HSEA-第二次作业-实验报告

191250026 丁云翔

任务一

在evolutionAgents.py 中实现演化算法，并用来玩pacman游戏中的bigMaze、openMaze、smallMaze三个地图。

具体代码在evolutionAgents.py中实现。

实现演化算法的关键步骤

演化算法玩Pacman的整体流程

首先由registerInitialState函数进行第一次getActions，将第一轮的动作序列存入self.actions。

```
    def registerInitialState(self, state):
        """
        This is the first time that the agent sees the layout of the game
        board. Here, we choose a path to the goal. In this phase, the agent
        should compute the path to the goal and store it in a local variable.
        All of the work is done in this method!

        state: a GameState object (pacman.py)
        """
        problem = self.searchType(state)  # Makes a new search problem
        self.actions = self.getActions(problem)  # Find a path
```

之后则调用参数为state的getAction函数，对它进行了修改使之能够在self.actions执行完时调用参数为problem的getAction函数更新新的动作序列，若新的动作序列为空，则到达目标，不需再执行。

```
    def getAction(self, state):
        """
        Returns the next action in the path chosen earlier (in
        registerInitialState).  Return Directions.STOP if there is no further
        action to take.

        state: a GameState object (pacman.py)
        """
        if 'actionIndex' not in dir(self): self.actionIndex = 0
        i = self.actionIndex
        self.actionIndex += 1
        if i < len(self.actions):  # You may need to add some conditions for
taking the action
            return self.actions[i]
        else:  # You may need to use getAction multiple times
            self.actionIndex = 0
            problem = self.searchType(state)
            self.actions = self.getActions(problem)
            if len(self.actions) > 0:
                action = self.actions[self.actionIndex]
                self.actionIndex += 1
```

```
            else:
                action = Directions.STOP
            return action
```

参数为problem的getAction函数是演化算法的主函数，依次进行了类变量的初始化、种群的初始化、迭代T轮、选取fitness最低（这里最低为最优）的解的动作序列。每轮迭代内依次进行了父辈选择、交叉生成子代、获取子代的fitness，直到产生了指定数量的后代，进行生存选择。

```
    def getActions(self, problem):
        '''
        The main iteration in Evolutionary algorithms.
        You can use getFitness, generateLegalActions, mutation, crossover and
other function to evolve the population.
        :param problem:
        :return: the best individual in the population
        '''
        self.problem = problem
        self.population = None
        self.generateLegalActions()
        for i in range(0, self.T):
            offspringList = []
            for j in range(0, int(self.offspringNum / 2)):
                parent1, parent2 = self.selectParents()
                offspring1, offspring2 = self.crossover(parent1, parent2)
                offspring1 = self.mutation(offspring1)
                offspring2 = self.mutation(offspring2)
                score1 = self.getFitness(offspring1[self.actionDim][0])
                score2 = self.getFitness(offspring2[self.actionDim][0])
                offspringList.append([offspring1, score1])
                offspringList.append([offspring2, score2])
            self.population = sorted(self.population, key=lambda x: x[1])
[:self.popSize - self.offspringNum]
            self.population = self.population + offspringList
        actions = []
        self.population = sorted(self.population, key=lambda x: x[1])
        for i in range(1, self.actionDim + 1):
            actions.append(self.population[0][0][i][1])
        if self.population[0][0][self.actionDim][0] in positionDict.keys():
            positionDict[self.population[0][0][self.actionDim][0]] += 1
        else:
            positionDict[self.population[0][0][self.actionDim][0]] = 1
        return actions
```

