人 工 智 能 （2023 春季）

实 习 报 告

班　　级 191211

学　　号 ~~20211000156~~

姓　　名 ~~孙鹤轩~~

评　　分 _____

中国地质大学（武汉）计算机学院

2023 年 04 月

目录

实习一：罗马尼亚度假问题

(1) 题目描述

分别采用代价一致宽度优先、贪婪算法和 A*算法实现罗马尼亚度假问题。

(2) 算法思想

代价一致宽度优先搜索：在顶点的邻接顶点中优先扩展代价最小的结点，即在搜索过程中记录父节点已经消耗的代价值，搜索下一个代价小的结点

贪婪算法：在顶点的邻接顶点中优先扩展距离目标点直线距离最短的点，即仅考虑启发函数值

A*算法：核心思想为避免扩展代价已经很高的结点，在邻接顶点中优先扩展 g(n)+h(n)最小的值，g(n)当前结点消耗的实际代价，h(n)为从当前节点到目标点的直线距离

(3) 软件说明

软件名：AI_Roma

编程环境：使用 windows64 位操作系统基于 vs2022 + Qt5.12.12 可视化框架编写

文件大小：437KB

(4) 关键代码

```
/****代价一致宽度搜索****/
std::stack<std::string> AI_firest::UCS(Graph g , std::string start , std::string target)
{
    clock_t r_start, r_finish;
    r_start = clock();
    search_node.clear();
    test_vertex.clear();
    std::stack<std::string> path;
    std::unordered_map <std::string, EList> vers = g.vertexs;//获取图的顶点表
    std::unordered_map<std::string, int> dis;//父节点消耗的路径代价
    std::unordered_map<std::string, bool> isVisited;//辅助数组
    std::vector<std::string>::iterator it = cityName.begin();

    while (it != cityName.end())
    {//初始化辅助数组
        isVisited.emplace((*it), false);
        it++;
    }

    SearchNode startCity(start, 0);
    oder_frige.push(startCity);//入队列
    isVisited[start] = true;
    search_node.emplace(start, startCity);
    test_vertex.push_back(start);
    dis.emplace(start, 0);//初始化起点

    while (!oder_frige.empty())
    {
        SearchNode temp = oder_frige.top();
        oder_frige.pop();//队头出队
```

```
            isVisited[temp.name] = true;//标记访问
            test_vertex.push_back(temp.name);
            if (temp.name == target) {
                path.push(temp.name);//记录路径
                SearchNode i = temp;
                while (path.top() != start)
                {
                    sum_cost[0] += i.cost;
                    i = search_node[i.father];
                    path.push(i.name);//放入完整路径
                }
                break;
            }
            ENode* p = vers[temp.name].head->link;//获取第一个邻接顶点
            while (p != NULL)
            {
                if (!isVisited[p->data])
                {
                    dis[p->data] = dis[temp.name] + p->cost;//获取邻接顶点的评估值
                    SearchNode path_city(p->data, dis[p->data]);
                    path_city.father = temp.name;
                    path_city.cost = p->cost;
                    search_node.emplace(p->data, path_city);
                    oder_frige.push(path_city);//入队列
                }
                p = p->link;
            }
        }

        while (!oder_frige.empty()) {//清空优先级队列
            oder_frige.pop();
        }
        r_finish = clock();
        run_time[0] = r_finish - r_start;
        return path;
        }
    /******A*算法******/
    std::stack<std::string> AI_firest::AStar(Graph g , std::string start , std::string target)
    {
        clock_t r_start, r_finish;
        r_start = clock();
        search_node.clear();
        test_vertex.clear();
        std::stack<std::string> path;
        std::unordered_map <std::string, EList> vers = g.vertexs;//获取图的顶点表
        std::unordered_map<std::string, bool> isVisited;//辅助数组
        std::vector<std::string>::iterator it = cityName.begin();
```

