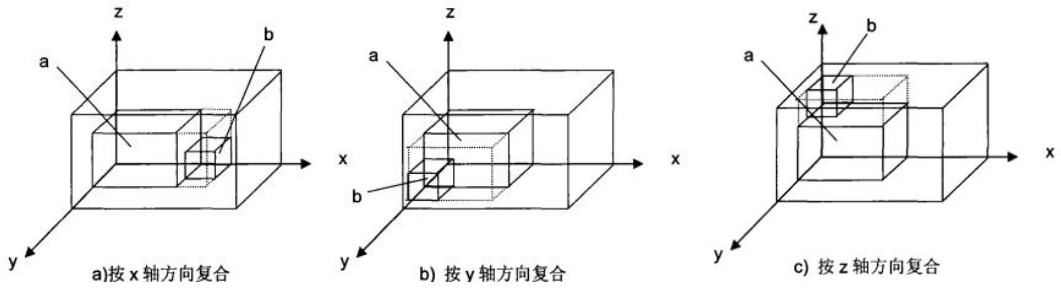


图 3-2 复合块

显然，按照前面的定义，复合块的数量将是物品数目的指数级，而且随意生成的复合块中可能有很多空隙，非常不利于装载。所以我们对复合块施加一定的限制是必要的，本实验限制复合块需要满足下面 8 个条件：

（1）复合块的大小不大于容器的大小。

（2）复合块需要的各个物品数目小于各个物品的可用数目。

（3）复合块中可以有空隙，但它的填充率至少要达到一定比例。复合块的填充率定义为复合块包含的所有物品体积/复合块体积。在复合块生成算法中，这个最低值为 MinFillRate。

（4）受复合块中空隙的影响，复合块顶部有支撑的可放置矩形可能很小，为了保证在后面的装载过程中剩余空间不会由于失去支撑浪费可用空间，我们限定顶部可放置矩形与相应的复合块顶部面积的比至少要达到 MinAreaRate。当然，在不考虑稳定约束时复合块不受

此约束限制。

（5）为控制复合块的复杂程度，定义复合块的复合次数如下：简单块的复杂次数为 0，其他复合块的复杂度为其子块的复杂次数的最大值加 1。块结构的 times 域描述了复合块的复杂程度，我们限制生成块的最大复杂次数为 MaxTimes。

（6）在考虑稳定性约束的情况下，按 x 轴方向、按 y 轴方向复合的时候，子块要保证顶部可放置矩形也能进行合并，也就是说子块要有相同的高度，并且在合并时顶部可放置矩形邻接。另外，在按 z 轴方向复合时，子块要保证复合满足上下稳定性约束，即上方的子块一定要放在下发子块的顶部可放置矩形上。

（7）拥有相同三边长度、物品需求和顶部可放置矩形的复合块被视为等价块，重复生成的等价块将被忽略。

（8）在满足以上约束的情况下，块数目仍然可能会很大，我们的生成算法将在块数目达到 MaxBlocks 时停止生成。

在考虑稳定性约束情况下，在块复合的同时，块顶部的可放置矩形也要同时进行复合。由于只有邻接的矩形才能进行复合。具体的复合条件和结果由表 3-1 描述。

表 3-1 复合的条件和结果

方向	a、b 满足的条件	复合块 c 的属性	复合块 c 应满足的条件
x 轴	a.ax = a.lx b.ax = b.lx a.lz = b.lz	c.ax := a.ax + b.ax c.ay := min(a.ay，b.ay) c.lx := a.1x + b.1x c.ly := max(a.1y，b.1y) c.lz := a. 1z	c. lx ≤ container.1x c.ly ≤ container.ly c. lz ≤ container. 1z c.require ≤ num c. volume / (c.1x * c.ly * c.1z) ≥MinFil1Rate (c.ax * c.ay) l (c.lx * c.ly)≥ MinAreaRate c.times ≤ MaxTimes
y 轴	a.ay = a.ly b.ay = b.ly a.lz = b.lz	c.ax =min(a. ax，b.ax) c.ay :=a.ay + b.ay c.lx := max(a.1x，b.1x) c.ly := a. ly + b.1y c.1z :=a.1z	
z 轴	a.ax ≥ b.lx a.ay ≥ b.1y	c.ax :=b.ax c.ay :=b.ay c.lx := a.1x c.ly := a.1y c.1z := a.1z + b.1z	

3.3 可行块列表生成

可行块列表生成算法 GenBlockList(space，avail)用于从 blockTable 中获取适合当前剩余

空间的可行块列表。其中，blockTable 是预先生成的所有可能的块的列表。这种分离式的设计使得基础启发式算法与块生成算法完全无关，块生成算法可以根据需要进行定制而不影响基础启发式算法。该算法扫描 blockTable，返回所有能放入剩余空间并且有足够剩余物品的块。由于 blockTable 是按块中物品总体积降序排列的，返回的可行块列表 blockList 也是按物品总体积降序排列的。

3.4 剩余空间切割与转移

在每个装载阶段一个剩余空间被装载，装载分为两种情况：有可行块，无可行块。在有可行块时，算法按照块选择算法选择可行块，然后将未填充空间切割成新的剩余空间。在无可行块时，当前剩余空间被抛弃，若其中的一部分空间可以被并入当前堆栈中的其他空间，则进行空间转移重新利用这些空间。

图 3-3 显示了在考虑稳定性约束剩余空间与块结合成为放置以后的状态。未填充空间将被按照不同情况，沿着块的三个面被分成三个剩余空间。需要注意的是，在考虑稳定性约束时，由于要保证所有剩余空间受到足够的支撑，z 轴上的新剩余空间在切割的时候必须沿着所选块的顶部可放置矩形进行。因此,块顶部的可放置矩形确定了一个新的剩余空间 spaceZ，其余的两个剩余空间为 x 轴方向上的 spaceX 和 y 轴方向上的 spaceY，所以只有两种切割方法。

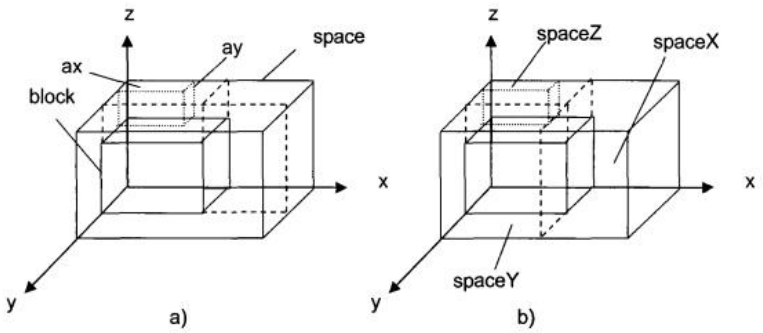


图 3-3 剩余空间切割与转移

如图 3-4 所示，各种切割方式本质上的不同就在于可转移空间的归属。我们希望在切割过程中尽量保证空间完整性，而衡量空间完整性的方法有很多，本实验选择的策略是另切割出的剩余空间尽可能的大。这里大小的判定以剩余空间在放置块以后在 x 轴、y 轴和 z 轴上的剩余长度 mx、my、mz 作为度量，将可转移空间分给剩余量较大的方向上的新空间。算法中 GenResidualSpace(space, block)执行未填充空间的切割，其返回的剩余空间 spaceX、spaceY、spaceZ 按照 mx、my、mz 的从小到大排列，并确保最后入栈的是包含可转移空间的剩余空间。表 3-2 列出了在考虑稳定性的情况下，剩余空间的切割方式以及三个空间的入

栈顺序。

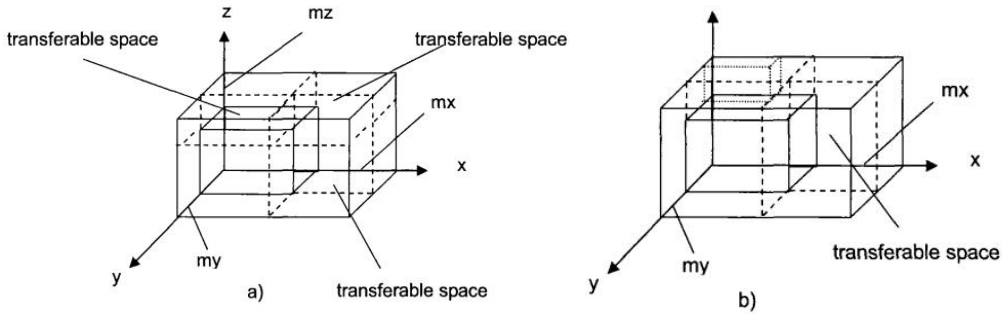


图 3-4 可转移空间

由于切割算法保证了包含可转移空间的剩余空间后入栈，所以其必然被先装载，若在装载过程中可行块列表为空，栈顶空间中的可转移空间可以被转移给剩余空间堆栈中来自同一次切割的其他空间以重新利用。观察图 3-4，我们发现可转移空间的重新分配实际上就是对未填充空间的重新切割。因此，我们可以通过重新切割未填充空间来达到再次利用可转移空间的目的，算法中的 TransferSpace(space，spaceStack)就是执行这样的任务，此过程判定当前剩余空间与栈顶的一个或两个剩余空间是否是由同一次切割而产生的,若是则将可转移空间转给相应的一个或两个剩余空间。具体的转移条件和结果在表 3-2 和表中描述。

表 3-2 剩余空间切割方式

切割条件	切割方法	入栈顺序	转移条件	转移结果
my ≥ mx	图 3-4a	spaceZ，spaceX，spaceY	spaceY 无可行块	图 3-4b
mx ≥ my	图 3-4b	spaceZ，spaceY，spaceX	spaceX 无可行块	图 3-4a

3.5 块选择算法

3.5.1 整体流程

本实验中的算法遍历整个可行块列表，尝试放置当前块到当前部分放置方案，然后用某种方式评估此状态，并将此评估值作为被选块的适应度，最终选取适应度最高的块作为结果。整个流程如图 3-5 所示。而具体的状态评估算法将在后几节详细介绍。

3.5.2 块放置和块移除算法

算法中块放置算法完成的工作包括将块和栈顶空间结合成一个放置加入当前放置方案，移除栈顶空间，扣除已使用物品，然后切割未填充空间并加入剩余空间堆栈。块移除算法完成的工作包括从当前部分放置方案中移除当前块所属的放置，恢复已使用物品，移除空间堆栈栈顶的三个切割出来的剩余空间，并将已使用剩余空间重新插入栈顶。

