

人 工 智 能

课 程 设 计 报 告

班级学号 191181-20181001095

姓 名 常文瀚

评 分 _____

中国地质大学（武汉）计算机学院

2020 年 12 月

一、分别采用随机重启爬山法、最小冲突法和遗传算法求解 n 皇后问题

1.1 题目描述

分别采用随机重启爬山法、最小冲突法和遗传算法求解 n 皇后问题。

1.2 基本思想

1.2.1 随机重启爬山法

对于给定的初始状态如果不能找到解，自己随机生成一个初始状态重启求解，直到找到最终的解为止。

1.2.2 最小冲突法

局部搜索对求解不少 constraint satisfy problem 有着很棒的效果。N 皇后问题就是一个约束满足问题，这里的约束，就是指“皇后之间不能冲突”。该算法使用完全状态的形式化：初始状态每个变量都赋一个值，后继函数每一次改变一个变量的取值，改变取值的依据是使冲突最小化——最小冲突启发式。如果最小冲突值有多个，那么按照一定的概率选择。

1.2.3 遗传算法

遗传算法是一种基于自然选择和群体遗传机理的搜索算法，它模拟了自然选择和自然遗传过程中的繁殖、杂交和突变现象。再利用遗传算法求解问题时，问题的每一个可能解都被编码成一个“染色体”，即个体，若干个个体构成了群体（所有可能解）。在遗传算法开始时，总是随机的产生一些个体（即初始解），根据预定的目标函数对每一个个体进行评估，给出一个适应度值，基于此适应度值，选择一些个体用来产生下一代，选择操作体现了“适者生存”的原理，“好”的个体被用来产生下一代，“坏”的个体则被淘汰，然后选择出来的个体，经过交叉和变异算子进行再组合生成新一代，这一代的个体由于继承了上一代的一些优良性状，因而在性能上要优于上一代，这样逐步朝着最优解的方向进化。因此，遗传算法可以看成是一个由可行解组成的群体初步进化的过程。

1.3 软件设计

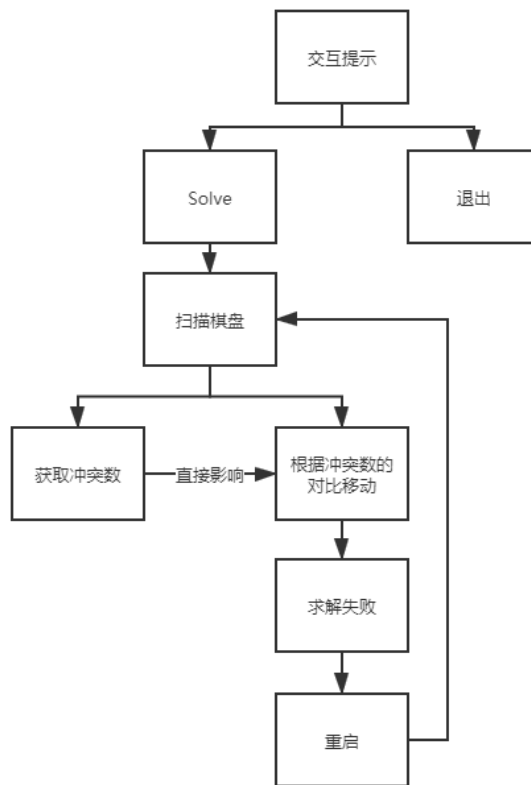
1.3.1 随机重启爬山法

（1）需求分析

通过使用随机重启爬山法，对 N 皇后问题求解，并将最终结果和程序运行时间打印出来。

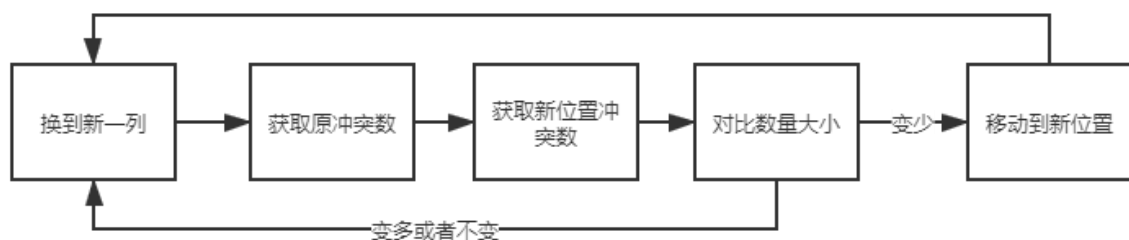
（2）总体设计

随机重启爬山法的总体设计主要有两个分支，其一是正常根据提示输入用户希望解决的 N 皇后问题的皇后数 N，而后算法进行求解，其二是直接退出，其中第二种分支也可以在程序运行期间选择。



