# 人工智能大作业一：推箱子游戏的实现

彭程　2020011075
清华大学　自动化系　自 02 班

日期：2022 年 11 月 1 日

**摘　　要**

本文为 2022 秋《人工智能基础》大作业一推箱子游戏（自命名“翻滚吧！鸡蛋”）的说明文档兼实验报告。对于最终实现的全部功能，前端的使用方法，算法的实现原理进行详细的说明和介绍。项目地址为 GitHub::pengc02/Sokoban-Game，将在提交截止后上传。

**关键词：** Sokoban, A* Search, Pygame

## 1　功能介绍

### 1.1　运行说明

本次实现了作业要求的全部基本任务，包括：地图生成，AI 完成鸡蛋与终点不一一对应的最简路径搜索，AI 完成鸡蛋与终点一一对应的最简路径搜索；此外，为保证游戏的完整性，额外增加了玩家模式的交互功能，可以像原始 Sokoban 游戏一样实现人机交互。以下是具体介绍：

运行可执行文件（或者 python 文件）后，界面如下图（左）示意，有开始游戏和游戏说明的选项，点击开始游戏，可选择游戏模式，任意选择一游戏模式即进入关卡选择页面，选择关卡后即可自行玩或观看 AI 玩，具体操作如下图的游戏说明或者可执行文件中的游戏说明。

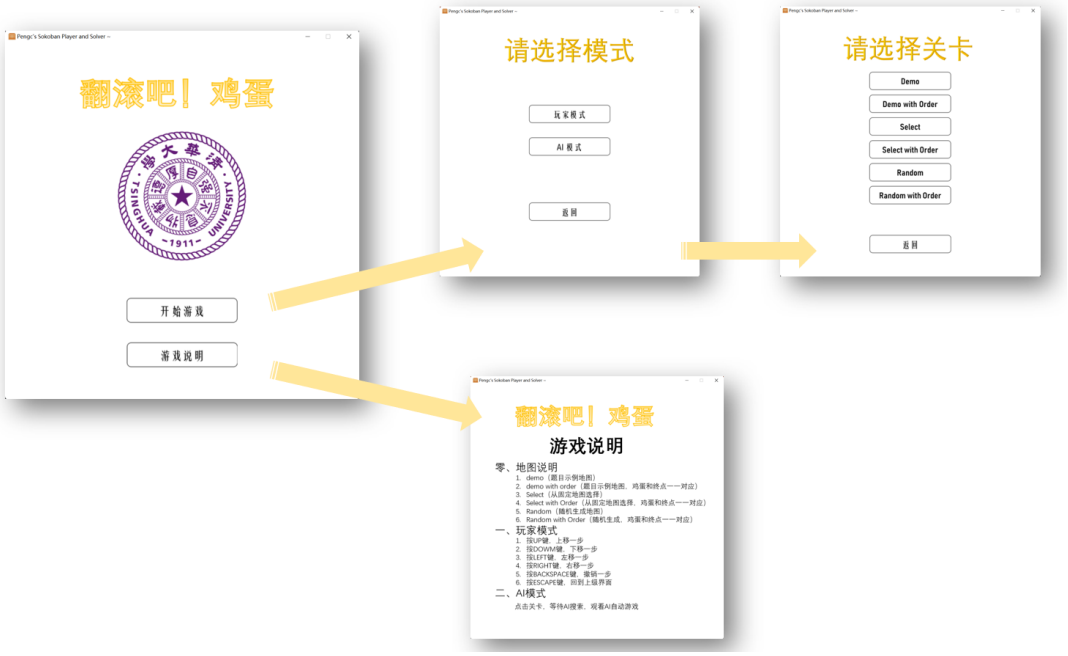


**图 1:** 界面说明

## 1.2 功能 1：地图生成功能

本次游戏提供三种地图选项，包括：demo(即作业说明中提供的地图)，select(从给定的一系列地图中随机抽取)，random(随机生成一种可行的地图)。每种选项都有对应的有序版本和无序版本。如下图所示，其中，demo 和 select 对应的地图数据存储在提前定义好的 json 文件中，random 是即时根据算法随机生成的地图。