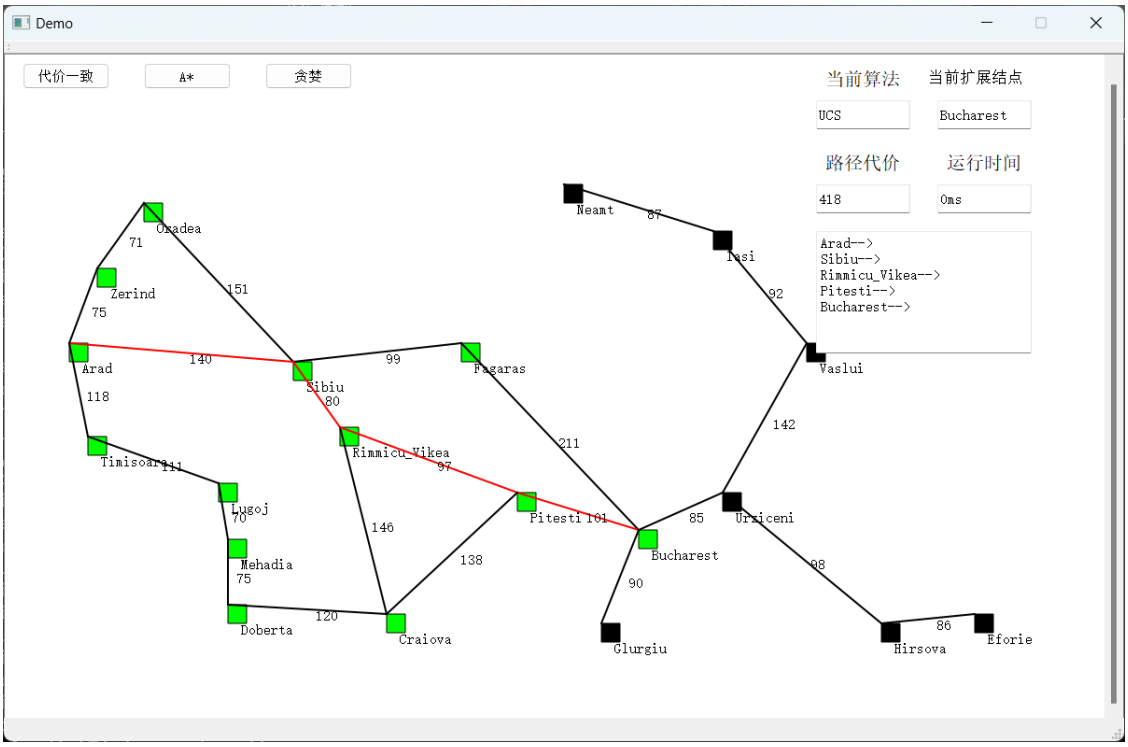
```
    while (it != cityName.end())
    {//初始化辅助数组
        isVisited.emplace((*it), false);
        it++;
    }

    int startValue = path_value[start];
    SearchNode startCity(start, startValue);
    oder_frige.push(startCity);//入队列
    isVisited[start] = true;
    test_vertex.push_back(start);
    search_node.emplace(start, startCity);
    while (!oder_frige.empty())
    {
        SearchNode temp = oder_frige.top();
        oder_frige.pop();//队头出队
        isVisited[temp.name] = true;
        test_vertex.push_back(temp.name);
        if (temp.name == target) {
            path.push(temp.name);
            SearchNode i = temp;
            while (path.top() != start)
            {
                sum_cost[1] += i.cost;
                i = search_node[i.father];
                path.push(i.name);
            }
            break;
        }

        ENode* p = vers[temp.name].head->link;//获取第一个邻接顶点
        while (p != NULL)
        {
            if (!isVisited[p->data])
            {
                SearchNode path_city(p->data, path_value[p->data] + p->cost);//评估值为直线距
离加当前结点消耗的代价
                path_city.father = temp.name;
                path_city.cost = p->cost;
                search_node.emplace(p->data, path_city);
                oder_frige.push(path_city);
            }
            p = p->link;
        }
    }
```

```
    while (!oder_frige.empty()) {
        oder_frige.pop();
    }
    r_finish = clock();
    run_time[1] = r_finish - r_start;
    return path;
}
/*******贪婪算法*******/
std::stack<std::string> AI_firest::Greedy(Graph g , std::string start , std::string target)
{
    clock_t r_start, r_finish;
    r_start = clock();
    search_node.clear();
    test_vertex.clear();
    std::stack<std::string> path;
    std::unordered_map <std::string, EList> vers = g.vertexs;//获取图的顶点表
    std::vector<std::string>::iterator it = cityName.begin();
    std::unordered_map<std::string, bool> isVisited;//辅助数组
    while (it != cityName.end())
    {//初始化辅助数组
        isVisited.emplace((*it), false);
        it++;
    }
    int startValue = path_value[start];
    SearchNode startCity(start, startValue);
    oder_frige.push(startCity);//入队列
    test_vertex.push_back(start);
    isVisited[start] = true;
    search_node.emplace(start, startCity);
    while (!oder_frige.empty())
    {
        SearchNode temp = oder_frige.top();
        oder_frige.pop();//队头出队
        isVisited[temp.name] = true;
        test_vertex.push_back(temp.name);
        if (temp.name == target) {
            path.push(temp.name);
            SearchNode i = temp;
            while (path.top() != start)
            {
                sum_cost[2] += i.cost;
                i = search_node[i.father];
                path.push(i.name);
            }
            break;
        }
        ENode* p = vers[temp.name].head->link;//获取第一个邻接顶点
```

```
            while (p != NULL)
            {
                if (!isVisited[p->data])
                {
                    SearchNode path_city(p->data, path_value[p->data]);
                    path_city.father = temp.name;
                    path_city.cost = p->cost;
                    search_node.emplace(p->data, path_city);
                    oder_frige.push(path_city);
                }
                p = p->link;
            }
        }
        while (!oder_frige.empty()) {
            oder_frige.pop();
        }
        r_finish = clock();
        run_time[2] = r_start - r_finish;
        return path;
}
```

