算法实践题

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。

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 l_i，w_i，h_i（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。

问题分解为基础和高级部分，完成基础部分可得 78 分以上，完成高级部分可得 85 分以上。

基础部分：

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；4. 算法时间不做严格要求，只要 1 天内得出结果都可。

高级部分：

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（每个面可以横着摆或竖着摆）；5. 算法需要实时得出结果，即算法时间小于等于 2 秒。

成员：2022202210017-史俊楠，2022202210031-高强

1、基础部分：

实验分析：

对于装箱问题的约束：本实验仅考虑两个约束：

(1) 方向约束：箱子平放即最大面朝下放置，这就决定了只有一条边可以作为箱子的高

(2) 稳定性约束：虽然题目没有明确说明，但根据常识，每个被装载的箱子必须得到容易底部或其他箱子的支撑，不能悬空

算法：

本实验采用启发式算法，采用剩余空间理论，即将剩余空间组织成堆栈，每次装箱从栈顶取一个剩余空间，若有可行块，按照块选择算法选择一个块放置在该空间，将剩余的未填充空间切分成新的剩余空间加入堆栈，若不存在可行块就抛弃该剩余空间，如此重复直至剩余空间堆栈为空为止。

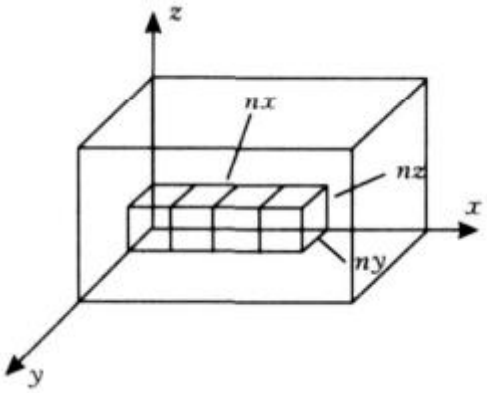
剩余空间：对于装箱问题中的每个箱子，都是通过其位置坐标和各个方向的边长来描述，位置坐标指靠近原点即车厢坐标(0,0,0)的左后下角的坐标，本实验对于箱子的装在都是在剩余空间中进行的，剩余空间初始化为车厢大小，经过装载后剩余空间指容器中未填充的长方体

空间（是长方体空间而非所有剩余空间），用（x,y,z）描述剩余空间的坐标，(lx,ly,lz)描述剩余空间三个维度的边长。
箱子：用 box_type 表明箱子类型，根据题目共有 5 类箱子数据，box_type3、box_type5 box_type8 、 box_type10 、 box_type15，每种类型的箱子有 5 组数据，box_type5_1……box_type5_5。给出初始数据，我们进行一定处理可得到数据三元组(container,box_list,num_list)，其中 container 指初始剩余空间，box_list 是箱子向量，每个元素是一个箱子的长宽高，num_list 是每个箱子的数量列表。
块和块表：

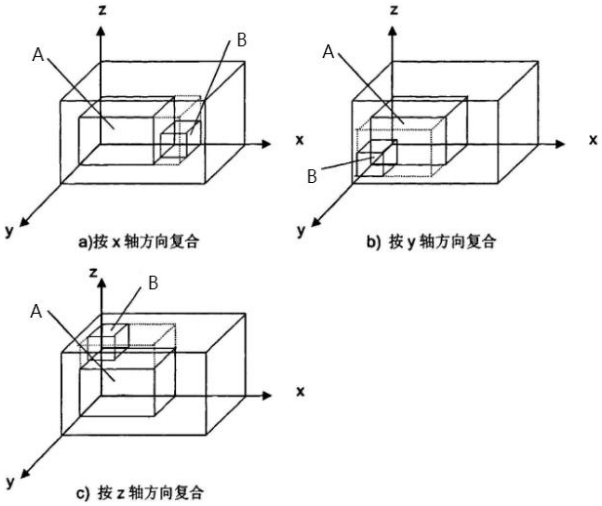
本实验采用的是基于块装载的启发式算法，块是其最小装载单位。一个块是包含了多个箱子的长方体，其中每个箱子的摆放都满足装箱问题的约束，用（volume,require_list,lx,ly,lz）描述一个块，其中 volume 描述块种箱子的总体积，require_list 描述了块中各种类型箱子的数目，lx,ly,lz 描述块三个维度的边长。由于块是由各种箱子组成的，因此块中可能存在空隙，在其中放置时要考虑稳定性约束。

