

# 人工智能原理大作业一：推箱子游戏

吴见灼 2021013460

## 1. 项目结构说明

此项目实现了作业要求的全部必做任务，以及，为了使游戏可视化方便调试，我还为此项目编写了交互界面。我所做的工作包括：

- (1) **UI:** 在 UI.py 文件中，我使用 pygame 编写了游戏界面，人类玩家可以游玩已有的地图，并可以选择让 AI 计算地图的可行解并可视化；
- (2) **箱子推到洞口不会消失，且箱子不存在与洞口一一对应的关系时的搜索算法:** 在 utility.py 文件中，我使用 A\*搜索+剪枝策略+数据结构优化搭建了智能体 (trivial\_astar\_search)，这个算法作为本次项目中智能体主要使用的算法。同时，为了减少状态数，我加入了路径搜索，使得智能体搜索的次数进一步减少 (advanced\_astar\_search)。最后，为了解决空旷地图下状态数过多，搜索过慢的问题，我提出了对本问题状态空间新的理解并提出新的算法 (more\_advanced\_astar\_search)，最终在空旷地图下取得了很好的效果。
- (3) **箱子推到洞口会消失，且箱子与洞口一一对应时的搜索算法:** 在 (2) 的基础上，我通过修改算法的启发式函数并调整游戏的相关逻辑，实现了搜索算法 (advanced\_astar\_search\_mode2)
- (4) **初始场景生成:** 在 make\_map.py 文件中，输入随机地图相关的参数（地图大小，箱子数量，游戏模式），即可生成随机地图。

相关代码存放于 Github::alicebob142857/Sokoban 中。

## 2. UI 说明

运行可执行文件，或者在安装项目依赖后运行 UI.py 文件，即可打开游戏界面，界面右下角有游戏的简要说明以及开发者信息。进入界面后，首先选择人类玩家体验游戏或者 AI 对运行结果进行可视化。



随后进行模式选择，arbitrary holes 表示“箱子推到洞口不会消失，且箱子不存在与洞口一一对应的关系”的模式，given holes 表示“箱子推到洞口会消失，且箱子与洞口一一对应”的模式。



完成模式选择后，进入关卡选择，点击包含数字的矩形框即可进入对应的关卡。在选择游玩模式和关卡的过程中，可以按‘q’回退到上一个选项。



人类玩家使用 wsad 控制小骑士推箱子，当箱子挡住洞口时，长按‘e’可以隐藏小骑士和箱子，以便观察洞口位置；按‘q’可退回开始界面。而 AI 模式下，将在计算后自动可视化智能体移动的过程，如果 AI 求解超时，将返回‘count limited’；如果地图无解，将返

