基于双向A*的推箱子求解与推箱子关卡生成

问题分析

推箱子作为一个游戏，既需要用图形界面和用户交互，其求解与关卡生成问题又和人工智能密切相关。Python的pygame库提供了丰富便捷的图形界面显示和用户输入处理功能；同时，Python作为一门流行于科学计算和人工智能的语言，也很适合在本次课程设计以外，在后续的探索中利用深度强化学习进行推箱子问题的求解，具有更多可能性。所以我们选用Python配合pygame库来实现推箱子。

问题形式化

推箱子很显然是可以通过搜索求解的问题。根据本学期课程用书《人工智能原理：现代方法》[1]中对搜索问题的形式化描述，推箱子是一个完全可观测的、单智能体的、在确定世界中的、已知的、序贯、静态、离散搜索问题。我们将推箱子问题写作一个类，其根据储存关卡初始化，所包含的方法全部参照书中内容。此外，推箱子问题的动作是有限的，最多为上下左右四种。推箱子问题的状态就是地图上每个位置的内容，这是一个矩阵。矩阵中的元素内容也是有限个，包含墙、空间、箱子、玩家、目标、在目标上的玩家和在目标上的箱子等。

推箱子求解

去除掉重复局面，每个推箱子问题的状态空间是有限的，那么在有限时间内（即使可能是超出多项式时间），用搜索算法搜索从初始状态到目标状态的路径就是可解的。问题的关键就是如何避免探索没有必要的状态，这包括达成目标的可能性很低的状态和后续已经无法达到目标的状态（称为死锁）。

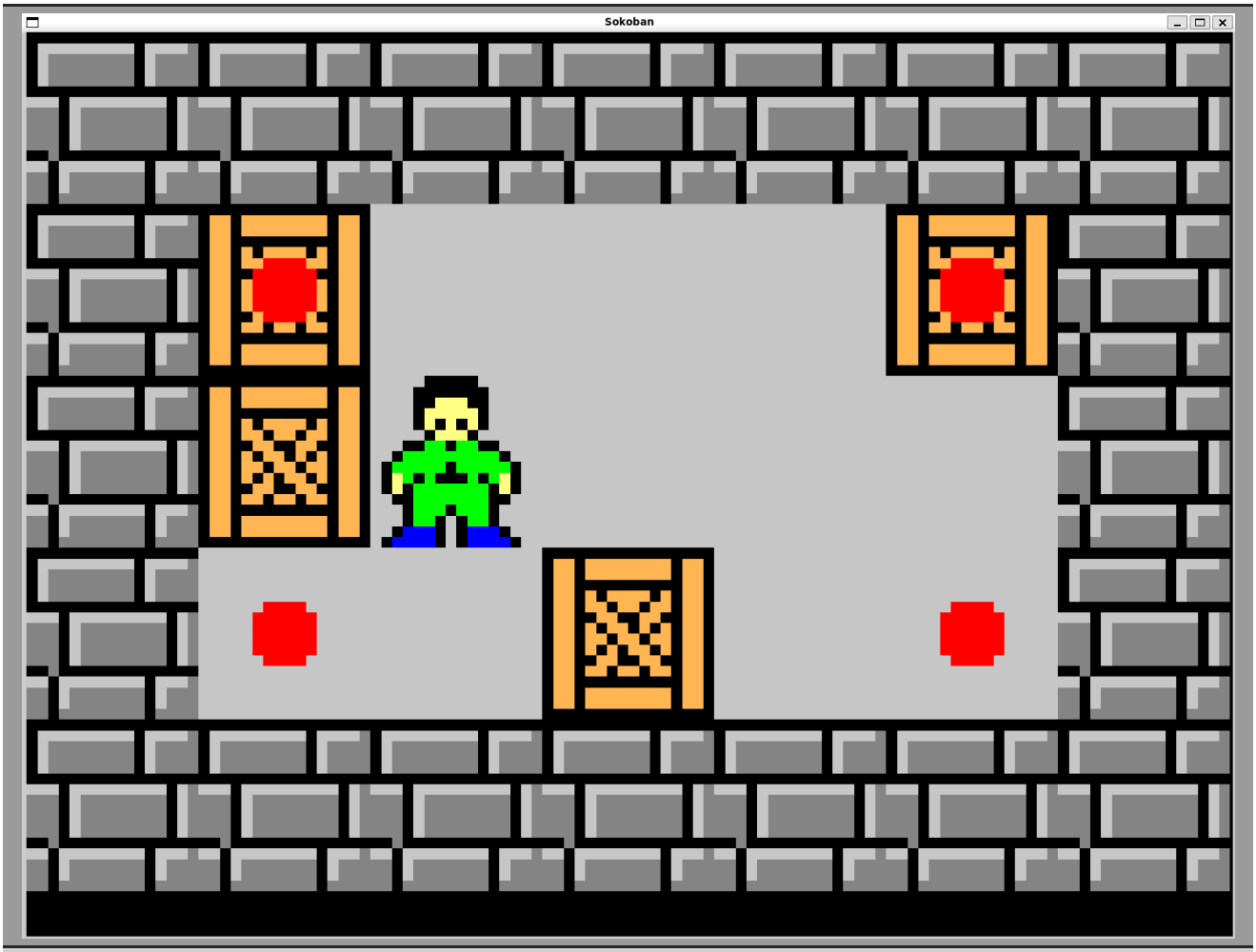

图1 死锁的一种情况（左侧两个箱子）

由于导致死锁的动作是单向的，并且推箱子问题又是完全可观测的，所以使用离线搜索是好过在线搜索（局部搜索）的。为了避免探索重复状态，甚至造成死循环，为状态类自定义哈希方法，并在搜索中比较即可。为了避免探索没有必要的状态，我们可以人工设计启发式函数和使用A* 算法。启发式函数需要精心设计。一个好的启发式函数可以大大减少搜索空间，提高搜索效率。启发式函数的主要要求是：

1. 一致性：启发式函数满足三角不等式，即对于任意状态n和其邻接状态n'，$h(n) \leq c(n, n') + h(n')$，其中$h(n)$为启发式函数值，$c(n, n')$为从$n$到$n'$的实际代价。
2. 可接受性：启发式函数永远不会高估从当前状态到目标状态的代价，即$h(n) \leq \hat{h}(n)$，其中$\hat{h}(n)$为从n到目标状态的实际最小代价。