(3) 详细设计

在获取冲突数的模块中，因为初始化时已经确定好了每个皇后各占一列使得在列上不会有冲突存在，所以我们只要对每一行和每一个格子对角线的冲突数进行计算，最后将冲突数传递给移动模块。移动模块在获取了获取冲突数的模块送达的数据后，会对当前冲突数和假设移动后的冲突数进行比对，如果冲突数变多则不移动，如果冲突数不变或减少则移动。



(4) 代码实现

// 计算当前棋盘存在的相互攻击的皇后对数

```

int getNumofConflicts(vector<int> *chessboard) {
    int numOfConflicts = 0;
    int width = this->N;
    for (int i = 0; i < width; i++) {
        for (int j = i + 1; j < width; j++) {

```

```

        // 当存在皇后位于对角线的时候 冲突数+1
        if (abs(j - i) == abs((*chessboard)[i] - (*chessboard)[j])) {
            numOfConflicts++;
        }
        // 当存在皇后位于同一列的时候，冲突数+1
        if ((*chessboard)[i] == (*chessboard)[j]) {
            numOfConflicts++;
        }
    }
}

return numOfConflicts;
}

```

```

int moveToTheBestPlace(vector<int>* chessboard, int row)
{
    // 记录为移动之前的位置
    int originPosition = (*chessboard)[row];
    int newPosition;
    int originConflicts = getNumofConflicts(chessboard);
    int numOfConflicts, index;
    if ((*chessboard)[row] < N-1) {
        //cout<<"first"<<(*chessboard)[row]<<endl;

        (*chessboard)[row]++;

        //cout<<"down"<<(*chessboard)[row]<<endl;

        index = (*chessboard)[row];

        numOfConflicts = getNumofConflicts(chessboard);

        (*chessboard)[row]--;

        //cout<<"back"<<(*chessboard)[row]<<endl;
    }

    else if ((*chessboard)[row] == N-1) {
        //cout<<"first"<<(*chessboard)[row]<<endl;
    }
}

```

```

        (*chessboard)[row]--;

        //cout<<"up"<<(*chessboard)[row]<<endl;

        index = (*chessboard)[row];

        numOfConflicts = getNumofConflicts(chessboard);

        (*chessboard)[row]++;

        //cout<<"back"<<(*chessboard)[row]<<endl;

    }

    if(originConflicts <= numOfConflicts){

        newPosition = originPosition;

        //cout<<"return "<<newPosition<<endl;

    }

    else if(originConflicts > numOfConflicts){

        newPosition = index;

        //cout<<"return "<<newPosition<<endl;

    }

    return newPosition;

}

// 每行选择最优位置
vector<int>* scanChessboard(vector<int>* chessboard, int row){

    (*chessboard)[row] = moveToTheBestPlace(chessboard,row);

    return chessboard;

}

// 求解，不断搜寻状态更好的情况，直到冲突数为 0
vector<int>* solve(vector<int> *chessboard) {

    // 随机播种

    srand(time(NULL));

    int resetTime = 0; // 重启步数

    step = 0; // 统计运行步数

```

```

// 当冲突数为 0 时终止爬山

cout<<"计算中"<<endl;

while (getNumofConflicts(chessboard) > 0) {

    if (step >= maxSteps) {

        cout<<step<<endl;

        reset(*chessboard);

        resetTime++;

        step = 0;

        printboard(*chessboard);

        //cout << "随机重启" << endl;

    }

    // 将 rowPosition 行的皇后移到同一行的最优位置

    chessboard = scanChessboard(chessboard,rowPosition++);

    // 判断 rowPosition 是否归零，防止越界

    rowPosition = rowPosition >= N ? rowPosition % N : rowPosition;

    step++;

}

cout << "Solved the problem, totally " << step << " steps. Including " << resetTime << " reset
times." << endl;

return chessboard;

}

};

