人工智能基础项目 1：推箱子

杨欣怡
自 13 2021011026

2024 年 4 月 17 日

1. 工程结构

本次项目实现了推箱子游戏，工程文件已同步上传到 Github（https://github.com/yyxxyy574/POAI24_project_1）。项目的设计分为 UI 界面、地图类、搜索算法三部分，各部分主要功能以及之间的相互关系如下图所示：

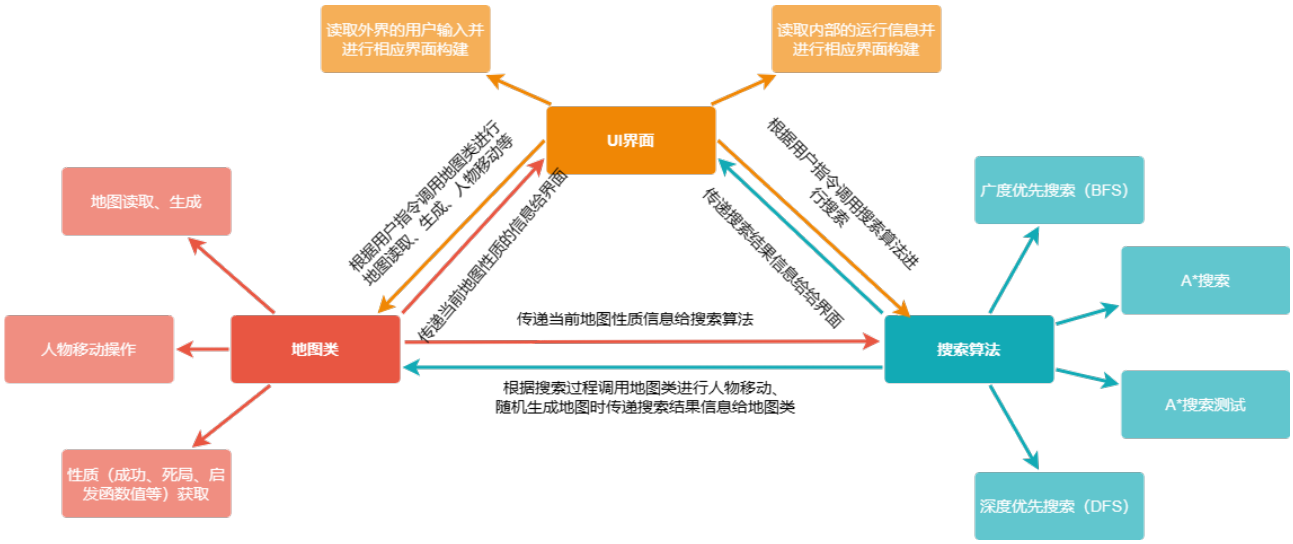

图 1: 项目设计示意图

另外，已提交文件的结构主要如下所示，可对照查看：
（1）code

- Assets：UI 界面所使用的图片和音效
- Maps：存储 demo1、demo2 的地图文件 map1.json、map2.json
- Solver：分别实现 A* 搜索算法、广度优先搜索算法、深度优先搜索算法的 astar.py、bfs.py、dfs.py
- map.py：定义地图类，包含地图读取、地图随机生成、移动、判断成功、判断死局、计算启发函数等功能
- sokoban.py：定义界面流程类，利用 pygame 实现开始界面、游戏模式选择界面、地图模式选择界面、主游戏界面、失败界面、死局界面，串联各个界面实现完整游戏流程
- sokoban.ico：游戏图标
- sokoban.exe：游戏可执行程序