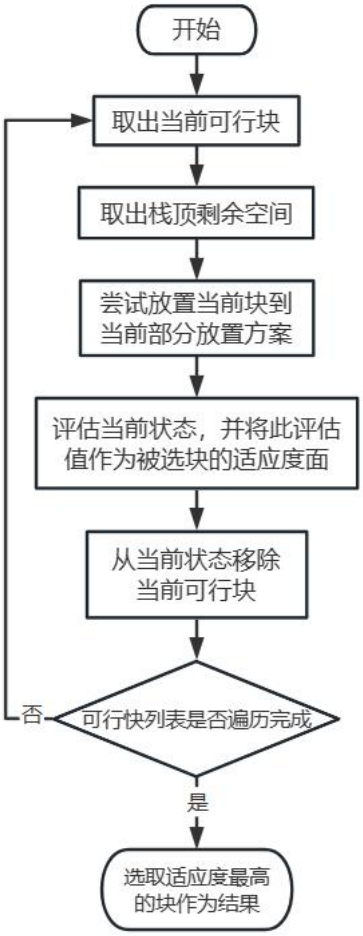


图 3-5 块选择算法流程

3.5.3 补全算法

除了块放置和块移除算法，另一个极其重要的算法是部分放置方案补全算法。我们知道，评估当前的部分放置方案好坏的最直接的方法是用某种方式补全它，并以最终结果的填充率作为当前状态的评估值。该算法实际上是整体基本启发式算法的一个简化版本，区别在于每个装载阶段算法都选择可行块列表中体积最大的块进行放置。由于可行块列表已经按照体积降序排列，实际上算法选择的块总是列表的第一个元素。算法不改变输入的部分放置方案，只是把最终补全的结果记录在此状态的 volume_complete 域作为该状态的评估值。

3.5.4 贪心算法

在块的选择算法过程中，也可以采用贪心算法，直接返回填充体积最大的块，由于可行块列表已经按照体积降序排列，实际上算法选择的块总是列表的第一个元素。

3.5.5 带深度限制的深度优先搜索算法

除了贪心算法和简单的补全算法，另一个很容易想到的方法是进行深度或广度优先搜索扩展当前放置方案，然后在叶子节点使用补全算法来评估叶节点的好坏，最终以搜索树中最

好的一个叶子节点的评估值作为当前状态的评估。但是，由于搜索过程中，每一个节点都有大量分支，采用宽度优先搜索需要海量的空间，因此是不现实的。实际应用中，一般采用的是带深度限制的深度优先搜索算法，通过深度来限制最多放置的块的数目。另外，由于每个阶段可行块列表包含大量的块，算法往往也限制每个节点的最大分支数。本文采用深度优先搜索算法扩展当前放置方案，算法的输入为一个部分放置方案，深度限制和最大分支数。该算法从一个部分放置方案出发，递归的尝试可行块列表中的块，在到达深度限制的时候调用补全函数得到当前方案的评估值，并记录整个搜索过程找到的最优的评估值作为输入部分放置方案的评估。

这里特别注意的是搜索深度代表通过深度优先搜索放置的块的最大个数。

如图 3-6 所示，节点生成两个子节点，直到节点深度为 3 时使用补全算法取得当前状态的评估值，最后采用整个算法中最好的结果作为算法的最终结果。图中，白色节点表示搜索时遍历的内部节点，黑色节点表示用补全算法计算出的节点，而灰色表示最终选择的最优节点。

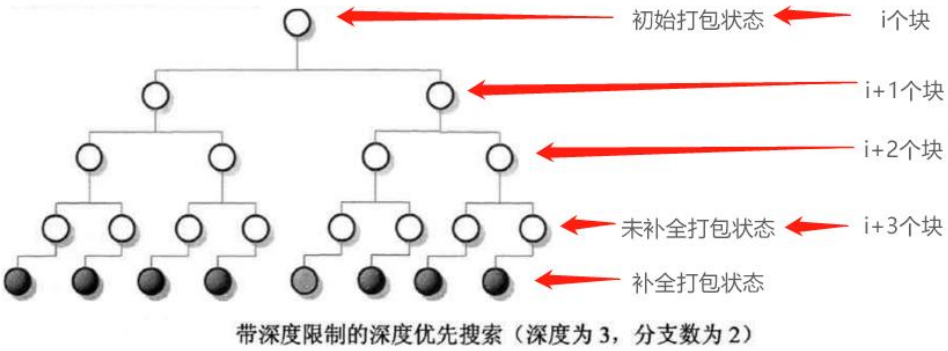


图 3-6 深度搜索算法

四、算法实现

4.1 基本数据结构

（1）箱子类

```
class Box:
    def __init__(self, lx, ly, lz, type=0):
        self.lx = lx    # 长
        self.ly = ly    # 宽
        self.lz = lz    # 高
        self.type = type    # 类型
    def __str__(self):
        return "lx: {}, ly: {}, lz: {}, type: {}".format(self.lx, self.ly, self.lz, self.type)
```

（2）剩余空间类

```python
class Space:
    def __init__(self, x, y, z, lx, ly, lz, origin=None):
        self.x = x    # 坐标
        self.y = y
        self.z = z
        self.lx = lx    # 长
        self.ly = ly    # 宽
        self.lz = lz    # 高
        self.origin = origin    # 表示从哪个剩余空间切割而来
    def __str__(self):
        return "x:{},y:{},z:{},lx:{},ly:{},lz:{}".format(self.x, self.y, self.z, self.lx, self.ly, self.lz)
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y and self.z == other.z and self.lx == other.lx and self.ly == other.ly a
```

（3）装箱问题类

```python
class Problem:
    def __init__(self, container: Space, box_list=[], num_list=[]):
        self.container = container    # 容器
        self.box_list = box_list    # 箱子列表
        self.num_list = num_list    # 箱子对应的数量
```

（4）块类

```python
class Block:
    def __init__(self, lx, ly, lz, require_list=[], children=[], direction=None):
        self.lx = lx    # 长
        self.ly = ly    # 宽
        self.lz = lz    # 高
        self.require_list = require_list    # 需要的物品数量
        self.volume = 0    # 体积
        self.children = children    # 子块列表，简单块的子块列表为空
        self.direction = direction    # 复合块子块的合并方向
        self.ax = 0    # 顶部可放置矩形尺寸
        self.ay = 0
        self.times = 0    # 复杂度，复合次数
        self.fitness = 0    # 适应度，块选择时使用
    def __str__(self):
        return "lx: %s, ly: %s, lz: %s, volume: %s, ax: %s, ay: %s, times:%s, fitness: %s, require: %s, children: " \
               "%s, direction: %s" % (self.lx, self.ly, self.lz, self.volume, self.ax, self.ay, self.times, self.fitness, self.require_list, se
    def __eq__(self, other):
        return self.lx == other.lx and self.ly == other.ly and self.lz == other.lz and self.ax == other.ax and self.ay == self.ay and (np.array(
```

（5）放置类

```python
class Place:
    def __init__(self, space: Space, block: Block):
        self.space = space    # 空间
        self.block = block    # 块
```

（6）装箱状态类

```python
class PackingState:
    def __init__(self, plan_list=[], space_stack: Stack = Stack(), avail_list=[]):
        self.plan_list = plan_list    # 已生成的装箱方案列表
        self.space_stack = space_stack    # 剩余空间堆栈
        self.avail_list = avail_list    # 剩余可用箱体数量
        self.volume = 0    # 已装载物品总体积
        self.volume_complete = 0    # 最终装载物品的总体积的评估值
```

4.2 算法基础模块

（1）合并块时通用校验项目

```
def combine_common_check(combine: Block, container: Space, num_list):
    if combine.lx > container.lx:    # 合共块尺寸不得大于容器尺寸
        return False
    if combine.ly > container.ly:
        return False
    if combine.lz > container.lz:
        return False
    if (np.array(combine.require_list) > np.array(num_list)).any():    # 合共块需要的箱子数量不得大于箱子总的数量
        return False
    if combine.volume / (combine.lx * combine.ly * combine.lz) < MIN_FILL_RATE:    # 合并块的填充体积不得小于最小填充率
        return False
    if (combine.ax * combine.ay) / (combine.lx * combine.ly) < MIN_AREA_RATE:    # 合并块的顶部可放置矩形必须足够大
        return False
    if combine.times > MAX_TIMES:    # 合并块的复杂度不得超过最大复杂度
        return False
    return True
```

（2）合并块时通用合并项目

```
def combine_common(a: Block, b: Block, combine: Block):
    combine.require_list = (np.array(a.require_list) + np.array(b.require_list)).tolist()    # 合并块的需求箱子数量
    combine.volume = a.volume + b.volume    # 合并填充体积
    combine.children = [a, b]    # 构建父子关系
    combine.times = max(a.times, b.times) + 1    # 合并后的复杂度
```

（3）生成简单块

```
def gen_simple_block(container, box_list, num_list):
    block_table = []
    for box in box_list:
        for nx in np.arange(num_list[box.type]) + 1:
            for ny in np.arange(num_list[box.type] / nx) + 1:
                for nz in np.arange(num_list[box.type] / nx / ny) + 1:
                    if box.lx * nx <= container.lx and box.ly * ny <= container.ly and box.lz * nz <= container.lz:
                        # xy可以颠倒 直接添加两个block
                        # 该简单块需要的立体箱子数量
                        requires = np.full_like(num_list, 0)
                        requires[box.type] = nx * ny * nz
                        # 简单块
                        block1 = Block(box.lx * nx, box.ly * ny, box.lz * nz, requires)
                        block2 = Block(box.ly * nx, box.lx * ny, box.lz * nz, requires)
```

```
                        # 顶部可放置矩形
                        block1.ax = box.lx * nx
                        block1.ay = box.ly * ny
                        block2.ax = box.ly * nx
                        block2.ay = box.lx * ny
                        # 简单块填充体积
                        block1.volume = box.lx * nx * box.ly * ny * box.lz * nz
                        block2.volume = box.lx * nx * box.ly * ny * box.lz * nz
                        # 简单块复杂度
                        block1.times = 0
                        block2.times = 0
                        block_table.append(block1)
                        block_table.append(block2)
    return sorted(block_table, key=lambda x: x.volume, reverse=True)
```

