武汉大学国家网络安全学院

实验报告

课程名称 高级算法

专业年级 2022级学硕1班

姓　　名 沈茹冰

学　　号 2022202210060

协 作 者 无

实验学期 2022-2023 学年 第一 学期

填写时间 2022 年 12 月

一、实验要求

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。

二、实验内容

装箱问题是典型的 NP 难问题，即在多项式的时间内找不到最优解，用传统的搜索技术求解这类问题，会产生“组合爆炸”现象。所以为了在合理的时间内找到近似最优解，一般多通过启发式方法来求解。

在本实验中采用基于“块”和“平面”的启发式算法进行求解，并采用贪心的策略来生成箱子摆放模式。

2.1 约束条件

该算法需要考虑的约束主要包括以下几个方面：

(1) 空间约束，即放入的箱子的边界不能超过车厢长宽高限制的范围。

(2) 稳定性约束，即没有悬空的箱子，上层货物需被下层货物完全支撑。

(3) 方向约束，即箱子可以以 6 个面中的哪几个面作为底面。

(4) 重量约束，即装入的箱子的总重量不能超过车厢限重，但在本次实验中不考虑限重问题。

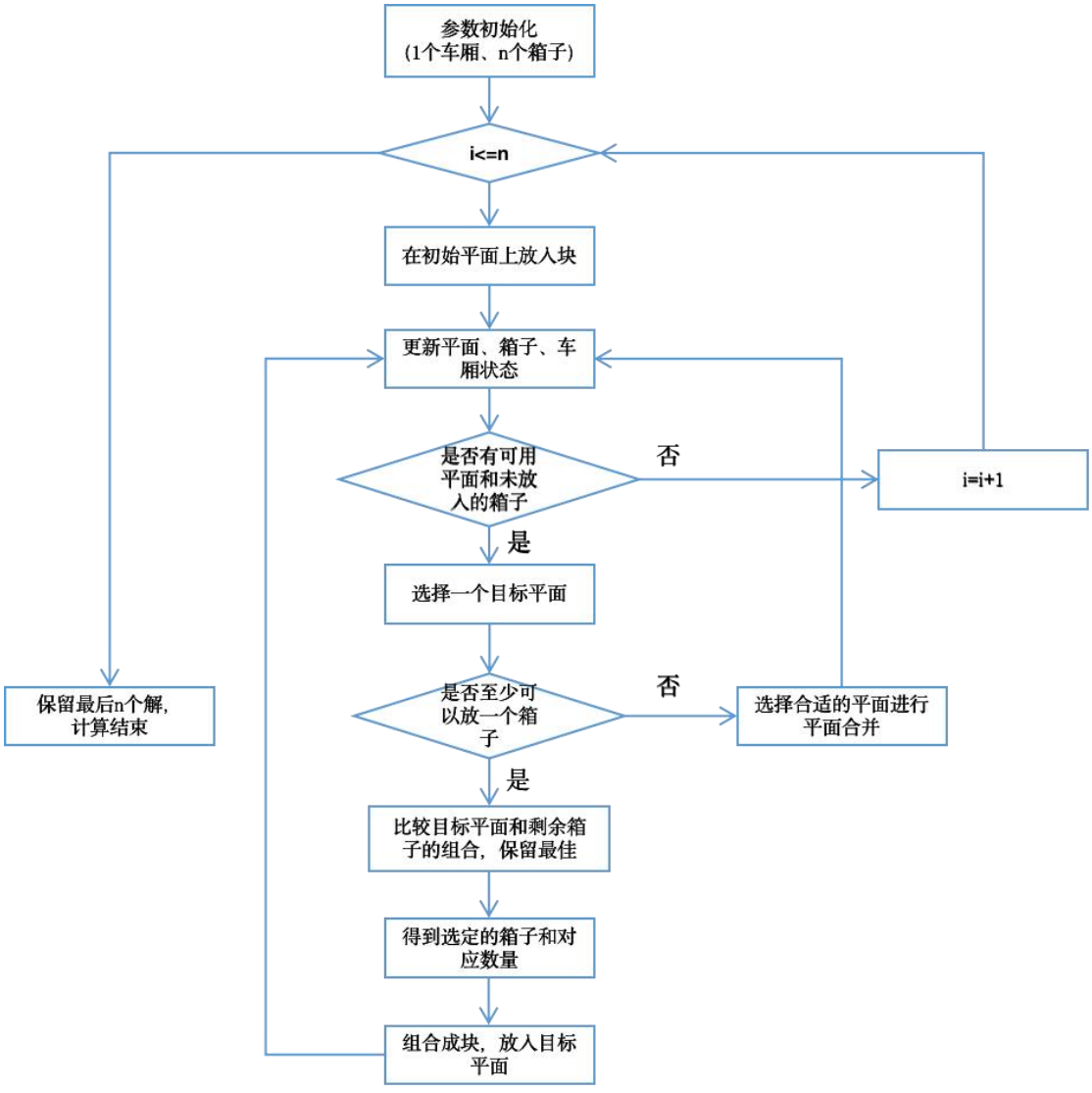
2.2 算法流程

这里采用自底向上的方式进行装箱，先铺满车厢底面，再向上堆放物品。算法的大致过程如下：首先由完全相同的箱子组成“块”，然后用块自底向上依次

填充车厢内的目标平面，并生成新的“平面”，不断重复上述过程。

其中，“块”指由相同类型的箱子按照相同摆放方向摆放所组成的单层结构，如果箱子不是同一类型或者摆放位置不同，都是指不同的块。“平面”指可用于摆放箱子的水平面，初始平面即车厢的整个底面，放入第一批箱子后，平面包括这批箱子顶面形成的面和车厢底面空余的部分。

算法流程图如下所示：



2.3 具体实现

（一）数据读取与存储

将数据处理成如下格式进行存储，共有 5 种规模的测试集，每种规模 5 个样例，共 25 个测试样例。[(l1, w1, h1, n1), (l2, w2, h2, n2)....] 分别表示各个种类箱子的长宽高和个数。同时，车厢的规模不变，(587 233 220)分别代表长宽高。

data.txt

```
E1-1 | [(108 76 30 40), (110 43 25 33), (92 81 55 39)]
E1-2 | [(91 54 45 32), (105 77 72 24), (79 78 48 30)]
E1-3 | [(91 54 45 32), (105 77 72 24), (79 78 48 30)]
E1-4 | [(60 40 32 64), (98 75 55 40), (60 59 39 64)]
E1-5 | [(78 37 27 63), (89 70 25 52), (90 84 41 55)]
E2-1 | [(108 76 30 24), (110 43 25 7), (92 81 55 22), (81 33 28 13), (120 99 73 15)]
E2-2 | [(49 25 21 22), (60 51 41 22), (103 76 64 28), (95 70 62 25), (111 49 26 17)]
E2-3 | [(88 54 39 25), (94 54 36 27), (87 77 43 21), (100 80 72 20), (83 40 36 24)]
E2-4 | [(90 70 63 16), (84 78 28 28), (94 85 39 20), (80 76 54 23), (69 50 45 31)]
E2-5 | [(74 63 61 22), (71 60 25 12), (106 80 59 25), (109 76 42 24), (118 56 22 11)]
E3-1 | [(108 76 30 24), (110 43 25 9), (92 81 55 8), (81 33 28 11), (120 99 73 11), (111 70 48 10)
E3-2 | [(97 81 27 10), (102 78 39 20), (113 46 36 18), (66 50 42 21), (101 30 26 16), (100 56 35 1
E3-3 | [(88 54 39 16), (94 54 36 14), (87 77 43 20), (100 80 72 16), (83 40 36 6), (91 54 22 15),