（2）demo

- 1_ 游戏模式 _ 搜索算法.mp4：demo1 的 AI 搜索出的推箱子方法
- i_ 游戏模式 _map.png：demoi 的地图
- i_ 游戏模式.png：demoi 的 AI 搜索出的最优推箱子步数和花费的搜索时间
- random_ 游戏模式 _n_map.png：随机生成的箱子数量为 n 的地图
- random_ 游戏模式 _n.png：随机生成的箱子数量为 n 的地图对应的 AI 搜索出的最优推箱子步数和花费的搜索时间
- fail.png：死局界面
- instruction.png：游戏说明界面
- select_map.png：地图模式选择界面
- select_mode.png：游戏模式选择界面
- start.png：开始界面
- success.png：成功界面

接下来分别对项目三个部分主要的设计思路、实现原理进行阐述。

2. UI 界面设计

游戏界面的流程示意如下图所示：

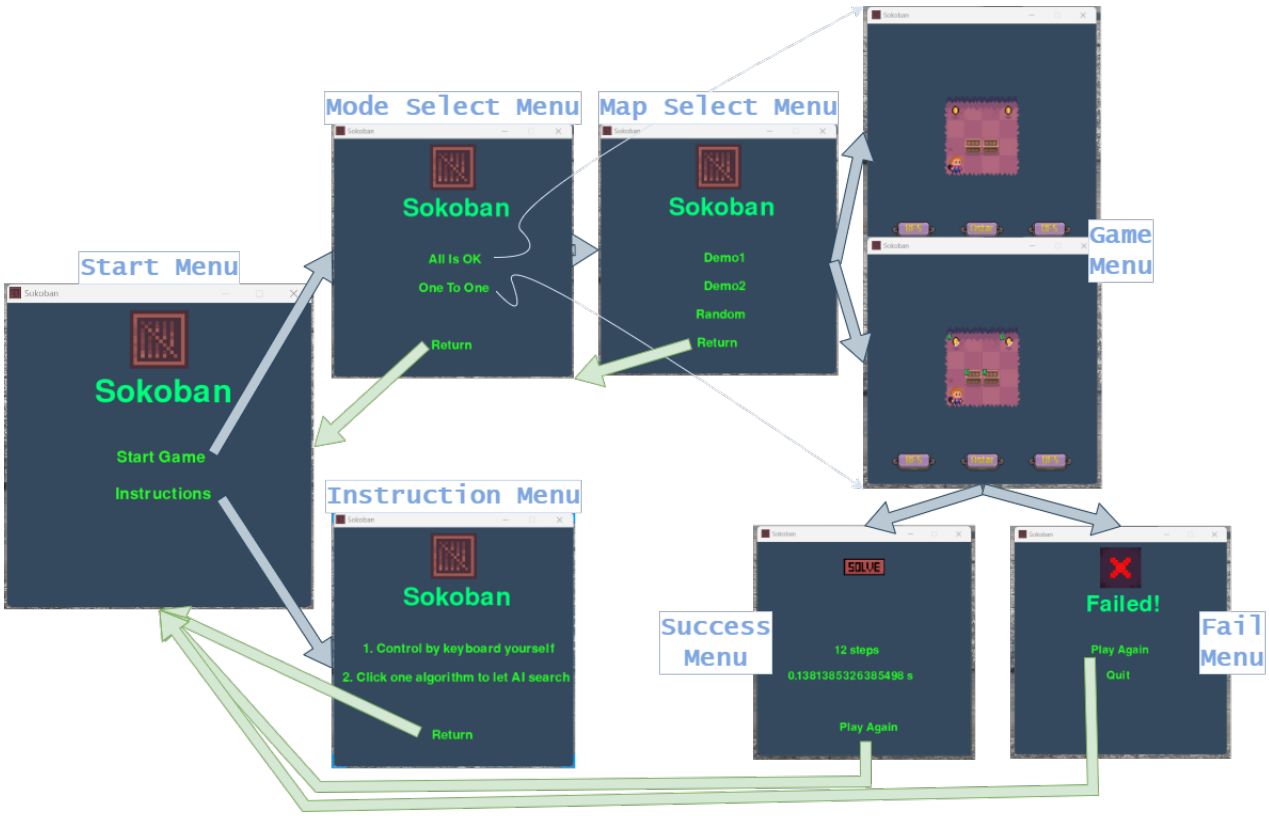

图 2: 游戏界面流程示意图

运行 code/sokoban.exe（或命令行进入 code 路径下，输入 python sokoban.py）后，进入 Start Menu，玩家可以点击 “Return” 返回 Start Menu；玩家可也以点击 “Instructions” 进入 Instruction Menu 查看游戏说明，查看完之后可以点击 “Return” 返回 Start Menu；玩家也可以点击 “Start Game” 开始游戏。