可视化后效果如下图所示，其中鸡蛋和目的地上带有的红色字母标记表示他们的对应关系：

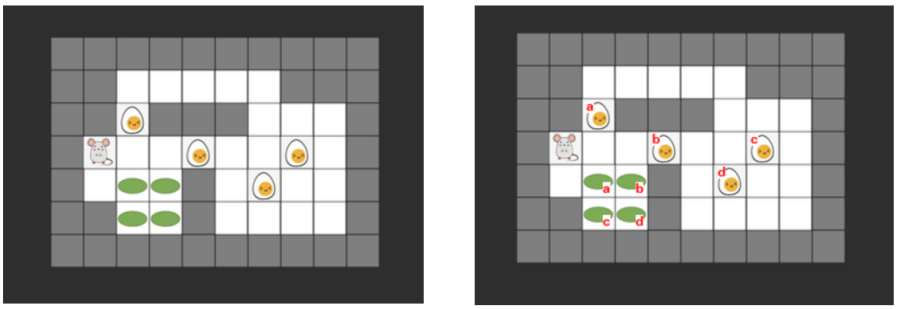

**图 2:** Demo 示例

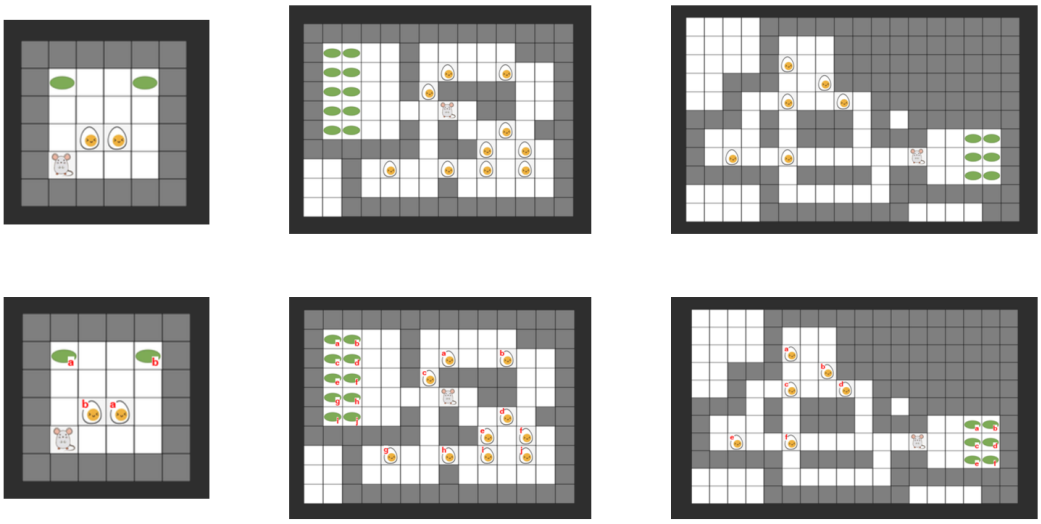

**图 3:** Select 示例

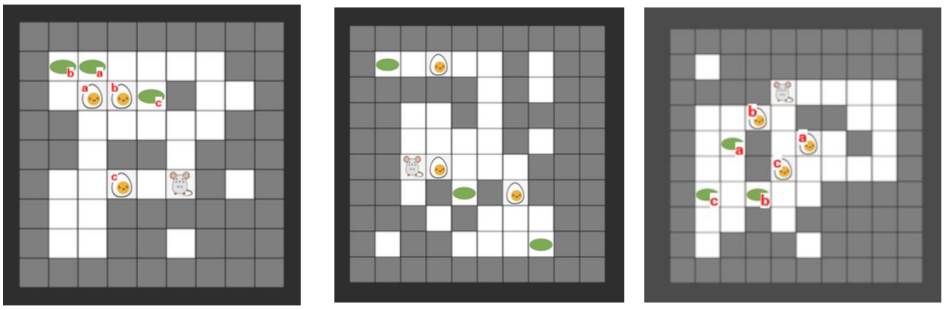

**图 4:** Random 示例

对于 Fig 4中随机生成的地图，经个人测试均有解而且具有一定难度，说明该随机生成算法的可靠性和有效性，从而说明了搜索算法的普适性，关于地图生成的具体策略算法见2.2节介绍。

同时也要注意，由于此处搜索难度较高，耗时较久，使用随机生成地图时需要等待一会，一般来说不会超过 1 分钟。

## 1.3 功能 2：AI 功能——位置不具有对应关系

该功能认为鸡蛋和目的地没有对应关系，目标是找到步数最少的路径实现鸡蛋的移动。

以题干描述中给出的题目为例，进入“开始”界面，点击“开始游戏”，点击“AI 模式”，点击“demo”关卡，等待 AI 搜索，待 AI 搜索得到答案后将在 UI 界面上展示出搜索过程和搜索步数，此处可见 AI 搜索得到的步数为 107 步，耗时约为 20s。