回’No Solution’，如果求解成功，将返回求解所需要的循环次数。AI 可视化播放速度可以在 settings.json 配置文件中更改。



### 3. 搜索算法介绍

本项目智能体基于 A\*算法实现。为了进一步提高算法的效率，我对算法进行改进，最后取得了较好的结果：

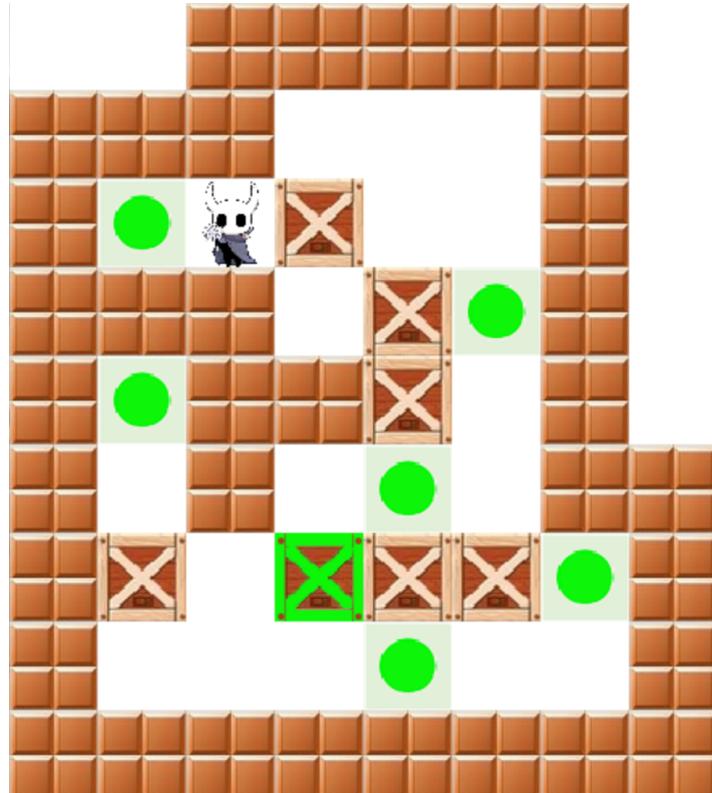
#### (1) trivial astar search

此算法较为平凡地使用地图中玩家的位置以及箱子的位置列表作为状态。玩家不断搜索其周围四个位置是否可行并进行移动。此算法中使用的启发式函数为每一个箱子与距离最近的洞口之间的曼哈顿距离之和，具体实现如下：

```
def distance(boxes, ends):  
    """  
        计算当前的价值函数。原理是，计算每个箱子到所有目标点的最短曼哈顿距离，然后对所有箱子进行求和  
        1. 对箱子到达指定位置进行奖励  
    """  
  
    result = 0  
    for box in boxes:  
        result += min([manhattan_distance(box, end) for end in ends])  
    return result
```

可以证明，此启发式函数一定小于真实的代价。而由于最近的曼哈顿距离在智能体运动的过程中变化是“光滑”的，不会出现突变，所以这个启发式函数能满足可用性和一致性，能得到最优解。与之相配合的，智能体每一步代价为 1。

最终该算法运行示例代码运行示例地图(地图 0)花费时间约为 2.2s，进行循环 10763 次



图片 1：示例地图

## (2) advanced astar search

在完成搜索算法后，我有以下思考：

在本任务中，本质上玩家与环境的交互仅限于推箱子，如果仅仅以玩家的移动作为我们的 action 的话，将很大程度忽视了箱子，墙壁，洞口等元素之间的交互信息。从这点上来看，与其说推箱子是玩家的搜索问题，不如说是箱子之间的“约束满足问题”。本质上不是人在动，而是箱子在可行的方向上移动，并与周围环境产生约束。

基于此我提出了改进算法：

我们将 action 视为箱子的移动，可行的推动位置由箱子和周围元素的位置关系得到，随后通过搜索玩家到此推动位置的路径，判断其是否可达并记录下此路径。

代码中核心的逻辑如下：

```
for j in range(len(current_boxes)):
    box = current_boxes[j]
    possible_moves = box_possible_point(current_map, box, width,
height)
    for possible_move in possible_moves:
        possible_point = get_movable_point(box, possible_move)
```

```

    box_path = bidirection_maze_search(current_map, current_pos,
possible_point, width, height)
    if box_path:
        ...

```

在搜索路径的过程中，为了加快这一步的搜索速度，我们使用双向 A\*搜索 (bidirection\_maze\_search)，维护两个开闭节点集，在每一端节点被优先级队列弹出时判断其是否在另一端的闭节点集中，从而实现更快地寻路。

```

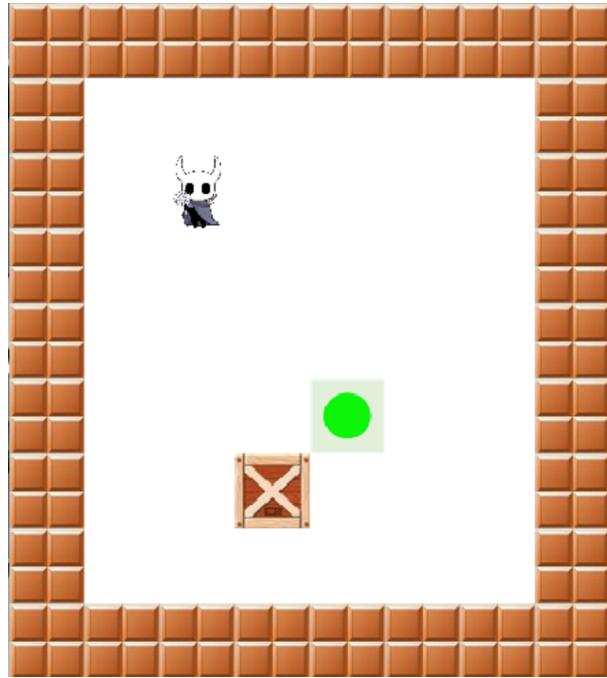
current_pos, path, direction = q.pop()
if direction:
    if bool(closed2) and do_hash(current_pos) in closed2.keys():
        new_path = closed2[do_hash(current_pos)][:]
        new_path.reverse()
        new_path = path + new_path
        return new_path
    if current_pos == end_point:
        return path
    directions, succ_pos = simple_move_directions(current_pos, map,
width, height)
    for i in range(len(directions)):
        if do_hash(succ_pos[i]) not in closed1:
            new_path = path + [directions[i]]
            new_cost = Manhattan_distance(succ_pos[i], end_point) +
len(new_path)
            q.push(new_cost, (succ_pos[i], new_path, True))
            closed1[do_hash(succ_pos[i])] = new_path