开始游戏后进入 Mode Select Menu，玩家可以点击 “Return” 返回 Start Menu；玩家也可以点击游戏模式，其中 “All is OK” 代表作业要求（2），箱子被推到洞口后不会消失、且箱子和洞口不存在一一对应的关系，“One to One” 代表作业要求（3），箱子被推到洞口后会消失，且箱子和洞口一一对应。

选择完游戏模式后，进入 Map Select Menu 选择地图模式，玩家可以点击 “Return” 返回 Mode Select Menu；玩家也可以点击地图模式，其中 “Demo1” 和 “Demo2” 代表 code/Maps 下的两种人工设计的地图，“Random” 代表随机生成地图。

选择完地图模式后，进入 Game Menu 游戏界面，示意图中展示的地图为 Demo1，玩家可以通过键盘上的 $'\uparrow''\downarrow''\leftarrow''\rightarrow'$ 键控制人物上下左右移动；玩家也可以点击底下的 “BFS”、“Astar”、“DFS” 按钮，让 AI 通过相应的搜索算法搜索出推箱子的方法，并进行方法的展示。若将所有的箱子都推到了相应的位置后，进入 Success Menu 展示成功信息，如果是键盘控制的则展示移动步数，如果是 AI 搜索的则还会展示搜索花费的时间，玩家可以点击 “Play Again” 返回 Start Menu；若出现了死局后，即箱子没有被推到相应位置且无法被推动，进入 Fail Menu，玩家可以点击 “Play Again” 返回 Start Menu，也可以点击 “Quit” 退出游戏。

具体代码见 code/sokoban.py。

3. 地图类设计

地图类将地图相关功能进行封装，包括从文件中读取地图信息、随机生成地图、进行人物移动、获取当前成功或死局状态、计算启发函数等。

3.1 地图表示

地图类（Class Map）中的参数有：

- game_mode：“all” 对应作业要求（2），“one” 对应作业要求（3）
- move_sequence：记录从初始地图到当前地图人物移动的序列
- size：地图尺寸
- walls：所有墙的坐标
- holes：所有洞口的坐标
- boxes：所有箱子的坐标
- player：人物的坐标
- map_matrix：地图的矩阵表示

在常见的推箱子游戏中，地图常常只用矩阵和人物坐标来进行表示，便于可视化、进行移动、判断地图状态等，本次项目也沿用了这一思路，矩阵中的元素用 “0” 代表空地、“1” 代表墙、“2” 代表洞口、“3” 代表没有推到洞口的箱子、“4” 代表已经推到洞口的箱子。另外，本次项目为了方便地图各种功能所需的信息获取，增加了 walls、holes、boxes 来储存相关的坐标信息，其中，walls 以集合形式存储，查找和删除速度快，便于地图生成过程中空地坐标的获取；holes 和 boxes 以字典形式存储，箱子和洞的序号用大写字母表示作为键，坐标作为值，便于成功、死局判断时的查询，以及 “one” 游戏模式下箱子和洞口的一一对应关系的表达。

3.2 人物移动

在对人物进行移动操作时，首先判断人物移动后的位置是否有箱子，若有，更改人物坐标，且进行推箱子操作，判断箱子被推后的位置是否有洞口，若有洞口且是“one”游戏模式，则根据箱子与洞口对应情况判断是否要删除该箱子，否则直接更改矩阵和箱子列表对应的值；若没有，则仅有人物移动，更改人物坐标即可。具体代码见 code/map.py/move()。

