Lab2

Problem 1

解题思路

A* 算法是一种很常用的路径查找和图形遍历算法。它有较好的性能和准确度。A * 算法通过下面这个函数来计算每个节点的优先级: $f(n) = g(n) + h(n)$.

其中：

- f (n) 是节点 n 的综合优先级。当我们选择下一个要遍历的节点时，我们总会选取综合优先级最高（值最小）的节点。
- g (n) 是节点 n 距离起点的代价。
- h (n) 是节点 n 距离终点的预计代价，这也就是 A * 算法的启发函数。

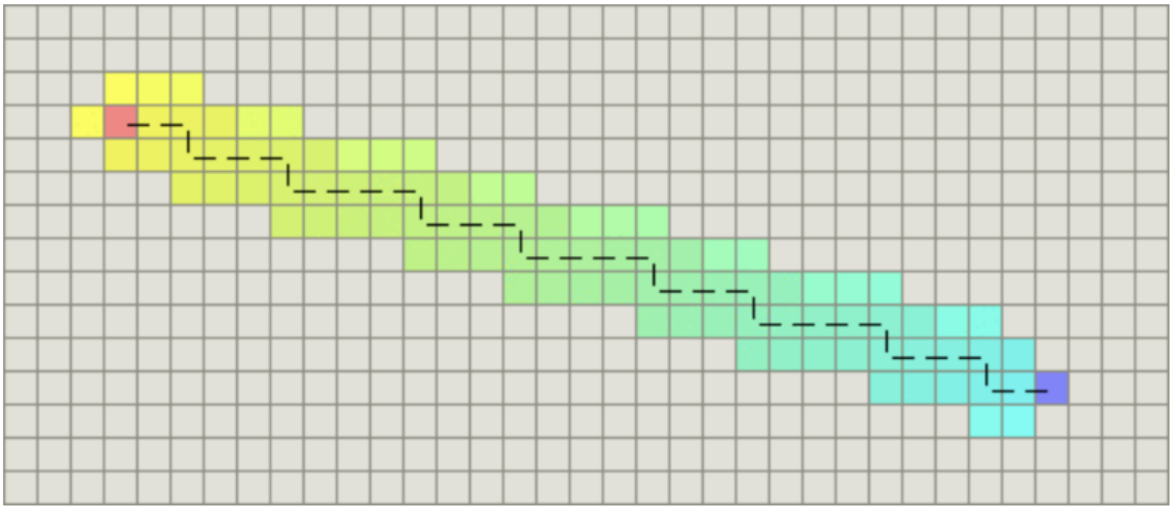
A * 算法在运算过程中，每次从优先队列中选取 f (n) 值最小（优先级最高）的节点作为下一个待遍历的节点。

那么，根据这个原理，我们可以写出以下写代码的思路：

- 初始化 open_set
- 将起点加入 open_set 中，并且将 open_set 设置为堆
    - 如果节点 n 是终点，则：
        - 输出需要移动它的步数
    - 如果节点 n 不是终点，则：
        - 将节点 n 从 open_set 中删除，加入 visited 中
        - 遍历节点 n 的所有邻近节点
            - 如果邻近节点在 visited 中，跳过
            - 如果邻近节点也不在 open_set 中：
                - 将 m 加入 open_set，根据节点 m 的 f 值进行堆排序，加入 open_set 这个堆中。

选取启发函数

如果图形中只允许朝上下左右四个方向移动，则启发函数可以使用曼哈顿距离。



计算曼哈顿距离的函数如下，这里的 D 是指两个相邻节点之间的移动代价，通常是一个固定的常数。

```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy)
```

我们这里的 g 值就采用移动的次数。然后在堆中我们将 g 值与 h 值相加作为 f 值进行堆排序。

代码实现

首先，定义一个 namedtuple 类型的变量，自定义类名为 State，拥有 board, zero_idx, g, h 属性。f 用来记录 f 值也同时作为堆排序的依据，board 用于记录每一个宝石的位置，zero_idx 用于记录 0 的位置，g 用于记录走过的步数，h 用于记录启发函数的值。

```
State = namedtuple("State", ["f", "board", "zero_idx", "g", "h"])
```

将初始状态命名为 initial_state 并且输入到 open_set 中然后进行 heapify，定义 visited 集合表示已经访问过的状态

```
open_set = [initial_state]
heapq.heapify(open_set)
visited = set()
```

当 open_set 不为空时，每次从 open_set 里面 pop out f 值最小的状态，如果 pop 出来的状态与最终状态相同，就直接返回 g 值。

否则，对原图空位旁边的宝石进行上下左右的四个方向的变换。在这里，由于变换的方式只和空位（0）有关，所以我们可以等效于移动 0 的位置，然后将移动的位置和 0 进行交换，得到新的状态。如果状态不在 visited 集合中，即没有被访问过，使用 heappush 方法插入到堆中，再对其使用 f 值进行排序。此处的 f 值等于 g 值 + h 值，g 值为移动步数，h 值为启发函数，即曼哈顿距离。

```
    while open_set:
        current_state = heapq.heappop(open_set)

        if current_state.board == goal_board:
            return current_state.g
```

```
        visited.add(current_state.board)

        x, y = current_state.zero_idx // 3, current_state.zero_idx % 3
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_zero_idx = new_x * 3 + new_y
                new_board = list(current_state.board)
                new_board[current_state.zero_idx], new_board[new_zero_idx] =
new_board[new_zero_idx], new_board[current_state.zero_idx]
                new_board = tuple(new_board)
                if new_board not in visited:
                    f = current_state.g + 1 + manhattan_distance(new_board)
                    new_state = State(f, new_board, new_zero_idx,
current_state.g + 1, manhattan_distance(new_board))
                    heapq.heappush(open_set, new_state)
```