块生成算法：
本实验包含了两种类型的块：简单块和复合块。

简单块是由同一种朝向的箱子堆叠而成，箱子和箱子之间没有空隙，堆叠的结果正好形成一个长方体，下图展示了一个简单块的例子，其中 nx,ny,nz 分别代表每个维度上的箱子数。



复合块是由多个简单块复合得到，简单块是最基本的复合块。对于两个简单块 A,B，有三种复合方式，按 x 轴方向复合、按 y 轴方向复合、按 z 轴方向复合得到复合块 C，C 也是一个长方体，C 的大小是包含了 A 和 B 的最小长方体，C 中可能有空隙。下图展示了三种复合方式。



由于复合块的多样性和存在空隙的问题，复合块的数量将是箱子数量的指数级，空隙问题也会造成利用率的下降，因此需要对复合块施加一定的限制：

(1) 复合块的大小不能超过容器。
(2) 复合块有空隙，为了减小空隙，需要对复合块的填充率做一定限制，即复合块的填充率大于等与 MIN_FILL_RATE。
(3) 块的顶部放置的块不能悬空
(4) 定义复合块的复杂度，简单块复杂度为 0，复合块的复杂度为其子块复杂度的最大值加 1，限制块的最大复杂度为 MAX_TIMES。
(5) 对于（volume,require_list,lx,ly,lz）相同的复合块视为等价块，重复生成的等价块将被忽略。

可行块生成：

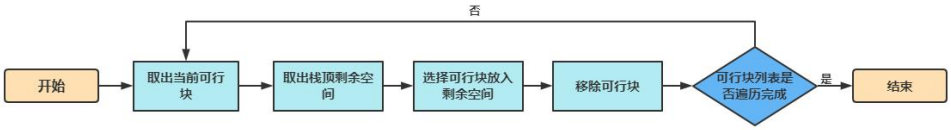
可行块生成算法 generate_block_list(space,avail,block_table)用于从块列表 block_table 中获取可用于当前剩余空间的可行块列表，将所有能放入空间 space 且 avail 中有足够的箱子满足 require_list 的块按体积降序排列并返回。

剩余空间的装载：

剩余空间的装载有两种情况：有可行块，无可行块。有可行块时根据块选择算法选择可行块进行装载，并将剩余的未填充空间切分为新的剩余空间。若无可行块则当前剩余空间被抛弃，对剩余的空间继续进行装载。

块的选择算法

算法的整体流程



块放置和移除：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。

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。

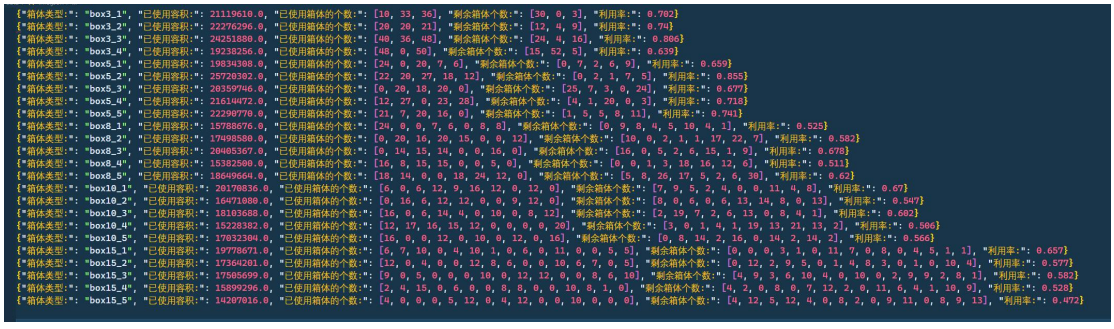
带深度限制的深度优先搜索算法：除了贪心算法，另一个方法是进行深度或广度优先搜索扩展当前放置方案，然后在叶子节点使用补全算法来评估叶节点的好坏，最终以搜索树中最好的一个叶子节点的评估值作为当前状态的评估。但是，由于搜索过程中，每一个节点都有大量分支，采用宽度优先搜索需要海量的空间，因此是不现实的。因此本实验采用的是带深度限制的深度优先搜索算法，通过深度来限制最多放置的块的数目。另外，由于每个阶段可行块列表包含大量的块，算法往往也限制每个节点的最大分支数。本文采用深度优先搜索算法扩展当前放置方案，算法的输入为一个部分放置方案，深度限制和最大分支数。该算法从一个部分放置方案出发，递归的尝试可行块列表中的块，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。