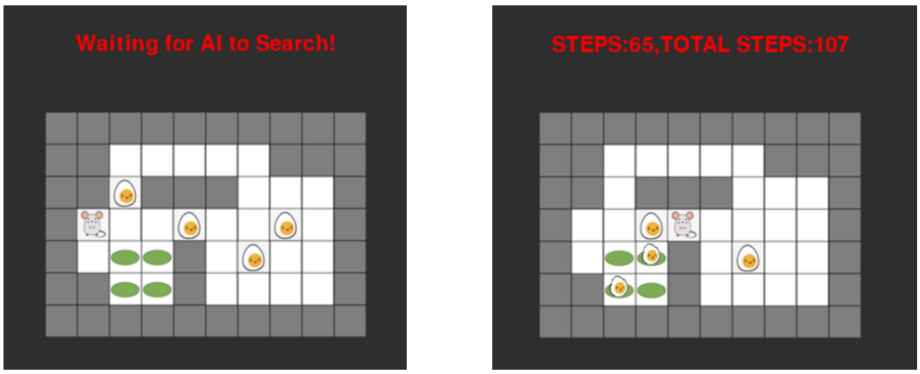


**图 5:** Demo 示例

详细结果如下（其中 udlr 分别代表上下左右移动）：

```
time cost:20.051385402679443
r r r r d d r r u r u u l d l u u l l l l d d d u u u r r r r d d l l l r r r u u l l l l d d
l d r u r r r d d r r u r u l l l l l d u r r r u u l l l l d d l d r u r r r d d r u r u l l
l l r r r u u l l l l d d
depth:107
```

## 1.4 功能 3：AI 功能——位置具有对应关系

同样以题干描述中给出的题目为例，我们改变对应关系，让对应关系与1.3节中不同。

同样，进入“开始”界面，点击“开始游戏”，点击“AI 模式”，点击“Demo with Order”关卡，等待 AI 搜索，AI 搜索得到答案后将在 UI 界面上展示出搜索过程和搜索步数，此处可见 AI 搜索得到的步数为 163 步，耗时约为 550s。可见设定了对应关系后，步数明显增多且复杂程度明显上升。

注意，由于此处搜索难度较高，耗时较久，需要等待一段时间。

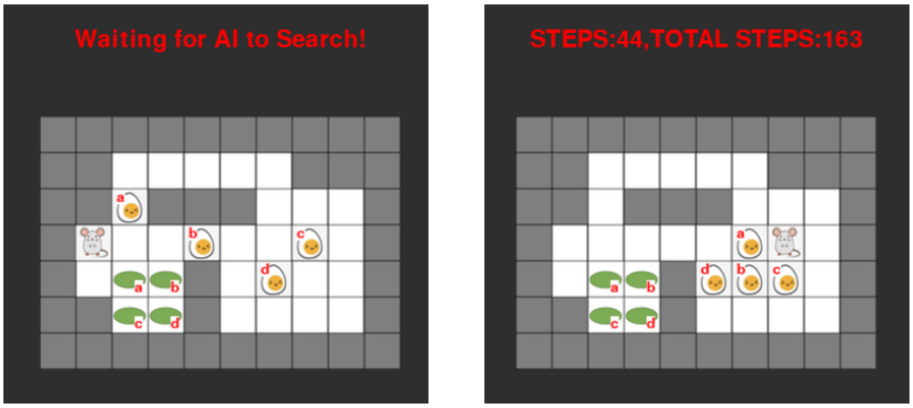


**图 6:** Demo with Order 示例

详细结果如下（其中 udlr 分别代表上下左右移动）：

```
time cost:542.9592864513397
r r r r d d r r u l r r u u l d u l d u u l l l l d d r d d l u l u r r r r r u r r d l r d d
l u d l l u r d r r u u u l l d l l l r r r u u l l l l d d l d r u r r r r u r r d l l l l l
d u r r r u u l l l l d d d u r r r d d r r u r u l l l l l r r r u u l l l l d d l d r u r r
r d d r u r u l l l l r r r u u l l l l l d d
depth:163
```

### 1.5 功能 4：玩家功能

为了方便自我体验和验证，在常规要求之外，我同样实现了玩家功能，即通过键盘的方向键键入移动目标完成鸡蛋的移动。此处不再单独展示该功能，具体操作方法如1.1节的介绍，点击“开始游戏”，点击“玩家模式”，选择任意关卡，进入后通过按“上下左右”键操控老鼠移动推动鸡蛋。