为了避免在死锁的状态下持续探索下去，我们首先在启发式函数中加入一个十分大的死锁惩罚项。有一种方法可以判断出大部分的死锁。多次遍历每个箱子，如果这个箱子可以被推动，就让他消失；继续遍历，如果某一轮还有箱子，但遍历一次并未消去任何箱子，那么可以知道后续的遍历也不能消去这些箱子，即局面出现了死锁。然而，另一种方式是从反方向解决问题，这天然地可以避免死锁的问题。

其次，在推箱子问题中，箱子需要被推到特定的目标位置上。为了更好地估计从当前状态到目标状态的代价，我们可以利用最小最优匹配来计算启发式函数。最小最优匹配其实就是找到如何将箱子分配给各个目标，使得箱子到目标的曼哈顿距离的总和最小。在求出耗费矩阵后，这个算法可以由 `scipy.optimize.linear_sum_assignment` 实现。

地图生成

生成的推箱子地图需要保证可解性和一定的难度。可解性和难度可以通过对生成地图的直接求解来实现，但是这意味着我们需要消耗指数级的时间。为了避免这个问题，我们选用了一种一定能够保证地图有解的生成方法，并选择了地图的几个特征用于评估生成地图的难度。我们随机生成大量的推箱子问题，并选择难度评分较高的进行输出。

地图难度评估

为了更好的评估地图的难度，我们选择了几个有代表性又便于求解的特征，这包括有效块数$P_b=$ `num_block_33` ，阻塞系数$P_c=$ `Congestion_metric` 和箱子的个数$n=$ `box_num` 。 有效块 `block_33` 指的是周围3*3范围内的八个位置不全是墙壁或空的块, 阻塞系数：

$$P_c=\sum_{i=1}^{n}\frac{\alpha b_i+\beta g_i}{\gamma\left(A_i-w_i\right)}$$

用于评估箱子与目标的密度。

在阻塞系数的公式中，n是地图中箱子的数目。按字典序排序，依次将所有的箱子与目标给予编号i，则A_i,b_i,g_i,w_i分别代表着在编号为i的箱子和目标之间包围的方框所包围的面积、箱子数，目标数，墙壁数，参数α,β,γ选取的值分别为10，5，1。

分别选取系数w_b,w_c,w_n,k为15，5，1，50，最终，我们可以将难度表达为：

$$\text{rewards}=\frac{w_bP_b+w_cP_c+w_nn}{k}$$

地图生成

为了避免繁琐的证明可解的过程，我们选择在生成时就能保证地图有解的策略。

我们首先将player的初始位置之外的所有的位置设置为障碍物。此时，我们可以进行的行动有移动 `(move)` 和冻结 `(freeze)` 。当选择移动时，我们会按照选择的方向在地图中进行移动，并且无视墙壁，将走过的所有位置都变成空。当选择冻结时，我们将无法再清除障碍物，并进入下一个状态。

进入第二个状态之后，我们能选择的行动有生成箱子 `generate_box` ，和结束生成 `end_gene_box` 。当选择生成箱子时，若选择的位置是空的，我们将在这里生成一个箱子。当选择结束生成时，我们无法再生成箱子，并进入下一个状态。

进入第三个状态之后，我们能选择的状态有移动 `move` 和结束 `end_all` 。当选择移动时，我们会仿照游戏的规则，在地图中进行移动，途中能够进行推箱子，也会受到障碍的影响，只是途中并没有设置目标。当选择结束后，我们除去没有移动过的所有箱子，然后在所有有箱子的地方设置目标。这使得所有目标都可达，同时防止我们生成的地图过于简单。

之后我们对生成的地图按照上面的标准进行评估，同时结束整个生成过程。若地图的得分低于预期设置的值，我们将重新生成，直到生成的地图超过我们的设置的阈值。

关卡文件的储存与爬取

经过调查，关卡文件的储存大都由字符表示地块类型，储存在plaintext文件中。主要有两套映射方式，一套是用 `#,$,@,.` 等符号，另一套是用 `W,B,P,G` 等字母，为了后续利用数据进行训练的便捷，我们储存了两套符号符号系统的转化字典和函数。

在设计要求提供的推箱子网站上，储存着许多关卡，我们编写了一个脚本，根据网站上图像排列的格式，转化为txt文件储存在 `levels/` 中。

数据结构设计与实现

各个类型的接口详见对应类的docstring.

状态地图 `Map`

`Map` 定义在 `game/map.py` 中，由表示所有地块的 `numpy` 数组 `tiles` 和表示玩家位置的 `player_x,player_y` 构成。为了增强代码可移植性和美观， `tiles` 中的数组元素为枚举类型 `Tile` 的对象，实际上相当于整数。

`Map` 包含一些常用的接口，例如：

1. `set_tiles(x, y, tiles)`：设置某一位置的地块为某个类型。
2. `is_XxX(x, y)`：判断某个位置的地块是否为某个类型。
3. `locate_XxX(x, y)`：给出某个类型的地块所在的位置，类型为$n \times 2$的数组，n为该类型地块的数量。借助 `numpy.where` 这个函数，可以很方便实现。
4. `p_move(dx, dy)`：使得玩家向某个方向移动，如果成功返回 `True` ，否则返回 `False` 。
5. `cost_matrix()`：给出一个局面的耗费矩阵，(i,j)为第i个箱子到第j个目标的曼哈顿距离。借助 `numpy` 的广播性质，这可以很方便实现。
6. `count_deadlock()`：利用上文提到的算法，进行并不精确的死锁判定，查全率可能不够高，但查准率是100%.

值得注意的是，由于 `Map` 的状态只和 `tiles` 有关，其哈希设计为 `tiles` 数组转化为二进制码后的哈希值。

推箱子问题规则 `SokobanProblem`

`SokobanProblem` 在 `game/problem.py` 实现，由储存关卡的文件路径初始化的属性 `level` 储存了关卡的初始局面。枚举类型 `SokobanAction` 定义了上下左右四个动作。问题规则形式化之后写作以下方法：

1. `initial_state()`：关卡的初始局面。
2. `actions(map)`：在某个状态下的所有可能动作。
3. `result(map, action)`：在某个状态下实施某个动作会导致的结果状态。
4. `is_goal(map)`：判断某个状态是否为目标状态。
5. `step_cost(map, action)`：由于每一步的耗费都是相等的，所以统一返回1.

