3.2

a.

状态：状态由机器人的位置与朝向确定。Agent 的位置数目由迷宫大小决定，朝向有四种：东南西北。

初始状态：迷宫正中间，面朝北

行动：朝东、西、南或北走一段距离

转移模型：行动会产生所期待的后果，在撞墙之前会停步

目标测试：测试当前位置是否可以走出迷宫

路径消耗：机器人所走路程

由于迷宫大小不确定，状态空间可能为无限大

b.

状态：状态由机器人所在路口以及机器人在路口转弯方向确定。朝向有四种：东南西北，路口决定了其是否可以转弯。

初始状态：迷宫正中间，面朝北

行动：沿着道路一直走并可以选择某一方向在交叉路口转弯

转移模型：行动会产生所期待的后果，在撞墙之前会停步

目标测试：测试当前位置是否可以走出迷宫

路径消耗：机器人所走路程

设迷宫交叉路口有 n 个，可选方向有东南西北 4 个，状态空间有 4n

c.

状态：状态由机器人所在路口确定。路口决定了其是否可以转弯。

初始状态：迷宫正中间，面朝北

行动：沿着道路一直走并可以选择任一方向在交叉路口转弯

转移模型：行动会产生所期待的后果，在撞墙之前会停步

目标测试：测试当前位置是否可以走出迷宫

路径消耗：机器人所走路程

设迷宫交叉路口有 n 个，状态空间有 n

d.

①忽略了其他智能体在迷宫中的运动

②机器人的朝向只能为东南西北(迷宫道路横平竖直)

③机器人传感器探测路径的能力(提早发现死胡同)

3.4

设一个八数码问题的最终状态为$\begin{matrix}1&2&3\\8&0&4\\7&6&5\end{matrix}$，此矩阵平铺表示为[1 2 3 8 0 4 7 6 5]

对平铺后的矩阵求逆序数并取奇偶数 R，可得 R=1。

对一个不可解的八数码问题表示为$\begin{matrix}2&1&3\\8&0&4\\7&6&5\end{matrix}$，平铺[2 1 3 8 0 4 7 6 5]，R=0。

当进行移位的时候，在平铺矩阵上表现为 0 与一个数交换位置，不会改变 R 的值。于是可以依据 R=0 与 R=1 划分为两个集合，两个集合不相交，处于同一个子集内的状态一定可达，不同子集的两个状态不可达。

于是设计算法，对每一个平铺矩阵求 R 值，根据 R 值判定当前状态属于哪个子集，根据上述分析过程可知这样对于随机生成的状态是有用的。

算法的 python 实现如下：

```
def R(x):
    N = x.shape[0]
    sum = 0
    for i in range(N - 1):
        sum += np.sum((x[i + 1:N] < x[i]).astype(float))
    return sum % 2
```

3.9

a.

