**博弈树搜索实验报告**

|  |  |
| --- | --- |
| **班级** | **9191062302** |
| **学号** |  |
| **姓名** |  |

目录

[1 实验目的 3](#_Toc125931801)

[2 实验要求 3](#_Toc125931802)

[3 实验内容 3](#_Toc125931803)

[4 实验代码 3](#_Toc125931804)

[5 实验结果 11](#_Toc125931805)

[5.1 不使用alpha和beta减枝 11](#_Toc125931806)

[5.2 使用alpha和beta减枝 12](#_Toc125931807)

# 实验目的

  通过实验让学生掌握与或树的知识表示方法，基于与或树的知识搜索一般过程。掌握基于与或树的知识表示的要素，掌握与或树的知识搜索流程图，通过博弈树求解问题培养应用与或树搜索解决实际问题的能力。

# 实验要求

C++或Python编写程序。严格按照与或树的搜索流程图编写程序，必须使用alpha和beta减枝，并比较它使用和不使用alpha和beta减枝的优缺点。

# 实验内容

实践博弈树搜索——“5x5格子的一字棋问题”，即五子棋，参照课件PPT上的例子来实现。 （课件PPT上是3x3）。 要求是Max方和Min方都用博弈树来决策，或者一方使用博弈树决策，一方随机或手工走棋，并使用alpha和beta减枝。

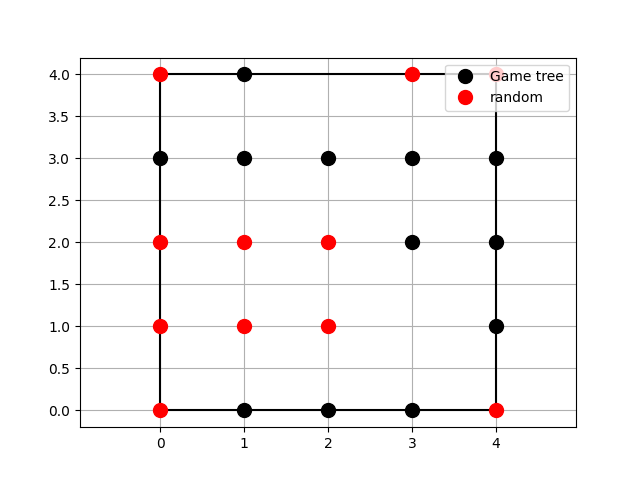
# 实验代码

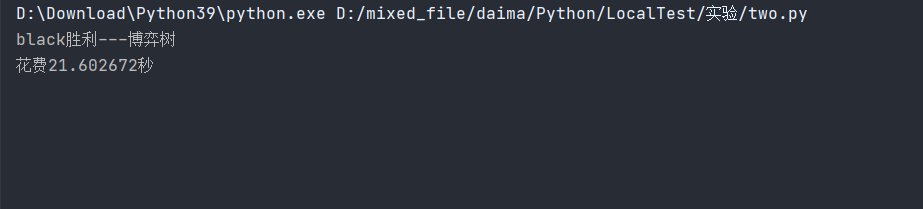
*from* time *import* time

*import* random  
*import* matplotlib.pyplot *as* plt  
  
shape\_score = [(50, (0, 1, 1, 0, 0)),  
 (50, (0, 0, 1, 1, 0)),  
 (200, (1, 1, 0, 1, 0)),  
 (500, (0, 0, 1, 1, 1)),  
 (500, (1, 1, 1, 0, 0)),  
 (5000, (0, 1, 1, 1, 0)),  
 (5000, (0, 1, 0, 1, 1, 0)),  
 (5000, (0, 1, 1, 0, 1, 0)),  
 (5000, (1, 1, 1, 0, 1)),  
 (5000, (1, 1, 0, 1, 1)),  
 (5000, (1, 0, 1, 1, 1)),  
 (5000, (1, 1, 1, 1, 0)),  
 (5000, (0, 1, 1, 1, 1)),  
 (50000, (0, 1, 1, 1, 1, 0)),  
 (99999999, (1, 1, 1, 1, 1))]  
  
  
*def* game\_tree(is\_prune):  
 *if* is\_prune:  
 maxmin\_with\_prune(*True*, DEPTH, -99999999, 99999999)  
 *else*:  
 maxmin(*True*, DEPTH)  
 *return* next\_point[0], next\_point[1]  
  