此外，问题某个局面的启发值有 `heuristic(map)` 方法输出。

`BiSokobanProblem` 在 `game/biproblem.py` 实现，继承于 `SokobanProblem`，多了一些满足双向搜索的方法：

1. `goal_states(num)`：返回长度至多为 `num` 的可能目标状态的列表。
2. `actions_to(map)`：可能到达某个状态的动作。
3. `reason(map, action)`：如果通过某个动作到达某个状态，返回其先前状态。

`re_heuristic(map)` 作为反向搜索的启发式函数，除了箱子到初始状态的最小最优匹配，还加入的玩家到初始玩家位置的曼哈顿距离。

搜索算法

在本学期人工智能原理的课程作业中，我们写过一个搜索算法的库 `sealgo`，意思是 `SEArch ALGOrithms`。这次在稍加完善后可以使用。`sealgo` 中包含许多局部搜索（`local_search.py`）与最佳优先搜索（`best_first_search.py`）算法，便于我们实验和分析哪种更适用于推箱子问题。每个算法接受一个问题作为参数初始化 `Search(problem)`，使用 `search()` 方法搜索答案并返回解的列表，每个解是一系列动作的列表。经过实验和分析，我们选择了A*算法，所以以下主要介绍A*算法。

```
class AStar(BestFirstSearch):
    def __init__(self, problem:HeuristicSearchProblem, weight:float|int=1):
        super().__init__(problem)
        self.eval_f = lambda s: self.g_costs[s] + weight *
self.problem.heuristic(s)
```

`AStar` 继承自抽象父类 `BestFirstSearch`，另外还包含一个权重参数，其对每个状态节点的评价函数 `eval_f` 为：

$$f(n) := g(n) + w \cdot h(n)$$

其中$g(n)$为累计代价，即到达该状态的路径代价的总和；$h(n)$为预估代价，即启发式函数给出的到达目标节点的近似路径代价；w为预估代价所占的权重，通过调节w也可以获得不同的搜索效率。

```
class BestFirstSearch(Search):
    def __init__(self, problem:SearchProblem) -> None:
        self.problem = problem
        self.frontier = PriorityQueue()
        init = self.problem.initial_state()
        if isinstance(init, list):
            self.g_costs = {} # cost so far
            self.predecessors = {}
            for state in init:
                self.g_costs[state] = 0
                self.predecessors[state] = (None, Action.STAY)
                self.frontier.put((-1, state))
        else:
            self.g_costs = {init: 0} # cost so far
            self.predecessors = {init: (None, Action.STAY)}
            self.frontier.put((-1, init))
        self.eval_f: Callable = lambda s: 0
        # self.eval_f must be defined in the subclass
```

```
    def search(self) -> List[List[Action]]:
        while not self.frontier.empty():
            state = self.frontier.get()[1]
            if self.problem.is_goal(state):
                return [self._reconstruct_path(state)]
            self._extend(state)
        return []
```

`BestFirstSearch` 用优先队列储存其探索的前沿，优先队列中状态的优先级即为 `eval_f(state)`，每次都取出优先级最高的节点探索，并将可拓展的节点加入优先队列。此外，`g_costs` 储存了到达每个节点的最小路径代价。`predecessors` 储存了每个节点的父节点，在搜索到目标节点时，就可以通过 `_reconstruct_path(state)` 重构到达目标节点的路径。

另外，在 `sealgo/bidirectional.py` 中，我们实现了双向搜索算法 `BiDirectional`：

```
class BiDirectional(Search):
    def __init__(self, problem: BiSearchProblem, f_algo: Type[BestFirstSearch],
b_algo: Type[BestFirstSearch]|None = None, b_weight: int = 1, *args, **kwargs) -
> None:
        self._init_problem(problem)
        if b_algo is None:
            b_algo = f_algo
        if args or kwargs:
            self.f_algo = f_algo(self.f_problem, *args, **kwargs)
            self.b_algo = b_algo(self.b_problem, *args, **kwargs)
        else:
            self.f_algo = f_algo(self.f_problem)
            self.b_algo = b_algo(self.b_problem)
        self.b_weight = b_weight

    def search(self) -> List[List[Action]]:
        b_times = 0
        while not self.f_algo.frontier.empty() and not
self.b_algo.frontier.empty():
            if b_times >= self.b_weight:
                b_times = 0
                # forward search
                f_state = self.f_algo.frontier.get()[1]
                self.f_algo._extend(f_state)
            # backward search
            b_state = self.b_algo.frontier.get()[1]
            if b_state in self.f_algo.predecessors:
                return [self._reconstruct_path(b_state)]
            self.b_algo._extend(b_state)
            b_times += 1
        return []
```

他可以接受任意搜索算法分别作为两端探索的算法，在这里我们在两端都使用A*算法。`b_weight` 参数指定当进行多少轮反向探索之后，进行一次前向探索，这在不同问题中的最佳值不同，会影响搜索效率。当两端的已探索区域相连时，返回初始状态到目标状态的路径。

显示 Display

`Display` 模块在 `ui/display.py` 实现，负责图形用户界面的显示，初始化时会加载材质，材质风格在 `assets/` 中储存，可以在启动时被命令行参数 `--icon-style` 指定。枚举类 `State` 定义了GUI的可能状态：开始菜单、游戏中、主菜单（暂停菜单）、胜利菜单、AI解决中、关卡生成中。每种菜单状态下分别由一种私有方法渲染，由方法 `run()` 判断。