E3-4 | [(49 25 21 16), (60 51 41 8), (103 76 64 16), (95 70 62 18), (111 49 26 18), (85 84 72 16),
E3-5 | [(113 92 33 23), (52 37 28 22), (57 33 29 26), (99 37 30 17), (92 64 33 23), (119 59 39 26)
E4-1 | [(49 25 21 13), (60 51 41 9), (103 76 64 11), (95 70 62 14), (111 49 26 13), (85 84 72 16),
E4-2 | [(97 81 27 8), (102 78 39 16), (113 46 36 12), (66 50 42 12), (101 30 26 18), (100 56 35 13
E4-3 | [(86 84 45 18), (81 45 34 19), (70 54 37 13), (71 61 52 16), (78 73 40 10), (69 63 46 13),
E4-4 | [(113 92 33 15), (52 37 28 17), (57 33 29 17), (99 37 30 19), (92 64 33 13), (119 59 39 19)
E4-5 | [(118 79 51 16), (86 32 31 8), (64 58 52 14), (42 42 32 14), (64 55 43 16), (84 70 35 10),
E5-1 | [(98 73 44 6), (60 60 38 7), (105 73 60 10), (90 77 52 3), (66 58 24 5), (106 76 55 10), (
E5-2 | [(108 76 30 12), (110 43 25 12), (92 81 55 6), (81 33 28 9), (120 99 73 5), (111 70 48 12)
E5-3 | [(49 25 21 13), (60 51 41 9), (103 76 64 8), (95 70 62 6), (111 49 26 10), (74 42 40 4), (
E5-4 | [(97 81 27 6), (102 78 39 6), (113 46 36 15), (66 50 42 8), (101 30 26 6), (100 56 35 7),
E5-5 | [(113 92 33 8), (52 37 28 12), (57 33 29 5), (99 37 30 12), (92 64 33 9), (119 59 39 12),
```

```python
def read_data(path):
    box_list = []
    cnt = 0
    with open(path) as f:
        tls = [l.strip() for l in f ]
        for tl in tls:
            cnt = 0
            box_l = []
            tid, blist = tl.split('|')
            list = "".join(blist)[2:-1]
            list = list.split(', ')
            # print(list)
            for bid in list:
                index = bid.split('(')[1].split(')')[0].split(' ')
                # print(index)
                box = {}
                box['test_id'] = tid[:-1]
                box['lx'] = index[0]
                box['ly'] = index[1]
                box['lz'] = index[2]
                box['num'] = index[3]
                box['type'] = cnt
                cnt += 1
                box_l.append(box)
            box_list.append(box_l)

    return box_list
```