  
*def* maxmin\_with\_prune(is\_black, depth, alpha, beta):  
 *"""  
 maxmin算法搜索 alpha + beta剪枝  
 """  
 if* is\_win(list1) *or* is\_win(list2) *or* depth == 0:  
 *return* evaluation(is\_black)  
  
 blank\_list = list(set(list\_all).difference(set(list3)))  
 order(blank\_list) *# 搜索顺序排序  
 for* next\_step *in* blank\_list:  
 *if not* has\_neighbor(next\_step):  
 *continue  
 if* is\_black:  
 list1.append(next\_step)  
 *else*:  
 list2.append(next\_step)  
 list3.append(next\_step)  
 value = -maxmin\_with\_prune(*not* is\_black, depth - 1, -beta, -alpha)  
 *if* is\_black:  
 list1.remove(next\_step)  
 *else*:  
 list2.remove(next\_step)  
 list3.remove(next\_step)  
 *if* value > alpha:  
 *if* depth == DEPTH:  
 next\_point[0] = next\_step[0]  
 next\_point[1] = next\_step[1]  
 *# alpha + beta剪枝点  
 if* value >= beta:  
 *return* beta  
 alpha = value  
 *return* alpha  
  
  
*def* maxmin(is\_black, depth):  
 *"""  
 maxmin算法搜索  
 """  
 # 游戏是否结束 | | 探索的递归深度是否到边界  
 if* is\_win(list1) *or* is\_win(list2) *or* depth == 0:  
 *return* evaluation(is\_black)  
 blank\_list = list(set(list\_all).difference(set(list3)))  
 order(blank\_list) *# 搜索顺序排序* value = -99999999  
 *if not* is\_black:  
 value = -value  
 *for* next\_step *in* blank\_list:  
 *if not* has\_neighbor(next\_step):  
 *continue  
 if* is\_black:  
 list1.append(next\_step)  
 *else*:  
 list2.append(next\_step)  
 list3.append(next\_step)  
 value = - maxmin(*not* is\_black, depth - 1)  
 *if* is\_black:  
 list1.remove(next\_step)  
 *else*:  
 list2.remove(next\_step)  
 list3.remove(next\_step)  
 *if* depth == DEPTH:  
 next\_point[0] = next\_step[0]  
 next\_point[1] = next\_step[1]  
 *return* value  
  
*# 离最后落子的邻居位置最有可能是最优点  
def* order(blank\_list):  
 last\_pt = list3[-1]  
 *for* i *in* range(-1, 2):  
 *for* j *in* range(-1, 2):  
 *if* i == 0 *and* j == 0:  
 *continue  
 if* (last\_pt[0] + i, last\_pt[1] + j) *in* blank\_list:  
 blank\_list.remove((last\_pt[0] + i, last\_pt[1] + j))  
 blank\_list.insert(0, (last\_pt[0] + i, last\_pt[1] + j))  
  
  
*def* has\_neighbor(pt):  
 *for* i *in* range(-1, 2):  
 *for* j *in* range(-1, 2):  
 *if* i == 0 *and* j == 0:  
 *continue  
 if* (pt[0] + i, pt[1] + j) *in* list3:  
 *return True  
 return False  
  
# 评估函数  
def* evaluation(is\_black):  
 *if* is\_black:  
 my\_list = list1  
 enemy\_list = list2  
 *else*:  
 my\_list = list2  
 enemy\_list = list1  
 *# 算自己的得分* score\_all\_arr = []  
 my\_score = 0  
 *for* pt *in* my\_list:  
 m = pt[0]  
 n = pt[1]  
 my\_score += cal\_score(m, n, 0, 1, enemy\_list, my\_list, score\_all\_arr)  
 my\_score += cal\_score(m, n, 1, 0, enemy\_list, my\_list, score\_all\_arr)  
 my\_score += cal\_score(m, n, 1, 1, enemy\_list, my\_list, score\_all\_arr)  
 my\_score += cal\_score(m, n, -1, 1, enemy\_list, my\_list, score\_all\_arr)  
 *# 算对手的得分* score\_all\_arr\_enemy = []  
 enemy\_score = 0  
 *for* pt *in* enemy\_list:  
 m = pt[0]  
 n = pt[1]  
 enemy\_score += cal\_score(m, n, 0, 1, my\_list, enemy\_list, score\_all\_arr\_enemy)  
 enemy\_score += cal\_score(m, n, 1, 0, my\_list, enemy\_list, score\_all\_arr\_enemy)  
 enemy\_score += cal\_score(m, n, 1, 1, my\_list, enemy\_list, score\_all\_arr\_enemy)  
 enemy\_score += cal\_score(m, n, -1, 1, my\_list, enemy\_list, score\_all\_arr\_enemy)  
  