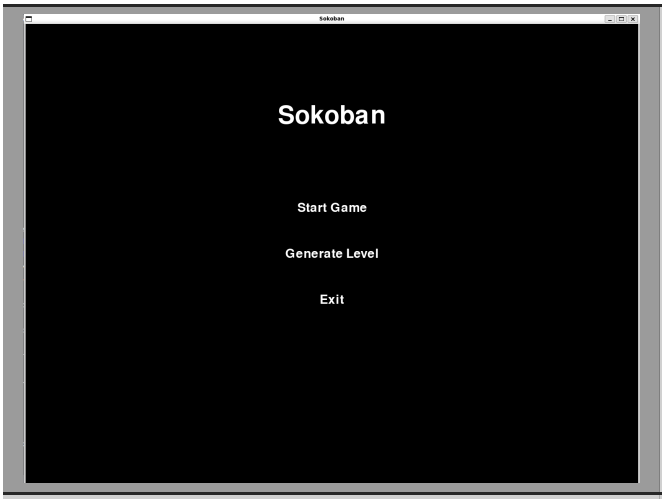

图2 开始菜单

图3 主菜单

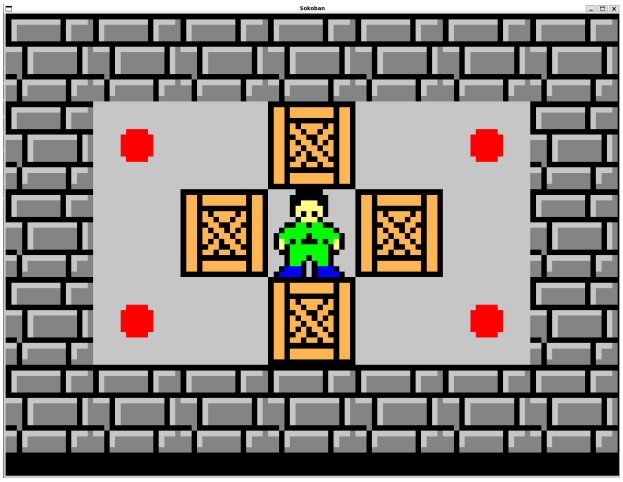

图4 游戏画面

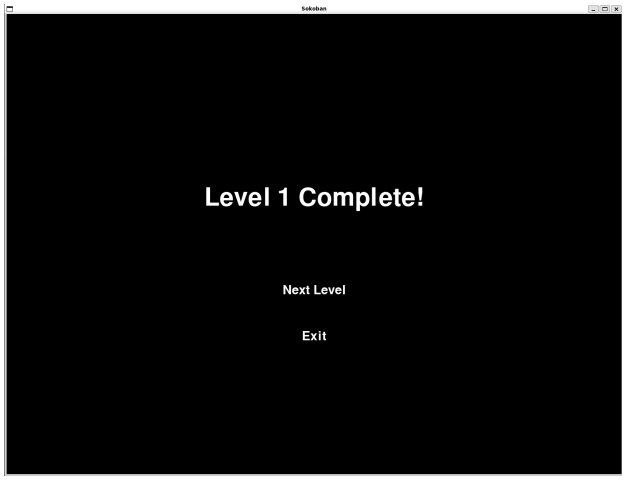

图5 胜利菜单

开始菜单 `_start_menu()` 在启动时出现，显示游戏标题，提供开始游戏、生成关卡、退出的选项；主菜单 `_main_menu()` 在暂停后出现，显示关卡号，提供继续游戏、重新开始、询问AI、退出的选项；胜利菜单 `_victory_menu()` 在每次通关后出现，提供下一关、退出的选项，如果所有关卡都通关，则返回主菜单；游戏中，渲染当前状态的可视化界面，每个地块的材质默认是从设计要求提供的推箱子网站上下载的。

用户输入处理 `InputHandler`

`InputHandler` 在 `ui/input_handler.py` 中实现，负责所有用户输入的处理。枚举类型 `Event` 定义了对游戏主逻辑造成的更新信号，例如 `ASK_AI`, `START`, `RESTART` 等。当用户输入与游戏主逻辑相关时，例如点击“Start”， `InputHandler` 返回对应 `Event`；当输入与推箱子问题相关时，例如按 `UP` 键， `InputHandler` 返回对应 `SokobanAction` 。

算法复杂度分析

A*搜索算法

最坏复杂度

在A*算法中，时间复杂度主要受以下因素影响：

1. **搜索空间大小**：即状态空间的大小，设为N。

2. **启发式函数的质量**：启发式函数越准确，搜索空间越小，时间复杂度越低。

在最坏情况下，A*算法相当于广度优先搜索，需要拓展深度d的b叉树上的每个节点，时间复杂度为$O(b^d)$，其中b为每个状态可能的后继状态数（分支因子），d为从初始状态到目标状态的最短路径长度。在推箱子问题中，分支因子小于4，因为每个箱子最多可以朝4个方向移动。所以其最坏时间复杂度为$O(b^4)$。

A*算法的最坏空间复杂度与时间复杂度相同，也是$O(b^4)$，因为A*算法需要在 `frontier` 中储存探索前沿的节点，同时在 `f_costs` 与 `predecessors` 中储存已经探索过的节点。

实际复杂度

在每个问题中的平均分支因子$\hat{b}$可以由：

$$\hat{b} = \log_d n$$

得到，其中d为路径长度，n为探索的节点数量。好的启发式函数可以减小平均分支因子，从而使得搜索时间得到指数级提升。

双向搜索算法

双向搜索算法从两端同时开始搜索，时间复杂度为$O(b^{d/2})$，其中b为每个状态可能的后继状态数（分支因子），d为从初始状态到目标状态的最短路径长度。与单向A*算法相比，双向搜索算法通过减小搜索深度，可以显著降低时间复杂度。

双向搜索算法的最坏空间复杂度也是$O(b^{d/2})$，因为需要同时存储从起始状态和目标状态进行的搜索前沿节点。

通过引入双向搜索算法，可以显著降低时间复杂度和空间复杂度，尤其在大规模搜索问题中，双向搜索算法的优势更加明显。

地图生成复杂度分析

最坏情形复杂度

由于我们设置没有到达预定分数就会不断循环，若设定的分数过高，可能会进入无限循环，此时的时间复杂度为无穷。

单个地图生成的时间复杂度

若不考虑没达到分数进行的循环，仅考虑生成一个地图，那么我们的时间复杂度由预设的最大移动次数决定，最大移动次数 `move_limit` 的数量为$O(n^3)$，其中n为地图的边长。即生成单个地图的时间复杂度为$O(n^3)$。

平均复杂度

假定达到某个预定分数平均所需要的迭代次数为a，则平均意义上的时间复杂度为$O(an^3)$

结果分析

推箱子求解

确定了启发式函数之后，前向A*搜索算法的超参数为预估权重 `weight` ，双向A*搜索的超参数为A*中的预估权重 `weight` 与后向-前向比例 `b_weight` 。

