人工智能第一次实验实验报告

舒文炫

2021 年 5 月 25 日

目录

Chapter 1

实验介绍

1.1 实验内容

本次实验分两个部分，分别是 Search 和 multiagent

- Search 目标是吃豆人寻找食物，即静态查找算法，本次实验需要实现 BFS 算法和 A* 算法。
- Multiagent 的目标是吃完所有食物同时避开鬼，其目的是在有对手的情况下做下一步决策使得自己利益最大化，这里需要实现 minimax 算法和 alpah-beta 剪枝。

1.2 实验环境

Python3.6，我使用了 anaconda 来管理 python 环境，脚本运行使用了 Git Bash。

Chapter 2

实验设计

2.1 part1：search

要完成 BFS 算法，需要有一个数据结构，可以保存当前节点的所有子节点，且子节点的访问顺序要在所有上一层节点访问完成之后，这样使用队列就很合适，队列可以实现数据先进先出，父节点先进，子节点在所有父节点后面。从前往后依次访问即可

对 A* 算法，需要一个启发式函数（这个助教已经给出，只要调用即可）以及当前路径总代价，加起来算做总代价的估计，我们每次选择代价最小的扩展，这样可以考虑使用优先队列，代价越小，优先级越高，就在队列越前面，这样依次扩展优先队列节点即可。

比较方便的是，助教已经给出了队列以及优先队列这些数据结构，可以直接使用。

2.2 part2:multiagent

对 minimax 算法，我们需要在博弈树上搜索，所提供的参数有搜索的深度，即往后看的回合数，以及 ghost 的数量，对每个状态，需要判断该状态是 agent 操作还是 ghost 操作，对 agent 节点我们要选最大的那个，对 ghost 节点我们要选最小的那个。由于这个博弈树，状态信息都已经封装好了，我在这里完成的为节点的选择。由于深度不一定为 1，我考虑的框架是调用 minimax 函数递归，遍历当前节点所有的子节点，对深度为 1，可以直接返回结果，大于 1 的可以通过深度为 1 的子问题间接得到。对深度为 1 的情况，由于 ghost 的数目不一定为 1，即取 min 结点的次数不会仅为 1 次，此时都是在同一深度，这里就需要判断在当前状态应该取 max 还是 min。观察发现：agent 节点需要取 max，ghost 节点要取 min，但是要注意一点，ghost 后可能还会接 ghost，当 ghost 后面是 agent 时，是当次博弈结束，如果深度为 1 可以直接返回结果，如果深度大于 1，此时深度要减 1。

对 alpha-beta 剪枝算法，这是对 minimax 的改进，剪枝使得我们没必要扩展所有的节点，我们可以将一些情况剪去，从而加快搜索。主体仍选择 minimax 算法所描述的方式，需要加上 alpha，beta 参数，保存到目前为止路径上发现的 MAX 极大值以及 MIN 极小值。在 max 节点处若发现拓展的点的值比 beta 值大，则后面的情况直接剪去，即返回到上层递归的位置。在 min 节点处若发现拓展的点的值比 alpha 值小，则后面的情况直接剪去，即返回到上层递归的位置。

Chapter 3

算法实现

下面我将展示我的代码来解释具体实现：

3.1 part1:search

```
def myBreadthFirstSearch(problem):
    visited = {}
    frontier = util.Queue()

    frontier.push((problem.getStartState(), None))

    while not frontier.isEmpty():
        state, prev_state = frontier.pop()

        if problem.isGoalState(state):
            solution = [state]
            while prev_state != None:
                solution.append(prev_state)
                prev_state = visited[prev_state]
            return solution[::-1]

        if state not in visited:
            visited[state] = prev_state

            for next_state, step_cost in problem.getChildren(state):
                frontier.push((next_state, state))

    return []
```

图 3.1: BFS

初始化 frontier 为 queue（队列），先将开始节点入队列，进入循环，只要 frontier 队列不空，将其第一个元素出队列，判断其是否为终止态，是则结束，此时找到了一条路径。若不是，更新 visited 字典，表示该节点已经扩展过，键是当前状态，值是其父节点，从而 visited 字典可以用来保存一条到达终点的路径。再循环将其所有子节点入队列，这个就实现了广度优先。这里的框架与 BFS 基本相同，我们用到了 priorityqueue 这个数据结构，维护它用到的是堆排序算法。计算优先级的方法，由于不能直接得到到当前节点路径的代价，只有每一步的代价，我保存了一个 pri 数组，用来储存每个节点 n 的 pri 值，即

$$pri(n) = h(n) + g(n)$$

```python
def myAStarSearch(problem, heuristic):
    visited={}
    pri={}
    frontier=util.PriorityQueue()
    g1=0
    pri[problem.getStartState()]=g1+heuristic(problem.getStartState())
    frontier.update((problem.getStartState(), None),g1+heuristic(problem.getStartState()))
    while not frontier.isEmpty():

        state, prev_state = frontier.pop()
        if problem.isGoalState(state):
            solution=[state]
            while prev_state!=None:
                solution.append(prev_state)
                prev_state = visited[prev_state]
            return solution[::-1]

        if state not in visited:
            visited[state] = prev_state

            for next_state, step_cost in problem.getChildren(state):
                pri[next_state]=heuristic(next_state)+pri[state]-heuristic(state)+step_cost
                frontier.update((next_state, state),pri[next_state])
    return []
```