解的表示和合法解的生成

解的表示包含在一个长度为actionDim+1的数组里。数组元素为一个列表，列表的0号位为执行当前动作后的位置和执行的当前动作组成的列表，1号位为该位置对应的fitness。数组0号位置存储当前演化算法的初始状态，后面依次为actionDim步。数组的1到actionDim号位的动作序列即为最后执行的动作序列。合法解的生成主要由下面两个函数保证。其中generateLegalActions函数是种群初始化时使用的，其使用self.problem.getSuccessors函数获取当前状态可选的动作，再随机选取其中的一个动作，保证每一步获取的动作都是合法的。checkValid函数则是保证突变和交叉后的解合法，其从第一个突变点/交叉点开始，基于前一步所到达的位置对后续的动作序列合法性进行检验，对于合法的动作，不改变动作，只更新位置；对于不合法的动作，则随机选取一个可选动作，并同时更新动作和位置，以保证最后得到的解合法。

```
    def generateLegalActions(self):
        '''
        generate the individuals with legal actions
        :return:
        '''
        self.population = []
        state = self.problem.getStartState()

        for i in range(0, self.popSize):
            individual = []
            currentState = state
            individual.append([state, Directions.STOP])
            for j in range(1, self.actionDim + 1):
                successors = self.problem.getSuccessors(currentState)
                next = random.choice(successors)
                currentState = next[0]
                individual.append([next[0], next[1]])
            score = self.getFitness(currentState)
            self.population.append([individual, score])


    def checkValid(self, individual, start):
        for i in range(start, self.actionDim + 1):
            successors = self.problem.getSuccessors(individual[i - 1][0])
            found = False
            for s in successors:
                if individual[i][1] == s[1]:
                    individual[i] = [s[0], s[1]]
                    found = True
                    break
            if not found:
                next = random.choice(successors)
                individual[i] = [next[0], next[1]]
        return individual
```

评估函数设计

对每个解进行评估，使用解的最后一个动作执行后到达的状态进行评估，一开始使用的曼哈顿距离，但容易陷入局部最优，即在离目标比较近但隔着墙的地方打转，为避免这种情况，我新增了一个positionDict字典，对每次使用演化算法后最后到达的位置进行计数，作为最终位置的次数对fitness影响的放大倍率乘以最后到达的位置之前作为最终位置的次数，再加上曼哈顿距离，得到fitness，当然，fitness越低越优。除此之外，如果不利用A*算法，很难想到其他有效的fitness计算方法可以绕开局部最优，但感觉如果使用A*算法就没有使用演化算法的必要了。

```
def positionHeuristic(state, times):
    '''
    A simple heuristic function for evaluate the fitness in task 1
    :param state:
    :return:
    '''
    if state in positionDict.keys():
        score = abs(state[0] - 1) + abs(state[1] - 1) + times *
positionDict[state]
    else:
        score = abs(state[0] - 1) + abs(state[1] - 1)
    return score
```

```
def getFitness(self, state):
    '''
    evaluate the individuals
    note that you should record the number of using getFitness, and report it at
the end of the task.
    :param state:
    :return:
    '''
    self.callFitness += 1

    return positionHeuristic(state)
```

演化算子设计

突变使用bit-wise，若该位发生突变，则在剩下的可选动作中随机选取一个。（若该状态只有原先的一个动作可选，则不变）

```
    def mutation(self, individual):
        first = 0
        for i in range(1, self.actionDim + 1):
            if random.random() < self.mutateProb:
                successors = self.problem.getSuccessors(individual[i - 1][0])
                otherAction = []
                for j in range(0, len(successors)):
                    if successors[j][1] != individual[i][1]:
                        otherAction.append([successors[j][0], successors[j][1]])
                if len(otherAction) != 0:
                    next = random.choice(otherAction)
                    individual[i] = [next[0], next[1]]
                if first == 0:
                    first = i
        if first == 0:
            return individual
        else:
            return self.checkValid(individual, first + 1)
```

交叉使用单点交叉。

```
    def crossover(self, parent1, parent2):
        crossPoint = random.randint(2, len(parent1[0]) - 1)
        offspring1 = parent1[0][:crossPoint] + parent2[0][crossPoint:]
        offspring2 = parent2[0][:crossPoint] + parent1[0][crossPoint:]
        return self.checkValid(offspring1, crossPoint),
self.checkValid(offspring2, crossPoint)
```