利用正向A*算法，使用最小最优匹配、人箱最近距离和死锁惩罚，用不同的预估权重在20个示例关卡上测试。经过实验，发现 `weight` 为3时A*表现较好，得到结果如下：

```
2024-07-22 12:11:26,804 - INFO - Game initialized at 2024-07-22 12:11:26.804829
2024-07-22 12:11:27,065 - INFO - Level 1: Solution found in 0.26 seconds.
2024-07-22 12:11:27,065 - INFO - The b-factor is 2.1955863578374766
2024-07-22 12:11:27,065 - INFO - Solution length: [25]
2024-07-22 12:11:28,642 - INFO - Level 2: Solution found in 1.57 seconds.
2024-07-22 12:11:28,642 - INFO - The b-factor is 2.622756694159562
2024-07-22 12:11:28,642 - INFO - Solution length: [27]
2024-07-22 12:11:28,950 - INFO - Level 3: Solution found in 0.31 seconds.
2024-07-22 12:11:28,949 - INFO - The b-factor is 2.1655796487703918
2024-07-22 12:11:28,950 - INFO - Solution length: [35]
2024-07-22 12:11:29,194 - INFO - Level 4: Solution found in 0.24 seconds.
2024-07-22 12:11:29,194 - INFO - The b-factor is 1.6870676516829783
2024-07-22 12:11:29,194 - INFO - Solution length: [71]
2024-07-22 12:11:30,372 - INFO - Level 5: Solution found in 1.17 seconds.
2024-07-22 12:11:30,372 - INFO - The b-factor is 2.277612976131103
2024-07-22 12:11:30,373 - INFO - Solution length: [41]
2024-07-22 12:11:33,062 - INFO - Level 6: Solution found in 2.69 seconds.
2024-07-22 12:11:33,062 - INFO - The b-factor is 2.811150845173945
2024-07-22 12:11:33,062 - INFO - Solution length: [30]
2024-07-22 12:11:33,286 - INFO - Level 7: Solution found in 0.23 seconds.
2024-07-22 12:11:33,286 - INFO - The b-factor is 1.836101573780415
2024-07-22 12:11:33,286 - INFO - Solution length: [52]
2024-07-22 12:11:44,935 - INFO - Level 8: Solution found in 11.65 seconds.
2024-07-22 12:11:44,935 - INFO - The b-factor is 2.4472097233439887
2024-07-22 12:11:44,935 - INFO - Solution length: [71]
2024-07-22 12:11:47,716 - INFO - Level 9: Solution found in 2.78 seconds.
2024-07-22 12:11:47,716 - INFO - The b-factor is 2.175626877770764
2024-07-22 12:11:47,716 - INFO - Solution length: [79]
2024-07-22 12:12:02,265 - INFO - Level 10: Solution found in 14.55
seconds.
2024-07-22 12:12:02,265 - INFO - The b-factor is 2.5029435109458373
2024-07-22 12:12:02,265 - INFO - Solution length: [76]
2024-07-22 12:12:02,756 - INFO - Level 11: Solution found in 0.49 seconds.
2024-07-22 12:12:02,755 - INFO - The b-factor is 1.7439535788979916
2024-07-22 12:12:02,756 - INFO - Solution length: [78]
2024-07-22 12:12:04,668 - INFO - Level 12: Solution found in 1.91 seconds.
2024-07-22 12:12:04,668 - INFO - The b-factor is 2.52037307175807
2024-07-22 12:12:04,668 - INFO - Solution length: [37]
2024-07-22 12:12:42,909 - INFO - Level 13: Solution found in 38.24 seconds.
2024-07-22 12:12:42,908 - INFO - The b-factor is 2.96284940146013
2024-07-22 12:12:42,909 - INFO - Solution length: [56]
2024-07-22 12:12:47,177 - INFO - Level 14: Solution found in 4.27 seconds.
2024-07-22 12:12:47,177 - INFO - The b-factor is 1.9528706242125256
2024-07-22 12:12:47,177 - INFO - Solution length: [144]
2024-07-22 12:12:55,734 - INFO - Level 15: Solution found in 8.56 seconds.
2024-07-22 12:12:55,734 - INFO - The b-factor is 2.2320129447429133
2024-07-22 12:12:55,735 - INFO - Solution length: [95]
2024-07-22 12:21:00,657 - INFO - Level 16: Solution found in 484.92 seconds.
2024-07-22 12:21:00,656 - INFO - The b-factor is 3.4026483702748584
2024-07-22 12:21:00,657 - INFO - Solution length: [60]
2024-07-22 12:21:51,155 - INFO - Level 17: Solution found in 50.50 seconds.
2024-07-22 12:21:51,155 - INFO - The b-factor is 2.8607833766234765
2024-07-22 12:21:51,155 - INFO - Solution length: [67]
2024-07-22 12:21:53,963 - INFO - Level 18: Solution found in 2.80 seconds.
2024-07-22 12:21:53,963 - INFO - The b-factor is 2.2596906302970865
2024-07-22 12:21:53,963 - INFO - Solution length: [63]
2024-07-22 12:35:24,376 - INFO - Level 19: Solution found in 810.42 seconds.
2024-07-22 12:35:24,375 - INFO - The b-factor is 2.84201976504555
2024-07-22 12:35:24,376 - INFO - Solution length: [164]
2024-07-22 12:35:27,046 - INFO - Level 20: Solution found in 2.67 seconds.
```

```
2024-07-22 12:35:27,046 - INFO - The b-factor is 2.3357095887171684
2024-07-22 12:35:27,046 - INFO - Solution length: [55]
```

经过实验，如此设计启发式函数，分支因子大多位于2~3。当个别关卡超过3时，搜索可能就需要超过一分钟甚至一刻钟！除了16和19关，该算法都能在较快的时间内找到答案。其中19关的时间尤其长，而16关的分支因子达到了3.4。可能是启发式函数设计不恰当，导致探索了很多没有发现的死锁；也可能是阻挡过多，导致最优匹配给出的距离过于低估了箱子需要前往目标的路程，难以发现正确的路径。

