人工智能 lab1

Search

- 目标：吃豆人寻找食物，即实现静态查找算法

BFS

简介

- 宽度优先搜索(breadth-first search）是简单搜索策略，先扩展根结点，接着扩展根结点的所有后继，然后再扩展它们的后继，依此类推。一般地，在下一层的任何结点扩展之前，搜索树上本层深度的所有结点都应该已经扩展过。
- 宽度优先搜索是一般图搜索算法的一个实例，每次总是扩展深度最浅的结点。这可以通过将边缘组织成FIFO队列来实现。也就是说，新结点（结点比其父结点深）加入到队列尾，这意味着浅层的老结点会在深层结点之前被扩展。
- 要注意的是，BFS 算法具有一般的图搜索框架，忽视所有到边缘结点或已扩展结点的新路径。可以容易地看出，这样的路径至少和已经找到的一样深。所以，宽度优先搜索总是有到每一个边缘结点的最浅路径。

伪代码

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

算法分析

- 完备性：BFS 是完备的。如果最浅的目标结点处于一个有限深度d，宽度优先搜索在扩展完比它浅的所有结点（假设分支因子b是有限的）之后最终一定能找到该目标结点。注意目标结点一经生成，它就一定是最浅的目标结点，原因是所有比它的浅的结点在此之前已经生成并且肯定未能通过目标测试。
- 最优性：最浅的目标结点不一定就是最优的目标结点。从技术上看，如果路径代价是基于结点深度的非递减函数，宽度优先搜索是最优的。
  - 最常见的情况就是所有的行动要花费相同的代价。
- 时间复杂度：假设搜索一致树(uniform tree）的状态空间中每个状态都有b个后继。搜索树的根结点生成第一层的 b 个子结点，每个子结点又生成b个子结点，第二层则有b^2个结点，依此类推。现在假设解的深度为d。在最坏的情况下，解是那一层最后生成的结点。这时的结点总数为:
  $b+b^2+\ldots+b^d=O(b^d)$

(如果算法是在选择要扩展的结点时而不是在结点生成时进行目标检测，那么在目标被检测到之前深度d上的其他结点已经被扩展，这时时间复杂度应为$O(b^{d+1})$)

- 空间复杂度：对任何类型的图搜索，每个已扩展的结点都保存在探索集中，空间复杂度总是在时间复杂度的b分之一内。特别对于宽度优先图搜索，每个生成的结点都在内存中。那么将有$O(b^{d-1})$个结点在探索集中，$O(b^d)$个结点在边缘结点集中。所以空间复杂度为$O(b^d)$，即它由边缘结点集的大小所决定。

Python 实现

```
def myBreadthFirstSearch(problem):
    visited = {}
    # Python list
    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 the adverse of solution
            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 []
```

- 和 DFS 类似，只需将栈替换为队列，队列中的元素为元组：(state, prev_state)
- 程序具体执行流程如下
    - 初始状态入队
    - 如果队列非空
        - 队首元素出队
        - 判断当前状态是否为目标状态
            - 如果是，则从当前状态开始，通过 visited 元组进行回溯，直到初始状态，然后返回从初始状态到当前状态的列表
            - 否则程序继续执行，判断当前状态是否访问过（图结构中可能存在环路）
                - 如果是，则标记当前状态已访问过，然后将当前状态的子状态入队
    - 否则返回空列表