代码及结果：

本文的实验均在 juypter notebook 环境中完成，其中箱子的利用率存储在 data/info.json 文件中，箱体的坐标存储在 data/dimensions.json 文件中，还有绘制的图片文件位于 pictures 文件夹中。

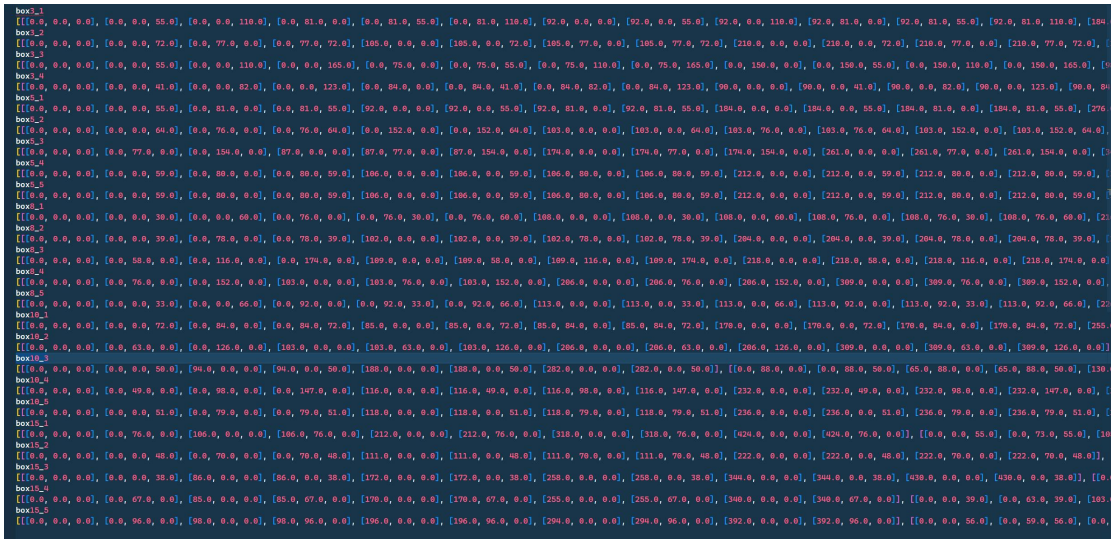
代码由于过长在此不做展示，请见 packing.ipynb 文件，此处只展示实验结果