3.3 成功判断

判断当前地图是否已经成功完成了推箱子任务，只需要判断还没有被推到相应洞口的箱子数量是否为0：对于“all”游戏模式，若所有箱子坐标在矩阵中的值都为“4”，则所有箱子都已经被推到了洞口；对于“one”游戏模式，由于人物移动操作时若箱子推到相应洞口，箱子列表中会删除该箱子，所以若箱子列表为空，则所有箱子都已经被推到了相应洞口。具体代码见 code/map.py/is_success()。

3.4 死局判断

死局即为没有被推到相应洞口的箱子无法被推动的情况，如下图所示：

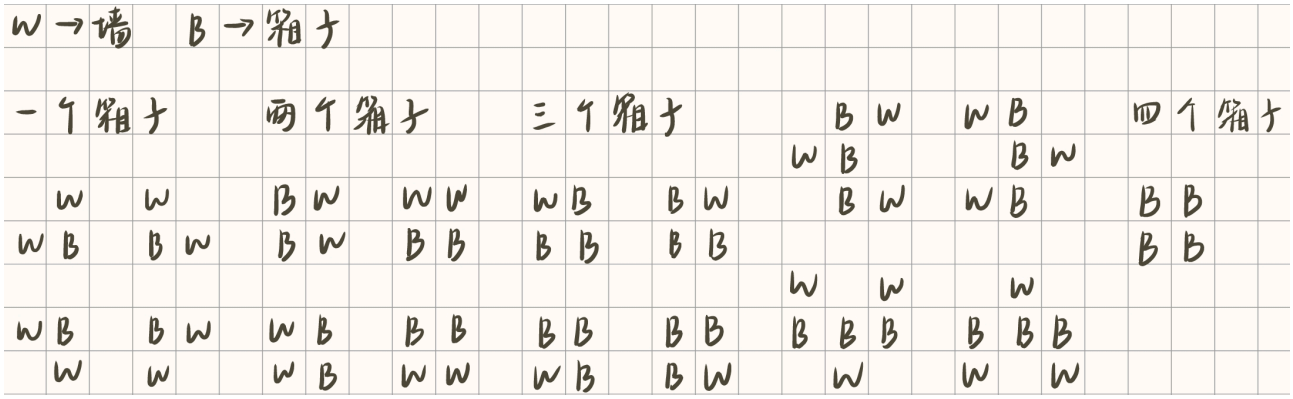

图 3: 死局情况示意图

若没有被推到相应的洞口的箱子出现上图所示的情况，则为死局。具体代码见 code/map.py/is_fail()。

3.5 随机生成

在随机生成地图时，为了保证生成的地图大部分是有解的，且搜索空间不太大使得搜索算法能在一定时间内找到解，采用了先随机初始化生成一个较易解的地图，再通过加墙进一步缩小搜索空间、增加地图复杂度。

3.5.1 初始化

随机初始化时，地图的长、宽为 6-10 之间的随机整数，箱子的数量为 1-4 之间的随机整数。首先，在地图的边缘加上一圈墙，再根据地图大小随机放一些墙；其次，在空地随机设置洞口的位置；接着，在放上箱子不导致死局的前提下，在空地（包括洞口）随机设置箱子的位置；最后，在空地（包括洞口）随机设置人物的位置。在设置洞口、箱子和人物位置时，都先判断了随机选择的位置会不会被墙壁围绕，一定程度上降低了生成无解地图的概率。具体代码见 code/map.py/random_initialize()。

3.5.2 加墙

随机生成了一个地图之后，通过 A* 算法搜索进行测试，若在一定搜索时间和移动步数内有解，则认为该地图较易解。基于该地图，在不形成死局的前提下随机加入墙，再对新的地图通过 A* 算法进行测试，若无解，则回溯去掉新加的墙，当无解次数超过三次，则认为地图具有一定复杂度，停止加墙；若搜索花费时间过长，则认为当前搜索空间太大，保留新加的墙，继续加墙；若移动步数过多，则认为当前地图具有一定复杂度，保留新加的墙，停止加墙。具体代码见 code/map.py/random_build()。