状态：状态由河的一岸传教士人数 C、野人人数 Y、船的位置 B(0 表示在河的一岸，1 表示在另一岸)的三元组(C,Y,B)表示

初始状态：(3,3,1)即野人、传教士、船都在船的一岸

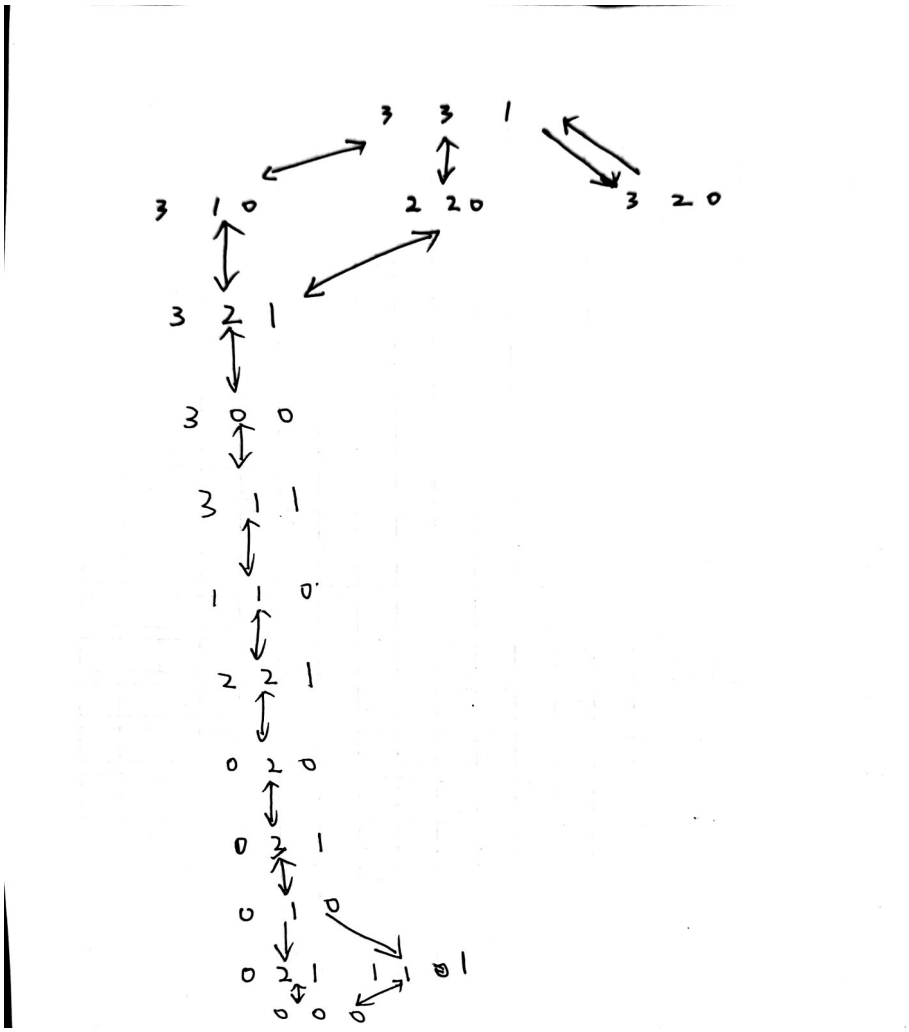
行动：野人和传教士划船在两岸移动

转移模型：在和一岸的人数随着船上、对岸人数进行变化，但必须保证岸上、船上的传教士必须大于等于野人人数，当且仅当某岸传教士为 0 时，野人数目可以大于传教士。当所有人到河的另一岸，结束

目标测试：所有人是否按照约束转移到河对岸

路径消耗：划船次数

状态空间图：



b.

使用基于深度度量的图搜索算法，由于基于深度，所以 open 表可以不进行重排操作，代码运行其中一个最短路径最优解如下

```
PS E:\UCAS\课程\人工智能\作业> python AI_3.py
当前状态: [3, 3, 1]
子节点可能有的状态: [[3, 1, 0], [3, 2, 0], [2, 2, 0]]
当前状态: [3, 1, 0]
子节点可能有的状态: [[3, 3, 1], [3, 2, 1]]
当前状态: [3, 2, 0]
子节点可能有的状态: [[3, 3, 1]]
当前状态: [2, 2, 0]
子节点可能有的状态: [[3, 3, 1], [3, 2, 1]]
当前状态: [3, 2, 1]
子节点可能有的状态: [[3, 0, 0], [3, 1, 0], [2, 2, 0]]
当前状态: [3, 0, 0]
子节点可能有的状态: [[3, 2, 1], [3, 1, 1]]
当前状态: [3, 1, 1]
子节点可能有的状态: [[3, 0, 0], [1, 1, 0]]
当前状态: [1, 1, 0]
子节点可能有的状态: [[2, 2, 1], [3, 1, 1]]
当前状态: [2, 2, 1]
子节点可能有的状态: [[1, 1, 0], [0, 2, 0]]
当前状态: [0, 2, 0]
子节点可能有的状态: [[0, 3, 1], [2, 2, 1]]
当前状态: [0, 3, 1]
子节点可能有的状态: [[0, 1, 0], [0, 2, 0]]
当前状态: [0, 1, 0]
子节点可能有的状态: [[0, 3, 1], [0, 2, 1], [1, 1, 1]]
当前状态: [0, 2, 1]
子节点可能有的状态: [[0, 0, 0], [0, 1, 0]]
当前状态: [1, 1, 1]
子节点可能有的状态: [[0, 0, 0], [0, 1, 0]]
结束
转移过程:
[3, 3, 1]
[3, 1, 0]
[3, 2, 1]
[3, 0, 0]
[3, 1, 1]
[1, 1, 0]
[2, 2, 1]
[0, 2, 0]
[0, 3, 1]
[0, 1, 0]
[0, 2, 1]
[0, 0, 0]
```

检查重复状态在本问题中十分重要，否则程序将会在某两个状态之间不断切换，如[3,3,1]<->[3,2,0]。详细 python 代码见附录。

c.