```

(5) 测试

通过对不同数据进行测试，程序可以正常运行，并且可以将解决一次 N 皇后问题所使用的时间以及结果打印到控制台，经过对结果的检查，认为算法运行正确无误。

1.3.2 最小冲突算法

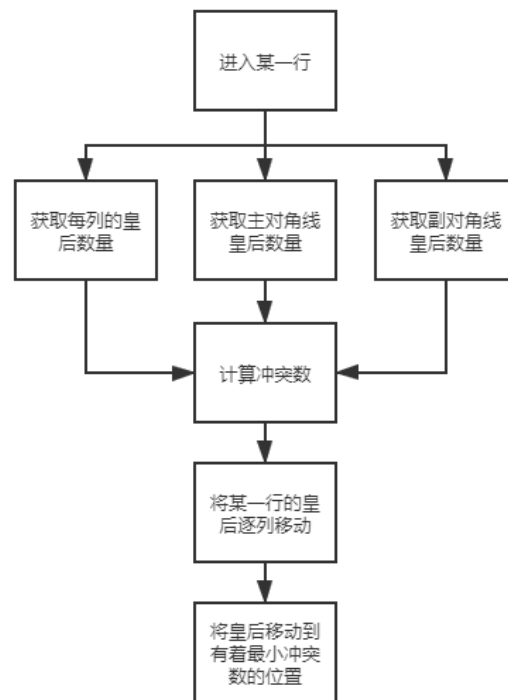
(1) 需求分析

通过使用最小冲突算法，对 N 皇后问题求解，并将最终结果和程序运行时间在控制台中打印出来。

(2) 总体设计

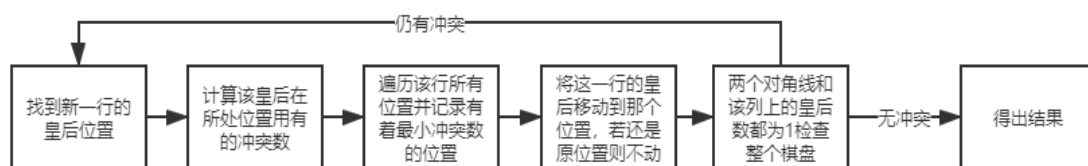
最小冲突法比较好理解，实现起来比较简单，本程序首先获取用户希望解决的皇后数量 N，然

后在确定每一行只有一个皇后的情况下，分别将每一行的皇后的位置在所处行移动，计算冲突数，最后将其移动到具有最小冲突数的地方。



(3) 详细设计

对每一行，先记录这一行皇后所处位置具有的冲突数，在这之后遍历这一行的所有位置，若有位置可以让这个皇后拥有更少的冲突数，那么就把他移动过去，最终如果这个皇后满足在列、主对角线、副对角线上的冲突数为 0，那么就要进行全局判断整个棋盘的皇后布局是否符合 N 皇后问题的求解要求，如果符合，那么就直接结束程序，反之继续对下一行进行遍历求解。



(4) 代码实现

```
bool qualify() {
```

```
    for(int i = 0; i < N; i++){//N queens
```

```
        //判断：如果每一个皇后对应的那一列，两条对角线只有一个皇后，那么就返回 true，反之返回 false
```

```
        //每一行只有一个皇后
```

```
        if(col[R[i]] != 1 || pdiag[getP(i, R[i])] != 1 || cdiag[getC(i, R[i])] != 1) {
```

```
            return false;
```

```

    }
}
return true;
}

```

//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 check 一下看看是否已经没有冲突，如果没有冲突（达到终止状态），返回 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) //如果冲突数变小，就换过去
        {
            min_conflict = conflict; //更新冲突数

            optimal_col = i;          //该行的皇后调整到当前列
        }
    }

    if (optimal_col != cur_col) {      //棋子需要移动的话，要更新 col,pdiag,cdiag

        col[cur_col]--;                //如果出现了移动，那么原先列的棋子减少一个

        pdiag[getP(row, cur_col)]--;  //原来的主对角线减少一个棋子

        cdiag[getC(row, cur_col)]--;  //原来的副对角线减少一个棋子

        col[optimal_col]++;            //如果出现了移动，那么当前列的棋子增加一个
    }
}

```



```

pdiag[getP(row, optimal_col)]++;//当前主对角线棋子增加
cdiag[getC(row, optimal_col)]++;//当前副对角线棋子增加

R[row] = optimal_col;           //该行的棋子设置在当前列

//当出现了这种情况的时候，也就是每一列和两个对角线没有冲突，检查是不是成功了
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