## 2 算法介绍

### 2.1 搜索算法

搜索算法的主要思想是 A* 搜索＋剪枝。关键在于 A* 搜索的函数设计和剪枝的策略。

#### 2.1.1 剪枝策略

对于本游戏我们的剪枝想法是将已经形成死局的节点丢弃，经过分析，我们可以发现如下图所示的情况均构成死局：

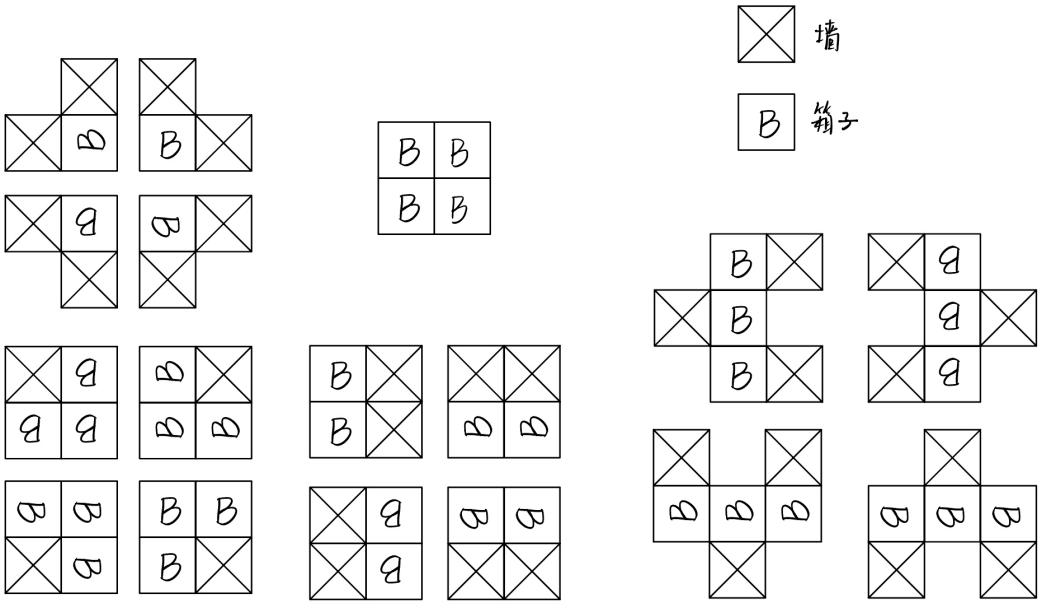


**图 7:** 需要剪枝的情况一览

我们可以认为以一个未到达终点的鸡蛋为中心的 3*3 范围内出现以上情况即为死局，于是我们在搜索时对上述情况进行检验，并将出现这些情况的节点抛弃。

### 2.1.2 A* 搜索

在问题的建模上，我将状态抽象为当前的老鼠和箱子的坐标（存储于 Node 类），状态转移为进行一次合法的移动（由 GameField 判定）。

在存储方面，为了应对复杂的“鸡蛋-目标”对应关系，我舍弃了常用的用字符表示元素，直接存储整张地图的表达方式，采用将地图大小、墙、老鼠的坐标，鸡蛋、目标的编号和坐标存储坐标值在 json 文件中的存储方法，如下例所示：

```
json = {
     "size": [6, 6],
     "goal": [["a", 1, 1], ["b", 1, 4]],
     "wall": [[0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [0, 5], [1, 0],
            [1, 5],[2, 0], [2, 5], [3, 0], [3, 5], [4, 0], [4, 5],
            [5, 0], [5, 1], [5, 2], [5, 3], [5, 4], [5, 5]],
     "box": [["a", 3, 3], ["b", 3, 2]],
     "player": [4, 1]
     }
```

在方法层面，我构建了 GameField 类，功能包括初始化地图，提供成功、失败检验，提供合法移动，进行移动操作，计算代价估计函数等。由于这是一个方法类，我们并不在此类中存放数据，故调用此类时应传入 Node 类中的老鼠和鸡蛋的位置。

A* 搜索的重点在于启发函数的选择，启发函数可以表示为 $f(n) = g(n) + h(n)$，其中 $g(n)$ 为从开始到当前节点的实际代价，$h(n)$ 为未来的估算代价。对于推箱子游戏来说，$g(n)$ 即为当前节点的深度，但 $h(n)$ 有多种选择方法，我的 $h(n)$ 设计方式如下，对于鸡蛋和终点一一对应的情况，只需计算对应点的曼哈顿距离即可；而对于没有一一对应的情况，则要略微复杂些，在保证正确性和较低的复杂性的基础上，

