计算概论B迷宫大作业

1. 代码说明

该项目所有的源码均存在于code文件夹下。

maze.py包含的迷宫存储、访问、生成和寻路的相关代码。如果直接运行，它会输出一个20*20的迷宫，并直接给出左上到右下的最短路径。

part1.py是第一部分程序以及其调用入口。

示例：

```
4 4
```

输出：

```
0 0 1 0
```

```
1 0 0 0
```

```
0 1 0 1
```

```
0 0 0 0
```

part2.py是第二部分程序以及其调用入口。

示例：

```
4 4
```

```
0 1 0 0
```

```
0 0 0 1
```

```
0 1 0 0
```

```
0 0 1 0
```

输出：

```
(0,0)(1,0)(1,1)(1,2)(2,2)(2,3)(3,3)
```

示例：

```
4 4
```

```
0 1 0 0
```

```
0 1 0 0
```

```
0 1 1 0
```

```
0 0 1 0
```

输出：

```
No possible path!
```

visualize.py包含一个函数visualize，它输入一个迷宫maze.Maze，输出它的可视化图片PIL.Image。如果直接运行，它会直接生成一个100*100的迷宫，找出左上到右下的路径，生成图片并直接调用系统自带的图片查看器进行显示。

part1-visualized.py和part2-visualized.py是part1和part2的可视化版本。注意，在弹出图片窗口之后，还会在控制台询问要把图片保存到哪里。

game.pyw使用了pygame对迷宫进行可视化。你可以不断生成任意你想要大小的**四维**迷宫，从中选取你想知道的两个点并找到它们之间的距离。你还可以通过鼠标自由地构造任何你脑海中的迷宫。

注意，一定要在code文件夹下运行，不要从外部导入part1或者part2，否则会报错ImportError。

---

2. 生成算法

首先让地图中充满了障碍；然后从起点开始不断挖掘，把障碍变成通路，最后形成迷宫。应具备如下条件：

1. 挖掘的目标一定需要在你所在的位置旁边；
2. 挖掘的目标周围只存在一个通路，那就是你所在的那个位置；其余地方全部是障碍。（保证任何位置到起点有且只有一条通路）
3. 如果成功挖掘了一个障碍，移动到这个新的地方并开始挖掘。
4. 如果不存在合适的挖掘目标，后退。

当通路充满了几乎整个网格后，你会后退到起点。这个时候算法结束。

由于这不一定能保证终点所在的位置被挖开，因此需要额外检查终点有没有被挖开；如果没有，那么重新生成迷宫，直到终点被挖开为止。

接下来看Maze.generate函数做了什么。

首先它把自己用障碍物1填满，这对应于地图中全是障碍。

接着进入了一个循环，循环条件正好是自己在右下角的那个位置是1，对应着如果没有挖到终点，那么就不断生成。第一次进入时，由于终点处被填上了1，因此会执行循环内的语句。

循环内，调用了_MazeGenerator的init函数和call函数。其中call函数调用了self.explore函数和randRemove函数。对explore函数的一次调用对应着一次挖掘或者后退。

进入explore函数，首先确认自己是障碍，而且自己周围除了自己的来源地（也就是上一次调用这个函数时，向exploreHistory列表中追加的地方），没有通路。然后把自己挖了，并进一步判断自己周围有没有合适的障碍。把合适的障碍放到targets列表中，随机访问targets中的元素，并以这个元素为中心，再一次调用explore函数。如果没有合适的障碍，explore函数结束，自然就回到了递归的上一级，也就相当于后退。

randRemove函数则是随机选择地图上一定比例的位置，并把这些地方的障碍挖掉。这样就有可能会形成回路，增加迷宫的复杂度。

Post Script:

你可能没有在explore函数中看到递归。这是因为我把递归拆成了一个循环和一个列表，以避免递归深度限制。当递归深度达到三千左右的时候，sys.setrecursionlimit函数也不管用了，唯一能做的就是取缔递归。

---

3. 寻路算法

构造一个前线作为推进基地，每次根据预期的路径长度选择其中一个进行探索，把前线不断向终点方向推进，直到它包含了终点为止。这样得到的路径一定是最短路径。具体过程如下：