//return qualify();
}

```

(5) 测试

通过对不同数据进行测试，程序可以正常运行，并且可以将解决一次 N 皇后问题所使用的时间以及结果打印到控制台，经过对结果的检查，认为算法运行正确无误。

1.3.3 遗传算法

(1) 需求分析

通过使用遗传算法，对 N 皇后问题求解，并将最终结果和程序运行时间在控制台中打印出来。

(2) 总体设计

遗传算法就是模拟自然选择的过程，将许多棋盘视为一个种群，每一个种群对应每一个个体。每两个个体进行结合产生一个新的个体，多次结合产生的新个体形成新的种群，然后再次繁殖，直到出现有适应度最高的个体，但是遗传算法，本算法中适应度函数为每个棋盘个体中不冲突的皇后对数



(3) 详细设计

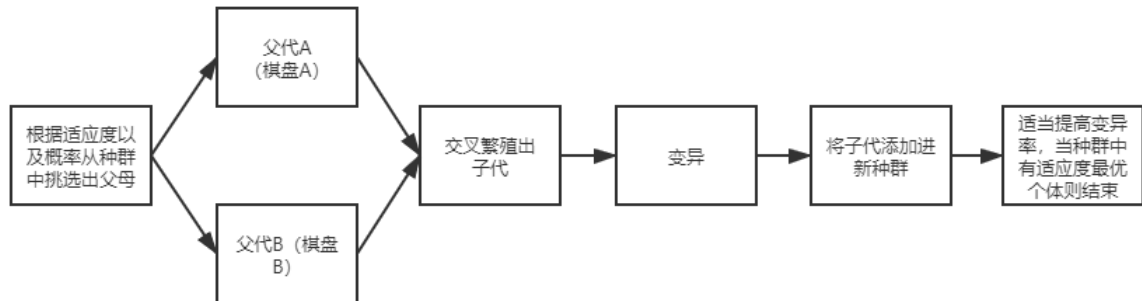
遗传算法有三个基本操作：选择，交叉，变异。

选择：选择一些个体来产生下一代。一种常用的选择策略是 “比例选择”，也就是个体被选中

的概率与其适应度函数值成正比。

交叉：两个棋盘交换部分的行得到一个新的物种，本次交叉采用两个变异点，交换两点之间的办法。

变异：随机挑选一个行的位置，随机改变该行皇后的位置。



(4) 代码实现

// 计算个体最大的不冲突个数

```
int calculateMaxNotConflicts(int N) {  
    int num = 0;  
    for (int i = N - 1; i > 0; i--) {  
        num += i;  
    }  
    return num;  
}
```

// 适应度函数，检查个体不冲突的皇后数

```
int getNumOfNotConflicts(vector<int> *chessboard) {  
    int numOfNotConflicts = 0;  
    int width = this->N;  
    for (int i = 0; i < width; i++) {  
        for (int j = i + 1; j < width; j++) {  
            // 当存在皇后不位于对角线的时候 且不在同一列时  
            //cout << i << " " << j << " " << N << " " << chessboard->size() << endl;  
            if (abs(j - i) != abs((*chessboard)[i] - (*chessboard)[j]) && (*chessboard)[i] !=  
                (*chessboard)[j]) {  
                numOfNotConflicts++;  
            }  
        }  
    }  
}
```

```

    }

    }

}

return numOfNotConflicts;

}

```

// 随机选择两个个体,轮盘赌算法: 思想是产生一个 0~1 的随机数, 按适应度的比例挑选个体

```

vector<vector<int>>>* select(const vector<vector<int>>& population,const vector<double>& fitness,
vector<vector<int>>& parents) {

    float m = 0;

    parents.clear();

    float p1 = (rand() % 100) * 1.0 / 100;

    float p2 = (rand() % 100) * 1.0 / 100;

    for (int i = 0; i < numOfIndividuals; i++) {

        m += fitness[i];

        if (p1 <= m) {

            // 加入第一个个体

            // 产生的随机数 p1 在 m ~m+fitness[i] 间则认为选择了 i

            //   cout << "select1 " << p1 << " " << m << " " << "i " << i << endl;

            parents.push_back(population[i]);

            break;

        }

    }

    m = 0;

    for (int i = 0; i < numOfIndividuals; i++) {

        m += fitness[i];

        // 加入第二个个体

        if (p2 <= m) {

            //   cout << "select2 " << p2 << " " << m << " " << "i " << i << endl;

            parents.push_back(population[i]);

            break;