运行结果

```
(ustc-ai) pqz@ubuntu:~/ai/LAB1/search$ python autograder.py -q q2
Starting on 5-28 at 19:55:40

Question q2
===========
*** PASS: test_cases/q2/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q2/graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases/q2/graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q2/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases/q2/pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 4/4 ###


Finished at 19:55:40

Provisional grades
==================
Question q2: 4/4
------------------
Total: 4/4
```

```
(ustc-ai) pqz@ubuntu:~/ai/LAB1/search$ python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs --frameTime 0
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.1 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

A*

简介

- $A*$算法对结点的评估结合了g(n)，即到达此结点已经花费的代价，和 h(n)，从该结点到目标结点所花代价:f(n)=g(n) + h(n)
- 由于g(n)是从开始结点到结点n的路径代价，而 h(n)是从结点n到目标结点的最小代价路径的估计值，因此 f(n)=经过结点n的最小代价解的估计代价
- 如果想要找到最小代价的解，首先扩展g(n)+ h(n)值最小的结点是合理的。可以发现这个策略不仅仅合理:假设启发式函数h(n)满足特定的条件，$A*$搜索既是完备的也是最优的。算法与一致代价搜索类似，除了$A*$搜索使用 g+h 而不是g。

算法分析

- 最优性：A* 有如下性质:
  - 如果h(n)是可采纳的，那么 A* 的树搜索版本是最优的
  - 如果h(n)是一致的，那么图搜索的A* 算法是最优的。
- 完备性：A* 算法具有完备性
- 时间复杂度：对于每步骤代价为常量的问题，时间复杂度的增长是最优解所在深度d的函数，这可以通过启发式的绝对错误和相对错误来分析。
- 空间复杂度：A* 算法在内存中保留了所有已生成的结点，常常在计算完之前就耗尽了内存。因此，A* 算法对于很多大规模问题，A*算法并不实用。

Python 实现

```
def myAStarSearch(problem, heuristic):
    visited = {}

    frontier = util.PriorityQueue()
    start_state = problem.getStartState()
    frontier.update((start_state, None, 0), heuristic(start_state))

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

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

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

            for next_state, step_cost in problem.getChildren(state):
                next_cost = cost + step_cost
                frontier.update((next_state, state, next_cost),
heuristic(next_state) + next_cost)

    return []
```

- 使用优先队列实现，队列中的元素为元组((state, prev_state, cost), heuristic)
- 程序具体执行流程如下
  - 初始状态入队
  - 如果队列非空
    - 队首元素出队
    - 判断当前状态是否为目标状态
      - 如果是，则从当前状态开始，通过 visited 元组进行回溯，直到初始状态，然后返回从初始状态到当前状态的列表
      - 否则程序继续执行，判断当前状态是否访问过（图结构中可能存在环路）

        - 如果是，则标记当前状态已访问过，然后将当前状态的子状态入队，其中子状态的 cost 由 父状态和当前步的 cost 相加得到，同时计算子状态的 f(n) = g(n) + h(n)
    - 否则返回空列表

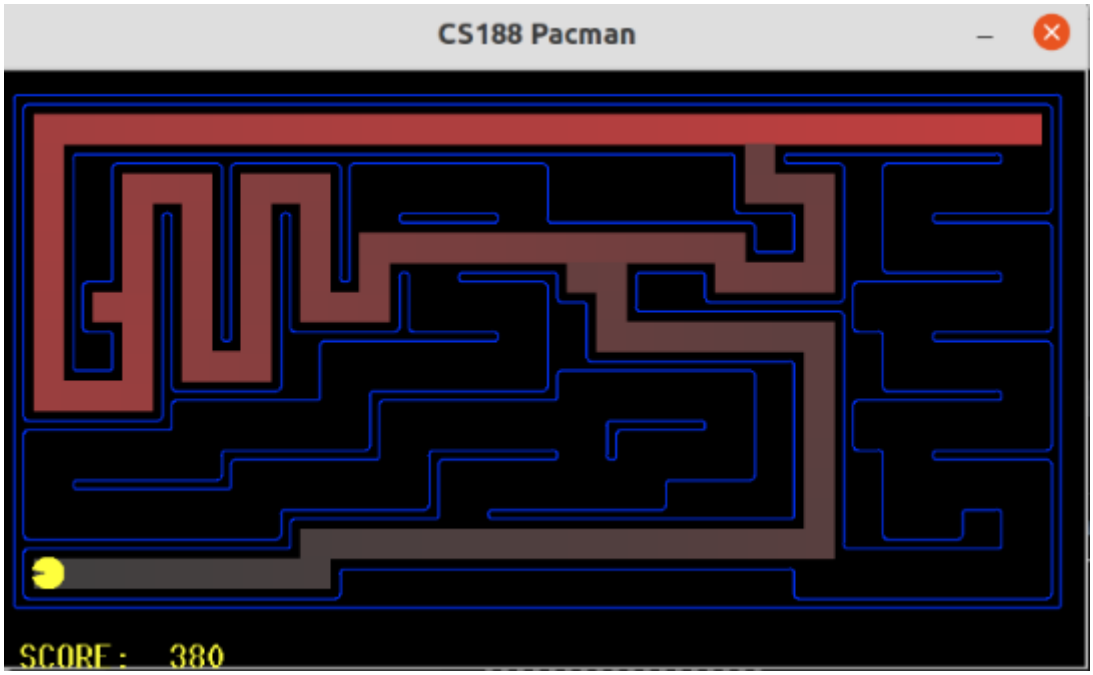
运行结果