（4）生成复合块

```
def gen_complex_block(container, box_list, num_list):
    block_table = gen_simple_block(container, box_list, num_list)    # 先生成简单块
    for times in range(MAX_TIMES):
        new_block_table = []
        for i in np.arange(0, len(block_table)):    # 循环所有简单块，两两配对
            a = block_table[i]    # 第一个简单块
            for j in np.arange(0, len(block_table)):
                if j == i:    # 简单块不跟自己复合
                    continue
                b = block_table[j]    # 第二个简单块
                if a.times == times or b.times == times:    # 复杂度满足当前复杂度
                    c = Block(0, 0, 0)
                    if a.ax == a.lx and b.ax == b.lx and a.lz == b.lz:    # 按x轴方向复合
                        c.direction = "x"
                        c.ax = a.ax + b.ax
                        c.ay = min(a.ay, b.ay)
                        c.lx = a.lx + b.lx
                        c.ly = max(a.ly, b.ly)
                        c.lz = a.lz
                        combine_common(a, b, c)
                        if combine_common_check(c, container, num_list):
                            new_block_table.append(c)
                            continue
```

```
                    if a.ay == a.ly and b.ay == b.ly and a.lz == b.lz:    # 按y轴方向复合
                        c.direction = "y"
                        c.ax = min(a.ax, b.ax)
                        c.ay = a.ay + b.ay
                        c.lx = max(a.lx, b.lx)
                        c.ly = a.ly + b.ly
                        c.lz = a.lz
                        combine_common(a, b, c)
                        if combine_common_check(c, container, num_list):
                            new_block_table.append(c)
                            continue
                    if a.ax >= b.lx and a.ay >= b.ly:    # 按z轴方向复合
                        c.direction = "z"
                        c.ax = b.ax
                        c.ay = b.ay
                        c.lx = a.lx
                        c.ly = a.ly
                        c.lz = a.lz + b.lz
                        combine_common(a, b, c)
                        if combine_common_check(c, container, num_list):
                            new_block_table.append(c)
                            continue
```

```
        block_table = block_table + new_block_table    # 加入新生成的复合块
        # 去重，拥有相同三边长度、物品需求和顶部可放置矩形的复合块被视为等价块，重复生成的等价块将被忽略
        block_table = list(set(block_table))
    return sorted(block_table, key=lambda x: x.volume, reverse=True)    # 按填充体积对复合块进行排序
```

（5）生成可行块列表

```
def gen_block_list(space: Space, avail, block_table):
    block_list = []
    for block in block_table:
        # 块中需要的箱子需求数量必须小于当前待装箱的箱子数量，块的尺寸必须小于放置空间尺寸
        if (np.array(block.require_list) <= np.array(avail)).all() and \
                block.lx <= space.lx and block.ly <= space.ly and block.lz <= space.lz:
            block_list.append(block)
    return block_list
```

（6）剩余空间切割

```
def gen_residual_space(space: Space, block: Block, box_list=[]):
    rmx = space.lx - block.lx    # 三个维度的剩余尺寸
    rmy = space.ly - block.ly
    rmz = space.lz - block.lz
    # 三个新裁切出的剩余空间（按入栈顺序依次返回）
    if rmx >= rmy:     # 可转移空间归属于x轴切割空间
        drs_x = Space(space.x + block.lx, space.y, space.z, rmx, space.ly, space.lz, space)
        drs_y = Space(space.x, space.y + block.ly, space.z, block.lx, rmy, space.lz, space)
        drs_z = Space(space.x, space.y, space.z + block.lz, block.ax, block.ay, rmz, None)
        return drs_z, drs_y, drs_x
    else:    # 可转移空间归属于y轴切割空间
        drs_x = Space(space.x + block.lx, space.y, space.z, rmx, block.ly, space.lz, space)
        drs_y = Space(space.x, space.y + block.ly, space.z, space.lx, rmy, space.lz, space)
        drs_z = Space(space.x, space.y, space.z + block.lz, block.ax, block.ay, rmz, None)
        return drs_z, drs_x, drs_y
```

（7）空间转移

```
def transfer_space(space: Space, space_stack: Stack):
    if space_stack.size() <= 1:    # 仅剩一个空间的话，直接弹出
        space_stack.pop()
        return None
    discard = space    # 待转移空间的原始空间
    space_stack.pop()    # 目标空间
    target = space_stack.top()
    # 将可转移的空间转移给目标空间
    if discard.origin is not None and target.origin is not None and discard.origin == target.origin:
        new_target = copy.deepcopy(target)
        if discard.lx == discard.origin.lx:       # 可转移空间原先归属于y轴切割空间的情况
            new_target.ly = discard.origin.ly
        elif discard.ly == discard.origin.ly:      # 可转移空间原来归属于x轴切割空间的情况
            new_target.lx = discard.origin.lx
        else:
            return None
        space_stack.pop()
        space_stack.push(new_target)
        return target    # 返回未发生转移之前的目标空间
    return None
```

（8）还原空间转移

```
def transfer_space_back(space: Space, space_stack: Stack, revert_space: Space):
    space_stack.pop()
    space_stack.push(revert_space)
    space_stack.push(space)
```

4.3 块放置算法

（1）块放置算法

```
def place_block(ps: PackingState, block: Block):
    space = ps.space_stack.pop()    # 栈顶剩余空间
    ps.avail_list = (np.array(ps.avail_list) - np.array(block.require_list)).tolist()    # 更新可用箱体数目
    place = Place(space, block)    # 更新放置计划
    ps.plan_list.append(place)
    ps.volume = ps.volume + block.volume    # 更新体积利用率
    cuboid1, cuboid2, cuboid3 = gen_residual_space(space, block)    # 压入新的剩余空间
    ps.space_stack.push(cuboid1, cuboid2, cuboid3)
    return place    # 返回临时生成的放置
```

（2）块移除算法

```
def remove_block(ps: PackingState, block: Block, place: Place, space: Space):
    ps.avail_list = (np.array(ps.avail_list) + np.array(block.require_list)).tolist()    # 还原可用箱体数目
    ps.plan_list.remove(place)    # 还原排样计划
    ps.volume = ps.volume - block.volume    # 还原体积利用率
    for _ in range(3):    # 移除在此之前裁切出的新空间
        ps.space_stack.pop()
    ps.space_stack.push(space)    # 还原之前的空间
```

（3）补全放置方案

```
def complete(ps: PackingState, block_table):
    tmp = copy.deepcopy(ps)    # 不对当前的放置状态进行修改
    while tmp.space_stack.not_empty():
        space = tmp.space_stack.top()        # 栈顶空间
        block_list = gen_block_list(space, ps.avail_list, block_table)        # 可用块列表
        if len(block_list) > 0:
            place_block(tmp, block_list[0])    # 放置块
        else:
            transfer_space(space, tmp.space_stack)    # 空间转移
    ps.volume_complete = tmp.volume    # 补全后的使用体积
```

（4）带深度限制的深度优先搜索算法

```
def depth_first_search(ps: PackingState, depth, branch, block_table):
    global tmp_best_ps
    if depth != 0:
        space = ps.space_stack.top()
        block_list = gen_block_list(space, ps.avail_list, block_table)
        if len(block_list) > 0:
            for i in range(min(branch, len(block_list))):    # 遍历所有分支
                place = place_block(ps, block_list[i])    # 放置块
                depth_first_search(ps, depth - 1, branch, block_table)    # 放置下一个块
                remove_block(ps, block_list[i], place, space)    # 移除刚才添加的块
        else:
            old_target = transfer_space(space, ps.space_stack)    # 转移空间
            if old_target:
                depth_first_search(ps, depth, branch, block_table)    # 放置下一个块
                transfer_space_back(space, ps.space_stack, old_target)    # 还原转移空间
    else:
        complete(ps, block_table)    # 补全该方案
        if ps.volume_complete > tmp_best_ps.volume_complete:    # 更新最优解
            tmp_best_ps = copy.deepcopy(ps)
```

（5）评价某个块

```
def estimate(ps: PackingState, block_table, search_params):
    global tmp_best_ps    # 空的放置方案
    # tmp_best_ps = PackingState()
    tmp_best_ps = PackingState([], Stack(), [])
    depth_first_search(ps, MAX_DEPTH, MAX_BRANCH, block_table)    # 开始深度优先搜索
    return tmp_best_ps.volume_complete
```

（6）查找下一个可行块

```
def find_next_block(ps: PackingState, block_list, block_table, search_params):
    return block_list[0]    #也可以采用贪心算法，直接返回填充体积最大的块
    best_fitness = 0    # 最优适应度
    best_block = block_list[0]    # 初始化最优块为第一个块（填充体积最大的块）
    for block in block_list:    # 遍历所有可行块
        space = ps.space_stack.top()    # 栈顶空间
        place = place_block(ps, block)    # 放置块
        fitness = estimate(ps, block_table, search_params)    # 评价值
        remove_block(ps, block, place, space)    # 移除刚才添加的块
        if fitness > best_fitness:    # 更新最优解
            best_fitness = fitness
            best_block = block
    return best_block
```

（7）基本启发式算法

```
def basic_heuristic(is_complex, search_params, problem: Problem):
    st = time.time()
    if is_complex:
        block_table = gen_complex_block(problem.container, problem.box_list, problem.num_list)    # 生成复合块
    else:
        block_table = gen_simple_block(problem.container, problem.box_list, problem.num_list)    # 生成简单块
    ps = PackingState(avail_list=problem.num_list)    # 初始化排样状态
    ps.space_stack.push(problem.container)    # 开始时，剩余空间堆栈中只有容器本身
    while ps.space_stack.size() > 0:    # 所有剩余空间均转满，则停止
        space = ps.space_stack.top()
        block_list = gen_block_list(space, ps.avail_list, block_table)
        if block_list:
            block = find_next_block(ps, block_list, block_table, search_params)    # 查找下一个近似最优块
            ps.space_stack.pop()    # 弹出顶部剩余空间
            ps.avail_list = (np.array(ps.avail_list) - np.array(block.require_list)).tolist()    # 更新可用物品数量
            ps.plan_list.append(Place(space, block))    # 更新排样计划
            ps.volume = ps.volume + block.volume    # 更新已利用体积
            cuboid1, cuboid2, cuboid3 = gen_residual_space(space, block)    # 压入新裁切的剩余空间
            ps.space_stack.push(cuboid1, cuboid2, cuboid3)
        else:
            transfer_space(space, ps.space_stack)    # 转移剩余空间
```