(5) 运行结果及分析

代价一致宽度优先搜索：



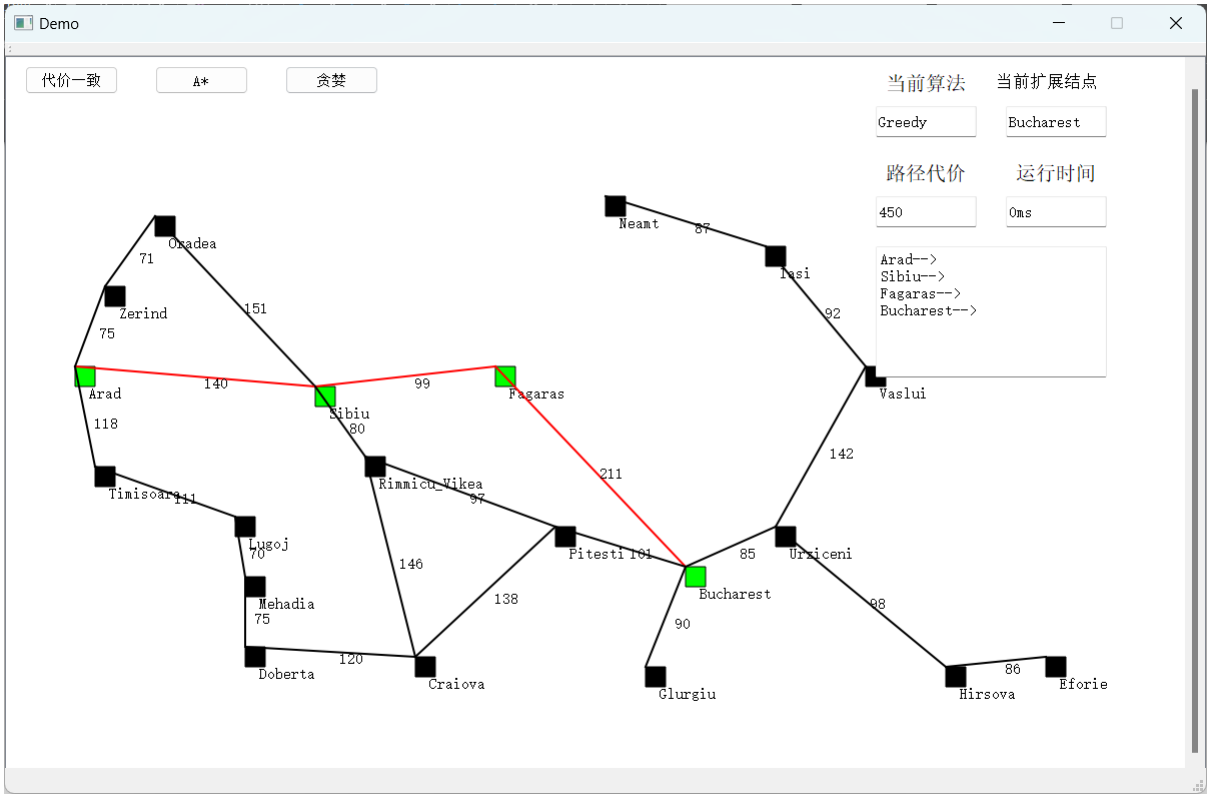
可以从软件中直观的看出：代价一致宽度优先算法可以找到到达目标点的最优路径

算法路径代价：418

路径：Arad-->Sibiu-->Rimmicu_Vikea-->Pitesti-->Bucharest

算法缺点：扩展结点较多，运行时间较慢

贪婪算法：



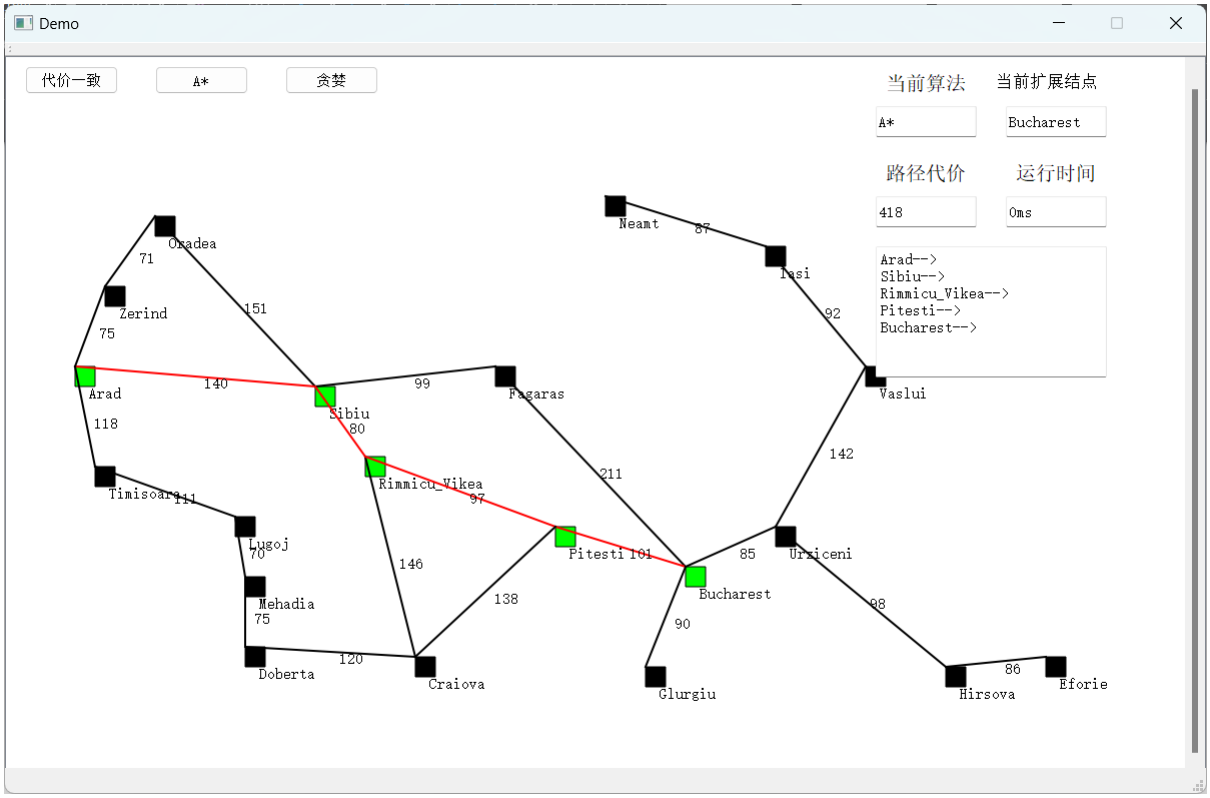
可以从软件中直观的看出：贪婪算法不可以找到到达目标点的最优路径

算法路径代价：450

路径：Arad-->Sibiu-->Fagaras-->Bucharest

算法缺点：往往不能找到最优解

A*算法：



可以从软件中直观的看出：A*算法可以找到到达目标点的最优路径

算法路径代价：418

路径：Arad-->Sibiu-->Rimmicu_Vikea-->Pitesti-->Bucharest

算法缺点：实时性差，每一节点计算量大、运算时间长，而且随着节点数的增多，算法搜索效率降低，而且在一些情况下 A* 算法并没有完全遍历所有可行解，所得到的结果不一定是最优解。