```
(ustc-ai) pqz@ubuntu:~/ai/LAB1/search$ python autograder.py -q q3
Starting on 5-28 at 19:57:19

Question q3
===========
*** PASS: test_cases/q3/astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q3/astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases/q3/astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases/q3/astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q3/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q3/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q3: 4/4 ###


Finished at 19:57:19

Provisional grades
==================
Question q3: 4/4
------------------
Total: 4/4
```

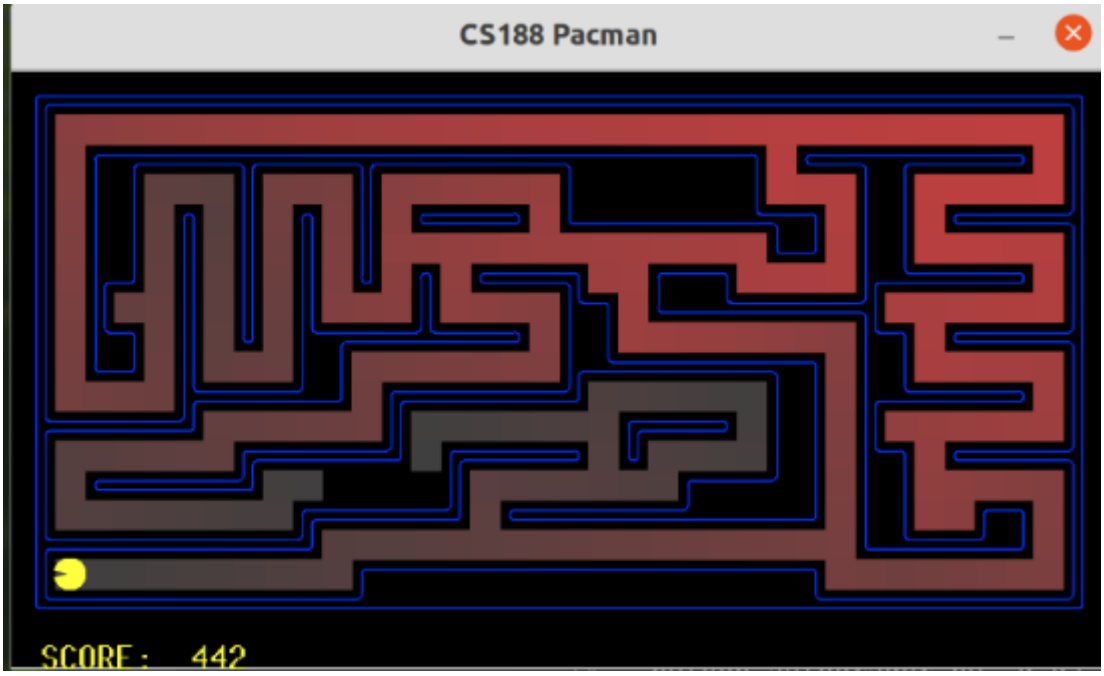
```
(ustc-ai) pqz@ubuntu:~/ai/LAB1/search$ python pacman.py -l mediumMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic --frameTime 0
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.1 seconds
Search nodes expanded: 221
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