 total\_score = my\_score - enemy\_score \* 0.1  
 *return* total\_score  
  
  
*# 每个方向上的分值计算  
def* cal\_score(m, n, x\_decrict, y\_derice, enemy\_list, my\_list, score\_all\_arr):  
 add\_score = 0 *# 加分项* max\_score\_shape = (0, *None*)  
 *for* item *in* score\_all\_arr:  
 *for* pt *in* item[1]:  
 *if* m == pt[0] *and* n == pt[1] *and* x\_decrict == item[2][0] *and* y\_derice == item[2][1]:  
 *return* 0  
 *for* offset *in* range(-5, 1):  
 *# offset = -2* pos = []  
 *for* i *in* range(0, 6):  
 *if* (m + (i + offset) \* x\_decrict, n + (i + offset) \* y\_derice) *in* enemy\_list:  
 pos.append(2)  
 *elif* (m + (i + offset) \* x\_decrict, n + (i + offset) \* y\_derice) *in* my\_list:  
 pos.append(1)  
 *else*:  
 pos.append(0)  
 tmp\_shap5 = (pos[0], pos[1], pos[2], pos[3], pos[4])  
 tmp\_shap6 = (pos[0], pos[1], pos[2], pos[3], pos[4], pos[5])  
 *for* (score, shape) *in* shape\_score:  
 *if* tmp\_shap5 == shape *or* tmp\_shap6 == shape:  
 *if* score > max\_score\_shape[0]:  
 max\_score\_shape = (score, ((m + (0 + offset) \* x\_decrict, n + (0 + offset) \* y\_derice),  
 (m + (1 + offset) \* x\_decrict, n + (1 + offset) \* y\_derice),  
 (m + (2 + offset) \* x\_decrict, n + (2 + offset) \* y\_derice),  
 (m + (3 + offset) \* x\_decrict, n + (3 + offset) \* y\_derice),  
 (m + (4 + offset) \* x\_decrict, n + (4 + offset) \* y\_derice)),  
 (x\_decrict, y\_derice))  
 *if* max\_score\_shape[1] *is not None*:  
 *for* item *in* score\_all\_arr:  
 *for* pt1 *in* item[1]:  
 *for* pt2 *in* max\_score\_shape[1]:  
 *if* pt1 == pt2 *and* max\_score\_shape[0] > 10 *and* item[0] > 10:  
 add\_score += item[0] + max\_score\_shape[0]  
 score\_all\_arr.append(max\_score\_shape)  
 *return* add\_score + max\_score\_shape[0]  
  
  
*def* is\_win(list):  
 *for* m *in* range(COLUMN):  
 *for* n *in* range(ROW):  
 *if* n < ROW - 4 *and* (m, n) *in* list *and* (m, n + 1) *in* list *and* (m, n + 2) *in* list *and* (  
 m, n + 3) *in* list *and* (m, n + 4) *in* list:  
 *return True  
 elif* m < ROW - 4 *and* (m, n) *in* list *and* (m + 1, n) *in* list *and* (m + 2, n) *in* list *and* (  
 m + 3, n) *in* list *and* (m + 4, n) *in* list:  
 *return True  
 elif* m < ROW - 4 *and* n < ROW - 4 *and* (m, n) *in* list *and* (m + 1, n + 1) *in* list *and* (  
 m + 2, n + 2) *in* list *and* (m + 3, n + 3) *in* list *and* (m + 4, n + 4) *in* list:  
 *return True  
 elif* m < ROW - 4 *and* n > 3 *and* (m, n) *in* list *and* (m + 1, n - 1) *in* list *and* (  
 m + 2, n - 2) *in* list *and* (m + 3, n - 3) *in* list *and* (m + 4, n - 4) *in* list:  
 *return True  
 return False  
  