备注：因为当前操作系统性能较高，c++的时钟函数精度较低，无法精确测量运行时间

(6) 小结

在本次实习中学习了三种算法，其中盲目搜索的是代价一致宽度优先搜索，带信息搜索的是贪婪算法和 A*算法。给我直观的感受是代价一直宽度优先搜索可以很好的找到最优路径解，能够在小规模数据中如鱼得水，可以精确的解决路径规划问题，缺点就是无法处理规模大的问题，这也是智能优化算法发展的推动点。

贪婪算法和 A*算法让我感受到了智能算法的强大。贪婪算法简洁高效，可以很快的找到解，在实际应用中我们往往只需要找到问题的可行解即可，所以贪婪算法是一种简洁高效的算法。A*算法作为启发式搜索算法让我直观感受到了启发式算法的优势，极大地开阔了我的眼界，感受到了算法改进在应用中的魅力。

这次实习通过实例来学习三种算法极大的提高了我的抽象能力，问题概括能力等等。

(7) 参考文献

https://blog.csdn.net/W_yu_cheng/article/details/89576917?ops_request_misc=&request_id=&biz_id=102&utm_term=c++%E7%BD%97%E9%A9%AC%E5%B0%BC%E4%BA%9A%E5%BA%A6%E5%81%87%E9%97%AE%E9%A2%98&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-0-89576917.142^v83^insert_down38,201^v4^add_ask,239^v2^insert_chatgpt&spm=1018.2226.3001.4187 罗马尼亚度假问题 人工智能搜索算法全代码 C++（深度优先，广度优先，等代价，迭代加深，有信息搜索，A*算法，贪婪算法）

实习二：采用最小冲突法求解 n 皇后问题

(1) 题目描述

n 皇后问题研究的是如何将 n 个皇后放置在 n×n 的棋盘上，皇后如果是同行同列或者是在同一斜对角上就会相互攻击。给定一个整数 n,，使皇后彼此之间不能相互攻击，使用最小冲突法返回一个可行解决方案，要求 n≥80

(2) 算法思想

（1）初始化 N 个皇后的一个放置，允许有冲突

（2）考虑某一行的某个皇后，她可能与 x 个皇后冲突，然后看看将这个皇后移动到这一行的哪个空位能使得与其冲突的皇后个数最少，就移动到那里。（也可以考虑列，是等价的）