read_data()函数用来读取和存储数据，定义 box 字典来存储不同箱子，test_id 表示第几个测试集，lx、ly、lz、num 分别表示箱子的长、宽、高和数量，type 表示箱子的种类。

（二）选择平面

因为在本问题情景中，只有一个车厢，所以初始只有一个备选平面，即车厢底面。后续在堆放过程中，会不断产生新的平面，所以需要制定规则来决定选取哪个平面堆放箱子。

在该算法中，平面的选择标准如下：

1) 优先选择离车厢底面更近的水平面，即选取 z 坐标最小的平面（避免堆积过高，影响货物堆放的稳定性）
2) 如果几个平面的空间位置 z 相同，优先选择面积较小的平面（面积小的平面在后续可能更难利用）
3) 若 1）2）都相同，则比较参考点都 x 坐标，选取 x 坐标最小的平面。若 x 坐标也相同，则比较 y 坐标，选取 y 坐标最小的平面。

具体实现在 select_plane 函数中，主要分了以上几种情况进行讨论，以选择较优的平面来堆放箱子。

```python
# 选择平面
def select_plane(ps: PackingState):
    # 选最低的平面
    min_z = min([p.z for p in ps.plane_list])
    temp_planes = [p for p in ps.plane_list if p.z == min_z]
    if len(temp_planes) == 1:
        return temp_planes[0]

    # 相同高度的平面有多个的话，选择面积最小的平面
    min_area = min([p.lx * p.ly for p in temp_planes])
    temp_planes = [p for p in temp_planes if p.lx * p.ly == min_area]
    if len(temp_planes) == 1:
        return temp_planes[0]

    # 较狭窄的
    min_narrow = min([p.lx/p.ly if p.lx <= p.ly else p.ly/p.lx for p in temp_planes])
    new_temp_planes = []
    for p in temp_planes:
        narrow = p.lx/p.ly if p.lx <= p.ly else p.ly/p.lx
        if narrow == min_narrow:
            new_temp_planes.append(p)
    if len(new_temp_planes) == 1:
        return new_temp_planes[0]

   # x坐标较小
    min_x = min([p.x for p in new_temp_planes])
    new_temp_planes = [p for p in new_temp_planes if p.x == min_x]
    if len(new_temp_planes) == 1:
        return new_temp_planes[0]

    # y坐标较小
    min_y = min([p.y for p in new_temp_planes])
    new_temp_planes = [p for p in new_temp_planes if p.y == min_y]
    return new_temp_planes[0]
```