图 3.2: A* search

。h(n) 的值我们可以直接调用 heuristic 函数得到。那么下一个节点 next 的优先值

$$pri(next) = pri(n) - h(n) + step\_cost + h(next)$$

然后使用 update 函数去更新队列，将最小值放在队列的头，方便之后 pop 出去。

3.2 part2:multiagent

minimax 函数用来递归，如果当前状态是终止状态则直接返回该状态的值，若不是，进入到下面主体，按照我在实验设计里面的描述，我将情况分为了三种

- 当前状态为 agent 下一个为 ghost，此时对所有子节点找 max，同时递归深度不变。
- 当前状态为 ghost 下一个为 ghost，此时对所有子节点找 min，同时递归深度不变。
- 当前状态为 ghost 下一个为 agent，此时对所有子节点找 min，递归深度减 1，此时进入到了下一回合，深度为 1 时需要直接返回，作为递归的出口。

最后返回最佳状态和最好的值。

alpha-betacut 算法和 minimax 主体相同，这里我直接在 getNextState 函数里面定义了 alphabetacut 函数方便进行递归。这里多出了两个参数 alpha，beta，其含义在实验设计里面已经提过。我这里仅指出与 minimax 函数不同的地方。中间变量 alpha1，beta1 用来保存临时的 alpha，beta 值

- 当前状态为 agent 下一个为 ghost，此时对所有子节点找 max，同时递归深度不变, 这时在原来的层，可能会修改 alpha 值，所以需要传入 alpha1，即改变后的 alpha 值。若发现拓展的点的值比 beta 值大，则后面的情况直接剪去，即返回到上层递归的位置。
- 当前状态为 ghost 下一个为 ghost，此时对所有子节点找 min，同时递归深度不变, 这时在原来的层，可能会修改 beta 值，所以需要传入 beta1，即改变后的 beta 值。若发现拓展的点的值比 alpha 值小，则后面的情况直接剪去，即返回到上层递归的位置。
- 当前状态为 ghost 下一个为 agent，此时对所有子节点找 min，递归深度减 1，此时进入到了下一回合，深度为 1 时需要直接返回，作为递归的出口。传参与上一条相同。

getNextState 函数主体：设置 alpha，beta 初始值以及调用了 alphabetacut 函数。

```python
def minimax(self, state, depth):
    if state.isTerminated():
        return None, state.evaluateScore()

    best_state, best_score = None, -float('inf') if state.isMe() else float('inf')
    for child in state.getChildren():
        if state.isMe() and not child.isMe():
                _,n_score=self.minimax(child,depth)
                if n_score>=best_score:
                    best_state=child
                    best_score=n_score
        if not state.isMe() and not child.isMe():
            _,n_score=self.minimax(child,depth)
            if n_score<best_score:
                best_state=child
                best_score=n_score
        if not state.isMe() and child.isMe():
            if depth==1:
                n_score=child.evaluateScore()
                if n_score<best_score:
                    best_state=child
                    best_score=n_score
            else:
                _,n_score=self.minimax(child,depth-1)
                if n_score<best_score:
                    best_state=child
                    best_score=n_score



    return best_state, best_score
```

图 3.3: minimax

```python
def alphabetacut(state,alpha,beta,depth):
    if state.isTerminated():
        return None, state.evaluateScore(),alpha,beta
    best_state, best_score = None, -float('inf') if state.isMe() else float('inf')
    alpha1=alpha
    beta1=beta
    for child in state.getChildren():

        if state.isMe() and not child.isMe():
            _,n_score,alpha1,beta1=alphabetacut(child,alpha1,beta,depth)
            if n_score>=best_score:
                best_state=child
                best_score=n_score
            if best_score>beta:
                return best_state, best_score, alpha1, beta1
            alpha1=max(alpha1,best_score)


        if not state.isMe() and not child.isMe():
            _,n_score,alpha1,beta1=alphabetacut(child,alpha,beta1,depth)
            if n_score<best_score:
                best_state=child
                best_score=n_score
            if best_score<alpha1:
                return best_state, best_score, alpha1, beta1
            beta1=min(beta1,best_score)
```