（3）不断执行（2），直到没有冲突为止

(3) 软件说明

软件名：AI_NQueen

编程环境：使用 windows64 位操作系统基于 vs2022 编写

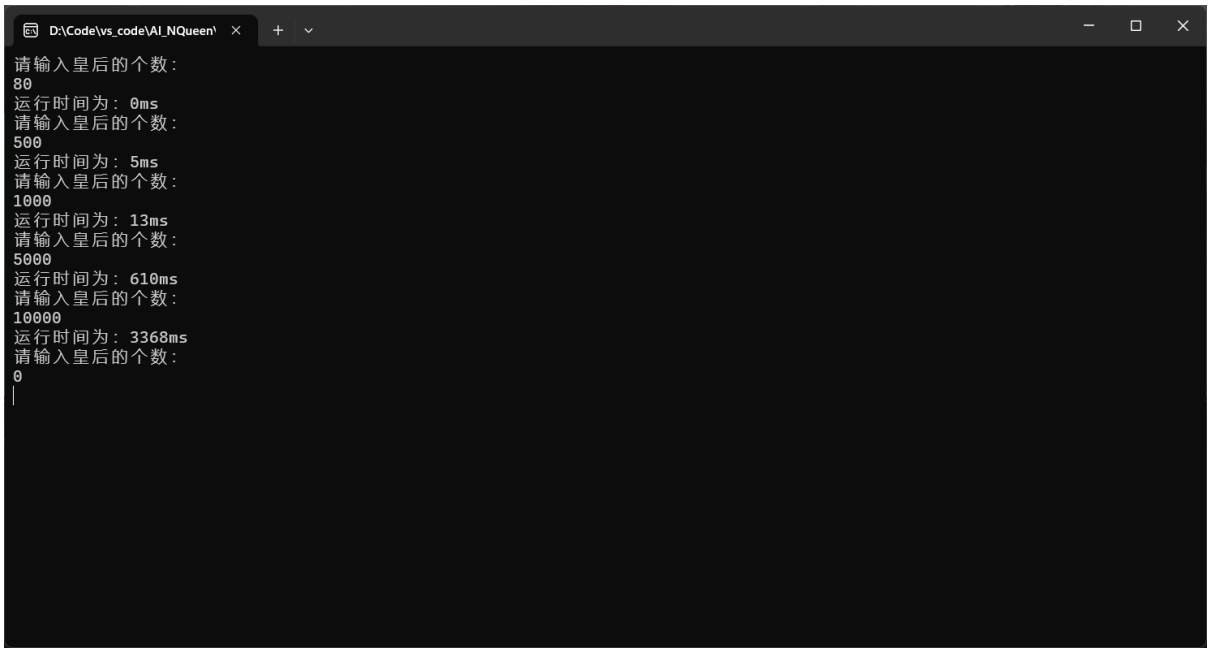
文件大小：89KB

(4) 关键代码

```
//用最小冲突算法调整第row行的皇后的位置（初始化时每行都有一个皇后，调整后仍然在第row行）
//调整过后检查一下看看是否已经没有冲突，如果没有冲突返回true
bool qualify() {
    for (int i = 0; i < N; i++) {//N queens
        if (col[R[i]] != 1 ||
            pdiag[getP(i, R[i])] != 1 ||
            cdiag[getC(i, R[i])] != 1) {
            return false;
        }
    }
    return true;
}
bool adjust_row(int row) {
    int cur_col = R[row];
    int optimal_col = cur_col;//最佳列号，设置为当前列，然后更新
    int    min_conflict = col[optimal_col] + pdiag[getP(row, optimal_col)] - 1
        + cdiag[getC(row, optimal_col)] - 1;//对角线冲突数为当前对角线皇后数减一
    for (int i = 0; i < N; i++) {//逐个检查第row行的每个位置
        if (i == cur_col) {
            continue;
        }
        int conflict = col[i] + pdiag[getP(row, i)] + cdiag[getC(row, i)];
        if (conflict < min_conflict) {
            min_conflict = conflict;
            optimal_col = i;
        }
    }
    if (optimal_col != cur_col) {//要更新col,pdiag,cdiag
        col[cur_col]--;
```

```
        pdiag[getP(row, cur_col)]--;
        cdiag[getC(row, cur_col)]--;

        col[optimal_col]++;
        pdiag[getP(row, optimal_col)]++;
        cdiag[getC(row, optimal_col)]++;
        R[row] = optimal_col;
        if (col[cur_col] == 1 && col[optimal_col] == 1
            && pdiag[getP(row, optimal_col)] == 1 && cdiag[getC(row, optimal_col)] == 1) {
            return qualify();//qualify相对更耗时，所以只在满足上面基本条件后才检查
        }
    }
    //当前点就是最佳点，一切都保持不变
    return false;//如果都没变的话，肯定不满足终止条件，否则上一次就应该返回true并终止了
    }
```

(5) 运行结果及分析



本次实验测试了 80，500，1000，5000，10000，均可以在秒级别解决，对于 10 万级别需要约 10 分钟解决。

由此可以看出最小冲突法可以较快的找出 n 皇后的一个解，比回溯法有了很大的突破。

(6) 小结

最小冲突法是局部搜索算法，对于像 n 皇后这种 csp 问题有很好的解决效果。经过此次实验，我很好的感受到了局部搜索算法的高效性，对于大规模 n(≤10000000)皇后级别的问题能在常数级别解决，最小冲突法几乎是独立于问题的规模，让我感受到了搜索算法的进步之快。

(7) 参考文献

https://www.cnblogs.com/fstang/archive/2013/05/12/3073598.html N 皇后——最小冲突算法一个相对高效的实现

实习三：使用联机搜索算法求解 Wumpus 怪兽世界问题

(1) 题目描述

Wumpus 世界是一个简单的世界示例，用于说明基于知识的 Agent 的价值并表示知识。Wumpus 世界是一个山洞，有 4×4 个房间，这些房间与通道相连。因此，共有 16 个房间相互连接。我们有一个以知识为基础的 Agent，他将在这个世界探索。这个山洞里有一间屋子，里面有个叫 Wumpus 的怪兽，他会吃掉进屋的任何人。Agent 可以射杀 Wumpus，但 Agent 只有一支箭。在 Wumpus 世界中，有一些陷阱 PIT，如果 Agent 落在深坑中，那么他将永远被困在那里。其中在一个房间里有可能找到 Gold。因此, Agent 的目标是找到金子并走出洞穴，而不会掉入坑或被 Wumpus 吞噬。如果 Agent 找到金子出来，他会得到奖励;如果被 Wumpus 吞下或掉进 PIT 中，他会受到惩罚。注意:这里的 Wumpus 是静态的，不能移动。

传感器:

1、如果 Agent 在 Wumpus 附近的房间里，他会感觉到恶臭 stench。(不是对角线的)。如果 Agent 在紧邻陷阱 PIT 的房间内，他会感觉到微风 Breeze。

2、Agent 可以感知到存在 Gold 的房间中的闪光。

3、射杀 Wumpus 时，它会发出可怕的尖叫声，在山洞的任何地方都可以感觉到。

性能指标:

如果 Agent 带着金从洞穴中出来，则可获得 1000 点奖励积分。被 Wumpus 吃掉或掉进坑里的点数为-1000 分。-1 表示每个操作，-10 表示使用箭头。如果 Agent 死亡或从山洞出来，游戏就会结束。

环境: 4*4 的房间网格。除了第一个正方形[1，1]以外，都是随机选择 Wumpus 和黄金的位置。洞穴的每个正方形都可以是第一个正方形以外的概率为 0.2 的坑(随机产生 PIT 陷阱).执行器:左转,右转,前进,抓,射击。