（三）生成块

当确定了目标平面、本次放入的箱子类型后，根据平面的大小和箱子的剩余数量计算该“块”中箱子的数量。设目标平面 X、Y 轴的长度分别为 L、W，待放入的箱子类型 i 的尺寸是 xi、yi、zi，此货物剩余 mi 个。优先在 x 方向上摆放，则 x、y 方向上的数量分别为：

$$Qx = \min\{m_i, \lfloor L / xi \rfloor\}$$

$$Qy = \min\{\lfloor m_i / Qx \rfloor, \lfloor W / yi \rfloor\}$$

则组成该块的箱子数量为$Q_x \times Q_y$。

```python
# 生成块
def gen_block(init_plane: Plane, box_list, num_list, max_height, can_rotate=False):
    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 <= init_plane.lx and box.ly * ny <= init_plane.ly and box.lz * nz <=
                        # 该简单块需要的立体箱子数量
                        requires = np.full_like(num_list, 0)
                        requires[box.type] = int(nx) * int(ny) * int(nz)
                        # 简单块
                        block = Block(lx=box.lx * nx, ly=box.ly * ny, lz=box.lz * nz, require_list=requ
                        # 简单块填充体积
                        block.volume = box.lx * nx * box.ly * ny * box.lz * nz
                        block_table.append(block)
                    # if can_rotate:
                    if True:
                        if box.ly * nx <= init_plane.lx and box.lx * ny <= init_plane.ly and box.lz * n
                            requires = np.full_like(num_list, 0)
                            requires[box.type] = int(nx) * int(ny) * int(nz)
                            # 简单块
                            block = Block(lx=box.ly * nx, ly=box.lx * ny, lz=box.lz * nz, require_list=
                            # 简单块填充体积
                            block.volume = box.ly * nx * box.lx * ny * box.lz * nz
                            block_table.append(block)
    return block_table
```

（四）块和平面组合规则

为了最大限度利用目标平面，块和平面的组合规则主要从“剩余面积”和“体积”考虑。首先考虑放入块之后目标平面剩余面积最小的组合，如果剩余面积相同，则比较块的体积大小，选择最大体积的块。具体实现在 find_block 函数中。

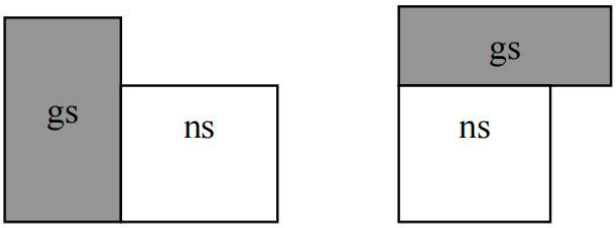
```python
# 查找下一个可行块
def find_block(plane: Plane, block_list, ps: PackingState):
    # 平面的面积
    plane_area = plane.lx * plane.ly
    # 放入块后，剩余的最小面积
    min_residual_area = min([plane_area - b.lx * b.ly for b in block_list])
    # 剩余面积相同，保留最大体积的块
    candidate = [b for b in block_list if plane_area - b.lx * b.ly == min_residual_area]
    # 可用平面最大高度
    max_plane_height = min([p.z for p in ps.plane_list])
    _candidate = sorted(candidate, key=lambda x: x.volume, reverse=True)

    return _candidate[0]
```

在每次向目标平面堆放“块”的过程中，采取**贪心思想**，每一步的填充过程中，都检测所有剩余的“块”和目标平面的组合效果，并将平面和块的最佳组合作为下次填充的对象。每一步只保留最佳的结果，这样的策略可以大大缩短计算时间，同时由于采用了贪心的思想，所得解的质量在可接受范围之内，即使无法取得全局最优解，但算法的时间复杂度较低，且获得的解是相对优的。

（五）合并平面

如果选择的平面不能容纳任何一个块，这时需要通过合并平面来增加空间的利用率。一个平面只能和相邻的平面进行合并，两者需要具有相同高度，至少有一边平齐。如下图所示，gs 表示目标平面，ns 表示相邻平面。首先 gs 在平面列表和备用平面列表中依次查找是否存在相邻平面 ns。其中，平面列表指还未使用的新平面列表，备用平面列表指已经计算过不能放入任何块的平面列表。