4.4 结果输出

（1）递归构建箱体坐标，用于绘图

```
def build_box_position(block, init_pos, box_list):
    if len(block.children) <= 0 and block.times == 0:    # 遇到简单块时进行坐标计算
        box_idx = (np.array(block.require_list) > 0).tolist().index(True)    # 箱体类型索引
        if box_idx > -1:
            box = box_list[box_idx]    # 所需箱体
            nx = block.lx / box.lx    # 箱体的相对坐标
            ny = block.ly / box.ly
            nz = block.lz / box.lz
            x_list = (np.arange(0, nx) * box.lx).tolist()
            y_list = (np.arange(0, ny) * box.ly).tolist()
            z_list = (np.arange(0, nz) * box.lz).tolist()    # dimensions是箱体的绝对坐标
            dimensions = (np.array([x for x in product(x_list, y_list, z_list)]) + np.array([init_pos[0], init_pos[1], init_pos[2]])).tolist()
            return sorted([d + [box.lx, box.ly, box.lz] for d in dimensions], key=lambda x: (x[0], x[1], x[2]))
        return []
```

```
    pos = []
    for child in block.children:
        pos += build_box_position(child, (init_pos[0], init_pos[1], init_pos[2]), box_list)
        if block.direction == "x":    # 根据子块的复合方向，确定下一个子块的左后下角坐标
            init_pos = (init_pos[0] + child.lx, init_pos[1], init_pos[2])
        elif block.direction == "y":
            init_pos = (init_pos[0], init_pos[1] + child.ly, init_pos[2])
        elif block.direction == "z":
            init_pos = (init_pos[0], init_pos[1], init_pos[2] + child.lz)
    return pos
```

（2）绘制立方体边框

```
def plot_linear_cube(ax, x, y, z, dx, dy, dz, color='red', linestyle=None):
    xx = [x, x, x+dx, x+dx, x]
    yy = [y, y+dy, y+dy, y, y]
    kwargs = {"alpha": 1, "color": color, "linewidth": 2.5, "zorder": 2}
    if linestyle:
        kwargs["linestyle"] = linestyle
    ax.plot3D(xx, yy, [z]*5, **kwargs)
    ax.plot3D(xx, yy, [z+dz]*5, **kwargs)
    ax.plot3D([x, x], [y, y], [z, z+dz], **kwargs)
    ax.plot3D([x, x], [y+dy, y+dy], [z, z+dz], **kwargs)
    ax.plot3D([x+dx, x+dx], [y+dy, y+dy], [z, z+dz], **kwargs)
    ax.plot3D([x+dx, x+dx], [y, y], [z, z+dz], **kwargs)
def cuboid_data2(o, size=(1, 1, 1)):# 立方体
    X = [[[0, 1, 0], [0, 0, 0], [1, 0, 0], [1, 1, 0]],
         [[0, 0, 0], [0, 0, 1], [1, 0, 1], [1, 0, 0]],
         [[1, 0, 1], [1, 0, 0], [1, 1, 0], [1, 1, 1]],
         [[0, 0, 1], [0, 0, 0], [0, 1, 0], [0, 1, 1]],
         [[0, 1, 0], [0, 1, 1], [1, 1, 1], [1, 1, 0]],
         [[0, 1, 1], [0, 0, 1], [1, 0, 1], [1, 1, 1]]]
    X = np.array(X).astype(float)
    for i in range(3):
        X[:, :, i] *= size[i]
    X += np.array(o)
    return X
```

（3）绘制立方体

```
def plotCubeAt2(positions, sizes=None, colors=None, **kwargs):
    if not isinstance(colors, (list, np.ndarray)):
        colors = ["C0"] * len(positions)
    if not isinstance(sizes, (list, np.ndarray)):
        sizes = [(1, 1, 1)] * len(positions)
    g = []
    for p, s, c in zip(positions, sizes, colors):
        g.append(cuboid_data2(p, size=s))
    return Poly3DCollection(np.concatenate(g), facecolors=np.repeat(colors, 6), **kwargs)
```

（4）绘制排样结果

```
def draw_packing_result(problem: Problem, ps: PackingState):
    fig = plt.figure()    # 绘制结果
    ax1 = mplot3d.Axes3D(fig, auto_add_to_figure=False)
    fig.add_axes(ax1)
    plot_linear_cube(ax1, 0, 0, 0, problem.container.lx, problem.container.ly, problem.container.lz)    # 绘制容器
    for p in ps.plan_list:
        box_pos = build_box_position(p.block, (p.space.x, p.space.y, p.space.z), problem.box_list)    # 箱子位置及尺寸
        positions = []
        sizes = []
        colors = ["yellow"] * len(box_pos)    # 箱子颜色
        for bp in sorted(box_pos, key=lambda x: (x[0], x[1], x[2])):
            positions.append((bp[0], bp[1], bp[2]))
            sizes.append((bp[3], bp[4], bp[5]))
        pc = plotCubeAt2(positions, sizes, colors=colors, edgecolor="k")
        ax1.add_collection3d(pc)
    plt.title('PackingResult')
    plt.show()
    # plt.savefig('3d_multilayer_search.png', dpi=800)
```

（5）打印结果

```
    et = time.time()
    cost_time = et - st
    all_volume = problem.container.lx * problem.container.ly * problem.container.lz
    #print("----------------E1----------------")
    print("--选取装填的货物：",end ="")
    for i in range(len(ps.avail_list)):
        print("第{}个箱子装载{}件；".format(i+1,problem.num_list[i] - ps.avail_list[i]),end ="")
    print("\n--运行时间：{:.4f}s".format(cost_time))
    print("--装填率：{:.4f}%".format(ps.volume / all_volume * 100))
    print("\n")
    '''
```

4.5 数据形式

```
def main():
    # 容器
    container = Space(0, 0, 0, 587, 233, 220)
    box_list = [Box(91, 54, 45, 0), Box(105, 77, 72, 1), Box(79, 78, 48, 2)]
    num_list = [32, 24, 30]
    '''
    all_box_list = [[Box(108, 76, 30, 0), Box(110, 43, 25, 1), Box(92, 81, 55, 2)],
                    [Box(91, 54, 45, 0), Box(105, 77, 72, 1), Box(79, 78, 48, 2)],
                    [Box(91, 54, 45, 0), Box(105, 77, 72, 1), Box(79, 78, 48, 2)],
                    [Box(60, 40, 32, 0), Box(98, 75, 55, 1), Box(60, 59, 39, 2)],
                    [Box(78, 37, 27, 0), Box(89, 70, 25, 1), Box(90, 84, 41, 2)],

                    [Box(108, 76, 30, 0), Box(110, 43, 25, 1), Box(92, 81, 55, 2), Box(81, 33, 28, 3), Box(120, 99, 73, 4)],
                    [Box(49, 25, 21, 0), Box(60, 51, 41, 1), Box(103, 76, 64, 2), Box(95, 70, 62, 3), Box(111, 49, 26, 4)],
                    [Box(88, 54, 39, 0), Box(94, 54, 36, 1), Box(87, 77, 43, 2), Box(100, 80, 72, 3), Box(83, 40, 36, 4)],
                    [Box(90, 70, 63, 0), Box(84, 78, 28, 1), Box(94, 85, 39, 2), Box(80, 76, 54, 3), Box(69, 50, 45, 4)],
                    [Box(74, 63, 61, 0), Box(71, 60, 25, 1), Box(106, 80, 59, 2), Box(109, 76, 42, 3), Box(118, 56, 22, 4)]]

                    [Box(108, 76, 30, 0), Box(110, 43, 25, 1), Box(92, 81, 55, 2), Box(81, 33, 28, 3), Box(120, 99, 73, 4), Box(111, 70, 48, 3)],
                    [Box(97, 81, 27, 0), Box(102, 78, 39, 1), Box(113, 46, 36, 2), Box(66, 50, 42, 3), Box(101, 30, 26, 4), Box(100, 56, 35, 3)],
                    [Box(88, 54, 39, 0), Box(94, 54, 36, 1), Box(87, 77, 43, 2), Box(100, 80, 72, 3), Box(83, 40, 36, 4), Box(91, 54, 22, 5)],
                    [Box(49, 25, 21, 0), Box(60, 51, 41, 1), Box(103, 76, 64, 2), Box(95, 70, 62, 3), Box(111, 49, 26, 4), Box(85, 84, 72, 5)],
                    [Box(113, 92, 33, 0), Box(52, 37, 28, 1), Box(57, 33, 29, 2), Box(99, 37, 30, 3), Box(92, 64, 33, 4), Box(119, 59, 39, 5)]
```

五、实验结果及分析

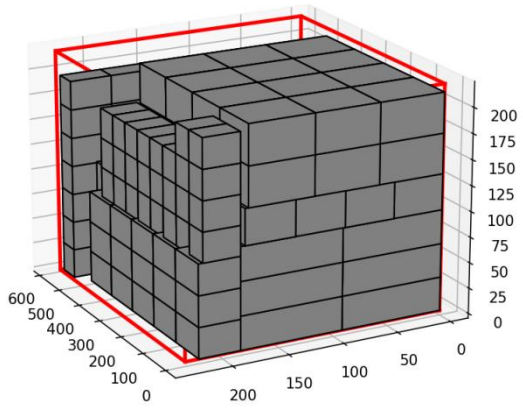
本实验的实验环境是：(1)算力：CPU-Intel(R) Core(TM) i7-12700KF @ 3.60GHz，16GB RAM；(2)环境架构：python 3.7