3.6 生成结果

随机生成的部分地图如下所示，下一行为上一行地图的 A* 搜索结果：

（1）“all”游戏模式

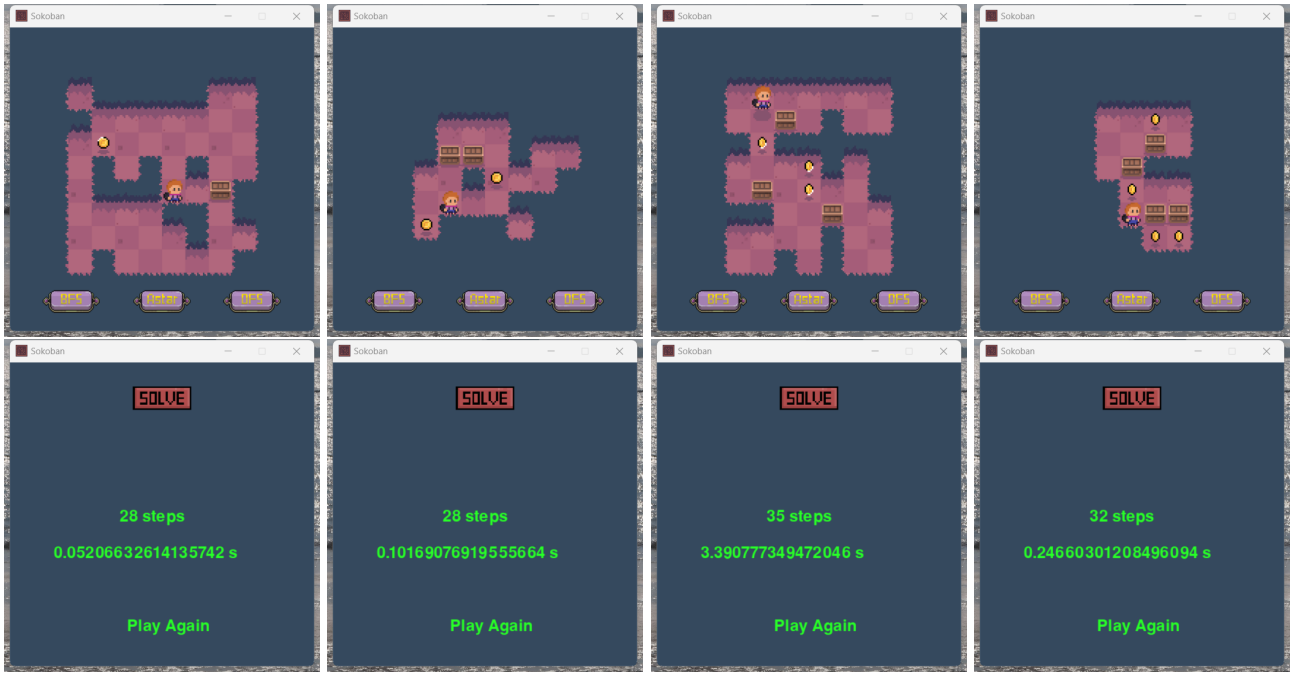

图 4: 随机生成地图示例（(“all”游戏模式）

（1）“one”游戏模式

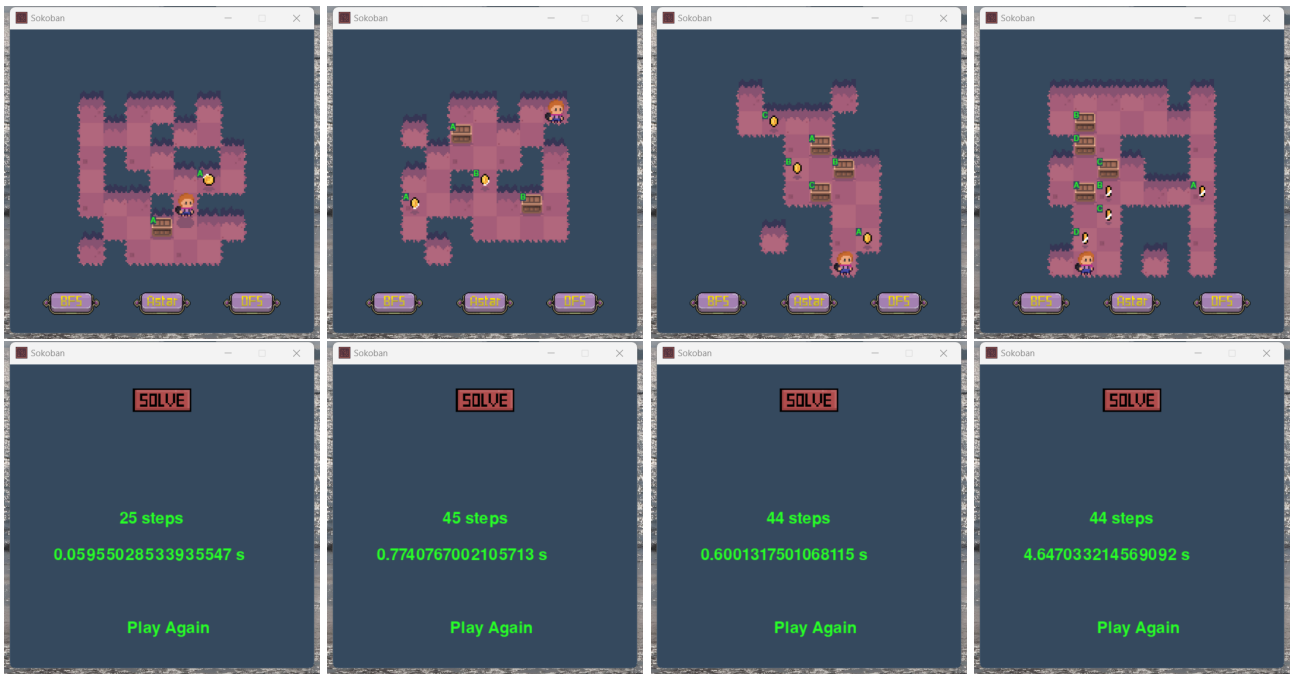

图 5: 随机生成地图示例（(“one” 游戏模式）

从地图和 A* 搜索结果可以看出，该随机生成地图的方法可以生成多种初始场景，包括初始位置、箱子和洞口的数量和分布情况。但生成的地图难度都不大，经过实验测试，这是因为初始化地图时设置的搜索时间阈值较短，提高该阈值可以增大生成地图的难度；但是由于提高该阈值意味着初始化地图的搜索空间变大，搜索花费时间较长，会导致随机生成地图的速度较慢。另外，由于加墙过程中若超出一定搜索时间或移动步数则保留当前加墙，为保证一定的生成速度没有完全搜索看是否有解，所以生成的地图可能是无解的，下图就是测试过过程中遇到的一种情况：

图 6: 随机生成地图无解的情况

4. 搜索算法设计

4.1 基础搜索：深度优先搜索、广度优先搜索

对于移动路线的搜索，本次项目首先采用了两种基础搜索算法：深度优先搜索和广度优先搜索，具体代码见 code/Solver/dfs.py 和 code/Solver/bfs.py，此处不再赘述。

4.2 最优搜索：A* 搜索

上述基础搜索算法的效果都不太好，其中深度优先搜索通常不能搜索出最优路径，而经过测试广度优先搜索的效率较低，故采用 A* 算法进行改进，基于 queue 库的 PriorityQueue() 构建开节点表，具体代