平面合并的过程将其拆分为两个子过程：试合并和正式合并。当目标平面找到相邻平面时，通过试合并决定是否保留。

设试合并后新产生的两块平面分别为 ms1 和 ms2，判断是否保留该次试合并及正式合并的算法流程如下：

（1）判断 ns 能否至少装入一个剩余的箱子：

a. 若不能，则判断 ms1、ms2 能否至少装入一个剩余的箱子；

a) 若 ms1、ms2 有平面满足，则将满足的平面插入平面列表，不满足的放入备用平面列表；

b) 若 ms1、ms2 均不满足，则判断合并后新平面的面积是否满足比原先两个平面都大；

i. 如果满足，则将 ms1 和 ms2 放入备用平面列表；

ii. 否则不保留这次试合并，重新寻找相邻平面；

b. 若能，转向（2）；

（2）计算放入箱子后 gs 和 ns 将耗费的总面积 ws1。根据 ms1、ms2 是否满足可以至少装下一个剩余箱子，计算合并后可能耗费的面积和 ws2，共有以下四种情况：

$$ws2 = WsSqr(ms1) + WsSqr(ms2)$$
$$ws2 = WsSqr(ms1) + Sqr(ms2)$$
$$ws2 = Sqr(ms1) + WsSqr(ms2)$$
$$ws2 = Sqr(ms1) + Sqr(ms2)$$

其中，第一种表示 ms1 和 ms2 都满足，第二种情况表示 ms1 满足、ms2 不满足，依此类推。

（3）比较 ws1 和 ws2 的大小：

a. 若 ws1>ws2，将平面合并，同时更新平面和备用平面列表；

b. 若 ws1<ws2，不保留此次试合并，继续寻找其他相邻平面。

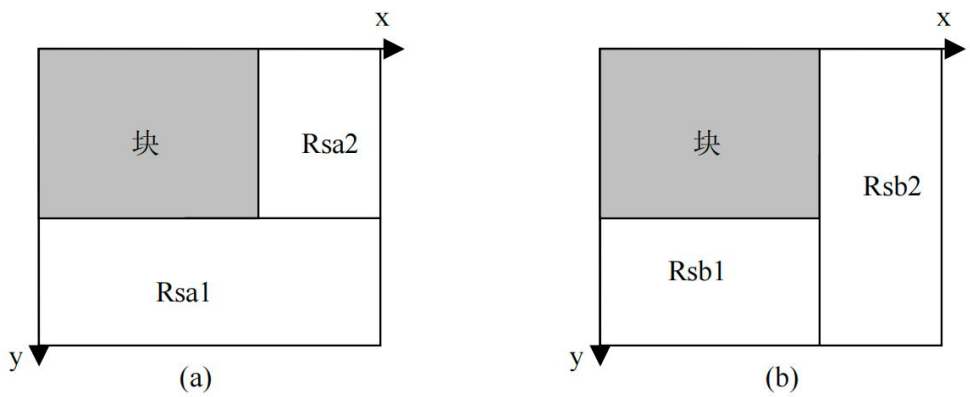
（4）若遍历所有平面列表和备用平面列表的元素后，仍没有找到可合并的平面，则将目标平面 gs 从平面列表移入备用平面列表。

合并平面部分的具体代码实现在 merge_plane()函数中，如下图所示：

```python
# 合并平面
def merge_plane(ps: PackingState, plane: Plane, block_table, max_height):
    for ns in ps.plane_list + ps.spare_plane_list:
        # 不和自己合并
        if plane == ns:
            continue
        # 找相邻平面
        is_adjacent, ms1, ms2 = plane.adjacent_with(ns)
        if is_adjacent:  # 有相邻平面
            block_list = gen_block_list(ns, ps.avail_list, block_table, max_height)
            # 相邻平面本身能放入至少一个剩余物品
            if len(block_list) > 0:
                block = find_block(ns, block_list, ps)
                # 计算相邻平面和原平面浪费面积的总和
                ws1 = ns.lx * ns.ly - block.lx * block.ly + plane.lx * plane.ly
                # 计算合并后平面的总浪费面积
                ws2 = plane_waste(ps, ms1, block_table, max_height) + plane_waste(ps, ms2, block_table, max_height)
                # 合并后，浪费更小，则保留合并
                if ws1 > ws2:
                    # 保留平面合并
                    ps.plane_list.remove(plane)
                    if ns in ps.plane_list:
                        ps.plane_list.remove(ns)
                    else:
                        ps.spare_plane_list.remove(ns)
                    if ms1:
                        ps.plane_list.append(ms1)
                    if ms2:
                        ps.plane_list.append(ms2)
                    return
                else:
                    # 放弃平面合并，寻找其他相邻平面
                    continue
```