实验数据：老师给出的 5 大类一种 25 组数据。

实验结果总结：最好的装载率可以达到 85.7443%(E1-2、E1-3)，其他数据基本在 80%上下。

5.1 装载坐标及示例图

以数据 E3-5 的贪心算法为例，展示装载结果的顺序和可视化：




```
--装载箱子坐标（按照摆放顺序）：
-第1 个箱子的装载坐标：[ 0 , 0 , 0 ]  -第2 个箱子的装载坐标：[ 0 , 0 ,33 ]  -第3 个箱子的装载坐标：[ 0 , 0 ,66 ]
-第4 个箱子的装载坐标：[ 0 , 0 ,99 ]  -第5 个箱子的装载坐标：[ 0 , 0 ,134]  -第6 个箱子的装载坐标：[ 0 , 0 ,173]
-第7 个箱子的装载坐标：[ 0 ,45 ,99 ]  -第8 个箱子的装载坐标：[ 0 ,59 ,134]  -第9 个箱子的装载坐标：[ 0 ,59 ,173]
-第10个箱子的装载坐标：[ 0 ,90 ,99 ]  -第11个箱子的装载坐标：[ 0 ,92 , 0 ]  -第12个箱子的装载坐标：[ 0 ,92 ,33 ]
-第13个箱子的装载坐标：[ 0 ,92 ,66 ]  -第14个箱子的装载坐标：[ 0 ,118,134]  -第15个箱子的装载坐标：[ 0 ,118,173]
-第16个箱子的装载坐标：[ 0 ,135,99 ]  -第17个箱子的装载坐标：[75 , 0 ,99 ]  -第18个箱子的装载坐标：[75 ,45 ,99 ]
-第19个箱子的装载坐标：[75 ,90 ,99 ]  -第20个箱子的装载坐标：[75 ,135,99 ]  -第21个箱子的装载坐标：[113, 0 , 0 ]
-第22个箱子的装载坐标：[113, 0 ,33 ]  -第23个箱子的装载坐标：[113, 0 ,66 ]  -第24个箱子的装载坐标：[113,92 , 0 ]
-第25个箱子的装载坐标：[113,92 ,33 ]  -第26个箱子的装载坐标：[113,92 ,66 ]  -第27个箱子的装载坐标：[119, 0 ,134]
-第28个箱子的装载坐标：[119, 0 ,173]  -第29个箱子的装载坐标：[119,59 ,134]  -第30个箱子的装载坐标：[119,59 ,173]
-第31个箱子的装载坐标：[119,118,134]  -第32个箱子的装载坐标：[119,118,173]  -第33个箱子的装载坐标：[150, 0 ,99 ]
-第34个箱子的装载坐标：[150,45 ,99 ]  -第35个箱子的装载坐标：[150,90 ,99 ]  -第36个箱子的装载坐标：[150,135,99 ]
-第37个箱子的装载坐标：[225, 0 ,99 ]  -第38个箱子的装载坐标：[225,45 ,99 ]  -第39个箱子的装载坐标：[225,90 ,99 ]
-第40个箱子的装载坐标：[225,135,99 ]  -第41个箱子的装载坐标：[226, 0 , 0 ]  -第42个箱子的装载坐标：[226, 0 ,33 ]
-第43个箱子的装载坐标：[226, 0 ,66 ]  -第44个箱子的装载坐标：[226,92 , 0 ]  -第45个箱子的装载坐标：[226,92 ,33 ]
-第46个箱子的装载坐标：[226,92 ,66 ]  -第47个箱子的装载坐标：[238, 0 ,134]  -第48个箱子的装载坐标：[238, 0 ,173]
-第49个箱子的装载坐标：[238,59 ,134]  -第50个箱子的装载坐标：[238,59 ,173]  -第51个箱子的装载坐标：[238,118,134]
-第52个箱子的装载坐标：[238,118,173]  -第53个箱子的装载坐标：[300, 0 ,99 ]  -第54个箱子的装载坐标：[300,45 ,99 ]
-第55个箱子的装载坐标：[300,90 ,99 ]  -第56个箱子的装载坐标：[300,135,99 ]  -第57个箱子的装载坐标：[339, 0 , 0 ]
-第58个箱子的装载坐标：[339, 0 ,33 ]  -第59个箱子的装载坐标：[339, 0 ,66 ]  -第60个箱子的装载坐标：[339,64 , 0 ]
-第61个箱子的装载坐标：[339,64 ,33 ]  -第62个箱子的装载坐标：[339,64 ,66 ]  -第63个箱子的装载坐标：[339,128, 0 ]
-第64个箱子的装载坐标：[339,128,33 ]  -第65个箱子的装载坐标：[339,128,66 ]  -第66个箱子的装载坐标：[357, 0 ,134]
```

```
-第67个箱子的装载坐标：[357, 0 ,173]  -第68个箱子的装载坐标：[357,59 ,134]  -第69个箱子的装载坐标：[357,59 ,173]
-第70个箱子的装载坐标：[357,118,134]  -第71个箱子的装载坐标：[357,118,173]  -第72个箱子的装载坐标：[375, 0 ,99 ]
-第73个箱子的装载坐标：[375,45 ,99 ]  -第74个箱子的装载坐标：[375,90 ,99 ]  -第75个箱子的装载坐标：[375,135,99 ]
-第76个箱子的装载坐标：[431, 0 , 0 ]  -第77个箱子的装载坐标：[431, 0 ,33 ]  -第78个箱子的装载坐标：[431, 0 ,66 ]
-第79个箱子的装载坐标：[431,64 , 0 ]  -第80个箱子的装载坐标：[431,64 ,33 ]  -第81个箱子的装载坐标：[431,64 ,66 ]
-第82个箱子的装载坐标：[431,128, 0 ]  -第83个箱子的装载坐标：[431,128,33 ]  -第84个箱子的装载坐标：[431,128,66 ]
-第85个箱子的装载坐标：[450, 0 ,99 ]  -第86个箱子的装载坐标：[450,45 ,99 ]  -第87个箱子的装载坐标：[450,90 ,99 ]
-第88个箱子的装载坐标：[450,135,99 ]  -第89个箱子的装载坐标：[523, 0 , 0 ]  -第90个箱子的装载坐标：[523, 0 ,33 ]
-第91个箱子的装载坐标：[523, 0 ,66 ]  -第92个箱子的装载坐标：[523,64 , 0 ]  -第93个箱子的装载坐标：[523,64 ,33 ]
-第94个箱子的装载坐标：[523,64 ,66 ]  -第95个箱子的装载坐标：[523,128, 0 ]  -第96个箱子的装载坐标：[523,128,33 ]
-第97个箱子的装载坐标：[523,128,66 ]  -第98个箱子的装载坐标：[531, 0 , 0 ]  -第99个箱子的装载坐标：[531, 0 ,49 ]
-第100个箱子的装载坐标：[531, 0 ,98 ]  -第101个箱子的装载坐标：[531, 0 ,147]  -第102个箱子的装载坐标：[531,52 , 0 ]
-第103个箱子的装载坐标：[531,52 ,49 ]  -第104个箱子的装载坐标：[531,52 ,98 ]  -第105个箱子的装载坐标：[531,52 ,147]
-第106个箱子的装载坐标：[531,104, 0 ]  -第107个箱子的装载坐标：[531,104,49 ]  -第108个箱子的装载坐标：[531,104,98 ]
-第109个箱子的装载坐标：[531,104,147]  -第110个箱子的装载坐标：[531,156, 0 ]  -第111个箱子的装载坐标：[531,156,28 ]
-第112个箱子的装载坐标：[531,156,56 ]  -第113个箱子的装载坐标：[531,156,84 ]  -第114个箱子的装载坐标：[531,156,112]
-第115个箱子的装载坐标：[531,156,140]  -第116个箱子的装载坐标：[531,156,168]  -第117个箱子的装载坐标：[531,193, 0 ]
-第118个箱子的装载坐标：[531,193,28 ]  -第119个箱子的装载坐标：[531,193,56 ]  -第120个箱子的装载坐标：[531,193,84 ]
-第121个箱子的装载坐标：[531,193,112]  -第122个箱子的装载坐标：[531,193,140]  -第123个箱子的装载坐标：[531,193,168]
-第124个箱子的装载坐标：[ 0 ,184, 0 ]  -第125个箱子的装载坐标：[ 0 ,184,30 ]  -第126个箱子的装载坐标：[ 0 ,184,60 ]
-第127个箱子的装载坐标：[ 0 ,184,90 ]  -第128个箱子的装载坐标：[ 0 ,184,119]  -第129个箱子的装载坐标：[ 0 ,184,148]
-第130个箱子的装载坐标：[57 ,184,90 ]  -第131个箱子的装载坐标：[57 ,184,119]  -第132个箱子的装载坐标：[57 ,184,148]
-第133个箱子的装载坐标：[99 ,184, 0 ]  -第134个箱子的装载坐标：[99 ,184,30 ]  -第135个箱子的装载坐标：[99 ,184,60 ]
-第136个箱子的装载坐标：[114,184,90 ]  -第137个箱子的装载坐标：[114,184,119]  -第138个箱子的装载坐标：[114,184,148]
```

```
-第139个箱子的装载坐标：[171,184,90 ]  -第140个箱子的装载坐标：[171,184,119]  -第141个箱子的装载坐标：[171,184,148]
-第142个箱子的装载坐标：[198,184, 0 ]  -第143个箱子的装载坐标：[198,184,30 ]  -第144个箱子的装载坐标：[198,184,60 ]
-第145个箱子的装载坐标：[228,184,90 ]  -第146个箱子的装载坐标：[228,184,119]  -第147个箱子的装载坐标：[228,184,148]
-第148个箱子的装载坐标：[285,184,90 ]  -第149个箱子的装载坐标：[285,184,119]  -第150个箱子的装载坐标：[285,184,148]
-第151个箱子的装载坐标：[297,184, 0 ]  -第152个箱子的装载坐标：[297,184,30 ]  -第153个箱子的装载坐标：[297,184,60 ]
-第154个箱子的装载坐标：[342,184,90 ]  -第155个箱子的装载坐标：[342,184,119]  -第156个箱子的装载坐标：[342,184,148]
-第157个箱子的装载坐标：[396,184, 0 ]  -第158个箱子的装载坐标：[396,184,30 ]  -第159个箱子的装载坐标：[396,184,60 ]
-第160个箱子的装载坐标：[399,184,90 ]  -第161个箱子的装载坐标：[399,184,119]  -第162个箱子的装载坐标：[399,184,148]
-第163个箱子的装载坐标：[ 0 ,184,177]  -第164个箱子的装载坐标：[57 ,184,177]
```