(2) 算法思想

使用联机搜索进行求解，联机问题在执行时，需要计算和行动交叉进行。联机搜索需要智能体能够记住先前的状态，不能访问下一个状态。下面是算法的基本流程：

（1）随机生成金子，怪兽，陷阱的位置，且它们的位置各不相同，使用 HashMap 进行信息维护

（2）从起始点开始 DFS，更新当前点的信息，对当前点的四个邻居根据已有信息进行评估，选择最有利的邻居点走下去，同时更新 agent 的维护地图。

（3）评估标准为：

1、首先寻找四周安全的点，如果没有则在地图上寻找最近安全点(BFS 实现）

2、如果没有安全点，则去往没有怪兽和陷阱的这些相对安全的点。

3、如果没有相对安全的点，则去往有怪兽的点杀死怪兽。

4、如果没有怪兽，则随机选择一个未被访问的点走下去，虽然很可能调入陷阱。

（4）重复上述过程直到找到金子然后最短路径回家或者掉入陷阱死亡。

(3) 软件说明

软件名：AI_Wwumpus

编程环境：使用 windows64 位操作系统基于 vs2022+Qt5.12.12 可视化框架编写

文件大小：1994KB

(4) 关键代码

```
//Dfs寻找最优扩展结点
void Widget::Dfs(Point current)
{
```

```
    vector<Point> neighbors;
    path_record[step_cnt++] = current;
    agent_world[current.xx][current.yy] |= real_world[current.xx][current.yy]; // 获取当前点信
息
    if ((agent_world[current.xx][current.yy] & GOLD) == GOLD) {   //如果遇到了金子
        find_gold = true;
        return;
    }
    if ((agent_world[current.xx][current.yy] & CAVE) == CAVE) {   // 掉进了洞穴
        game_over = true;
        return;
    }
    PutFlag(agent_world, current, VISITED); // 当前点已访问
    UpdateNeighborsInformation(current);    // 更新周围点的信息
    agent_world[current.xx][current.yy] &= ~CURRENT; // 不在当前点

    int Next[4][2] = { {0, 1}, {0, -1}, {1, 0}, {-1, 0} };// 上，下，左，右
    for (int i = 0; i < 4; ++i) {
        int next_x = current.xx + Next[i][0];   // 邻居点x坐标
        int next_y = current.yy + Next[i][1];   // 邻居点y坐标
        if (next_x >= 0 && next_x < ROW_NUM && next_y >= 0 && next_y < COL_NUM) {// 边界检查
            neighbors.push_back({ next_x, next_y });
        }
    }
    for (int i = 0; i < neighbors.size(); ++i) { // 周围可能有金子
        if ((agent_world[neighbors[i].xx][neighbors[i].yy] & GOLD) == GOLD) {
            PutFlag(agent_world, neighbors[i], CURRENT);
            Dfs(neighbors[i]); // 下一步
            if (find_gold || game_over) return;
        }
    }
    // 周围没有金子
    if (find_gold == false) {
        vector<Point> safe_place;
        for (int i = 0; i < neighbors.size(); ++i) {
            if (((agent_world[neighbors[i].xx][neighbors[i].yy] & SAFE) == SAFE)        // 安
全
                && ((agent_world[neighbors[i].xx][neighbors[i].yy] & VISITED) == 0)) { // 未访
问
                safe_place.push_back(neighbors[i]);
            }
        }
        if (safe_place.size() > 0) {    // 存在安全区域
            int rand_next_pos = rand() % safe_place.size();   // 随机选择一个安全区域
            PutFlag(agent_world, safe_place[rand_next_pos], CURRENT);
            Dfs(safe_place[rand_next_pos]); // 下一步
            if (find_gold || game_over) return;
```