（六）新平面产生与划分

在每次放入“块”之后，原目标平面被分成三个子平面，如下图左右两种情况所示。所以这里需要确定如何对剩余平面进行划分以产生新平面。这里使用最大面积的方式来划分子平面，即对于两种划分方法，选择产生最大面积子平面的生成方法。



这部分具体实现在 gen_new_plane()函数中，如下所示。

```python
# 裁切出新的剩余空间（有稳定性约束）
def gen_new_plane(plane: Plane, block: Block):
    # 块顶部的新平面
    rs_top = Plane(plane.x, plane.y, plane.z + block.lz, block.lx, block.ly)
    # 底部平面裁切
    if block.lx == plane.lx and block.ly == plane.ly:
        return rs_top, None, None
    if block.lx == plane.lx:
        return rs_top, Plane(plane.x, plane.y + block.ly, plane.z, plane.lx, plane.ly - block.ly), None
    if block.ly == plane.ly:
        return rs_top, Plane(plane.x + block.lx, plane.y, plane.z, plane.lx - block.lx, block.ly), None
    # 比较两种方式中面积较大的两个子平面，选择有最大面积子平面的生成方式
    rsa1 = Plane(plane.x, plane.y + block.ly, plane.z, plane.lx, plane.ly - block.ly)
    rsa2 = Plane(plane.x + block.lx, plane.y, plane.z, plane.lx - block.lx, block.ly)
    rsa_bigger = rsa1 if rsa1.lx * rsa1.ly >= rsa2.lx * rsa2.ly else rsa2
    rsb1 = Plane(plane.x, plane.y + block.ly, plane.z, block.lx, plane.ly - block.ly)
    rsb2 = Plane(plane.x + block.lx, plane.y, plane.z, plane.lx - block.lx, plane.ly)
    rsb_bigger = rsb1 if rsb1.lx * rsb1.ly >= rsb2.lx * rsb2.ly else rsb2
    if rsa_bigger.lx * rsa_bigger.ly >= rsb_bigger.lx * rsb_bigger.ly:
        return rs_top, rsa1, rsa2
    else:
        return rs_top, rsb1, rsb2
```

（七）启发式算法总流程

启发式算法的总函数在 basic_heuristic()中，首先是由同样大小的箱子生成块。初始时，在剩余空间堆栈中只有车厢本身。然后循环遍历平面列表，选择平面，并查找可用块。如果存在可用块，则用近似最优块填充平面，并更新车厢的最大使用高度；如果不存在可用块，则合并相邻平面。这一过程循环直到平面使用完毕。

```python
# 启发式算法
def heuristic(problem: Problem):
    # 生成简单块
    block_table = gen_block(problem.container, problem.box_list, problem.num_list, problem.height
    # 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.plane_list.append(Plane(problem.container.x, problem.container.y, problem.container.z, pro
    max_used_high = 0
    # 循环直到所有平面使用完毕
    while ps.plane_list:
        # 选择平面
        plane = select_plane(ps)
        # 查找可用块
        block_list = gen_block_list(plane, ps.avail_list, block_table, problem.height_limit)
        if block_list:
            # 查找下一个近似最优块
            block = find_block(plane, block_list, ps)
            # 填充平面
            fill_block(ps, plane, block)
            # 更新最大使用高度
            if plane.z + block.lz > max_used_high:
                max_used_high = plane.z + block.lz
        else:
            # 合并相邻平面
            merge_plane(ps, plane, block_table, problem.height_limit)
```

2.4 实验结果

对测试集中的所有样例都进行了测试，并计算车厢的填充率和算法的执行时间。下图两个结果分别是是否考虑箱子摆放状态的结果。

（1）箱子的最大面朝下摆放

```
(bbp) shenrubing@shenrubingdeMacBook-Pro code % python main.py
E1-1  62.26  0.05s
E1-2  70.722  0.02s
E1-3  70.722  0.02s
E1-4  78.664  0.06s
E1-5  62.856  0.07s
E2-1  67.972  0.03s
E2-2  71.651  0.08s
E2-3  61.713  0.08s
E2-4  72.164  0.06s
E2-5  66.517  0.04s
E3-1  64.228  0.06s
E3-2  73.963  0.06s
E3-3  66.921  0.07s
E3-4  71.149  0.10s
E3-5  58.074  0.19s
E4-1  67.038  0.22s
E4-2  65.499  0.11s
E4-3  69.677  0.15s
E4-4  66.392  0.21s
E4-5  73.351  0.20s
E5-1  75.856  0.21s
E5-2  75.218  0.28s
E5-3  75.39  0.37s
E5-4  63.4  0.19s
E5-5  70.72  0.31s
```