实验结果列出了每种类型的箱子数据，使用容积以及使用个数和未使用个数和利用率。



{"箱体类型:": "box3_1", "已使用容积:": 21119610.0, "已使用箱体的个数:": [10, 33, 36], "剩余箱体个数:": [30, 0, 3], "利用率:": 0.702}
{"箱体类型:": "box3_2", "已使用容积:": 22276296.0, "已使用箱体的个数:": [20, 20, 21], "剩余箱体个数:": [12, 4, 9], "利用率:": 0.74}
{"箱体类型:": "box3_3", "已使用容积:": 24251880.0, "已使用箱体的个数:": [40, 36, 40], "剩余箱体个数:": [24, 4, 16], "利用率:": 0.806}
{"箱体类型:": "box3_4", "已使用容积:": 19238256.0, "已使用箱体的个数:": [48, 0, 50], "剩余箱体个数:": [15, 52, 5], "利用率:": 0.639}
{"箱体类型:": "box5_1", "已使用容积:": 19834308.0, "已使用箱体的个数:": [24, 0, 20, 7, 6], "剩余箱体个数:": [0, 7, 2, 6, 9], "利用率:": 0.659}
{"箱体类型:": "box5_2", "已使用容积:": 25720302.0, "已使用箱体的个数:": [22, 20, 27, 18, 12], "剩余箱体个数:": [0, 2, 1, 7, 5], "利用率:": 0.855}
{"箱体类型:": "box5_3", "已使用容积:": 20359746.0, "已使用箱体的个数:": [0, 20, 18, 20, 0], "剩余箱体个数:": [25, 7, 3, 0, 24], "利用率:": 0.677}
{"箱体类型:": "box5_4", "已使用容积:": 21614472.0, "已使用箱体的个数:": [12, 27, 0, 23, 28], "剩余箱体个数:": [4, 1, 20, 0, 3], "利用率:": 0.718}
{"箱体类型:": "box5_5", "已使用容积:": 22290770.0, "已使用箱体的个数:": [21, 7, 20, 16, 0], "剩余箱体个数:": [1, 5, 5, 8, 11], "利用率:": 0.741}
{"箱体类型:": "box8_1", "已使用容积:": 15788676.0, "已使用箱体的个数:": [24, 0, 0, 7, 6, 0, 8, 8], "剩余箱体个数:": [0, 9, 8, 4, 5, 10, 4, 1], "利用率:": 0.525}
{"箱体类型:": "box8_2", "已使用容积:": 17498580.0, "已使用箱体的个数:": [0, 20, 16, 20, 15, 0, 0, 12], "剩余箱体个数:": [10, 0, 2, 1, 1, 17, 22, 7], "利用率:": 0.582}
{"箱体类型:": "box8_3", "已使用容积:": 20485367.0, "已使用箱体的个数:": [0, 14, 15, 14, 0, 0, 16, 0], "剩余箱体个数:": [16, 0, 5, 2, 6, 15, 1, 9], "利用率:": 0.678}
{"箱体类型:": "box8_4", "已使用容积:": 15382580.0, "已使用箱体的个数:": [16, 8, 15, 15, 0, 0, 5, 0], "剩余箱体个数:": [0, 0, 1, 5, 18, 16, 12, 6], "利用率:": 0.511}
{"箱体类型:": "box8_5", "已使用容积:": 18649664.0, "已使用箱体的个数:": [18, 14, 0, 0, 18, 24, 12, 0], "剩余箱体个数:": [5, 8, 26, 17, 5, 2, 6, 30], "利用率:": 0.62}
{"箱体类型:": "box10_1", "已使用容积:": 20170836.0, "已使用箱体的个数:": [6, 0, 6, 12, 9, 16, 12, 0, 12, 0], "剩余箱体个数:": [7, 9, 5, 2, 4, 0, 0, 11, 4, 8], "利用率:": 0.67}
{"箱体类型:": "box10_2", "已使用容积:": 16471080.0, "已使用箱体的个数:": [0, 16, 6, 12, 12, 0, 0, 9, 12, 0], "剩余箱体个数:": [8, 0, 6, 0, 6, 13, 14, 8, 0, 13], "利用率:": 0.547}
{"箱体类型:": "box10_3", "已使用容积:": 18103688.0, "已使用箱体的个数:": [16, 0, 6, 14, 4, 0, 10, 0, 8, 12], "剩余箱体个数:": [2, 19, 7, 2, 6, 13, 0, 8, 4, 1], "利用率:": 0.602}
{"箱体类型:": "box10_4", "已使用容积:": 15228382.0, "已使用箱体的个数:": [12, 17, 16, 15, 12, 0, 0, 0, 0, 20], "剩余箱体个数:": [3, 0, 1, 4, 1, 19, 13, 21, 13, 2], "利用率:": 0.506}
{"箱体类型:": "box10_5", "已使用容积:": 17032304.0, "已使用箱体的个数:": [16, 0, 0, 12, 0, 10, 0, 12, 0, 16], "剩余箱体个数:": [0, 8, 14, 2, 16, 0, 14, 2, 14, 2], "利用率:": 0.566}
{"箱体类型:": "box15_1", "已使用容积:": 19778671.0, "已使用箱体的个数:": [6, 7, 10, 0, 4, 16, 1, 0, 6, 0, 11, 0, 0, 5, 5], "剩余箱体个数:": [0, 0, 0, 3, 1, 0, 11, 7, 0, 8, 0, 4, 5, 1, 1], "利用率:": 0.657}
{"箱体类型:": "box15_2", "已使用容积:": 17364281.0, "已使用箱体的个数:": [12, 0, 4, 0, 0, 12, 0, 6, 0, 0, 10, 6, 7, 0, 5], "剩余箱体个数:": [0, 12, 2, 9, 5, 0, 1, 4, 8, 3, 0, 1, 0, 10, 4], "利用率:": 0.577}
{"箱体类型:": "box15_3", "已使用容积:": 17505699.0, "已使用箱体的个数:": [9, 0, 5, 0, 0, 0, 10, 0, 12, 12, 0, 0, 0, 6, 10], "剩余箱体个数:": [4, 9, 3, 6, 10, 4, 0, 10, 0, 2, 9, 9, 2, 8, 1], "利用率:": 0.582}
{"箱体类型:": "box15_4", "已使用容积:": 15899296.0, "已使用箱体的个数:": [2, 4, 15, 0, 6, 0, 0, 8, 8, 0, 0, 10, 8, 1, 0], "剩余箱体个数:": [4, 2, 0, 8, 0, 7, 12, 2, 0, 11, 6, 4, 1, 10, 9], "利用率:": 0.528}
{"箱体类型:": "box15_5", "已使用容积:": 14207016.0, "已使用箱体的个数:": [4, 0, 0, 0, 5, 12, 0, 4, 12, 0, 0, 10, 0, 0, 0], "剩余箱体个数:": [4, 12, 5, 12, 4, 0, 8, 2, 0, 9, 11, 0, 8, 9, 13], "利用率:": 0.472}