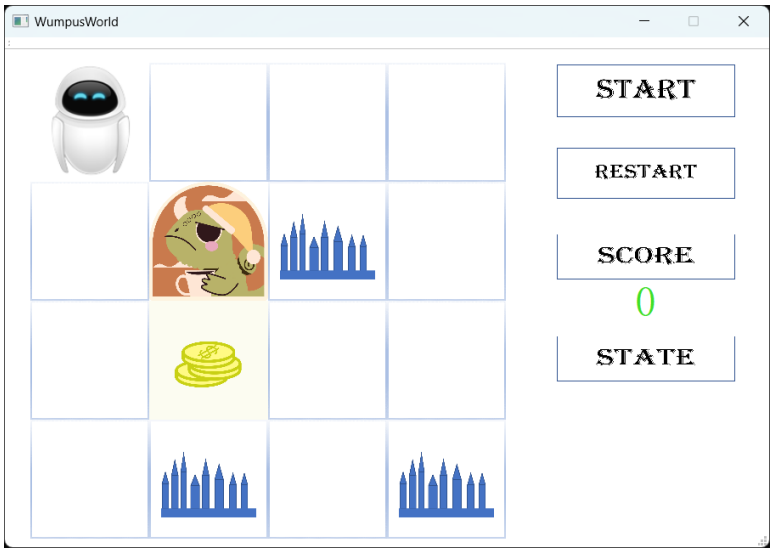
```
        }
        else {        // 没有安全区域，寻找最近的安全地带
            Point nearest_safe_pos = { -1, -1 }; // 最近的安全点初始化
            for (int i = 0; i < ROW_NUM; ++i) {
                for (int j = 0; j < COL_NUM; ++j) {
                    if ((agent_world[i][j] & SAFE) == SAFE) {
                        if (nearest_safe_pos == Point{ -1, -1 })  // 第一个安全的点
                            nearest_safe_pos = { i, j };
                        else {
                            int dismin = Bfs(current, nearest_safe_pos, false); // 最近安全点
距离
                            int discur = Bfs(current, Point{ i, j }, false);    // 当前安全点距
离
                            if (discur < dismin) {   // 当前安全点更近
                                nearest_safe_pos = { i, j };  // 更新最近安全点
                            }
                        }
                    }
                }
            }
            // 机器人地图上有安全点
            if (nearest_safe_pos != Point{ -1, -1 }) {
                PutFlag(agent_world, nearest_safe_pos, CURRENT);
                Bfs(current, nearest_safe_pos, true);// 记录行动路线
                Dfs(nearest_safe_pos);  // 下一步
                if (find_gold || game_over) return;
            }
            else {    // 没有安全点，选择一个相对安全的地点
                vector<Point> good_neighbor;
                for (int i = 0; i < neighbors.size(); ++i) {
                    if (((agent_world[neighbors[i].xx][neighbors[i].yy] & VISITED) == 0)// 未
被访问
                        && ((agent_world[neighbors[i].xx][neighbors[i].yy] & CAVE) == 0)// 没
有陷阱
                        && ((agent_world[neighbors[i].xx][neighbors[i].yy] & WUMPUS) == 0)) //
没有怪兽
                        good_neighbor.push_back(neighbors[i]);
                }

                if (good_neighbor.size() > 0) {  // 有相对安全的点
                    int rand_next_goodpos = rand() % (good_neighbor.size());    // 随机选取一个
相对安全的点
                    Dfs(good_neighbor[rand_next_goodpos]);    // 下一步
                    if (find_gold || game_over) return;
                }
                else {// 没有相对安全的点，选择杀死怪兽
                    vector<Point> kill_pos, cave_pos;
```

```
                    for (int i = 0; i < neighbors.size(); ++i) {
                        if ((agent_world[neighbors[i].xx][neighbors[i].yy] & VISITED) == 0) //
未被访问过
                            if ((agent_world[neighbors[i].xx][neighbors[i].yy] & WUMPUS) ==
WUMPUS) {    // 是怪兽
                                kill_pos.push_back(neighbors[i]);
                            }
                            else {
                                cave_pos.push_back(neighbors[i]);
                            }
                    }
                    if (kill_pos.size() > 0) {   // 存在怪兽则杀死怪兽
                        int kill_wumpus = rand() % kill_pos.size();
                        vector<Point> wumpus_neighbors;
                        GetNeighborPosition(kill_pos[kill_wumpus], wumpus_neighbors);
                        agent_world[kill_pos[kill_wumpus].xx][kill_pos[kill_wumpus].yy] &=
~WUMPUS;// 取消怪兽标记
                        // 将杀死的怪兽周围的臭气标记取消
                        for (int i = 0; i < wumpus_neighbors.size(); ++i) {
                            agent_world[wumpus_neighbors[i].xx][wumpus_neighbors[i].yy] &=
~STENCH;
                        }
                        PutFlag(agent_world, kill_pos[kill_wumpus], CURRENT);
                        Dfs(kill_pos[kill_wumpus]);  // 下一步
                        if (find_gold || game_over) return;
                    }
                    else if (cave_pos.size()) {  // 没有怪兽只有陷阱
                        int jump_cave = rand() % cave_pos.size();
                        PutFlag(agent_world, cave_pos[jump_cave], CURRENT);
                        Dfs(cave_pos[jump_cave]);
                    }
                }
            }
        }
    }
}

//使用Bfs获取最短路径
int Widget::Bfs(Point src, Point tar, bool record)
{
    // 上下左右四个方向
    int Next[4][2] = { {0, 1}, {0, -1}, {-1, 0},{1, 0} };
    int steps = 0;
    map<Point, Point> bfs_path;
    bool flag = false;
    if ((agent_world[tar.xx][tar.yy] & VISITED) == 0) {
        agent_world[tar.xx][tar.yy] |= VISITED;
```

```
        flag = true;
    }

    queue<Point> Q;
    Q.push(src);
    bool vis[ROW_NUM][COL_NUM];     // 遍历标记
    memset(vis, false, sizeof(vis));
    vis[src.xx][src.yy] = true;
    while (!Q.empty()) {
        Point cur = Q.front();
        Q.pop();
        ++steps;
        if (cur == tar) {
            break;
        }
        for (int i = 0; i < 4; ++i) {
            int dx = cur.xx + Next[i][0];   // 下一个点的坐标
            int dy = cur.yy + Next[i][1];
            if (dx >= 0 && dx < ROW_NUM && dy >= 0 && dy < COL_NUM && ((agent_world[dx][dy] &
VISITED) == VISITED) && vis[dx][dy] == false) {
                if (record == true)// 需要记录路径
                    bfs_path[Point{ dx,dy }] = cur;
                vis[dx][dy] = true;
                Q.push({ dx, dy });
            }
        }
    }
    if (record == true) {   // 需要记录则根据搜索路径找到搜索结果
        GetBfsPath(tar, src, tar, bfs_path);
    }
    if (flag) agent_world[tar.xx][tar.yy] &= ~VISITED;// 取消访问标记
    return steps;   // 返回最短路径长度
    }
```

(5) 运行结果及分析

初态：



末态：

在本实验中陷阱的数量设置为 3，点击 start 按钮开始自动寻路，通过 score 标签返回得分，state 标签显示机器人的状态。因为此次随机生成的地图比较容易，agent 能很好地得到高分，在其他恶劣环境中（比如陷阱将金子包围）dfs 算法收效很小，因为在算法设计中采用的是激进的策略，当遇到问题不会进行回溯，而是随机向下试探，这样就导致得分风险很大。

(6) 小结

在此次实验中，我首次感受到了 agent 的设计过程，由于时间原因，我并没有深入设计 agent 的学习、推理机制，所以这并不是一个成功的 agent。在算法设计中同时存在许多问题，例如陷阱位置的推断，假如告知 agent 的陷阱数量，可以用概率算法去推断陷阱位置，可以很大程度上避免死亡，同时没有遍历到所有的 safe 结点，导致 agent 的稳定性很差等等。所以此程序还有很大的改进空间，需要继续努力。