DFS、BFS、A* 比较

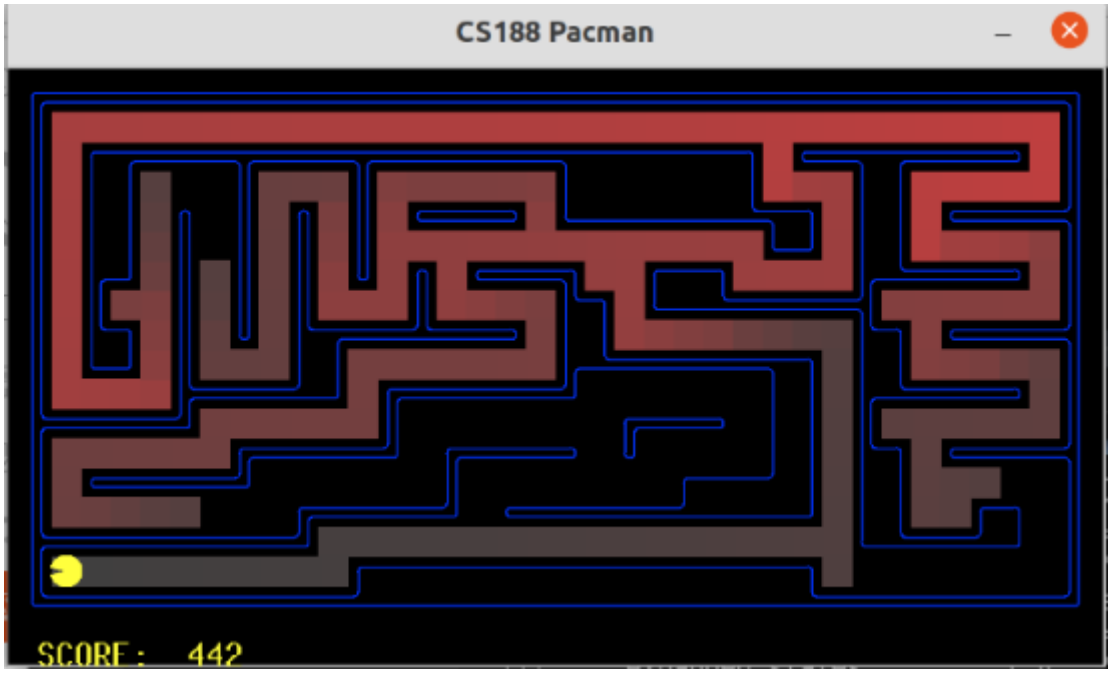
DFS



BFS



A*



对比可知

DFS 重复走过的路径最少，但是走了一条很长的路径

BFS 重复走过的路径最多

A* 走过的路径数处于 DFS 和 BFS 之间

Multiagent

- 目标：吃豆人吃完所有食物，同时避开鬼 。即在有对手的情况下做出下一步决策使自己的利益最大化 。游戏中目标 agent 为吃豆人，其他 agent 为鬼
- 深度 depth，它指的是每个 agent 所走的步数。例如 depth=2，有 1 个 pacman 和 2 个 ghost，则从搜索树的最顶层到最底层应该经过 pacman->ghost1->ghost2->pacman->ghost1->ghost2，操作应该为 max->min->min->max->min->min。

minimax

简介

- 极小极大算法从当前状态计算极小极大决策。它使用了简单的递归算法计算每个后继的极小极大值，直接实现上面公式的定义。递归算法自上而下一直前进到树的叶结点，然后随着递归回溯通过搜索树把极小极大值回传

伪代码

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

算法分析

- 极小极大算法对博弈树执行完整的深度优先探索。如果树的最大深度是m，在每个结点合法的行棋有b个，那么极小极大算法的时间复杂度是 $O(b^m()$。
- 一次性生成所有的后继的算法，空间复杂度是O(bm)，而每次生成一个后继的算法，空间复杂度是O(m)。