```

运行示例地图，我们所需时间约为 4.4s，循环次数为 5576 次。可见所需的循环次数大幅度减小。而所需时间的增加也是可预见的：因为示例地图很多情况下，智能体只有一种移动方向，而且箱子的可推动的方向也很少，使用改进算法势必会造成大部分时间用在判断路径的是否可达上。

然而当我们关注比较空旷的地图时，改进算法将不会浪费时间在空地盲目的探索上，相反，通过更快速的寻路算法，智能体可以精准地找到能够推动箱子的位置，每一步都将用于箱子的推动。在较为空旷的地图（地图 1）下，改进算法效率很高：

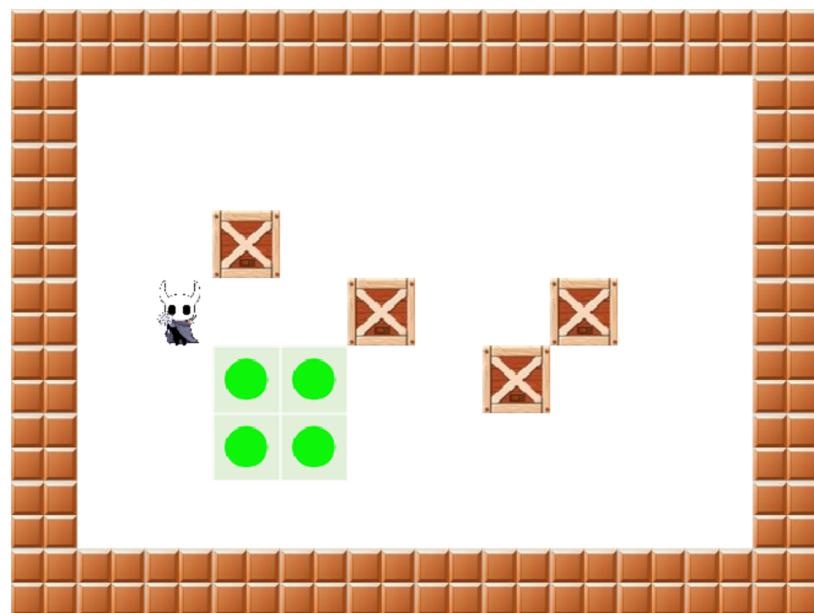
	time	Cycle numbers
trivial	0.003s	49
advanced	0.0006s	4