父辈选择使用轮盘赌法，fitness越小，被选中的概率越大。

```
    def selectParents(self):
        self.population = sorted(self.population, key=lambda x: x[1])
        return self.rouletteWheel()

    def rouletteWheel(self):
        sumFitness = 0
        choosen = []
```

```
        if self.population[0][1] != 0:
            for i in range(0, self.popSize):
                sumFitness += 1.0 / self.population[i][1]
            for _ in range(0, 2):
                temp = 0
                randomNum = random.uniform(0, sumFitness)
                for i in range(self.popSize - 1, -1, -1):
                    temp += 1.0 / self.population[i][1]
                    if randomNum <= temp:
                        choosen.append(self.population[i])
                        break
        else:
            for i in range(0, self.popSize):
                sumFitness += 1.0 / (self.population[i][1] + 1)
            for _ in range(0, 2):
                temp = 0
                randomNum = random.uniform(0, sumFitness)
                for i in range(self.popSize - 1, -1, -1):
                    temp += 1.0 / (self.population[i][1] + 1)
                    if randomNum <= temp:
                        choosen.append(self.population[i])
                        break
        return choosen[0], choosen[1]
```

生存选择使用精英策略，优先保留fitness低的。但这样容易导致陷入局部最优，考虑修改为优先保留新的。

实验效果

具体参数设定以及使用评估函数的次数及分数

smallMaze

动作序列长度actionDim=20，迭代轮数T = 100，种群大小popSize = 100，每位的突变概率mutateProb = 0.2，每轮产生的后代数量offspringNum = 20，作为最终位置的次数对fitness影响的放大倍率times = 10。

运行了五次，结果如下

	使用评估函数的次数	分数
1	23100	309
2	16800	351
3	23100	309
4	18900	339
5	18900	337

openMaze

动作序列长度actionDim=30，迭代轮数T = 200，种群大小popSize = 200，每位的突变概率mutateProb = 0.2，每轮产生的后代数量offspringNum = 40，作为最终位置的次数对fitness影响的放大倍率times = 20。

运行了五次，结果如下

	使用评估函数的次数	分数
1	16400	450
2	32800	418
3	32800	402
4	24600	420
5	24600	432

运行时可发现可以很好的跳出局部最优。

bigMaze

动作序列长度actionDim=10，迭代轮数T = 100，种群大小popSize = 100，每位的突变概率mutateProb = 0.2，每轮产生的后代数量offspringNum = 20，作为最终位置的次数对fitness影响的放大倍率times = 50。

运行了五次，结果如下

	使用评估函数的次数	分数
1	504000	-1886
2	539700	-2058
3	478800	-1768
4	380100	-1298
5	302400	-924

针对这种局部最优特别多的迷宫型地图，只能降低每次演化算法的计算负担，让它在较短的时间内在局部最优附近徘徊，并将他们作为最后位置的数量刷上去，这样才能尽快跳出，故按照这个理念设置了参数。但在实际运行时，虽然能够跳出局部最优，但算法仍然难以将正确路径和断头路区分开，实际运行下来仍然需要花费大量时间。

任务二

这个任务要求你在Search和Classic环境中测试你在任务一中实现的演化算法，具体测试地图：

- Search1、 Search2 、 Search3；
- minimaxClassic、originalClassic、powerClassic。

测试结果

由于此前演化算法的启发式函数是基于找出口而设计的，在找食物的环境中表现其实比较一般。

Search1

动作序列长度actionDim=5，迭代轮数T = 50，种群大小popSize = 50，每位的突变概率mutateProb = 0.2，每轮产生的后代数量offspringNum = 10，作为最终位置的次数对fitness影响的放大倍率times = 50。

运行了五次，结果如下

	使用评估函数的次数	分数
1	3850	506
2	4950	498
3	4400	504
4	3850	506
5	3850	506

因为这个地图很小，其实动作序列不需要很长，由于动作序列短，种群的大小、迭代次数、每次迭代的后代数都可以相应减少，放大倍率可以设置的大一些，以偏向于探索未访问的地方，更快吃到食物而不是在原先问题的目标附近徘徊。