箱体坐标，由于内容过大，此处只能展示部分，详见 data/dimensions.json 文件

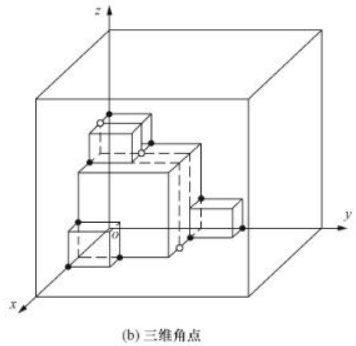


2、高级部分

1 理论研究

货物装箱问题属于完全多项式非确定性问题的范畴，即一种组合优化的NP 完全问题，为当前世界七大数学难题之一。根据对装箱问题已知信息的不同，可以分为在线装箱问题和离线装箱问题。其中，在线装箱问题是指仅知已经装入箱体的货物信息和当前待装入箱体的货物信息，来确定当前货物的放置位置。

首先对问题进行建模分析，图1摘自《求解货物在线装箱问题的融合算法》[1]，该文作者比较详细描述了在线装箱问题并对相关问题进行建模与求解。箱子是三维实体，需要使用某特定点表示箱体位置，论文中选取的表示方法为采取箱体的角点，即一个箱子的左、下、后点的坐标。为了更充分利用空间，每个箱体的候选位置都选取已放置箱体的顶点，如图1中标识的黑色点。



(b) 三维角点

图1 装箱问题的空间表示

因此，在每次放入新的箱体时，候选角点的位置都会进行更新。而一个箱子总共有6（3*2，

[1] 张长勇,刘佳瑜,王艳芳.求解货物在线装箱问题的融合算法[J].科学技术与工程,2021,21(11):4513-4518.

即3种类型的面，水平放置或竖直放置）种放置形态，设候选角点位置数量为m，则箱子的放置选择共有6m种。接下来需要在这6m个位置种选取最佳位置，需要有算法评估每种放法的优越程度，比较简单的实现是首个合适的位置放置，比较复制的方式是根据车厢的放置情况设计判别函数，之后使用遗传算法、模拟退火、2opt等算法得出较优的放置点。