图片 2：比较空旷的地图

### (3) more advanced astar search

尽管改进算法在空旷地图很优秀，但是其仍然无法在能够接受的时间内运行更空旷的地图（地图 4）。究其原因，我分析如下：



图片 3：更空旷的地图

- (a) 改进算法重新定义了 action，但是对 state 的改进仍然不够，本质上其状态仍然是箱子的位置。但是箱子的坐标是无法提供更多的约束信息的。所以

在进一步改进中，我将箱子和环境的交互矩阵作为状态。交互矩阵即，箱子在其四个方向上发出一条射线，记录这条射线最终到达的元素的种类（墙，另一个箱子，洞口），以及其所经过的距离，这样能更好地表征出元素之间的约束。当然，为了减小空间复杂度，我使用一定规律对此信息进行编码，并在将状态加入优先级队列时仍然以坐标列表的形式加入，只是在弹出时再求取其交互矩阵。例如，左向的交互向量求解如下：

```
Def left_detection(current_map, possible_move, box, end_list,
width, height):
    count = 0
    if box[0] > 0:
        while box[0] > count + 1 and current_map[box[0] - count
- 1, box[1]] == 0 and (box[0] - count - 1, box[1]) not in
end_list:
            count += 1
            tmp_count = count
            if (box[0] - tmp_count - 1, box[1]) in end_list:
                index = end_list.index((box[0] - tmp_count - 1,
box[1]))
                count += height * (index + 2)
            if current_map[box[0] - tmp_count - 1, box[1]] == 2:
                count += height
    return count
```

在求取状态矩阵之后，我们在移动箱子时观察其移动方向正交方向的状态变化：如果正交方向状态改变（即其对应的元素种类改变，比如右侧从墙变为洞口，或者其距离改变，比如在拐角处从贴墙变为距离墙 5 个单位），则视为进入下一个状态；否为视为状态未发生变化，箱子继续向其移动方向移动，直到发生变化为止（keep\_move 函数）。以下为 keep\_move 函数的核心逻辑：

```
Info = [info1, info2]
tmp_info = info[:]
count = 0
while True:
    new_box = box_move(new_box, direction, width, height)
    new_box_list[current_box_index] = new_box
    if direction == Move.UP or direction == Move.DOWN:
        info1 = left_detection(map, Move.LEFT, new_box, end_list,
width, height)
        info2 = right_detection(map, Move.RIGHT, new_box, end_list,
width, height)
    else:
```

```

    info1 = up_detection(map, Move.UP, new_box, end_list, width,
height)
    info2 = down_detection(map, Move.DOWN, new_box, end_list,
width, height)
    info = [info1, info2]
    if not new_box:
        new_box = tmp_new_box[:]
        new_box_list = tmp_new_box_list[:]
        info = tmp_info[:]
        return new_box_list, new_current_pos, new_path, count
    if map[new_box] != 0:
        new_box = tmp_new_box[:]
        new_box_list = tmp_new_box_list[:]
        info = tmp_info[:]
        return new_box_list, new_current_pos, new_path, count
    if info != tmp_info and count >= 1:
        new_box = tmp_new_box[:]
        new_box_list = tmp_new_box_list[:]
        info = tmp_info[:]
        return new_box_list, new_current_pos, new_path, count
    if new_box in end_list and info == tmp_info:
        new_path.append(direction)
        new_current_pos = box_move(new_current_pos, direction, width,
height)
    return new_box_list, new_current_pos, new_path, count + 1

```

这样直观的理解是：如果智能体进入一片空旷的空间，它将一直往前走，直到其左边或右边地形变化，或者遇到了可交互的方块（其他箱子，洞口）为止；或者智能体通过狭长的通道时，将一直走到其距离底部一格为止。

- (b) 同时，本任务中，当启发式函数为曼哈顿距离时，观察程序的运行，可以发现此时更加偏向“广度优先搜索”，即智能体在其出发点周围逐渐探索。但是我们对最优的评判不一定是行走的步数，也可以是推箱子的次数，所以启发式函数不一定要拘泥于曼哈顿距离和已行走的步数。我们使用箱子推动的次数作为代价，每个箱子和距离最近的洞口之间不同坐标值的数量之和作为启发式函数。之所以这样设计，是因为理想情况下，智能体可能一次走很远的路程，如果智能体能像象棋中的车一样走直线距离，那么它最快也需要 1 或者 2 次才能到达目标。启发函数代码如下：

```

def special_distance(boxes, ends):
    """
    这个只统计重合数量
    """
    result = 0
    set1 = set(boxes)
    set2 = set(ends)
    diff1 = set1.difference(set2)
    diff2 = set2.difference(set1)
    for ele1 in diff1:
        min_dist = float('inf')
        min_ele = None
        for ele2 in diff2:
            current_dist = Manhattan_distance(ele1, ele2)
            if current_dist < min_dist:
                min_ele = ele2
                min_dist = current_dist
        if min_ele:
            if ele1[0] == min_ele[0] or ele1[1] == min_ele[1]:
                result += 1
            else:
                result += 2
    return result