        }

    }

}

```

```

    }

}

return &parents;

}

```

// 对两个个体进行杂交，产生一个后代,双点+双侧杂交

```

vector<int>* crossover(const vector<int>& chessboard1, const vector<int>& chessboard2, vector<int>&son) {

    //srand((unsigned)time(NULL));

    int pos1 = 0, pos2 = 0;

    son.clear();

    while (pos1 >= pos2) {

        pos1 = rand() % N;

        pos2 = rand() % N;

    }

    for (int i = 0; i < this->N; i++) {

        if (i < pos1 || i > pos2) {

            son.push_back(chessboard1[i]);

        }

        else {

            son.push_back(chessboard2[i]);

        }

    }

    //cout << "crossover: " << pos1 << " " << pos2 << endl;

    return &son;

}

```

// 求解

```

vector<int>* solve(vector<int>& chessboard) {

    /*
        设置种群参数

        pMutate: 变异发生的概率

        pCrossover: 交叉发生的概率
    */

    double pMutate = 0.2;

    double pCrossover = 0.9;

    vector<int> individual;

    population = *createPopulation(population,numOfIndividuals); // 初始化种群

    int numOfGeneration = 0; // 进化的代数

    srand((unsigned)time(NULL));

    // 开始进化

    do {

        // 计算种群中每一个个体的适应度值,结果保存在 fitness

        calculateFitness(population, fitness);

        vector<vector<int>> newPopulation;

        do {

            // 随机挑选两个个体作为父母

            vector<vector<int>> parents;

            parents = *select(population, fitness, parents);

            // 只产生更优的儿子

            while (1) {

                // 防止不交叉, son 为空

                vector<int> son = parents[0];

                // 随机播种, 产生随机数

                // 一定概率发生交叉, 产生儿子,儿子适应度更高

                float pc = rand() % 100 * 1.0 / 100;

                if (pc < pCrossover) {

                    son = *(crossover((parents).at(0), (parents).at(1), son));

```

```

    }

    // 一定概率发生变异，改变儿子

    float pm = rand() % 100 * 1.0 / 100;

    if (pm < pMutate) {

        son = *mutate(son);

    }

    // 若儿子优于或等于父母，则添加到种群中

    if (getNumOfNotConflicts(&son) >= getNumOfNotConflicts(&parents[0]) &&
getNumOfNotConflicts(&son) >= getNumOfNotConflicts(&parents[1])) {

        //cout    <<    getNumOfNotConflicts(&son)    <<    "    "    <<

getNumOfNotConflicts(&parents[0]) << " " << getNumOfNotConflicts(&parents[0]) << endl;

        // 将儿子加入新的种群中

        /*if (getNumOfNotConflicts(&son) == 27) {

            printChessboard(son);

            system("pause");

        }*/

        newPopulation.push_back(son);

        son.clear();

        parents.clear();

        break;

    }

    else {

        if(pMutate <= 0.98)

            pMutate += 0.02;

    }

}

} while(newPopulation.size() != numOfIndividuals);

// 新种群替代旧种群

population.clear();

for (int i = 0; i < numOfIndividuals; i++) {

    population.push_back(newPopulation[i]);

```

```

    }

    numOfGeneration++;

} while (positionOfNotConflict(population) == -1);

// 得到种群中满足适应度的个体

chessboard = population[positionOfNotConflict(population)];

cout << "total generations: " << numOfGeneration << endl;

return &chessboard;

}

};

```

(5) 测试

通过对不同数据进行测试，程序可以正常运行，并且可以将解决一次 N 皇后问题所使用的时间以及结果打印到控制台，经过对结果的检查，认为算法运行正确无误。

1.4 运行结果与分析

1.4.1 随机重启爬山法运行结果

在对随机重启爬山法的测试中，我选用的皇后数目为：10

第一次测试结果：834ms

```

1 --- Hill Climbing Algorithm (随机重启爬山法)
棋盘初始化并打印
计算中
Solved the problem, totally 68 steps. Including 13 reset times.
[ 0] total time: 834 ms.