1. 从前线中找到一个预期长度最短的点，作为这一轮扩张的推进基地；
2. 遍历推进基地周围的点。如果这个点还没有被探索过，记录走到这里的距离；如果这个点已经被探索过，比较两条路线的长短，使用其中较短的路径。
3. 记录这些点的上一个点是推进基地。
4. 把这些新的点加入前线；把现在的这个推进基地移出前线。

等到探索结束后，我们从终点就可以一步步回到起点，这就能获得中间走过的点的列表。

预期长度：走到这个点已经走过的路程，加上这个点到终点的1-范数。

在_PathFinder类中，history: dict[tuple, np.ndarray[int32]]属性记录了为了走到这一个点，它的前一个点是什么；frontier: list[np.ndarray[int32]]属性记录了当前的前线，其中frontier已经按照预期路径从短到长进行排序；costs: dict[tuple, int]属性记录了从起点走到这个点需要走过多少距离。

---

4. Maze类

Maze类具有以下属性：

data: np.ndarray[np.uint8] 这个数组存储了迷宫的数据。0表示可以经过的路，1表示不能经过的障碍。

shape: tuple 迷宫的形状。理论上可以用self.data.shape代替。

dimension: int 迷宫的维数。理论上可以用len(self.data.shape)代替。

ticker: np.ndarray[np.int32] 一个一维数组，用于描述这个迷宫中的某个位置。

directions: list[np.ndarray[np.int32]] 所有可能的前进方向。这是一个一维数组列表，列举了所有可能的移动方向。二维情形下它的值为[[1,0],[0,1],[-1,0],[0,-1]]。

Maze类具有以下成员函数：

__init__(data=..., shape=(4,4))->None: 给定迷宫数据或者形状，初始化迷宫。如果同时给定数据和形状，忽略形状。如果只给定形状，其数据为全0.

__getitem__(ticker)->int: 给定一个ticker，返回迷宫中对应位置的数据。

__setitem__(ticker, value)->None: 给定一个ticker，把迷宫中对应位置的数据改为value。

__str__()->str: 把迷宫转换为字符串，行内用空格分割，行间用换行符分割。结尾没有换行符。如果是三维或者以上的迷宫，不会输出其中的内容。

legalTicker(ticker)->bool: 返回ticker对应的位置是否位于迷宫内。

neighbors(ticker)->list[np.ndarray[np.int32]]: 给一个位置，返回它所有旁边的位置。

deepcopy()->Maze: 复制自身。

generate()->None: 重新生成一个存在通路的迷宫。

findpath(self, start=None, end=None)->list[tuple]: 给定两个位置，从中寻路。如果不给参数，默认从左上角到右下角寻路。返回路径上各点对应的坐标构成的列表。

---

5. 关于game.pyw

实现了四维迷宫的生成和显示。当w和z长度均为1时，它退化为了二维迷宫。

使用方法：

1. 直接运行game.pyw，自动生成一个(20*20*2)大小的高维迷宫。
2. 使用鼠标左键点击地图上的点将其选中，再选一个点即自动从两个点之间开始寻路。
3. 使用鼠标滚轮，或者按下+/-按键，缩放地图。缩放中心和鼠标指针的位置密切相关。
4. 使用鼠标右键拖动地图，或者按下四个方向键，改变它在屏幕中的相对位置。按下F键使迷宫自动归位。
5. 使用I/J/K/L四个按键改变显示迷宫的Z和W坐标。这样你可以自由地选取四维迷宫中的任何一点。
6. 按下E键或者Edit按钮进入编辑模式。再次按下E键或者PathFind按钮退出编辑模式。
7. 在编辑模式下，使用鼠标左键点击地图上的点将其挖去，使用鼠标右键点击地图上的点放置障碍物。
8. 在编辑模式下，按下C键或者Clear按钮，清空整个地图中的全部障碍物。
9. 在编辑模式下，按下G键或者Generate按钮，重新随机生成迷宫。
10. 按下S键或者Settings按钮，进入设置模式，编辑整个地图的尺寸和维数。如果你不想保存修改，按下Discard按钮；如果希望保存修改，按下Save按钮。
11. 如果发现窗口未响应，属于正常现象，程序正在进行计算，无暇处理事件；但是如果窗口没有发生未响应但是仍然显示异常，尝试按下R键以强制刷新。
12. 支持文本输入。你需要做的是把所有的输入以part2部分的格式放在一个文本文件中，然后将文本文件拖入pygame窗口。请参考sample.txt。
13. img文件夹并非必要，可以删去，并不影响程序正常运行。