Python 实现

```
class MyMinimaxAgent():

    def __init__(self, depth):
        self.depth = depth

    def minimax(self, state, depth):
        if state.isTerminated():
            return None, state.evaluateScore()
        if state.isMe() and depth == 0:
            return state, state.evaluateScore()
        if state.isMe():
            depth -= 1
        # initialize
        best_state, best_score = None, -float('inf') if state.isMe() else
float('inf')

        for child in state.getChildren():
            # YOUR CODE HERE
            _, current_score = self.minimax(child, depth)
            # max
            if state.isMe():
                if current_score >  best_score:
                    best_state = child
                    best_score = current_score
            # min
            else:
                if current_score < best_score:
                    best_state = child
                    best_score = current_score
```

```
        return best_state, best_score

    def getNextState(self, state):
        best_state, _ = self.minimax(state, self.depth)
        return best_state
```

主体思路如下

- 如果到达终止状态，则返回效用值
- 如果当前为吃豆人且深度为 0，则返回当前状态和效用值
- 每当遇到吃豆人时，深度减 1（认为根结点深度最大，叶结点深度为0）
- 初始化
- 遍历当前状态的每一个子状态
    - 递归调用 minimax 算法
    - 如果当前为吃豆人，则需最大化效用值，并视情况更新最优状态和最优效用
    - 如果当前为鬼，则需小化效用值，并视情况更新最优状态和最优效用

运行结果

```
(ustc-ai) pqz@ubuntu:~/ai/LAB1/multiagent$ python autograder.py -q q2 --no-graphics
Starting on 5-28 at 22:38:09

Question q2
===========

*** PASS: test_cases/q2/0-eval-function-lose-states-1.test
*** PASS: test_cases/q2/0-eval-function-lose-states-2.test
*** PASS: test_cases/q2/0-eval-function-win-states-1.test
*** PASS: test_cases/q2/0-eval-function-win-states-2.test
*** PASS: test_cases/q2/0-lecture-6-tree.test
*** PASS: test_cases/q2/0-small-tree.test
*** PASS: test_cases/q2/1-1-minmax.test
*** PASS: test_cases/q2/1-2-minmax.test
*** PASS: test_cases/q2/1-3-minmax.test
*** PASS: test_cases/q2/1-4-minmax.test
*** PASS: test_cases/q2/1-5-minmax.test
*** PASS: test_cases/q2/1-6-minmax.test
*** PASS: test_cases/q2/1-7-minmax.test
*** PASS: test_cases/q2/1-8-minmax.test
*** PASS: test_cases/q2/2-1a-vary-depth.test
*** PASS: test_cases/q2/2-1b-vary-depth.test
*** PASS: test_cases/q2/2-2a-vary-depth.test
*** PASS: test_cases/q2/2-2b-vary-depth.test
*** PASS: test_cases/q2/2-3a-vary-depth.test
*** PASS: test_cases/q2/2-3b-vary-depth.test
*** PASS: test_cases/q2/2-4a-vary-depth.test
*** PASS: test_cases/q2/2-4b-vary-depth.test
*** PASS: test_cases/q2/2-one-ghost-3level.test
*** PASS: test_cases/q2/3-one-ghost-4level.test
*** PASS: test_cases/q2/4-two-ghosts-3level.test
*** PASS: test_cases/q2/5-two-ghosts-4level.test
*** PASS: test_cases/q2/6-tied-root.test
*** PASS: test_cases/q2/7-1a-check-depth-one-ghost.test
*** PASS: test_cases/q2/7-1b-check-depth-one-ghost.test
*** PASS: test_cases/q2/7-1c-check-depth-one-ghost.test
*** PASS: test_cases/q2/7-2a-check-depth-two-ghosts.test
*** PASS: test_cases/q2/7-2b-check-depth-two-ghosts.test
*** PASS: test_cases/q2/7-2c-check-depth-two-ghosts.test
*** Running MinimaxAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running MinimaxAgent on smallClassic after 1 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases/q2/8-pacman-game.test

### Question q2: 5/5 ###


Finished at 22:38:10

Provisional grades
==================
Question q2: 5/5
------------------
Total: 5/5
```