```

最终我们运行地图 4, trivial\_astar\_search 和 advanced\_astar\_search 都无法在可以接受的时间内求出结果（时长超过 1min，循环超过 20w 次）。而 more\_advanced\_astar\_search 可以在 7.9s 内运行出结果，同时只需要 4101 次循环。同时，令人惊喜的是，改进后的算法运行示例地图（地图 0），也需要 1.3s，循环 872 次，相对 trivial\_astar\_search 时间和循环次数都大大优化。

#### (4) advanced astar search mode 2

此算法是在 advanced astar search 算法的基础上，稍微修改以完成“箱子推到洞口会消失，且箱子与洞口一一对应”的任务。除了游戏逻辑上的修改以及重写了部分工具类函数，算法上仅仅是替换了启发式函数，计算箱子和其对应洞口的曼哈顿距离。由于箱子消失导致其搜索的步数大大减小，所以即使在不剪枝的情况下，示例地图也能在 0.2s 得到结果，循环次数为 435 次。

```

Def mode2_distance(box_list, end_list):
    num = len(end_list)
    dist = 0

```

```

for i in range(num):
    dist += manhattan_distance(box_list[i], end_list[i])
return dist

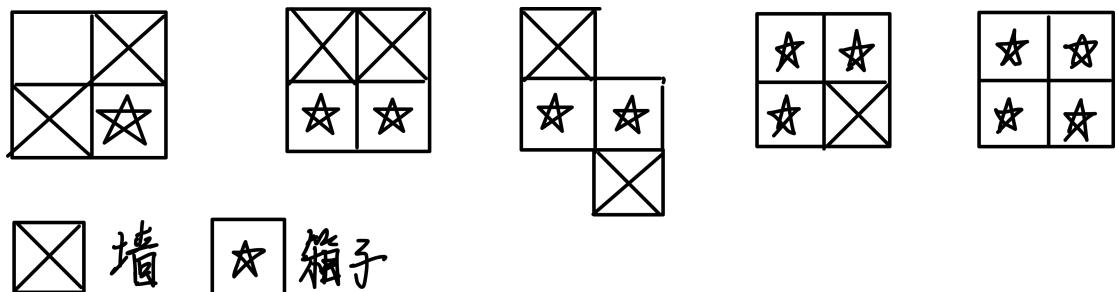
```

## 4. 优化技巧介绍

为了提高算法运行速度，在项目中我还使用了一些技巧进行优化：

### (1) 剪枝策略

推箱子游戏中一些失败的情况可以提前进行判断，一些节点可以被剪枝以提高智能体运行速度。以下是常见的失败情况：



在程序中，我们检查以箱子为中心  $3 \times 3$  的范围内是否包含以下构型。

### (2) 数据结构优化

在维护开闭节点集时，需要反复对元素进行查找。列表的时间复杂度为  $O(n)$ ，而 `set()` 和 `dist()` 内部都是以哈希表实现的，查找的时间复杂度为  $O(1)$ 。在优化数据结构之后，算法运行的速度大幅提升。

```

Def do_hash(target):
    """
    将列表转化为整型元组进行哈希查询
    """
    # 对 x, y 坐标进行进制转化
    return tuple([x[0] * width + x[1] for x in target])