利用双向A*算法，将 `weight` 设置为3，使用不同的 `b_weight` （当为 `np.inf` 时退化为反向A*搜索），在20个示例关卡上测试，得到结果如下：

weight	level	elapsed_time	b_factor	length
1	0	0	1.89	3
1	1	0.02	1.95	18
1	2	0.02	1.72	23
1	3	0.18	2.12	36
1	4	0.11	1.64	68
1	5	0.73	2.45	36
1	6	0.14	2.15	29
1	7	0.12	1.96	39
1	8	0.43	1.95	68
1	9	0.29	1.83	78
1	10	1.4	2.23	67
1	11	0.82	2.42	41
1	12	0.19	2.13	34
1	13	0.23	1.91	55
1	14	1.88	1.94	141
1	15	3.07	2.35	72
1	16	50.08	3.52	37
1	17	6.83	2.76	48
1	18	1.76	2.33	60
1	19	18.77	2.86	60
1	20	0.78	2.31	46
10	0	0	1.89	3
10	1	0.01	1.8	18
10	2	0.01	1.77	23
10	3	0.12	2.18	28
10	4	0.05	1.49	68

weight	level	elapsed_time	b_factor	length
10	5	0.58	2.45	38
10	6	0.06	1.95	29
10	7	0.06	1.92	33
10	8	0.26	1.87	64
10	9	0.52	2	72
10	10	0.65	2.11	65
10	11	0.31	2.22	41
10	12	0.08	1.96	34
10	13	0.09	1.73	55
10	14	1.44	1.93	141
10	15	2.3	2.22	72
10	16	21.05	3.38	37
10	17	1.87	2.57	48
10	18	0.94	2.26	60
10	19	6.7	2.65	60
10	20	0.29	2.12	46
100	0	0	1.89	3
100	1	0	1.77	18
100	2	0.02	1.76	23
100	3	0.09	2.14	28
100	4	0.04	1.46	68
100	5	0.58	2.59	32
100	6	0.16	2.29	29
100	7	0.06	1.88	33
100	8	0.21	1.85	64
100	9	0.71	2.18	56
100	10	0.72	2.11	69
100	11	0.24	2.18	41
100	12	0.09	1.99	34
100	13	0.06	1.7	55
100	14	1.26	1.91	141
100	15	1	2.18	72
100	16	18.86	3.36	37
100	17	3.44	2.74	48
100	18	0.85	2.26	60

weight	level	elapsed_time	b_factor	length
100	19	10.29	2.75	62
100	20	0.31	2.21	42
1000	0	0	1.89	3
1000	1	0.01	1.77	18
1000	2	0.01	1.76	23
1000	3	0.09	2.13	28
1000	4	0.07	1.48	68
1000	5	0.59	2.59	32
1000	6	0.16	2.29	29
1000	7	0.05	1.88	33
1000	8	0.21	1.85	64
1000	9	0.71	2.17	56
1000	10	0.79	2.15	67
1000	11	0.39	2.29	41
1000	12	0.1	1.98	34
1000	13	0.06	1.7	55
1000	14	1.26	1.91	141
1000	15	1.07	2.19	72
1000	16	18.74	3.36	37
1000	17	5.74	2.73	48
1000	18	0.88	2.25	60
1000	19	8.46	2.75	62
1000	20	0.32	2.21	42
Infinity	0	0	1.89	3
Infinity	1	0.01	1.77	18
Infinity	2	0.01	1.76	23
Infinity	3	0.1	2.13	28
Infinity	4	0.04	1.48	68
Infinity	5	0.62	2.59	32
Infinity	6	0.17	2.29	29
Infinity	7	0.05	1.88	33
Infinity	8	0.21	1.85	64
Infinity	9	0.74	2.17	56
Infinity	10	3.24	2.15	71
Infinity	11	0.4	2.29	41

weight	level	elapsed_time	b_factor	length
Infinity	12	0.09	1.98	34
Infinity	13	0.07	1.7	55
Infinity	14	1.3	1.91	141
Infinity	15	1.09	2.19	72
Infinity	16	18.84	3.36	37
Infinity	17	3.48	2.73	48
Infinity	18	0.88	2.25	60
Infinity	19	11.07	2.75	62
Infinity	20	0.31	2.21	42

画出搜索用时和分支因子随 `b_weight` 变化的曲线图如图6：

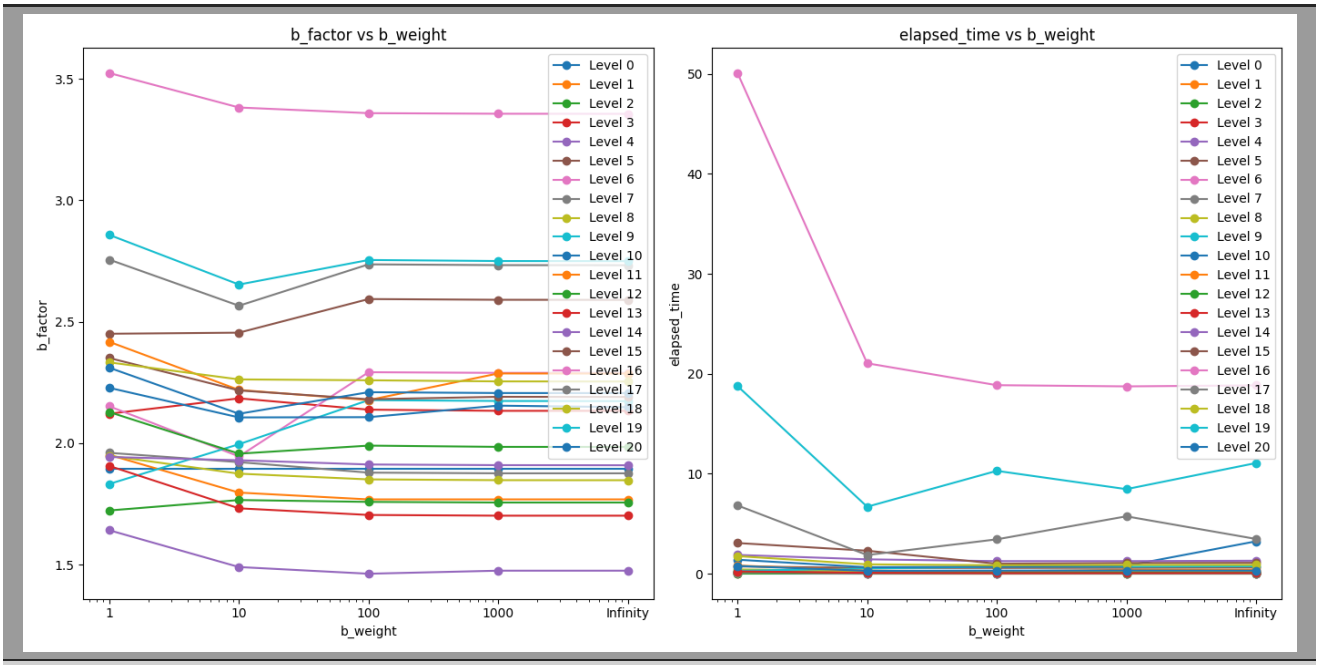

图6 搜索用时和分支因子随 `b_weight` 变化

可见，虽然相比纯正向搜索有大幅提升，但均衡的正向和反向搜索效果最差，而反向搜索的权重增大到100以上，以及纯反向搜索时，对不同关卡的表现各有差异。这可能是因为正向搜索可以只有一个初始状态，而反向搜索有多个，这在箱子较多的场景可能会有性能的下降；但是反向搜索不会陷入死锁，这在容易造成死锁的场景十分高效。总的来说，在推箱子求解任务中，反向搜索更有效，但正向搜索也起到补充作用。