主要是因为存在约束条件，以及会出现两个状态一直切换无法跳出的情况。

3.12

当每个问题实例都通过单个超级组合行动进行求解时，在搜索过程中，深度搜索和广度搜索的效果是完全一致的，二者没有任何差别。

但是我不认为这是一个加速问题求解过程的实用方法，加速求解是在合理的情况下简化 agent 的动作，如 Go(Sibiu)的点火、刹车等动作对于选择路径代价几乎产生不了影响，所以可以简化掉。但 Go(*)活动的组合简化许多动作，将会造成搜索空间变大，并不利于加速问题的求解。

附录

```
import numpy as np
import queue

op_list = np.array([[0, 2], [0, 1], [1, 1], [1, 0], [2, 0]])  # 船上人员可以存在的集合，第一维为传教士，第二维为野人


def inside(x, a, b):  # a<=x<=b
    if ((x >= a) & (x <= b)):
        return 1
    else:
        return 0


def get_nextstatus(status, op_list):  # 产生下一个状态待选集合
    next_status = []
    a = []
    C = status[0]
    Y = status[1]
    B = status[2]
    for i in range(op_list.shape[0]):
        if (B == 1):
            if (inside(C - op_list[i, 0], 0, 3) & inside(Y - op_list[i, 1], 0, 3) & inside(3 - C + op_list[i, 0], 0,
                                                                                           3) & inside(
                3 - Y + op_list[i, 1], 0, 3)):
                if (((((C - op_list[i, 0]) >= (Y - op_list[i, 1])) & (
                        (3 - C + op_list[i, 0]) >= (3 - Y + op_list[i, 1])))) | (
                            C - op_list[i, 0] == 0) | (3 - C + op_list[i, 0] == 0))):
                    next_status.append([C - op_list[i, 0], Y - op_list[i, 1], 0])
        else:
            if (inside(C + op_list[i, 0], 0, 3) & inside(Y + op_list[i, 1], 0, 3) & inside(3 - C - op_list[i, 0], 0,
                                                                                           3) & inside(
                3 - Y - op_list[i, 1], 0, 3)):
                if ((((C + op_list[i, 0]) >= (Y + op_list[i, 1])) & (
                        (3 - C - op_list[i, 0]) >= (3 - Y - op_list[i, 1])))) | (
                        3 - C - op_list[i, 0] == 0) | (C + op_list[i, 0] == 0)):
                    next_status.append([C + op_list[i, 0], Y + op_list[i, 1], 1])
    return next_status


class agent():  # agent 定义：包括 father 以及当前状态
```

```
    def __init__(self, status):
        self.father = None
        self.status = status

    def getfather(self, father):  # father 赋值
        self.father = father


father_status = [3, 3, 1]  # 初试化状态
open = queue.Queue()
close = queue.Queue()  # open 和 close 表
closed_list = []  # 探索过节点的 list，用于防止重复探索
show_list = []  # 探索路径的展示--需要通过 father 一步一步回调

father = agent(father_status)  # 生成 father
open.put(father)  # open 初试化
while (~open.empty()):
    now = open.get()
    close.put(now)  # 取 open 以及放入 close
    if now.status == [0, 0, 0]:
        print('结束')
        show_list.append(now.status)
        while (now.father != None):
            now = now.father
            show_list.append(now.status)
        print('转移过程：')
        for i in range(len(show_list)):
            print(show_list[len(show_list) - 1 - i])
        break
    if (closed_list.count(now.status) == 0):
        next_status = get_nextstatus(now.status, op_list)
        print('当前状态:', now.status)
        print('子节点可能有的状态:', next_status)
        for x in next_status:
            if (closed_list.count(x) == 0):
                if (now.father != None):
                    if (x != now.father.status):
                        a = agent(x)
                        a.getfather(now)
                        open.put(a)
                else:
                    a = agent(x)
                    a.getfather(now)
```

```
                open.put(a)  # 生成子节点依赖关系，初始状态注意需要特殊处理
    closed_list.append(now.status)  # 扩展已经探索过的状态
```