= = = = = Q =
= = = = Q = = =
Q = = = = = =
= = Q = = = = =
= = = = = = = Q
= = = = = Q = =
= = = Q = = = =
= = = = = Q =
= = = Q = = = =
= Q = = = = =

-----
时间： 834 ms

```

1.4.2 最小冲突法运行结果

在对随机重启爬山法的测试中，我选用的皇后数目为：10

第一次测试结果：8ms

在实际运用中，爬山算法不会考虑与当前状态不直接相邻的状态，只会选择比当前状态价值更好的相邻状态，所以简单来说，爬山算法是向价值增长方向持续移动的循环过程。

(2) 最小冲突算法

该算法最精彩的地方，是时间复杂度可以优化到 $O(n^2)$ ，这是另两种算法完全无法比拟的。

(3) 遗传算法

遗传算法是一种基于自然选择和群体遗传机理的搜索算法，它模拟了自然选择和自然遗传过程中的繁殖、杂交和突变现象，体现了“适者生存”的原理，选择出来的个体，经过交叉和变异再组合生成新一代，这一代的个体由于继承了上一代的一些优良性状，因而在性能上要优于上一代，这样逐步朝着最优解的方向进化。

1.5 使用说明

- (1) 本程序使用控制台作为用户与计算机交互的平台，用户只需要输入数字作为对程序指挥的命令
- (2) 程序在运行后会输出运行的结果以及运行时间，可以作为算法对不同数量皇后问题求解速度的直观反映，也可以用于某一种算法与其他算法求解速度的对比。
- (3) 在输入了一次皇后数量 N 后无法修改皇后数量，若想要使用该程序对其他有数量的皇后问题求解则需要重新启动该程序。

1.6 小结

通过将三种算法应用到实际问题当中，我更加清晰的了解到了每一种算法运行的思路，并在将三种算法进行性能对比时探究了它们具有性能差距的原因，使我更加透彻的掌握三种算法，希望以后可以更多的将三种算法应用到实际问题的解决中。

1.7 参考文献

- [1] https://www.iteye.com/resource/qq_16547997-9335887
- [2] <https://blog.csdn.net/a19990412/article/details/83304430>
- [3] <https://blog.csdn.net/CVSsvsvsvs/article/details/82959061>
- [4] <https://www.cnblogs.com/ktao/p/7721811.html>
- [5] <https://blog.csdn.net/jzp1083462154/article/details/80032987>
- [6] <https://blog.csdn.net/u013390476/article/details/50011261>

二、使用联机搜索求解 Wumpus 怪兽世界问题

2.1 题目描述

- Wumpus World PEAS 描述:
- 性能度量: gold +1000, death -1000, -1 per step, -10 for using the arrow
- 环境描述:
 - Squares adjacent to wumpus are smelly