推箱子生成

连续生成三次推箱子关卡，平均用时15s得到的结果如下图：

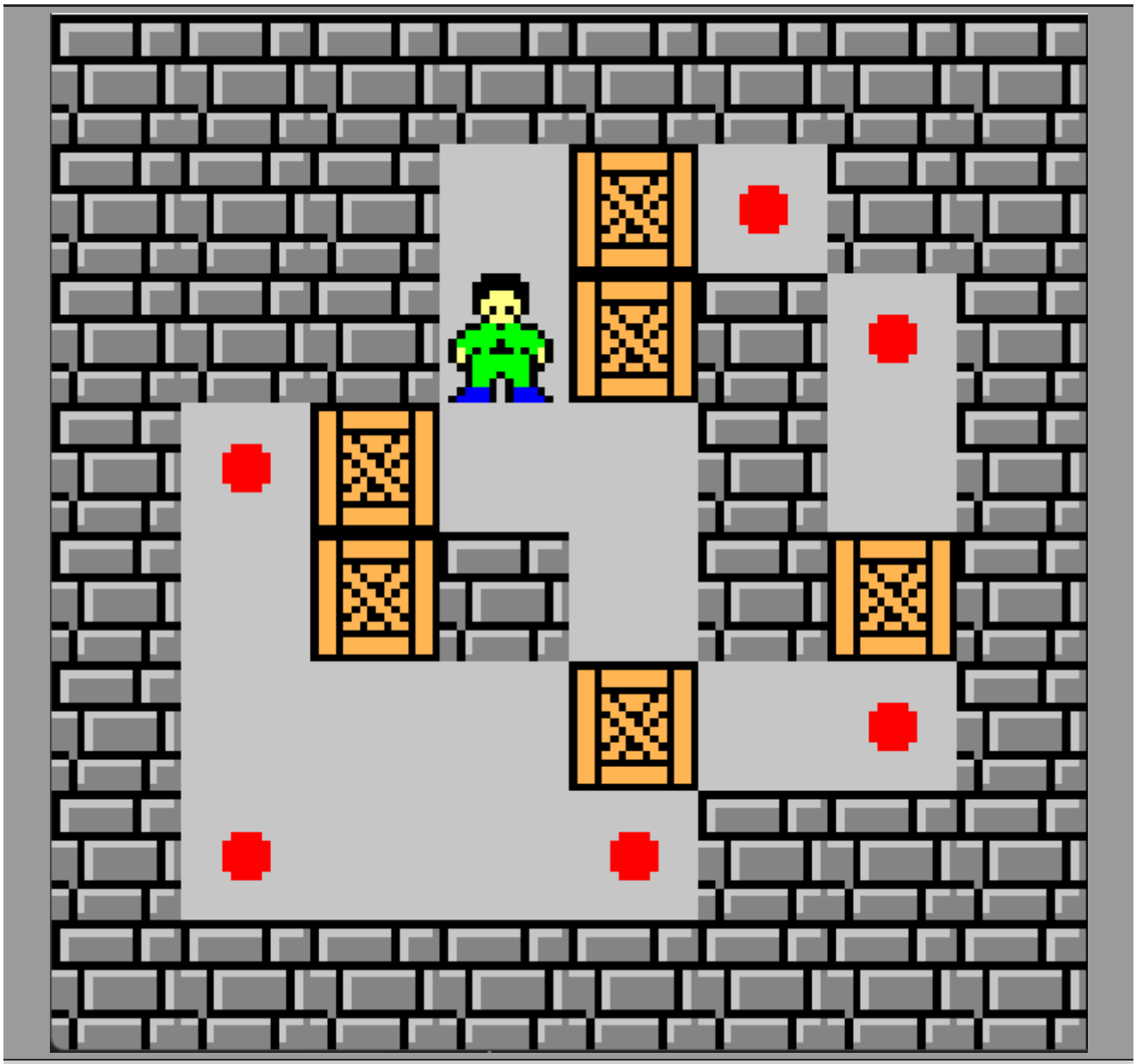

图7 第一次生成

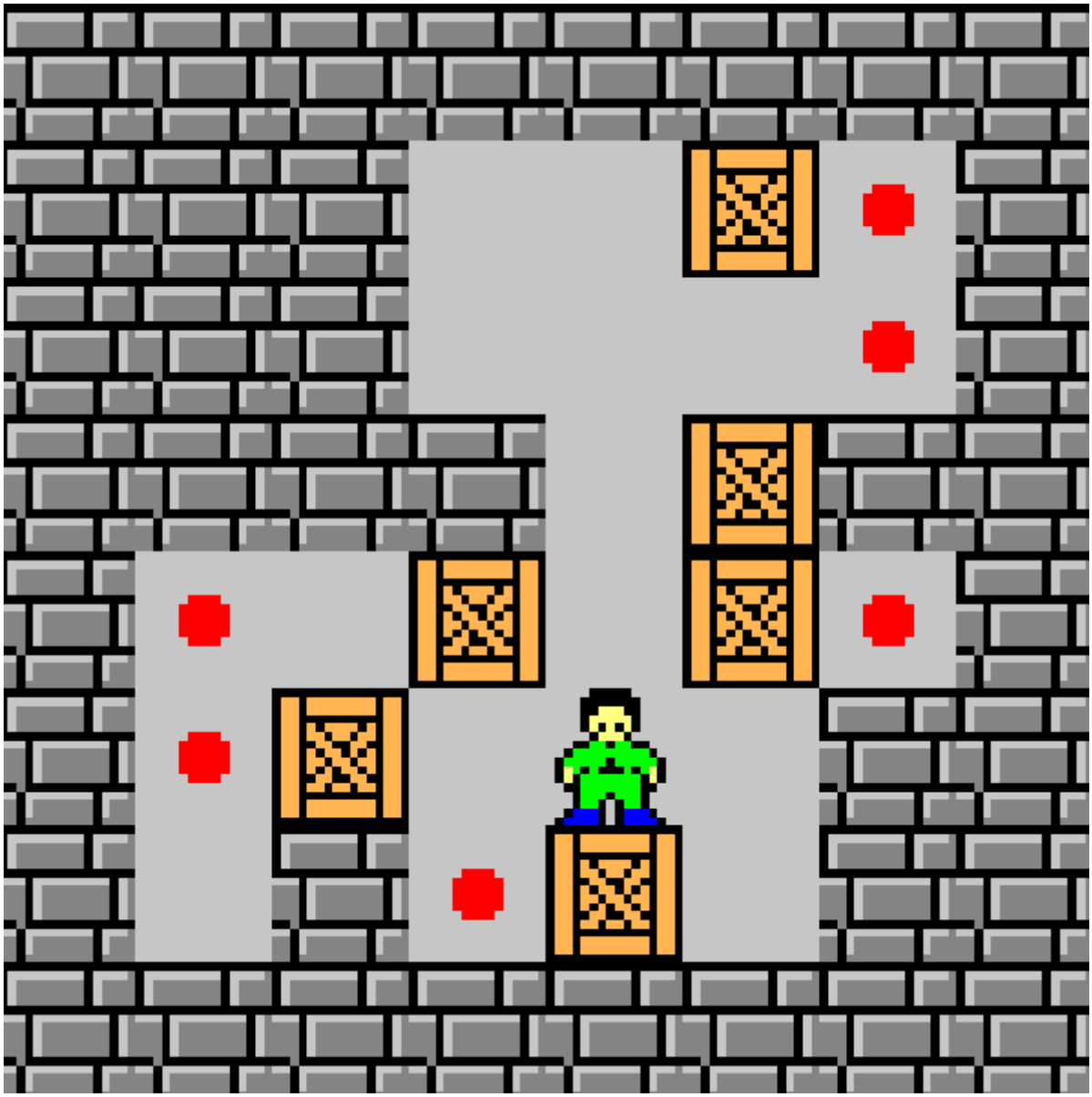

图8 第二次生成