码见 code/Solver/astar.py。

4.2.1 启发函数设计

对于启发函数 $f(n)=g(n)+h(n)$，其中 $g(n)$ 为从开始节点到当前节点的实际代价，在本次项目里即为从开始节点到当前节点的移动路径的长度；$h(n)$ 为当前节点到达成功节点的估算代价，在本次项目里，在“all”游戏模式下，采用排序后的箱子和洞口的横纵坐标差值之和，在“one”游戏模式下，采用对应箱子和洞口的曼哈顿距离之和。在保证较低复杂度的前提下，该启发函数可以满足要求。具体代码实现如下，可见 code/map.py/cost()：

```
def cost(self):
    cost = 0
    if self.game_mode == "all":
        sorted_holes_x = sorted(self.holes[hole][0] for hole in self.holes)
        sorted_holes_y = sorted(self.holes[hole][1] for hole in self.holes)
        sorted_boxes_x = sorted(self.boxes[box][0] for box in self.boxes)
        sorted_boxes_y = sorted(self.boxes[box][1] for box in self.boxes)
        for i in range(len(self.boxes)):
            cost += np.abs(sorted_holes_x[i] - sorted_boxes_x[i]) + \
            np.abs(sorted_holes_y[i] - sorted_boxes_y[i])
    else:
        for box in self.boxes:
            cost += np.abs(self.holes[box][0] - self.boxes[box][0]) + \
            np.abs(self.holes[box][1] - self.boxes[box][1])
    return cost
```

经过测试，A* 算法有效地提高了搜索效率，但效果依于实时游戏来说依然不是特别理想，故对每种搜索算法都采取了以下两种改进策略：

4.3 优化策略：死局情况剪枝

在推箱子游戏中，有些时候即使出现了死局情况，人物依然可以继续移动，在该节点下继续搜索必定无法找到正确解法，所以可以进行剪枝。故在访问节点时，首先判断其是否在闭节点表中，若在则将其加入闭节点表，接着判断其是否符合死局情况，若不符合才将其加入开节点表，这样一方面进行了剪枝，另一方面在下一次访问到该节点时不需要重新判断是否死局。

4.4 优化策略：*hash* 闭节点表查询实现

本次项目刚开始基于集合构建闭节点表，存储 Map 类对象，每次判断新节点是否在闭节点表中时，需要遍历一遍闭节点表，比较 map_matrix 和 player 是否相同，效率较低。故通过 hash 查询的优化策略实现闭节点表的查询，在“all”游戏模式下，以玩家位置、排序后的箱子位置构成的字符串作为 key，在“one”模式下，以玩家位置、箱子位置构成的字符串作为 key，提高了查询效率。构建 hash key 的具体代码如下所示：

```
def hash(game_map):
    map_list = [game_map.player[0], game_map.player[1]]
    if game_map.game_mode == "all":
        sorted_boxes = sorted(game_map.boxes[box] for box in game_map.boxes)
        map_list += [box_pos_xy for box_pos in sorted_boxes for box_pos_xy in box_pos]
    else:
        map_list += [box_pos_xy for box_id in game_map.boxes \
```

```
        for box_pos_xy in game_map.boxes[box_id]]
    return ''.join(map(str, map_list))
```