5.2 不同数据和算法装载率

贪心算法对于 E1 数据的实验结果如下：

```
----------------E1-1----------------
--选取装填的货物：第1个箱子装载39件；第2个箱子装载32件；第3个箱子装载24件；
--运行时间：10.1383s
--装填率：77.1828%
```

```
----------------E1-2----------------
--选取装填的货物：第1个箱子装载28件；第2个箱子装载20件；第3个箱子装载18件；
--运行时间：5.3268s
--装填率：76.9634%
```

```
----------------E1-3----------------
--选取装填的货物：第1个箱子装载28件；第2个箱子装载20件；第3个箱子装载18件；
--运行时间：5.4519s
--装填率：76.9634%
```

```
----------------E1-4----------------
--选取装填的货物：第1个箱子装载64件；第2个箱子装载28件；第3个箱子装载64件；
--运行时间：63.4382s
--装填率：83.3179%
```

```
----------------E1-5----------------
--选取装填的货物：第1个箱子装载16件；第2个箱子装载36件；第3个箱子装载48件；
--运行时间：44.9425s
--装填率：72.2237%
```

深度搜索算法对于 E1 数据的实验结果如下：

```
----------------E1-1----------------
--选取装填的货物：第1个箱子装载39件；第2个箱子装载33件；第3个箱子装载24件；
--运行时间：222.6494s
--装填率：77.5758%


----------------E1-2----------------
--选取装填的货物：第1个箱子装载16件；第2个箱子装载23件；第3个箱子装载30件；
--运行时间：182.4249s
--装填率：85.7443%


----------------E1-3----------------
--选取装填的货物：第1个箱子装载16件；第2个箱子装载23件；第3个箱子装载30件；
--运行时间：219.5270s
--装填率：85.7443%


----------------E1-4----------------
--选取装填的货物：第1个箱子装载64件；第2个箱子装载28件；第3个箱子装载37件；
--运行时间：12944.0801s
--装填率：70.9295%


----------------E1-5----------------
--选取装填的货物：第1个箱子装载63件；第2个箱子装载52件；第3个箱子装载16件；
--运行时间：11179.6475s
--装填率：59.7131%
```

	E1-1	E1-2	E1-3	E1-4	E1-5
贪心算法时间	10.1383s	5.3268s	5.4519s	63.4382s	44.9425s
贪心算法装载率	77.1828%	76.9634%	76.9634%	83.3179%	72.2237%
深度搜索时间	222.6494s	182.4249s	219.5270s	12944.0801s	11179.6475s
深度搜索装载率	77.5758%	85.7443%	85.7443%	70.9295%	59.7131%

从两种算法的实验结果可以看出，对于 E1 数据深度搜索算法的时间要比贪心算法长的多，这是因为贪心算法直接将可行块的最大块进行装填，而深度搜索算法会往下探索去评估每个块的装填效果再返回决定最终的装载块。但是在消耗更多时间成本的基础上，深度搜索算法并不是在所有数据中都得到了比贪心算法更高的装载率，因此在下面四组数据中，我们只采用贪心算法进行实验。

贪心算法对于 E2 数据的实验结果：

```
----------------E2-1----------------
--选取装填的货物：第1个箱子装载24件；第2个箱子装载7件；第3个箱子装载8件；第4个箱子装载13件；第5个箱子装载14件；
--运行时间：5.7955s
--装填率：76.8728%
```

```
----------------E2-2----------------
--选取装填的货物：第1个箱子装载12件；第2个箱子装载0件；第3个箱子装载28件；第4个箱子装载18件；第5个箱子装载16件；
--运行时间：43.2867s
--装填率：79.8299%
```

```
----------------E2-3----------------
--选取装填的货物：第1个箱子装载24件；第2个箱子装载24件；第3个箱子装载0件；第4个箱子装载20件；第5个箱子装载24件；
--运行时间：98.3505s
--装填率：77.1762%
```

```
----------------E2-4----------------
--选取装填的货物：第1个箱子装载14件；第2个箱子装载24件；第3个箱子装载20件；第4个箱子装载12件；第5个箱子装载28件；
--运行时间：27.3685s
--装填率：81.3523%
```

```
----------------E2-5----------------
--选取装填的货物：第1个箱子装载18件；第2个箱子装载6件；第3个箱子装载20件；第4个箱子装载24件；第5个箱子装载0件；
--运行时间：10.3569s
--装填率：80.1424%
```

	E2-1	E2-2	E2-3	E2-4	E2-5
贪心算法时间	5.7955s	43.2867s	98.3505s	27.3685s	10.3569s
贪心算法装载率	76.8728%	79.8299%	77.1762%	81.3523%	80.1424%

贪心算法对于 E3 数据的实验结果：

```
---------------E3-1----------------
-选取装填的货物：第1个箱子装载24件；第2个箱子装载9件；第3个箱子装载0件；第4个箱子装载11件；第5个箱子装载6件；第6个箱子装载8件；第7个箱子装载12件；第8个箱子装载9件；
-运行时间：19.1830s
-装填率：71.8808%
```

```
---------------E3-2----------------
-选取装填的货物：第1个箱子装载0件；第2个箱子装载20件；第3个箱子装载18件；第4个箱子装载20件；第5个箱子装载16件；第6个箱子装载0件；第7个箱子装载16件；第8个箱子装载12件；
-运行时间：101.3043s
-装填率：69.3383%
```

```
--------------E3-3---------------
-选取装填的货物：第1个箱子装载15件；第2个箱子装载14件；第3个箱子装载0件；第4个箱子装载16件；第5个箱子装载6件；第6个箱子装载15件；第7个箱子装载17件；第8个箱子装载6件；
-运行时间：90.4631s
-装填率：78.5226%
```

```
--------------E3-4---------------
-选取装填的货物：第1个箱子装载16件；第2个箱子装载0件；第3个箱子装载16件；第4个箱子装载16件；第5个箱子装载0件；第6个箱子装载12件；第7个箱子装载15件；第8个箱子装载0件
-运行时间：57.0022s
-装填率：73.1041%
```

```
--------------E3-5---------------
-选取装填的货物：第1个箱子装载18件；第2个箱子装载14件；第3个箱子装载26件；第4个箱子装载15件；第5个箱子装载18件；第6个箱子装载24件；第7个箱子装载12件；第8个箱子装载28件；
-运行时间：235.4289s
-装填率：83.1642%
```

	E3-1	E3-2	E3-3	E3-4	E3-5
贪心算法时间	19.1830s	101.3043s	90.4631s	57.0022s	235.4289s
贪心算法装载率	71.8808%	69.3383%	78.5226%	73.1041%	83.1642%

贪心算法对于 E4 数据的实验结果：

```
----------------E4-1----------------
--选取装填的货物：第1个箱子装载13件；第2个箱子装载9件；第3个箱子装载10件；第4个箱子装载12件；第5个箱子装载0件；第6个箱子装载12件；第7
--运行时间：65.2959s
--装填率：78.2156%
```

```
---------------E4-2---------------
--选取装填的货物：第1个箱子装载8件；第2个箱子装载16件；第3个箱子装载12件；第4个箱子装载12件；第5个箱子装载18件；第6个箱子装载0件；第7个箱子装载7件；
--运行时间：80.0124s
--装填率：71.0252%
```

```
---------------E4-3---------------
-选取装填的货物：第1个箱子装载8件；第2个箱子装载16件；第3个箱子装载12件；第4个箱子装载15件；第5个箱子装载0件；第6个箱子装载9件；第7个箱子装载9件；第
-运行时间：75.7165s
-装填率：69.2038%
```

```
---------------E4-4---------------
-选取装填的货物：第1个箱子装载11件；第2个箱子装载0件；第3个箱子装载17件；第4个箱子装载19件；第5个箱子装载13件；第6个箱子装载18件；第7个箱子装载1
-运行时间：139.6234s
-装填率：80.7570%
```

```
---------------E4-5----------------
-选取装填的货物：第1个箱子装载16件；第2个箱子装载0件；第3个箱子装载14件；第4个箱子装载0件；第5个箱子装载8件；第6个箱子装载0件；
-运行时间：89.2906s
-装填率：80.7303%
```

	E4-1	E4-2	E4-3	E4-4	E4-5
贪心算法时间	65.2959s	80.0124s	75.7165s	139.6234s	89.2906s
贪心算法装载率	78.2156%	71.0252%	69.2038%	80.7570%	80.7303%

贪心算法对于 E5 数据的实验结果：

```
---------------E5-1---------------
--选取装填的货物：第1个箱子装载6件；第2个箱子装载0件；第3个箱子装载10件；第4个箱子装载3件；第5个箱子装载5件；第6个箱子装载10件；第7个箱子装载12件
--运行时间：61.1833s
--装填率：76.5442%
```

```
---------------E5-2---------------
--选取装填的货物：第1个箱子装载12件；第2个箱子装载12件；第3个箱子装载6件；第4个箱子装载9件；第5个箱子装载0件；第6个箱子装载12件；第7个箱子装载8件；第8
--运行时间：124.1752s
--装填率：80.8138%
```

	E5-1	E5-2	E5-3	E5-4	E5-5
贪心算法时间	61.1833s	124.1752s	160.8630s	120.4421s	123.8327s
贪心算法装载率	76.5442%	80.8138%	77.7866%	81.3736%	78.4692%

六、实验总结

（1）初步设计摆放算法时没考虑一种箱子的两种摆放方式，导致摆放结果不理想。在思考和分析后，将两种摆放方式加入简单块生成的模块中，在生成简单块时就考虑不同的摆放方式会产生的所有可能的简单块结果。

（2）实验室据的箱子种类不多，并且每种箱子也达到了一定的数量，因此可以看到贪心算法也能直接得到较高的装载率。

II 高级实现部分

一、问题概述

与基础部分不同的是，高级部分的约束条件有所不同：

1. 参数考虑小数点后两位；
2. 实现在线算法，也就是箱子是按照随机顺序到达，先到达先摆放；
3. 需要考虑箱子的摆放顺序，即箱子是从内到外，从下向上的摆放顺序；
4. 因箱子共有 3 个不同的面，所有每个箱子有 6 种不同的摆放状态；
5. 算法需要实时得出结果，即算法时间小于等于 2 秒。