最终我们选用对于鸡蛋和目标的 x 坐标排序后对位相减后取绝对值，y 坐标同理，然后对 x,y 的结果求和。可以证明，这样得到的类曼哈顿距离 ≤ 最邻近点的曼哈顿距离 ≤ 未来实际代价。具体代码实现如下：

```
def cost_predict(self, posBox):
  posGoals = self.goal
  cost = 0

  if self.mode: # 对于有序情况，计算对应位置的曼哈顿距离
     for i in range(self.numofbox):
        cost+=np.abs(posBox[i][1]-posGoals[i][1]) + np.abs(posBox[i][2]-posGoals[i][2])
  else: # 对于无序情况，对x,y排序后计算相应距离
     box_x = sorted([i[1] for i in posBox])
     box_y = sorted([i[2] for i in posBox])
     goal_x = sorted([i[1] for i in posGoals])
     goal_y = sorted([i[2] for i in posGoals])
     for i in range(self.numofbox):
        cost += np.absolute(box_x[i] - goal_x[i]) + np.absolute(box_y[i] - goal_y[i])

  return cost
```

在搜索节点层面，我构建了 Node 类，存储当前老鼠和鸡蛋位置，具有生成子节点功能。

在搜索算法层面，我采用了 A* 搜索加上剪枝的方法，提高效率的关键因素之一在于使用 set 维护开闭节点实现哈希查询。代码及简略说明如下：

```
def Astar_search(problem):
  # 思路 A*搜索，用优先队列和set维护
  # 优先级： F = G（移动代价） + H（预估代价）

  # 维护一些初始化的信息
  sokoban = problem # 定义所用的方法，即为GameField类
  p = sokoban.init_player
  b = sokoban.init_box
  node_start = Node(p, b)
  pqu = PriorityQueue(node_start, sokoban)
  state_hash = dohash(sokoban.mode,p,b) # dohash函数实现了将鸡蛋和老鼠坐标变成字符串，实现可哈希
  close = Set() # 此处闭节点表和开节点表用set实现哈希查询，极大提高了搜索效率
  open = Set()
  open.add(state_hash)
  # 开始A*搜索
  while not pqu.empty():
     node = pqu.pop()
     posPlayer = node.posPlayer
     posBox = node.posBox
     state_hash = dohash(sokoban.mode,posPlayer, posBox)
     open.remove(state_hash)
     close.add(state_hash)
     actions = sokoban.actions(posPlayer,posBox) # 当前状态下所有可能的移动
     for action in actions:
```

```
            new_Player,new_Box = sokoban.move(posPlayer,posBox,action)
            new_node = node.child_node(new_Player,new_Box,action)
            new_state_hash = dohash(sokoban.mode,new_Player,new_Box)
            if sokoban.isSucceed(new_Box): # 成功就返回成功节点
                return new_node
            if sokoban.isFailed(new_Box): # 出现死局就进行剪枝，放入闭节点表
                close.add(new_state_hash)
            elif not close.find(new_state_hash) and not open.find(new_state_hash): #
                 未搜索过则放入优先队列
                pqu.push(new_node)
                open.add(new_state_hash)

    return 0
```

关于具有一一对应关系的任务和不具有一一对应关系的任务，两者仅有启发函数和对于成功的判定方式有差异（只体现在 GameField 中），其余均为相同。

### 2.2 地图生成算法

由于推箱子是一个 NP-hard 问题[1], 并不存在多项式级别的求解办法，因此推箱子问题的地图生成也就没有简单的方案。为了生成具有一定使用意义的地图，我采用了搜索加回溯的策略。

首先初始化一张十分简单的地图，包含鸡蛋、终点、老鼠和少量的墙，保证这张图有解（无解则回溯或丢弃）。之后不断向其中加入墙，如出现无解则回溯至上一次有解重新加，无解次数超过阈值则输出最近一次的可解地图。

当然这样仍然出现了许多问题，例如既是一张十分空旷明显有解的地图，也会因为其约束太少、状态空间过大而搜索不出可行解，为此，我设计了搜索时间的阈值，超过阈值则认为空间太大，于是向地图中加入新的墙，由于我们保证初始简单地图有解，因此最差结果即回溯至初始状态，实验证明在这样的约束下确实可以生成有一定实用意义，结果路径并不是非常简单的地图（见 Fig 4），具体代码实现见”buildmap.py”，此处不再赘述。

## 3 致谢

感谢 Github::KnightofLuna/sokoban-solver对剪枝策略的指导和 Github::ThoseBygones/Sokoban对前端界面流程及 pygame 使用的指导，感谢Github::ElegantLaTeX/ElegantBook的 LaTeX 模板。

此外，还要感谢 gjw20、wzq20 和 cjz20 三位同学在本次作业完成过程中和我的讨论，这给了我很多启发和帮助。

最后，感谢老师和助教对课程辛勤的付出。

[1] M. Fryers, M.T. Greene, Sokoban, Eureka 54 (1995).