从上图实验结果可以看出算法的执行速度都很快，时间都在 1s 内。车厢的填充率最低为 58.074%，最高 78.664%，普遍在 70%左右。

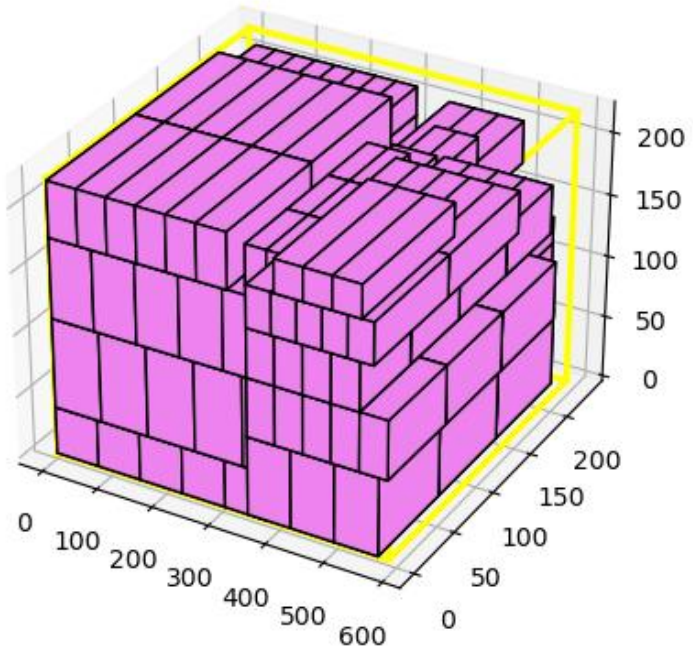
（2）考虑箱子不同的摆放状态

```
(bbp) shenrubing@shenrubingdeMacBook-Pro code % python main.py
E1-1  80.413  0.11s
E1-2  79.169  0.05s
E1-3  79.169  0.05s
E1-4  83.286  0.07s
E1-5  71.311  0.14s
E2-1  70.813  0.08s
E2-2  82.658  0.14s
E2-3  67.3  0.08s
E2-4  74.134  0.10s
E2-5  80.352  0.09s
E3-1  69.42  0.13s
E3-2  78.216  0.18s
E3-3  73.445  0.17s
E3-4  73.788  0.31s
E3-5  66.152  0.31s
E4-1  66.606  0.34s
E4-2  71.513  0.24s
E4-3  68.896  0.27s
E4-4  82.502  0.52s
E4-5  68.333  0.29s
E5-1  74.231  0.35s
E5-2  79.034  0.48s
E5-3  74.048  0.62s
E5-4  73.123  0.59s
E5-5  75.424  0.61s
```

从上图实验结果可以看出，当考虑箱子不同的摆放状态时，车厢填充率普遍提高，最高达 83.286%，最低为 66.152%。同时，算法执行时间普遍比单状态的时间久，但执行时间仍在 1s 内，可以实时得出结果。

（3）装箱结果可视化

同时为了更直观的看箱子在车厢中的摆放情况，对每个测试样例的摆放结果进行可视化，其中一个结果如下图所示：



三、实验总结

在本次实验中，使用基于块和平面的启发式算法解决了三维装箱问题，实验完成了基础内容和部分高级内容。由于基于生成块的算法，不适合转换为在线装箱，所以没有实现在线版本。但在实现的装箱算法中考虑了：

1）箱子的摆放顺序，按照从内到外、从下到上的摆放顺序；

2）由于箱子有 3 个不同的面，故共有 3 种不同的摆放状态；

3）算法的实时性，从实验结果可以看出算法执行时间均小于 1s；

在实验过程中，对启发式算法、贪心算法等思想也有了更进一步的认识和思考，通过在一个具体情境下运用其解决问题，而不仅仅是书本上的例题知识，也学习了如何灵活运用这些算法，如何设计约束条件、算法流程等来解决实际问题，感觉有更大收获。