(a) alpha-beta cut part1

```python
    if not state.isMe() and child.isMe():
        if depth==1:
            n_score=child.evaluateScore()
            if n_score<best_score:
                best_state=child
                best_score=n_score
            if best_score<alpha1:
                return best_state, best_score, alpha1, beta1
            beta1=min(beta1,best_score)
        else:
            _,n_score,alpha1,beta1=alphabetacut(child,alpha,beta1,depth-1)
            if n_score<best_score:
                best_state=child
                best_score=n_score
            if best_score<alpha1:
                return best_state, best_score, alpha1, beta1
            beta1=min(beta1,best_score)


return best_state, best_score, alpha1, beta
```

(b) alpha-beta cut part2

```python
alpha=-float('inf')
beta=float('inf')
best_state, a, b, c= alphabetacut(state,alpha,beta,self.depth)
return best_state
```

图 3.4: getnextstate

Chapter 4

实验测试以及结果分析

4.1 part1: search

测试结果以图形化界面表示：这三幅图，分别对应三种算法搜索扩展的节点，红色表示扩展的路径节

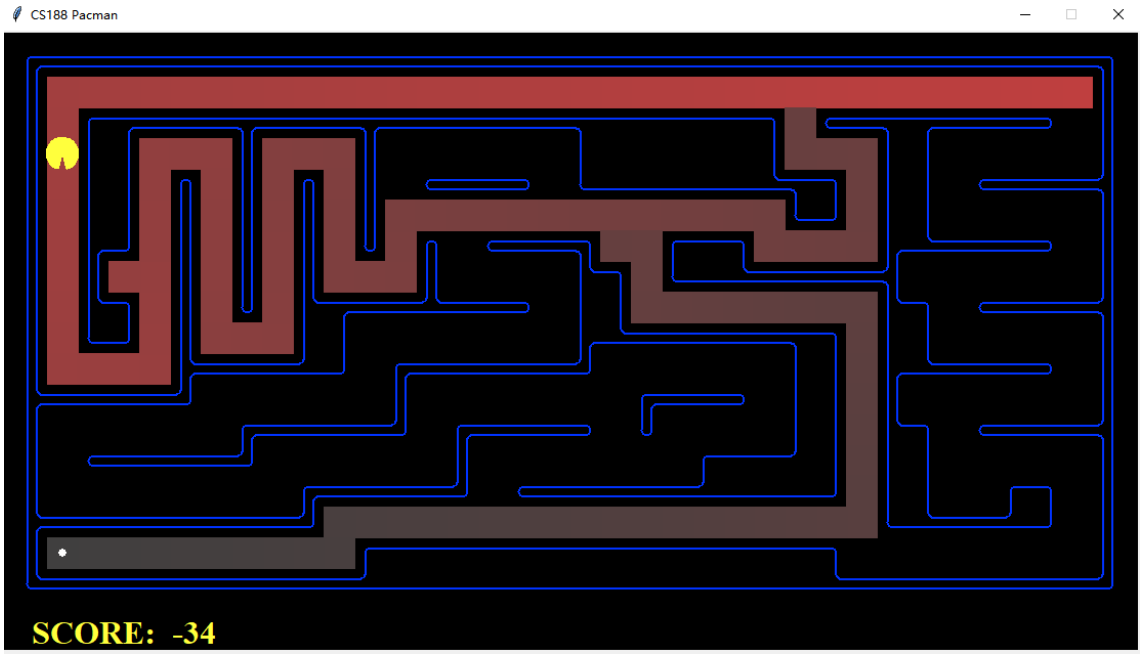

图 4.1: DFS

点，比较三幅图，我们发现 DFS 算法，扩展的节点是最少的，这是因为该算法会对一条路径走到底，如果发现了目标就结束，这里扩展的路径比较好，很快找到了目标。BFS 算法扩展的节点是最多的，这是它每下一层，都会把该层所有节点遍历一遍，然而在大多数情况下，这些扩展是没有意义的。这两个算法是无信息搜索，A* 算法是有信息搜索，我们知道了启发式函数以及每一步的代价，扩展的节点少一些，这里扩展出了一条代价最小的路径。