  
def* main(is\_prune):  
 mode = 0  
 *for* i *in* range(COLUMN):  
 *for* j *in* range(ROW):  
 list\_all.append((i, j))  
 g = 0  
 change = 0  
 *while True*:  
 *# black  
 if* change % 2 == mode:  
 *if* change == 0:  
 pos = ((ROW + 1) / 2, (COLUMN + 1) / 2)  
 *else*:  
 pos = game\_tree(is\_prune)  
 list1.append(pos)  
 list3.append(pos)  
 *# AI胜利  
 if* is\_win(list1):  
 print("black胜利---博弈树")  
 *break* change = change + 1  
  
 *# red  
 else*:  
 *while True*:  
 x = random.randint(0, ROW - 1)  
 y = random.randint(0, COLUMN - 1)  
 *if not* (x, y) *in* list3:  
 *break* list2.append((x, y))  
 list3.append((x, y))  
 *# 胜利  
 if* is\_win(list2):  
 print("red胜利---随机")  
 *break* change = change + 1  
  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 t1 = time()  
 COLUMN = 5  
 ROW = 5  
 list1 = [] *# game tree* list2 = [] *# random* list3 = [] *# game tree + random* list\_all = [] *# all* next\_point = [0, 0] *# 下一步最应该下的位置* DEPTH = 3  
 main(is\_prune=*True*)  
 x1 = []  
 y1 = []  
 x2 = []  
 y2 = []  
 *for* p *in* list1:  
 x1.append(p[0])  
 y1.append(p[1])  
 *for* p *in* list2:  
 x2.append(p[0])  
 y2.append(p[1])  
 print('花费%f秒' % (time() - t1))  
 plt.plot([0, 0, 4, 4, 0], [0, 4, 4, 0, 0], '-k')  
 plt.plot(x1, y1, '.k', markersize=20, label='Game tree')  
 plt.plot(x2, y2, '.r', markersize=20, label='random')  
 plt.legend(loc="upper right")  
 plt.grid(*True*)  
 plt.axis('equal')  
 plt.show()

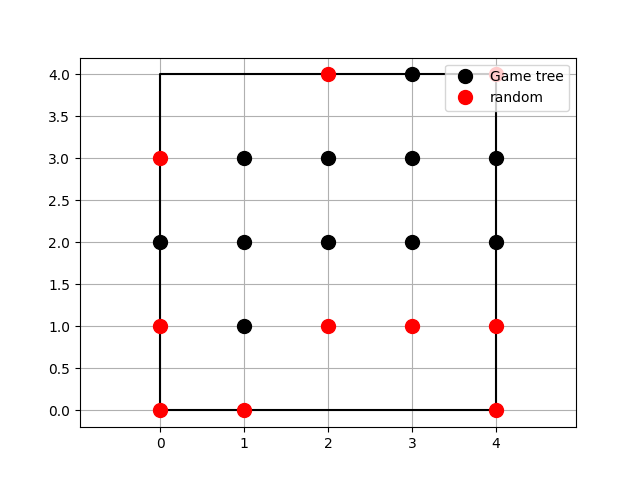
# 实验结果

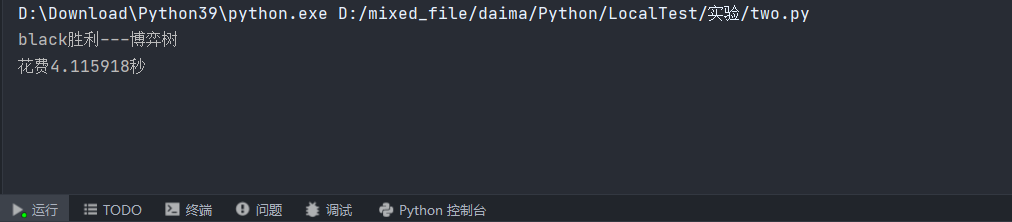
## 不使用alpha和beta减枝





## 使用alpha和beta减枝





使用alpha和beta减枝，减少了程序执行时间，但也使代码更加复杂。