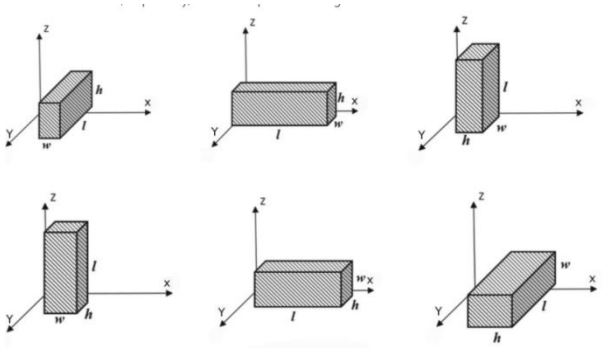


图2 箱体的六种放置形态

表1是论文中作者使用的评估位置点价值的方式，其公式 $F=0.4\frac{n_1}{N_0}+0.2\frac{n_2}{N_0}+0.2t_3$，接着通过模拟退火算法跳出局部最优。

表1 论文中的判别函数设计

序号	规则	权重	排序	原始值
1	轴优先级	0.4	z>y>x	n1/N0
2	待放货物体积与角点旁边空间容积差值	0.2	不减小	n2/N0
3	待放与角点左侧货物等	0.2	等高:0 不等高:1	0,1

模型构建比较容易，但是论文中提到的细节有限，想要复现较难，其中的难点是新放入箱体后如何寻找候选角点，论文中提出了由二维角点到三维角点的复现方式，但细节有限，因此最终未能复现。除了使用传统算法之外，还可以采用强化学习方式进行求解。

本次实验的代码部分采取较为简单的候选角点判断方式，即选取每个已放置箱体的固定点作为可能的候选角点，之后判定该点是否悬空，待放入箱体是否可以合法放入该点。候选角点的选取规则是优先选取坐标点离原点更近的点，即优先在角落堆放箱体，一个箱体有六种形态，因此需要尝试每一种形态是否可以合法放入，选则首先适应的状态即可。

2 代码实现

代码使用python实现，共设计3类，Item、Bin以及Packer，Item表现箱体，Bin代表车厢，

Packer代表装入操作。Item的含有width、length、weight等表示物体信息的属性，因为每个箱体有六种摆放形态，属性rotation_type表示箱体的角度，rotation_type确定后，箱体和车厢的长宽高的对应关系也随之确定，属性position代表箱体在车厢内的存放位置，即箱体左、后、下的坐标点，Bin的主要属性包括车厢的长宽高以及其中已经放入的箱体。

（一）箱体放入前，查看所有可能的放置位置。实现逻辑如下列代码，这一步寻找的范围比较粗略，遍历所有已经放入的箱体，选取和角点某一维坐标相同的顶点，即角点的上方，前方，右方的三个顶点作为候选位置，这些位置并非都是合法的，新的箱体放入这些位置可以使得车厢利用更加紧凑。

```
for ib in items_in_bin:
            pivot = [0, 0, 0]
            if axis == Axis.LENGTH:  # axis = 0/ x-axis
                pivot = [ib.position[0] + ib.get_dimension()[0],
                         ib.position[1],
                         ib.position[2]]
            elif axis == Axis.WIDTH:  # axis = 1/ y-axis
                pivot = [ib.position[0],
                         ib.position[1] + ib.get_dimension()[1],
                         ib.position[2]]
            elif axis == Axis.HEIGHT:  # axis = 2/ z-axis
                pivot = [ib.position[0],
                         ib.position[1],
                         ib.position[2] + ib.get_dimension()[2]]
```

（二）判别候选点的信息，并选择最佳候选点。该步骤是关键的一步，在本步骤中可以使用不同的算法，但由于对三维空间的建模比较困难，本实验采取简单的判别原则，设候选坐标为$P_0(x,y,z)$、$P_1(x,y,z)$、$P_2(x,y,z)$、$P_3(x,y,z)$.....，令

$value_i = \|p_i\|_{-\infty} = \min(x_i, y_i, z_i)$，选取拥有最小value的坐标点作为候选坐标点，当最低的维度相同时，比较第二个维度，选择数值相对较小的点，即选取的候选坐标点离原点在某一维度上最近，优先向角落里堆放物品。