alpha-beta 剪枝

简介

- 极小极大值搜索的问题是必须检查的游戏状态的数目是随着博弈的进行呈指数级增长。不幸的是，指数增长无法消除，不过还是可以有效地将其减半。这里的技巧是可能不需要遍历博弈树中每一个结点就可以计算出正确的极小极大值。
- 借用剪枝思想尽可能消除部分搜索树。这种特别技术称为α-β剪枝。将此技术应用到标准的极小极大搜索树上，会剪掉那些不可能影响决策的分支，仍然返回和极小极大算法同样的结果
- α-β剪枝可以应用于任何深度的树，很多情况下可以剪裁整个子树，而不仅仅是剪裁叶结点。一般原则是：考虑在树中某处的结点n，选手选择移动到该结点。如果选手在n的父结点或者更上层的任何选择点有更好的选择m，那么在实际的博弈中就永远不会到达n。所以一旦发现关于n的足够信息（通过检查它的某些后代），能够得到上述结论，就可以剪裁它
- a-β剪枝的效率很大程度上依赖于检查后继状态的顺序。
- α-β搜索中不断更新 α 和 β的值，并且当某个结点的值分别比目前的 MAX 的 α 或者 MIN 的 β 值更差的时候剪裁此结点剩下的分支 (即终止递归调用)

伪代码

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a), α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Python 实现

```python
class MyAlphaBetaAgent():

    def __init__(self, depth):
        self.depth = depth


    def alpha_beta(self, state, depth, alpha, beta):
        if state.isTerminated():
            return None, state.evaluateScore()
        if state.isMe() and depth == 0:
            return state, state.evaluateScore()
        if state.isMe():
            depth -= 1
        # initialize
```

```
        best_state, best_score = None, -float('inf') if state.isMe() else
float('inf')

        for child in state.getChildren():
            # YOUR CODE HERE
            _, current_score = self.alpha_beta(child, depth, alpha, beta)
            # max
            if state.isMe():
                if current_score > beta:
                    return child, current_score
                alpha = max(alpha, current_score)
                if current_score >  best_score:
                    best_state = child
                    best_score = current_score
            # min
            else:
                if current_score < alpha:
                    return child, current_score
                beta = min(beta, current_score)
                if current_score < best_score:
                    best_state = child
                    best_score = current_score

        return best_state, best_score

    def getNextState(self, state):
    # YOUR CODE HERE
        best_state, _ = self.alpha_beta(state, self.depth, -float('inf'),
float('inf'))
        return best_state
```