```

我们将开闭节点中的坐标作为一个以 `width` 为进制的整型进行表示，实现了坐标和整数值的一一对应，然后转化为元组即可在哈希表中查询。

## 5. 地图生成算法

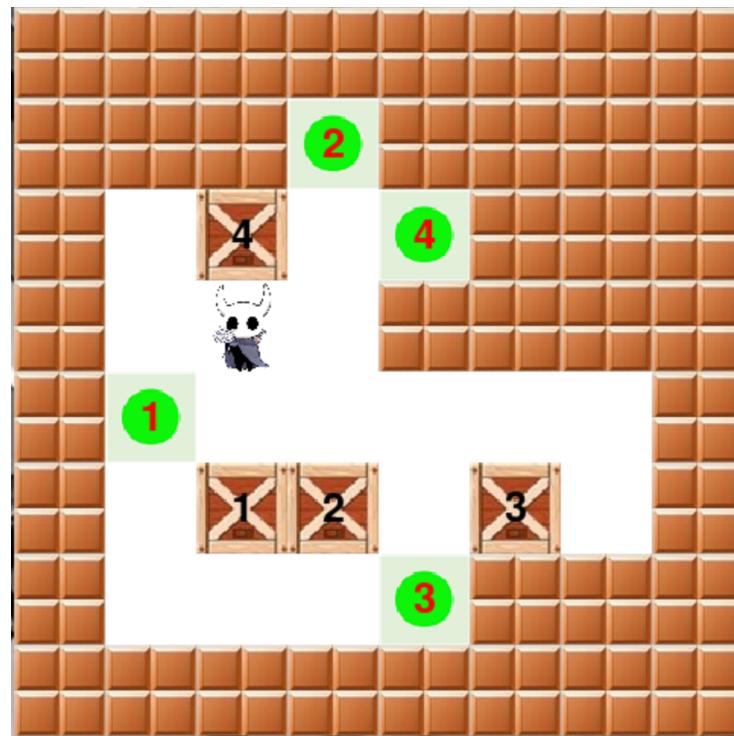
得益于 more advanced astar search，程序能够在很短的时间内完成在空旷地图下的搜索。生成地图时，我们给定生成地图的大小，箱子的数量，通过随机数在地图距离边界两个

以内随机生成箱子和洞口（地图外围为围墙，箱子和墙之间间隔一格将使可解的概率大大增加）。随后我们使用算法在“箱子推到洞口不会消失，且箱子不存在于洞口一一对应的关系”的模式下进行求解，使用一个掩码矩阵记录智能体和箱子移动中经过的位置，并在搜索结束后将为未经过的位置设置为墙。如果设计“箱子推到洞口消失，且箱子与洞口一一对应”的模式的地图，则记录搜索完成时箱子和洞口对应关系，并恢复到初始的矩阵中即可。个人认为这个功能属于开发者模式，所以并未集成进 UI 中。如果无解或者超时（循环 10000 次），那么程序将返回生成失败，需要再次运行。经实测无解概率很低，大约在 20% 以下。添加墙以及生成模式 2 地图的核心逻辑如下：

```
res, count = more_advanced_astar_search(map, end, initial_pos, width, height)
if mode == 'given_map':
    map = given_mode_map.copy()
print(res)
if res and res != -1:
    tmp_map = map.copy()
    mask[map != 0] = 1
    mask[end != 0] = 1
    for direction in res:
        position = box_move(initial_pos, direction, width, height)
        initial_pos = position
        if tmp_map[position] == 2 or tmp_map[position] % 10 == 2:
            next_position = box_move(position, direction, width, height)
            mask[next_position] = 1
            mask[position] = 1
            tmp_map[next_position] = tmp_map[position]
            tmp_map[position] = 0
        else:
            mask[position] = 1

    map[mask == 0] = 1
    if mode == 'given_map':
        same_indices = np.where(tmp_map > 10)
        coordinates = [(x, y) for x, y in zip(same_indices[0], same_indices[1])]
        for coordinate in coordinates:
            end[coordinate] = int(tmp_map[coordinate] / 10)
return map, end
else:
    return None, None
```

生成的地图如下：



图片 4：自动生成的地图

## 6. 总结

在这次项目中，我通过不断改进算法，最终完成了作业要求的任务，并编写了 UI 以便观察。在此过程中不仅锻炼了我编程的能力，也让我复习了数据结构，oop 等课程的知识。更重要的是，在不断改进算法的过程中，我更明白了 A\*算法的原理，A\*算法仅仅是能让我们最快地向我们估计的方向进行搜索，同时保证搜索过程中一定能最优，针对具体问题，状态空间和动作的设计，启发式函数的估计，对算法效率的影响是巨大的。

这次项目我的心路历程也是非常曲折，我由于找地图时忘记了输入墙的位置，阴差阳错得到了十分空旷地图 4，完成基础的 A\*算法之后只有这个地图一直无法完成，和它较劲也耗费了我很多时间。但是一次次通过思考改进算法，最后解决了空旷地图的问题的时候，那种喜悦和兴奋是难以忘怀的。

最后，感谢 GitHub::pengc02/Sokoban-Game 对数据结构优化策略的启发。感谢老师和助教对本课程的辛勤付出。