4.2 part2:multiagent

这里测试的时候，对所有测试样例都 PASS，说明我的算法实现正确。

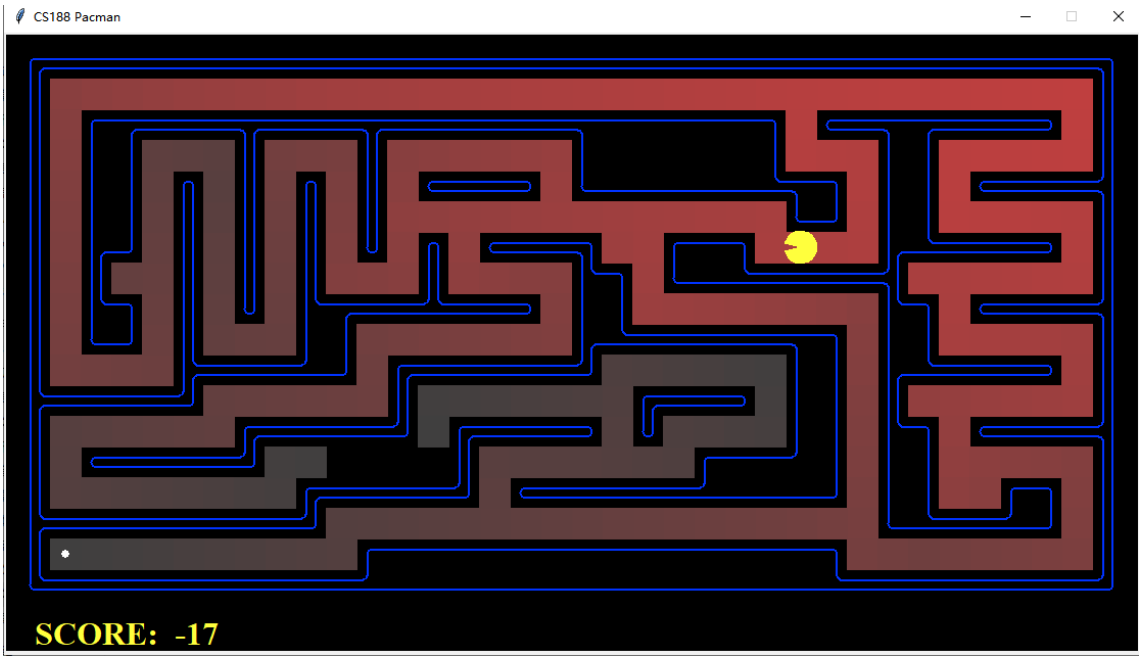

图 4.2: BFS

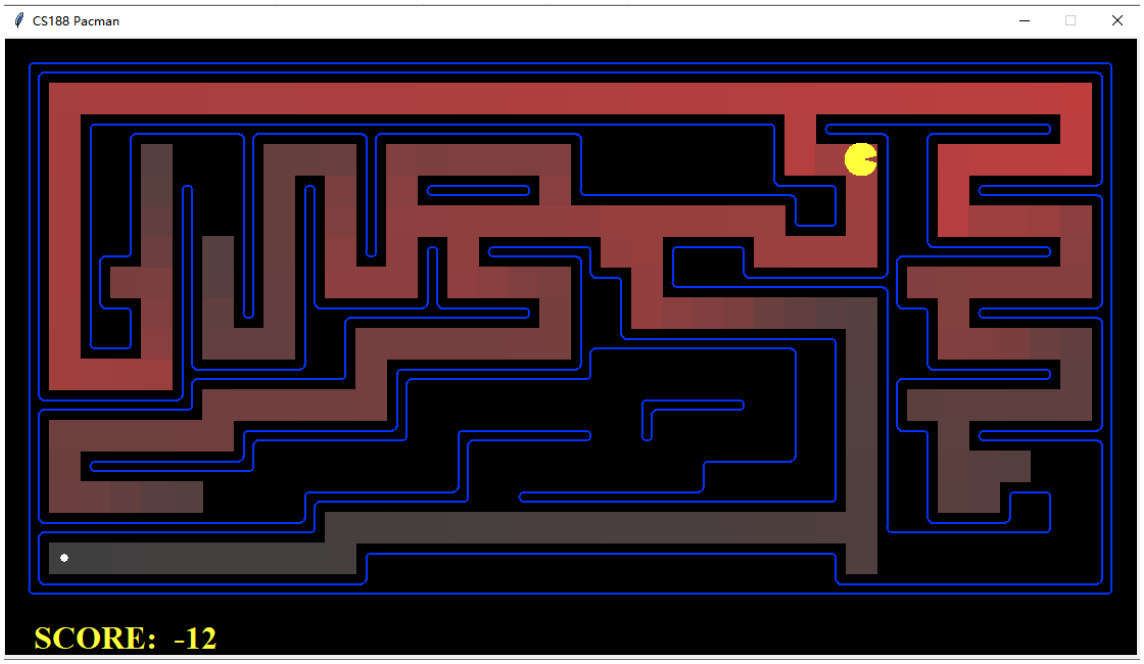

图 4.3: A*

Chapter 5

实验总结

本次实验，我实现了 DFS,A*,minimax,alphabetacut 四个算法，进一步了加深了对算法的理解，以及算法实现能力，收获很大。