（三）获得候选角点能够利用的三维空间，以边缘处的箱体Item为例，在x轴上，其可用空间为Bin.Length-(Item.position[0]+Item.Length),其主要代码如下所示，除了车厢的限制外，其它已放置的箱体也可能成为限制因素。

```
if (
        ib_neighbor.position[0] > ib.position[0] + ib.get_dimension()[0] and
        ib_neighbor.position[1] + ib_neighbor.get_dimension()[1] > ib.position[1] and
        ib_neighbor.position[2] + ib_neighbor.get_dimension()[2] > ib.position[2]
):
    right_neighbor = True
    x_distance = ib_neighbor.position[0] - (ib.position[0] + ib.get_dimension()[0])
    ib_neigh_x_axis.append(x_distance)

                    if not right_neighbor:
x_distance = bin.length - ib.position[0]
ib_neigh_x_axis.append(x_distance)
```

（四）选择箱体的放入角度，判断箱体是否会超过车厢范围，箱体的放入是否会导致箱体重叠以及箱体是否悬空放置，选择箱体放入角度的规则设计为，箱体在放入后，其某一维度剩余可利用空间最大，判别方式与第（二）步相同。下图是悬空判别函数，主要方法是查看放入点的纵坐标，若为0，则不悬空，若不为0，遍历所有已放入箱体，判断放置点是否处于其底层箱体的上表面中。

```
def judge_support_face(item, priovt):
    if priovt[2] == 0:
        return True
    position = item.position
    dimention = item.get_dimension()
    face_height = position[2] + dimention[2]
    if face_height != priovt[2]:
        return False
    else:
        if position[0] <= priovt[0] <= position[0] + dimention[0] and position[1] <= priovt[1] <= position[1] + \
                dimention[0]:
            return True
        else:
            return False
```

下图代码分别判顶装入后是否会超出车厢上限和判别箱体是否会重叠。

```
if (
    pivot[0] + dimension[0] <= self.length and  # x-axis
    pivot[1] + dimension[1] <= self.width and  # y-axis
    pivot[2] + dimension[2] <= self.height  # z-axis
):

    fit = True
```

```
def rect_intersect(item1, item2, x, y):
    d1 = item1.get_dimension()
    d2 = item2.get_dimension()

    cx1 = item1.position[x] + d1[x] / 2
    cy1 = item1.position[y] + d1[y] / 2
    cx2 = item2.position[x] + d2[x] / 2
    cy2 = item2.position[y] + d2[y] / 2

    ix = max(cx1, cx2) - min(cx1, cx2)
    iy = max(cy1, cy2) - min(cy1, cy2)

    return ix < (d1[x] + d2[x]) / 2 and iy < (d1[y] + d2[y]) / 2
```

为了验证代码的效率，编写Init_Car_box模块用以初始化Bin以及Item，所得箱体列表的顺序是随机的生成的，程序将会依次尝试装入箱体列表中的箱体，即相当于箱体的随机到达。如果当前箱体不能装入指定的候选点中，则直接放弃该箱体，装入下一个箱体。最后的填充率为$\frac{\sum_{i=1}^{n}\text{装入车厢的箱体体积}}{\text{车厢的体积}}$。

3 实验结论

实验中共给出了25组数据，但其中一组是重复的，故最终测试24组数据进行装入，其装入率稳定在0.65左右。本实验依旧存在很多缺陷，如当前箱体无法放入指定点后，实质上车厢仍旧有大量其余空间，可以放置，而且对箱体的状态判别也比较困难，因为箱体的大小和数量不确定，其摆放方式也有很多可能，一个新箱体摆放后对车厢空间的影响比较难以建模，想要获得更佳的结果，需要利用更多信息，引入更复杂的计算。

表1　在线算法放置不同箱体填充率与时间开销（单位：s）

箱体类别	1	2	3	4	5	6	7	8	9	10	11	12

填充率	0.65	0.67	0.69	0.68	0.64	0.67	0.64	0.60	0.57	0.65	0.65	0.66
识别时间（s）	0.08	0.04	0.18	0.20	0.03	0.10	0.07	0.10	0.06	0.05	0.13	0.07

箱体类别	13	14	15	16	17	18	19	20	21	22	23	24
填充率	0.61	0.60	0.61	0.68	0.60	0.67	0.62	0.69	0.62	0.65	0.60	0.64
识别时间(s)	0.09	0.23	0.12	0.16	0.14	0.23	0.14	0.09	0.08	0.11	0.11	0.11