对于每一个输入，我们使用 python 里面的 map 方法先转化为 iterable 的元素，再将其转化为元组，然后对其应用以上的 A* 算法进行搜索。

```
if __name__ == "__main__":
    input_str = ["135720684", "105732684", "015732684", "135782604",
"715032684"]
    for i in range(len(input_str)):
        initial_board = tuple(map(int, input_str[i]))
        print(solve(initial_board))
```

运行结果：

```
1
1
2
1
3
```

Problem2

解题思路

根据题目意思，这道题的建模应该从实现 A* 算法的树搜索入手。为了准确地计算 f 值，我们需要选取合适的 g 值和 h 值。有关 g 值的选取很简单，只要计算当前节点距离起始节点所总共移动过的代价就可以了。关于启发函数的选取，我的思路是这样的：

对于第 2 行到第 M+1 行的每一行输入，我们可以得到 x_i 与 y_i 的距离。但是由于我们的路径是从 N 号到 1 号，并且不能走回头路，所以可以这样想：如果 x_i < y_i 那么将 x_i 与 y_i 交换，这样就可以让每一次都是下降的。而启发函数的选取原则就是小于等于达到目标所需要的真实代价，所以无论 y_i 是否为目标，我们都可以将这一条小路径上面的 cost 值作为启发函数的值。

定义一个邻接表 adj_list，用于存储每一行输入里面，所到达的目标节点，以及需要消耗的代价。那么对于任意一个起始点，将其作为下标输入到邻接表中，都可以返回它的目标和代价。若某个下标所对应的值为空，则返回 0.

我们就把这里的代价作为启发函数的值。

```
def heuristic(room, adj_list):
    return min(adj_list[room], default=0)
```

思路与上题一样，在具体实现方面，需要通过代码来解释具体如何设置的。

代码实现

首先定义一个 namedtuple，存储当前节点的 f 值，位置，g 值，h 值。f 值用来作为堆排序的依据。

```
State = namedtuple("State", ["f", "room", "g", "h"])
```

定义一些参数。设置起始位置为 N，目标状态为 1. 定义 visited 集合为空集，如果一个节点被访问过就加入这个集合。paths 作为所有可行路径的长度集合。

```
start_room = N
goal_room = 1
visited = set()
paths = []
```

由于初始状态的 g 值为 0，那么将其到任意下一个节点的 cost 作为启发函数的值，也就是 f 值。定义初始状态，将 open_set 作为一个容纳所有状态的列表，并首先加入初始状态，然后进行 heapify.

```
f = heuristic(start_room, adj_list)[1]
initial_state = State(f, start_room, 0, heuristic(start_room, adj_list))
open_set = [initial_state]
heapq.heapify(open_set)
```

然后我们就可以逐个计算与初始节点相邻的节点的 f 值，并且加入堆中，进行堆排序。每一次排序之后都取堆顶的值也就是最小值，跳到这个节点并且遍历其相邻节点，计算出 f 值，再重新加入堆中重新进行排序，直到到达终点。

由于这里有一个路径长度为 K 的限制，所以循环条件中，既要让 open_set 不为空，也要让 paths 的长度小于 K。若 open_set 为空，且 paths 的长度小于 K 了，就在循环的外面补上 -1.

如果已经到达了终点，就将其 g 值加入到 path 列表中，然后跳过循环的剩余部分。跳过循环的剩余部分是不想将 State(f, goal_room, g, 0) 加入到 open_set 中，使得多条相同长度的路径所得到的结果，也就是长度，可以得到输出。

```
    while open_set and len(paths) < K:
        current_state = heapq.heappop(open_set)

        if current_state.room == goal_room:
            paths.append(current_state.g)
            continue

        visited.add(current_state)
```

如果没有到达终点，就查看它的邻接节点。如果它的邻接节点的 State 不在 open_set 中，就跳入下一节点，同时更新 State 的各个元素，放入 open_set 中。

这里需要注意的是，如果 next_room 没有后继节点, adj_list[next_room] 应该是 0. 所以 heuristic(next_room, adj_list) 为 0，就不是一般情况下 adj_list 取下标返回的元组类型了。所以这里需要分类讨论一下，如果是元组，就取 [1] 下标；若不是就加 0.

然后我们就可以更新状态，然后放进 open_set 中了。

```
        for next_room, cost in adj_list[current_state.room]:
            if State(f, next_room, current_state.g + cost, heuristic(next_room,
adj_list)) not in visited:
                if isinstance(heuristic(next_room, adj_list), tuple):
                    f = current_state.g + cost + heuristic(next_room, adj_list)
[1]
                else:
                    f = current_state.g + cost
                new_state = State(f, next_room, current_state.g + cost,
heuristic(next_room, adj_list))
                heapq.heappush(open_set, new_state)
```

如果 paths 的长度小于 K，就在最后全部补上 -1. 最后返回 paths

```
    if len(paths) < K:
        paths.extend([-1] * (K-len(paths)))

    return paths
```

运行结果：

所给的五个测试输入的输出分别是：

```
3 3 -1 -1

4 5 6 7

5 5 6 6 7 7

4 4 5 -1 -1 -1 -1

5 5 6 6 6 8 -1 -1
```