(7) 参考文献

https://github.com/Marko-M/wumpus-world/

实习四：采用 α-β 剪枝算法实现井字棋游戏。随机选取先手后手

(1) 题目描述

采用 α-β 剪枝算法实现井字棋游戏，要求随机选取先手后手。

(2) 算法思想

【1】 极小极大算法

采用极小极大搜索树进行搜索，算法步骤如下：

1、选取先手；

2、轮到 MAX 走时，从当前状态扩展博弈树到所设置深度 h(根据允许时间要求所能的深度)；

3、计算每个叶子节点的评估值（e(s)= 在所有空位放上 MAX 后成 3 子一线的个数-在所有空位放上 MIN 后成 3 子一线的个数)；

4、根据不同的叶子节点采用不同的方法倒推计算各节点值:

对于 MAX 节点选取其后继节点中最大的值为其评估值;

对于 MIN 节点选取其后继节点中最小的值为其评估值

5、选择移动到具有最大倒推值的 MIN 节点；

6、重复直到达到终止条件

【2】 α－β 剪枝

定义：

α：沿着 MAX 路径上的任意选择点，迄今为止我们已经发现的最高值。

β：沿着 MIN 路径上的任意选择点，迄今为止我们已经发现的最低值。

在搜索过程中完成以下操作：

边搜索边更新 α 和 β 的值并且一旦得知当前节点的值比当前 MAX 或 MIN α 或 β 值更差，则在此结点剪去其余的分枝

(3) 软件说明

软件名：AI_Tic_tac_toe

编程环境：使用 windows64 位操作系统基于 vs2022+Qt5.12.12 可视化框架编写

文件大小：479KB

(4) 关键代码

```
//MINMAX算法
int Widget::MinMaxSolution(int depth, int alpha, int beta)
{
    int cur_value = 0, best_value = 0, cnt = 0;
    Point location[10];
    int win_people = GetWinPeople();

    if (win_people == -1 || win_people == 1 || depth == 0) {
        // qDebug() << CalEvalute()<<endl;
        return CalEvalute();
    }

    //深度未耗尽，给定初始值
    if (cur_player == -1) best_value = -N;
    else if (cur_player == 1) best_value = N;
```

```
//获取棋盘上剩余的位置
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
        if (chess[i][j] == 0) {
            //qDebug()<<i<<","<<j<<endl;
            location[cnt].x = i;
            location[cnt].y = j;
            cnt++;
        }
    }
}
cnt -= select_level;    //困难级别
if (cnt < 1) {
    best_point = location[0];
    return best_value;
}

for (int i = 0; i < cnt; ++i) {
    Point cur_pos = location[i];
    int x = cur_pos.x, y = cur_pos.y;
    chess[x][y] = cur_player;   //当前点下一个棋
    cur_player = (cur_player == 1) ? -1 : 1;

    cur_value = MinMaxSolution(depth - 1, alpha, beta);  //向下递归

    chess[x][y] = 0;  //取消下棋
    cur_player = (cur_player == 1) ? -1 : 1;

    if (cur_player == -1) {   // 当前玩家是Max节点
        if (cur_value > best_value) {
            best_value = cur_value;
            if (depth == cur_depth) best_point = cur_pos;
            alpha = best_value;
        }
        if (beta <= alpha) return beta;  // Max中上界小于下界 返回较小值
    }
    else if (cur_player == 1) {   // 当前是Min节点
        if (cur_value < best_value) {
            best_value = cur_value;
            if (depth == cur_depth) best_point = cur_pos;
            beta = best_value;
        }
        if (beta <= alpha) return alpha; // Min中上界小于下界 返回较大值
    }
}
return best_value;
}
```

(5) 运行结果及分析

Hard 级别计算机首次落子

通过场上剩余的格子数量 cnt 减去选择的难度系数 select_level 计算剩余搜索深度，例如对于 hard，cnt 初始值为 9，select_level 值为 0，则搜索深度为 9，下一次递归 cnt 值为 7，则剩余搜索深度为 7，直到对抗结束。

从图中可以以及代码中可以看出当搜索深度为 9 时，计算机优先在四个角落子，找到了最优落子点，并且通过 α－β 剪枝极大的提高了搜索效率，程序运行流畅。

(6) 小结

MINMAX 算法很好的诠释了博弈思想，这也是我首次了解到的具体的对抗搜索算法，让我进一步感受到了智能算法对算法发展的推动作用。

而 α - β 剪枝算法这种搜索树树优化算法可以给我很好的启发，让我明白了算法改进优化的重要性。

(7) 参考文献

https://blog.csdn.net/weixin_45826187/article/details/125408872?ops_request_misc=&request_id=&biz_id=102&utm_term=%E4%BD%BF%E7%94%A8%CE%B1-%CE%B2%E5%89%AA%E6%9E%9D%E5%AE%9E%E7%8E%B0%E4%BA%95%E5%AD%97%E6%A3%8B%E6%B8%B8%E6%88%8F&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-0-125408872.142^v83^insert_down38,201^v4^add_ask,239^v2^insert_chatgpt&spm=1018.2226.3001.4187 **基于 Alpha-Beta 剪枝树的井字棋人机博弈实现**

实 习 评 语

评阅时间：

评阅记录

实习题								
成绩								
上机成绩：				实习报告成绩：				
成绩：				评阅人（签字）：				