二、算法基本思想

箱子按照先上后前再右的顺序摆放。第一个到达的箱子摆在车厢的左后角，第二到达的箱子检测是否可以摆在已有箱子的上面，若不能则摆在前一个箱子的前面，如图 2-1 所示。直到下一个箱子无法摆放到已有箱子的上面，也无法再往前摆，则开始摆第二列，那么第一列的空间就此舍弃不再摆放。第一列中最宽的箱子会形成第一列的边界，当第二列箱子进行摆放时不允许超过此边界。如图 2-2 所示，假设图中第一列已摆满，并且蓝色块最宽。

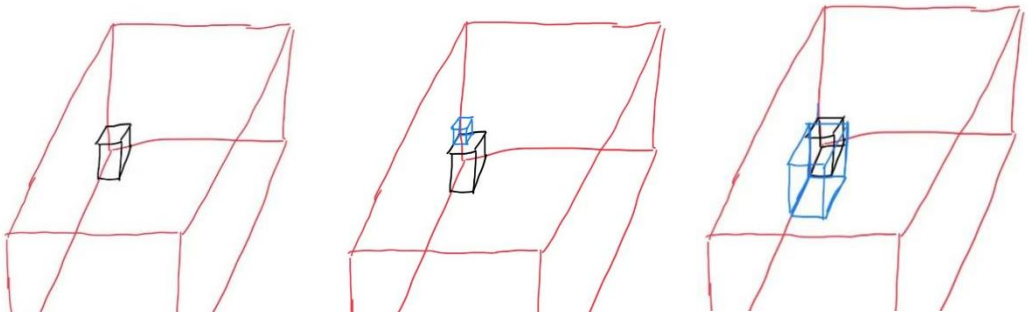


图 2-1 第一列摆放

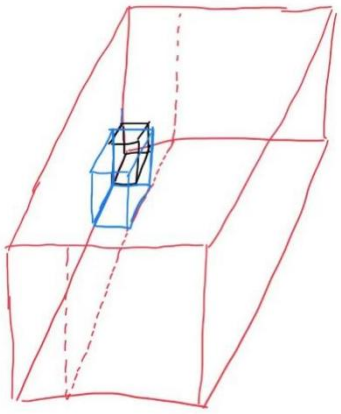


图 2-2 第一列摆满时的空间分割

选择这种摆放方式的原因是：

（1）若贴着最里面的车厢宽进行摆放那么下一个到来块的摆放空间的选择较少，而贴着车厢的长边进行摆放时，下一个到来块的可选择性更多，这样浪费的空间更少。如图 2-3 所示，贴着车厢长边摆放下一个到来的箱子会有更多选择，这样浪费的上部空间更少。

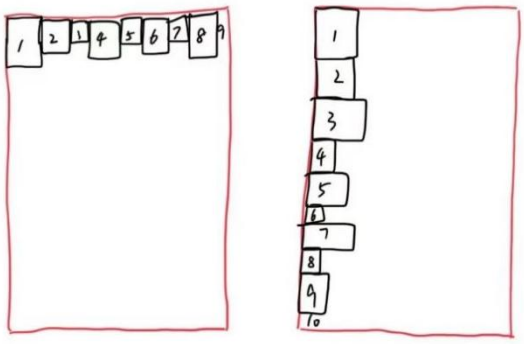


图 2-3 贴着不同边的摆放效果

（2）我们这样设计的主要原因在于这种一列一列摆放的方式可以避免去考虑复杂多样的摆放约束（不考虑装载员的体积即装载人员是否可以到最后一列进行摆放），比如前右两个块会挡住最里面块的摆放，这是提高效率的方式，也是我们需要进一步改进从而提升装载率的地方。

三、算法具体内容

基本实现与基本部分大致相同，不同的点在于由于每次只放置一个箱子，因此无法再生成简单块以及复杂块。关于裁切空间，对于一个块的放置后，只裁切其上面的块以及前面的块，右边的块待选不作为本列放置空间的参考范围。其余的细节问题将在下面小章节中仔细介绍。

3.1 数据读取

模拟箱子按照不同顺序依次进入，利用一个列表来放置不同种类箱子的顺序，再通过随机打乱函数实现不同箱子的随机顺序到达。

3.2 箱子的摆放

对于每一列的第一个箱子，由于没有任何参照物以及评估值，因此我们选择将其以最大面为放置面，且最长边贴着 x 轴。因此需要对箱子进行判断，将其三个边中的最大值设为 lx、其次大的值设为 ly，最小值设为 lz。

对于不是每一列的第一个箱子来说，我们考虑它的六种摆放方式，六种摆放方式中首先选择可以放置到当前最小剩余空间的可行块，再在可行块中挑选放置面（x*y）面积最大的块。若最小剩余空间无法放置六种情况的任何一种，则进入第二小剩余空间进行判断，同理

若还不满足则再进入下一个剩余空间。

3.3 裁切空间以及剩余空间

关于裁切空间，我们同样按照基本部分的裁切方式进行，不同的在于我们不再判断其剩余的 x 和 y 长度，而是直接裁切出三个空间进行返回。其中 x 方向和 z 方向的裁切空间用于放到剩余空间中进行放置，y 方向的裁切空间需要记录下来，随着后面不同摆放方式有可能增大 y 的起始点需要对 y 方向的裁切空间进行更新。最终下一列的剩余空间即上一列完成后更新得到的 y 方向的裁切空间。

关于剩余空间，最一开始就是整个容器。后面通过栈结构来保存剩余空间。比如放置第一个块之后，剩余空间会加入本次放置得到的 x 方向和 z 方向的裁切空间，且与基本部分不同的是，剩余空间的摆放顺序是从小到大，即每次弹出的剩余空间都是当前所有剩余空间中最小的。这样也算是贪心算法的思想，如果第一个剩余空间无法放置块，则跳到下一个块，依次直到找到可以放下这个块的剩余空间，若找不到，则直接结束程序。需要注意的是，在依次寻找合适的剩余空间时，需要把跳过的剩余空间进行保存，结束本次块的放置后需要把这些跳过的剩余空间再 push 到剩余空间堆栈中，防止后面来了更小的块，本来可以放到最小剩余空间但剩余空间被丢弃引发的浪费。到了新的一列，其初始的剩余空间就是上一列结束所有摆放更新得到的 y 方向的裁切空间，这时需要 clear 剩余空间堆栈，防止再到上一列进行摆放，clear 之后再把 y 方向的裁切空间 push 进行，重复每一列的放置计算即可。

四、算法实现

基本数据结构与简单部分相同，不同的模块如下：

（1）第一个箱子选择最大面当作第一个块

考虑每一列第一个箱子没有参考物以及评判标准，因此按照前面提到的算法基本思想，以最大面为放置面，且最长边靠着 x 轴。

```
def gen_one_block(cur_box,num_list):
    box = Box(cur_box.lx, cur_box.ly, cur_box.lz, cur_box.type)
    #第一个箱子选择最大面放置
    xyz = []
    xyz.append(int(box.lx))
    xyz.append(int(box.ly))
    xyz.append(int(box.lz))
    xyz.sort(reverse=True)
    box.lx = xyz[0]
    box.ly = xyz[1]
    box.lz = xyz[2]
    #一次只看一个块，因此只需要一个箱子数量
    requires = np.full_like(num_list, 0)
    requires[box.type] = 1
    #块的尺寸
    block = Block(box.lx, box.ly, box.lz, requires)
    block.ax = box.lx    #顶部可放置地址
    block.ay = box.ly
    block.volume = box.lx * box.ly * box.lz    #块的体积
    block.times = 0
    return block
```

（2）其他箱子的六种摆放方式

每个边各做一次长宽高，即 3！=6 种摆放方式。

```
def gen_six_block(curr_box,num_list):
    box = Box(curr_box.lx, curr_box.ly, curr_box.lz, curr_box.type)
    cur_block_table = []
    #六种可能
    box_xyz = [int(box.lx), int(box.ly), int(box.lz)]
    xyz_type = [[0,1,2],
                [0,2,1],
                [1,0,2],
                [1,2,0],
                [2,0,1],
                [2,1,0]]
    for i in range(6):
        cur_box = Box(curr_box.lx, curr_box.ly, curr_box.lz, curr_box.type)
        cur_box.lx = box_xyz[xyz_type[i][0]]
        cur_box.ly = box_xyz[xyz_type[i][1]]
        cur_box.lz = box_xyz[xyz_type[i][2]]
```

```
        #一次只到一个块，因此只需要一个箱子数量
        requires = np.full_like(num_list, 0)
        requires[cur_box.type] = 1
        #块的尺寸
        block = Block(cur_box.lx, cur_box.ly, cur_box.lz, requires)
        #顶部可放置矩阵
        block.ax = cur_box.lx
        block.ay = cur_box.ly
        #块的体积
        block.volume = cur_box.lx * cur_box.ly * cur_box.lz
        block.times = 0
        cur_block_table.append(block)

    return cur_block_table
```

（3）裁切出新的剩余空间（有稳定性约束）

与基本部分基本一致，不同的在于不需要考虑 x 方向和 y 方向裁切空间的大小，只考虑方向。不考虑大小的原因在于在一列的摆放中只对 x 方向的剩余空间进行放置，y 方向的裁切空间是待修剪的下一列的初始剩余空间。

```
def gen_xz_space(space: Space, block: Block, box_list=[]):
    # 三个维度的剩余尺寸
    rmx = space.lx - block.lx
    rmy = space.ly - block.ly
    rmz = space.lz - block.lz
    # 三个新裁切出的剩余空间（按入栈顺序依次返回）
    drs_x = Space(space.x + block.lx, space.y, space.z, rmx, space.ly, space.lz, space)
    drs_y = Space(space.x, space.y + block.ly, space.z, block.lx, rmy, space.lz, space)
    drs_z = Space(space.x, space.y, space.z + block.lz, block.ax, block.ay, rmz, None)
    return drs_x, drs_y, drs_z
```