α：到目前为止路径上发现的 MAX 的最佳选择

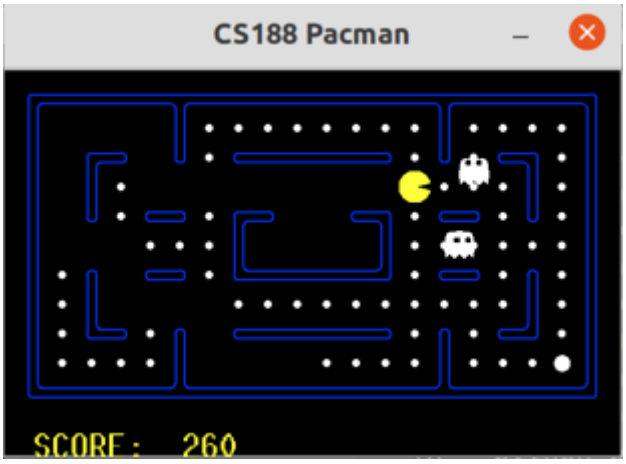
β：到目前为止路径上发现的 MIN 的最佳选择

主体思路如下

- 如果到达终止状态，则返回效用值
- 如果当前为吃豆人且深度为 0，则返回当前状态和效用值
- 每当遇到吃豆人时，深度减 1（认为根结点深度最大，叶结点深度为0）
- 初始化
- 遍历当前状态的每一个子状态
    - 递归调用 minimax 算法
    - 如果当前为吃豆人，则需最大化效用值
        - 如果当前分数大于 β，那么剩下的子结点没有必要再探索了，因为往上一层的 MIN 结点肯定不会选取它们，所以直接返回当前状态和效用值，即剪枝
        - 否则更新 α，并视情况更新最优状态和最优效用
    - 如果当前为鬼，则需小化效用值
        - 如果当前分数小于 α，那么剩下的子结点没有必要再探索了，因为往上一层的 MAX 结点肯定不会选取它们，所以直接返回当前状态和效用值，即剪枝
        - 否则更新 β，并视情况更新最优状态和最优效用

运行结果

```
(ustc-ai) pqz@ubuntu:~/ai/LAB1/multiagent$ python autograder.py -q q3 --no-graphics
Starting on 5-28 at 22:40:32

Question q3
===========

*** PASS: test_cases/q3/0-eval-function-lose-states-1.test
*** PASS: test_cases/q3/0-eval-function-lose-states-2.test
*** PASS: test_cases/q3/0-eval-function-win-states-1.test
*** PASS: test_cases/q3/0-eval-function-win-states-2.test
*** PASS: test_cases/q3/0-lecture-6-tree.test
*** PASS: test_cases/q3/0-small-tree.test
*** PASS: test_cases/q3/1-1-minmax.test
*** PASS: test_cases/q3/1-2-minmax.test
*** PASS: test_cases/q3/1-3-minmax.test
*** PASS: test_cases/q3/1-4-minmax.test
*** PASS: test_cases/q3/1-5-minmax.test
*** PASS: test_cases/q3/1-6-minmax.test
*** PASS: test_cases/q3/1-7-minmax.test
*** PASS: test_cases/q3/1-8-minmax.test
*** PASS: test_cases/q3/2-1a-vary-depth.test
*** PASS: test_cases/q3/2-1b-vary-depth.test
*** PASS: test_cases/q3/2-2a-vary-depth.test
*** PASS: test_cases/q3/2-2b-vary-depth.test
*** PASS: test_cases/q3/2-3a-vary-depth.test
*** PASS: test_cases/q3/2-3b-vary-depth.test
*** PASS: test_cases/q3/2-4a-vary-depth.test
*** PASS: test_cases/q3/2-4b-vary-depth.test
*** PASS: test_cases/q3/2-one-ghost-3level.test
*** PASS: test_cases/q3/3-one-ghost-4level.test
*** PASS: test_cases/q3/4-two-ghosts-3level.test
*** PASS: test_cases/q3/5-two-ghosts-4level.test
*** PASS: test_cases/q3/6-tied-root.test
*** PASS: test_cases/q3/7-1a-check-depth-one-ghost.test
*** PASS: test_cases/q3/7-1b-check-depth-one-ghost.test
*** PASS: test_cases/q3/7-1c-check-depth-one-ghost.test
*** PASS: test_cases/q3/7-2a-check-depth-two-ghosts.test
*** PASS: test_cases/q3/7-2b-check-depth-two-ghosts.test
*** PASS: test_cases/q3/7-2c-check-depth-two-ghosts.test
*** Running AlphaBetaAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running AlphaBetaAgent on smallClassic after 1 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases/q3/8-pacman-game.test

### Question q3: 5/5 ###


Finished at 22:40:34

Provisional grades
==================
Question q3: 5/5
------------------
Total: 5/5
```




```
(ustc-ai) pqz@ubuntu:~/ai/LAB1/multiagent$ python pacman.py -p AlphaBetaAgent -l mediumClassic --frameTime 0
Pacman emerges victorious! Score: 1609
Average Score: 1609.0
Scores:        1609.0
Win Rate:      1/1 (1.00)
Record:        Win
```

总结

- 通过本次实验，对四种搜索算法有了更加深刻的认识
- 通过比较 DFS、BFS 和 A* 算法，也从直观上看出了算法搜索路径的区别
- 在助教给出的代码框架下进行实验，不用自己搭建整体框架，同时也了解了算法的核心思想