Search2

动作序列长度actionDim=5，迭代轮数T = 50，种群大小popSize = 50，每位的突变概率mutateProb = 0.2，每轮产生的后代数量offspringNum = 10，作为最终位置的次数对fitness影响的放大倍率times = 100。

运行了五次，结果如下

	使用评估函数的次数	分数
1	7700	560
2	6600	571
3	7700	560
4	6600	572
5	7700	561

地图变大了，放大倍率可以再设置的大一些，以偏向于探索未访问的地方，更快吃到食物而不是在原先问题的目标附近徘徊。

Search3

动作序列长度actionDim=10，迭代轮数T = 20，种群大小popSize = 20，每位的突变概率mutateProb = 0.2，每轮产生的后代数量offspringNum = 10，作为最终位置的次数对fitness影响的放大倍率times = 150。

运行了五次，结果如下

	使用评估函数的次数	分数
1	5280	579
2	4840	590
3	4620	604
4	10780	321
5	5060	587

地图又变大了，同样放大倍率可以再设置的大一些，同时可以也适当增大一下动作序列长度，并减少迭代轮数和种群大小，以使解不容易偏向向原先问题的目标聚集。

minimaxClassic

动作序列长度actionDim=5，迭代轮数T = 20，种群大小popSize = 20，每位的突变概率mutateProb = 0.2，每轮产生的后代数量offspringNum = 10，作为最终位置的次数对fitness影响的放大倍率times = 50。

运行了五次，结果如下

	使用评估函数的次数	分数
1	440	-499
2	440	-497
3	220	-492
4	220	-493
5	220	-493

地图比较小，故动作序列长度也设置的小一点，但地图太小，精灵的密度太大，启发式函数又没用到食物信息，这种地图该演化算法基本无法获胜。

originalClassic

动作序列长度actionDim=10，迭代轮数T = 20，种群大小popSize = 20，每位的突变概率mutateProb = 0.2，每轮产生的后代数量offspringNum = 10，作为最终位置的次数对fitness影响的放大倍率times = 150。

运行了五次，结果如下

	使用评估函数的次数	分数
1	4620	-47
2	1100	-354
3	3300	-267
4	1540	-366
5	3080	-253

地图比较大，同样采用类似Search3的思路设置参数。

powerClassic

动作序列长度actionDim=10，迭代轮数T = 20，种群大小popSize = 20，每位的突变概率mutateProb = 0.2，每轮产生的后代数量offspringNum = 10，作为最终位置的次数对fitness影响的放大倍率times = 100。

运行了五次，结果如下

	使用评估函数的次数	分数
1	1760	409
2	1980	-5
3	3740	1004
4	2200	-1
5	2860	622

地图大小中等，故可将放大倍率稍微调小一些，食物密度大，且大食物多，比较适合这个算法。

任务三

对所实现的演化算法进行改进，并在任务二中的几个环境中进行测试。

可能的改进方向：

- 解的表示改进
- 评估函数改进
- 演化算子改进
- 参数优化

改进思路

首先是要改进评估函数，即启发式函数的设计，使之能够利用食物、精灵、和大食物位置信息，因为之前的启发式函数实际上是基于固定位置目标设计的，其实并不适用于吃食物和躲避精灵。改进后应使得趋向于食物和大食物的解的fitness小（更优），当处于无敌状态时，趋向于精灵的解的fitness小（更优），反之，远离精灵的解的fitness小（更优），使fitness的计算更加准确。

其次可以将生存选择从保留较优的改为保留较年轻的，这样有助于跳出局部最优。

参数优化与场景本身的性质关系比较大，包括地图大小、墙的布局、任务类型等，可以对不同的场景使用不同的参数组合进行多次测试再取平均，找到更适合每个场景的参数组合。

具体改进和改进后的效率

由于时间关系，未能完成改进的实现，故这一部分略过。