（4）基本启发式算法

初始化算法，将初始剩余空间定义为整个容器大小。

```
def basic_heuristic(is_complex, search_params, problem: Problem, type_list):
    print("----------------E1-4----------------")
    # 初始化排样状态
    ps = PackingState(avail_list=problem.num_list)
    # 开始时，剩余空间堆栈中只有容器本身
    ps.space_stack.push(problem.container)
    #用于记录下一次重起列的位置
    newcontainer = Space(problem.container.x, problem.container.y, problem.container.z, problem.container.lx, problem.container.ly, probl
    block_table = []
    for i in range(len(type_list)):
        st = time.time()
        cur_box = problem.box_list[type_list[i]]
        print("-第{}个箱子（长：{}，宽：{}，高：{}）摆放位置：".format(int(i+1), cur_box.lx, cur_box.ly, cur_box.lz),end = "")
```

第一列第一个块的放置，其中需要对 x 方向和 z 方向的剩余空间进行判断，保证最小剩余空间在剩余空间栈顶，放置第一个块之后将代表整个容器大小的剩余空间弹出。

```
        if i == 0:
            block_table.append(gen_one_block(cur_box, problem.num_list))
            space = ps.space_stack.top()
            block_list = gen_block_list(space, ps.avail_list, block_table)
            #清空块列表
            block_table = []
            if block_list:
                block = block_list[0]
                ps.space_stack.pop()
                ps.avail_list = (np.array(ps.avail_list) - np.array(block.require_list)).tolist()
                ps.plan_list.append(Place(space, block))
                ps.volume = ps.volume + block.volume
                print("坐标（{},{},{}）,边界点（{},{},{}）".format(space.x, space.y, space.z, space.x + block.lx, space.y + block.ly, space.z
                cuboidx, cuboidy, cuboidz = gen_xz_space(space, block)    #本列其他可放置空间（按照横截面积大小加入空间堆栈）
                if cuboidx.lx * cuboidx.ly < cuboidz.lx * cuboidz.ly:
                    ps.space_stack.push(cuboidz, cuboidx)
                else:
                    ps.space_stack.push(cuboidx, cuboidz)
                newcontainer.y = cuboidy.y    #下一列空间（后面根据y来判定是否需要右移）
                newcontainer.ly = cuboidy.ly
            else:
                break #第一部分
```

对于一列中不是第一个箱子的放置，需要对六种摆放方式进行可行块判定，若可行块存在则挑选放置面最大的摆放情况进行放置；若可行块为空，需要对当前剩余空间进行记录并跳到下一个更大的剩余空间进行判断能否放置，直到找到可以放置的剩余空间，对该剩余空间进行弹出并将之前保存的较小剩余空间需要按照从大到小压入栈内，保证栈顶剩余空间最小。

```
        else:                                                                                    ⚠ 14  ⚠ 161  ✓ 15
            block_table = block_table + gen_six_block(cur_box, problem.num_list)
            tmp_block_table = []
            while ps.space_stack.size() > 0:
                space = ps.space_stack.top()
                block_list = gen_block_list(space, ps.avail_list, block_table)
                if block_list:
                    #清空块列表
                    block_table = []
                    max_index = 0
                    max_rate = 0
                    for j in range(len(block_list)):
                        rate = (block_list[j].lx * block_list[j].ly) / (space.lx * space.ly)
                        if rate > max_rate:
                            max_rate = rate
                            max_index = j
                    block = block_list[max_index]
                    ps.space_stack.pop()
                    ps.avail_list = (np.array(ps.avail_list) - np.array(block.require_list)).tolist()
                    ps.plan_list.append(Place(space, block))
                    ps.volume = ps.volume + block.volume
                    print("坐标（{},{},{}）,边界点（{},{},{}）".format(space.x, space.y, space.z, space.x + block.lx, space.y + block.ly, sp
```

```
                #本列其他可放置空间（按照横截面积大小加入空间堆栈）
                cuboidx, cuboidy, cuboidz = gen_xz_space(space, block)
                if cuboidx.lx * cuboidx.ly < cuboidz.lx * cuboidz.ly:
                    ps.space_stack.push(cuboidz, cuboidx)
                else:
                    ps.space_stack.push(cuboidx, cuboidz)
                #下一列空间（后面根据y来判定是否需要右移）
                if cuboidy.y > newcontainer.y:
                    newcontainer.y = cuboidy.y
                    newcontainer.ly = cuboidy.ly
                #把其他丢弃掉的space还原，以防后面来的小箱子造成浪费
                for k in range(len(tmp_block_table)):
                    ps.space_stack.push(tmp_block_table[len(tmp_block_table)-1-k])
                break
            elif len(block_list)==0:
                tmp_block_table.append(ps.space_stack.top())
                ps.space_stack.pop()#二
```

对于一个块，若当前列的所有剩余空间都无法放置该块，则另起一列，其中这列为上一列 y 方向的裁切空间且其起始点的 y 坐标为上一列所有摆放边界点的最大值。

```
        if ps.space_stack.size() == 0:
            ps.space_stack.clear()    #另起一列    #将剩余栈空间都清除
            #将重起下一列的空间装入
            ps.space_stack.push(newcontainer)
            block_table.append(gen_one_block(cur_box, problem.num_list))
            space = ps.space_stack.top()
            block_list = gen_block_list(space, ps.avail_list, block_table)
            block_table = []    #清空块列表
            if block_list:
                block = block_list[0]
                ps.space_stack.pop()
                ps.avail_list = (np.array(ps.avail_list) - np.array(block.require_list)).tolist()
                ps.plan_list.append(Place(space, block))
                ps.volume = ps.volume + block.volume
                print("坐标（{},{},{}）,边界点（{},{},{}）".format(space.x, space.y, space.z, space.x + block.lx, space.y + block.ly, spac
                cuboidx, cuboidy, cuboidz = gen_xz_space(space, block)    #本列其他可放置空间（按照横截面积大小加入空间堆栈）
                if cuboidx.lx * cuboidx.ly < cuboidz.lx * cuboidz.ly:
                    ps.space_stack.push(cuboidz, cuboidx)
                else:
                    ps.space_stack.push(cuboidx, cuboidz)
                newcontainer.y = cuboidy.y    #下一列空间（后面根据y来判定是否需要右移）
                newcontainer.ly = cuboidy.ly
            else:
                break#二
```

下面是结果的输出方式。

```
    et = time.time()
    cost_time = et - st
    print("-本次运行时间为：{:.10f}s".format(cost_time))
    print("")
all_volume = problem.container.lx * problem.container.ly * problem.container.lz
#print(problem.container)
print("\n\n\n-选取装填的货物：",end="")
for i in range(len(ps.avail_list)):
    print("第{}个箱子装载{}件；".format(i+1,problem.num_list[i] - ps.avail_list[i]),end="")
print("\n-装填率：{:.4f}%".format(ps.volume / all_volume * 100))
print("\n")
'''
print("--装载箱子坐标（按照摆放顺序）：")
index = 0
for i,p in zip(range(len(ps.plan_list)),ps.plan_list):
    box_pos = build_box_position(p.block, (p.space.x, p.space.y, p.space.z), problem.box_l
    for item in sorted(box_pos, key=lambda x: (x[0], x[1], x[2])):
        index = index + 1
        print("-第{:^2}个箱子的装载坐标：[{:^3},{:^3},{:^3}]  ".format(index,int(item[0]),int(
        if index % 3 == 0:print("")
'''
# 绘制排样结果图
draw_packing_result(problem, ps)
```

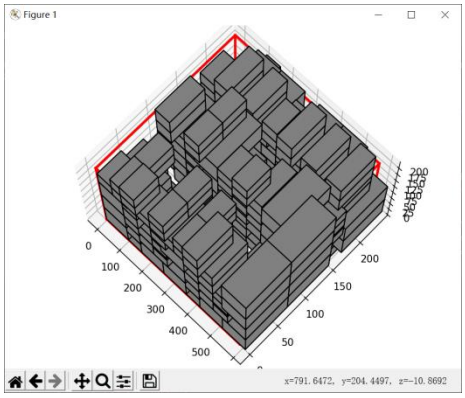
五、实验结果及分析

实验的实验环境是：(1)算力：CPU-Intel(R) Core(TM) i7-12700KF @ 3.60GHz，16GB RAM；(2)环境架构：python 3.7

实验数据：老师给出的 5 大类一种 25 组数据。

实验结果总结：装载率基本都在 50-70%，运行时间都在 0.0001s 左右。

以数据 E1-4 为例对算法进行检测得到结果：




```
-选取装填的货物：第1个箱子装载37件；第2个箱子装载30件；第3个箱子装载41件；
-装填率：68.5604%
```

```
-第1个箱子（长：98，宽：75，高：55）摆放位置：坐标（0,0,0）,边界点（98,75,55）
-本次运行时间为：0.0000000000s

-第2个箱子（长：98，宽：75，高：55）摆放位置：坐标（0,0,55）,边界点（98,75,110）
-本次运行时间为：0.0000000000s

-第3个箱子（长：98，宽：75，高：55）摆放位置：坐标（0,0,110）,边界点（98,75,165）
-本次运行时间为：0.0000000000s

-第4个箱子（长：60，宽：40，高：32）摆放位置：坐标（0,0,165）,边界点（60,40,197）
-本次运行时间为：0.0009980202s

-第5个箱子（长：60，宽：40，高：32）摆放位置：坐标（60,0,165）,边界点（92,60,205）
-本次运行时间为：0.0000000000s

-第6个箱子（长：60，宽：59，高：39）摆放位置：坐标（98,0,0）,边界点（158,59,39）
-本次运行时间为：0.0000000000s

-第7个箱子（长：60，宽：59，高：39）摆放位置：坐标（98,0,39）,边界点（158,59,78）
```

全部块的坐标结果详见文档“高级部分摆放坐标.txt”。

从实验结果可以看出，箱子按照一列一列的顺序进行摆放，装载率也达到了 68.5604%，并且将箱子摆放坐标进行计算，未超过车厢最大边界点。算法用时未超过两秒。

六、实验总结

高级部分的主要思想就是贪心，对于基本部分的组合和复合不再考虑。且为了防止考虑因为摆放顺序引起的复杂多样的摆放约束，我们直接讨论如何一列一列的摆放，这样无论何时，可以摆放的剩余空间都是完全开放的。

此外，我们在设计算法时还讨论了先上下还是先平铺还是先前后（大多都是凭直觉，下面的工作需要对我们的想法进行进一步的验证）的顺序关系，以及每种方法的优缺点，最终确定此方案。

参考文献

[1]彭煜. 求解三维装箱问题的启发式分层搜索算法[D].厦门大学,2009.