4.5 搜索结果

经过测试，demo1 和 demo2 通过改进后的 A* 搜索算法得到的结果如下所示：

（1）demo1

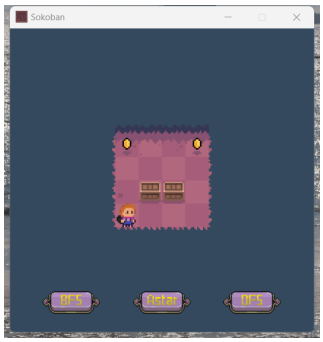

图 7: “all”模式地图

图 8: “all”模式结果

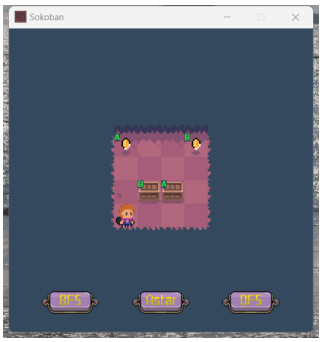

图 9: “one”模式地图

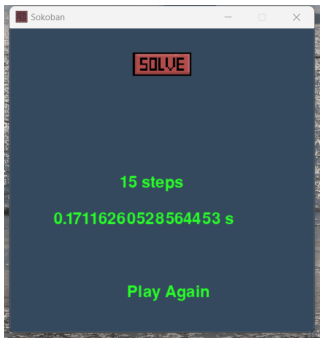

图 10: “one”模式结果

移动路径分别为：

- “all”游戏模式：→↑→↓→↑↑←←↓←↑
- “one”游戏模式：→↑↑→→↓←←↓←↑↑↑→→

另外，demo 中有通过改进后的三种算法进行搜索的游戏过程视频，可进行对照查看，其中，A* 算法花费时间最短，符合改进要求。

（2）demo2

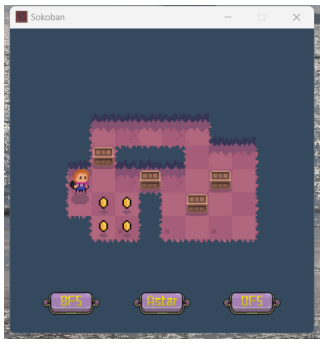

图 11: “all”模式地图

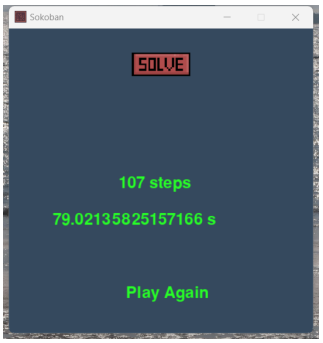

图 12: “all”模式结果

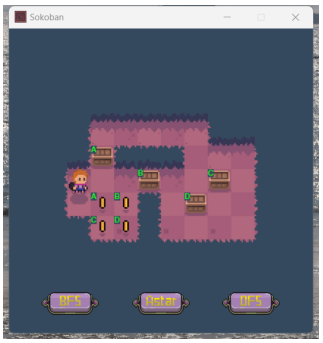

图 13: “one”模式地图

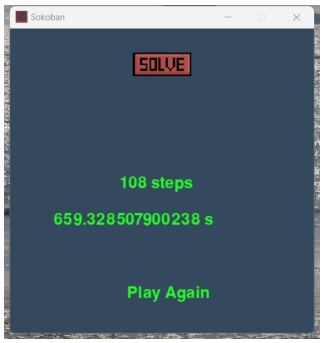

图 14: “one”模式结果

移动路径分别为：

- “all”游戏模式：→→→→↓↓→→→↑↑↑←↓←↑↑←←←←↓↓↓↑↑↑→→→→↓↓←←←→→→↑↑←←←←↓↓←↓→↑→→→↓↓→→↑→↑←←←←←↓↑→→→↑↑←←←←↓↓←↓→↑→→→↓↓→↑→↑←←←←→→→↑↑←←←←↓↓
- “one”游戏模式：→→→→↓↓→→↑←↓←↑→→→↑↑←←↑←←←←↓↓↑↑→→→→↓↓←←←→→→↑↑←←←←↓↓←↓→↑→→→↑→→↓←←←←←→→→↑↑←←←←↓↓↓↑→→→↓↓→↑→↑←←←←↓↑→→→↑↑←←←←↓↓←↓→

5. 参考资料

界面素材来源https://github.com/rumbleFTW/sokobot,剪枝思路来源https://github.com/KnightofLuna/sokoban-solver。