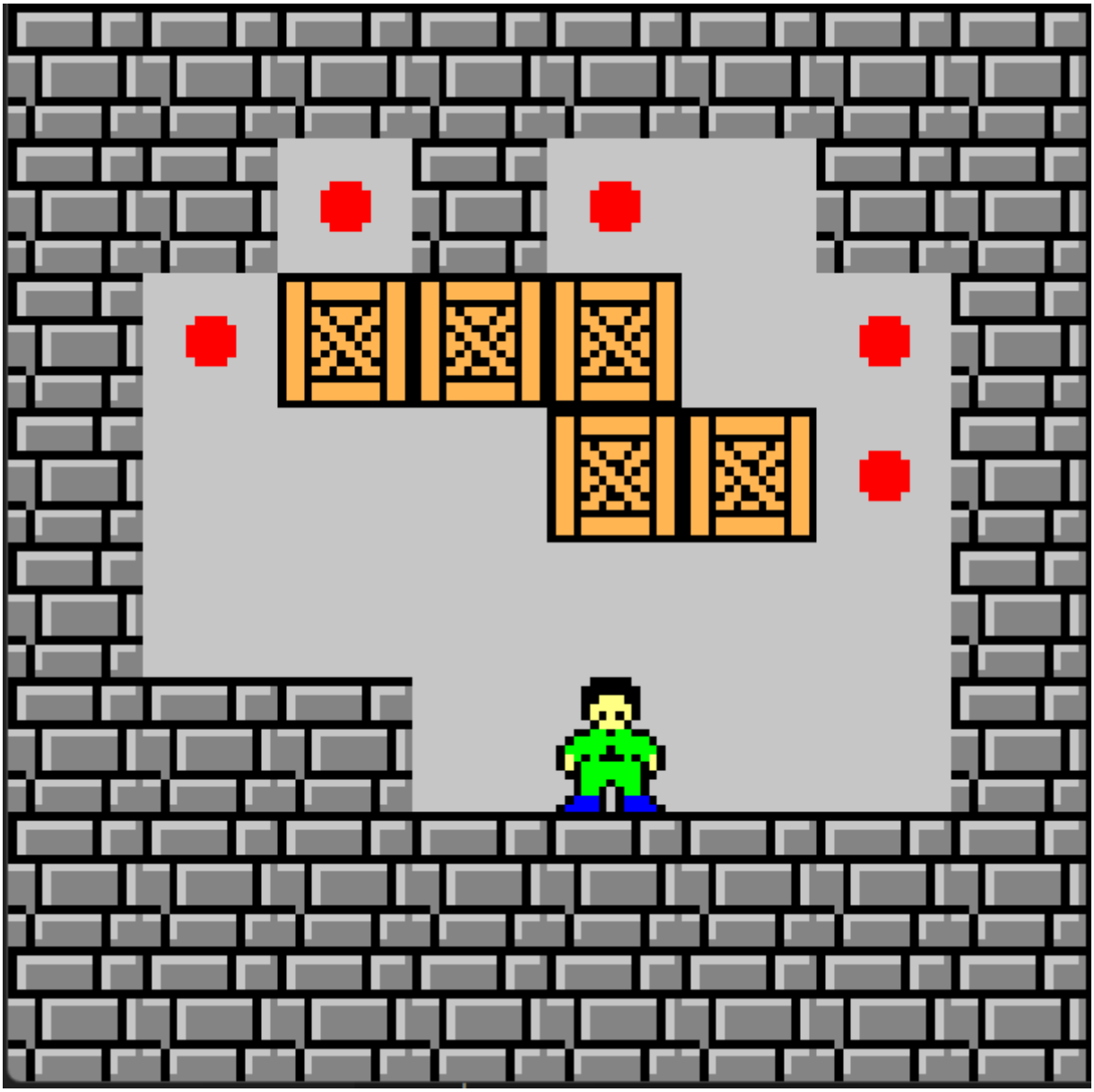

图9 第三次生成

可以看到，生成的地图都是可解的，并且有一定的多样性和难度。

成员分工

- 林佳豪：基本要求1、2，进阶项1
- 田翔宇：进阶项2 我们分别完成了各自负责部分的技术报告。