 - Squares adjacent to pit are breezy
 - Glitter iff gold is in the same square
 - Shooting kills wumpus if you are facing it
 - Shooting uses up the only arrow
 - Grabbing picks up gold if in same square
 - Releasing drops the gold in same square
- 传感器: Stench, Breeze, Glitter, Bump, Scream
- 执行器: Left turn, Right turn, Forward, Grab, Release, Shoot

2.2 基本思想

(1) 随机生成金子, 怪兽, 陷阱的位置, 且它们的位置各不相同。从起始点开始, 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。重复上述过程直到找到金子然后回家或者掉入陷阱死亡。

(2) 联机搜索

联机搜索问题只能通过 Agent 执行行动来求解, 它不是纯粹的计算过程。我们假设环境是确定的和完全可观察的, 我们规定 Agent 只知道以下信息:

- $ACTIONS(s)$, 返回状态 s 下可能进行的行动列表;
- 单步耗散函数 $c(s, a, s')$ ——注意 Agent 知道行动的结果为状态 s' 时才能用;
- $GOAL-TEST(s)$ 。

在每个活动之后, 联机 Agent 都能接收到感知信息, 告诉它到达的当前状态; 根据此信息, Agent 可以扩展自己的环境地图。可以用当前地图来决定下一步往哪里走。这种规划和活动的交替是联机搜索算法和前面讲的脱机搜索算法的不同点。

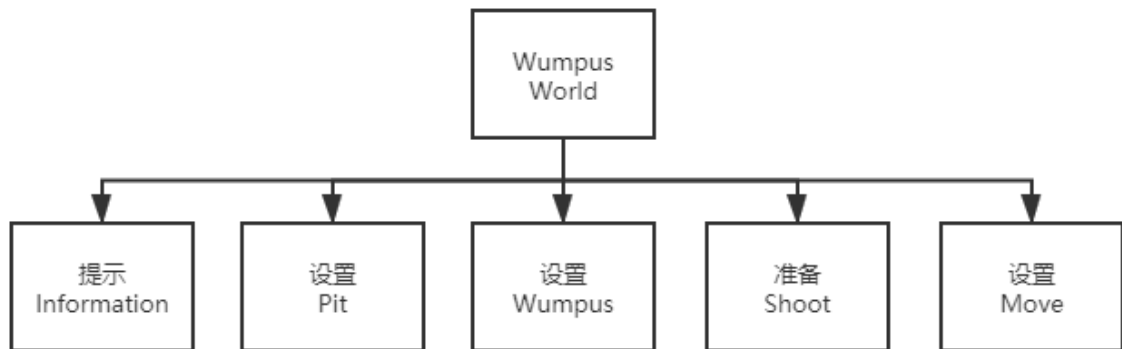
2.3 软件设计

(1) 需求分析

Wumpus 世界是由多个房间组成并相连接起来的山洞。在洞穴的某处隐藏着一只 Wumpus (怪兽), 它会吃掉进入它房间的任何入。Agent 可以射杀 Wumpus, 但是 Agent 只有一枝箭。某些房间是无底洞, 任何人漫游到这些房间都会被无底洞吞噬 (Wumpus 除外, 它由于太大而幸免)。生活在该环境下的唯一希望是存在发现一堆金子的可能性, 并找到黄金带回出发点。

(2) 总体设计

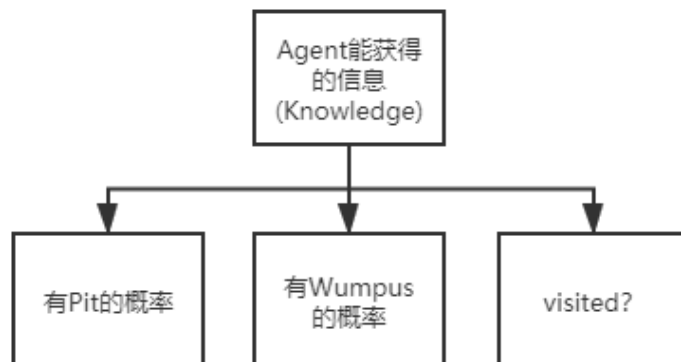
整个程序总体设计包括五个模块：设置 Pit 模块、设置 Wumpus 模块、移动模块、射击模块、信息提示模块。



地图中的每一个方格记录了五个信息：是否有臭味、是否有风、是否有黄金、是否被怪物杀掉、是否掉进了洞里。



Agent 每次移动都可以获得三个信息：位置有 Pit 的概率、位置有 Wumpus 的概率、是否走过。



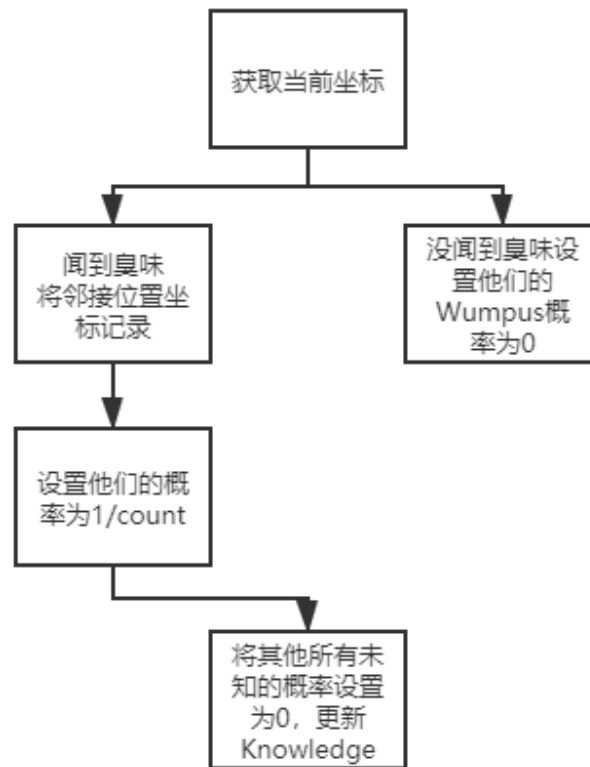
(3) 详细设计

首先，信息提示模块会把当前位置是否有臭味、风或者黄金报告出来，在这之后将会打印出当前位置临接位置存在 Wumpus 或者存在 Pit 的概率。

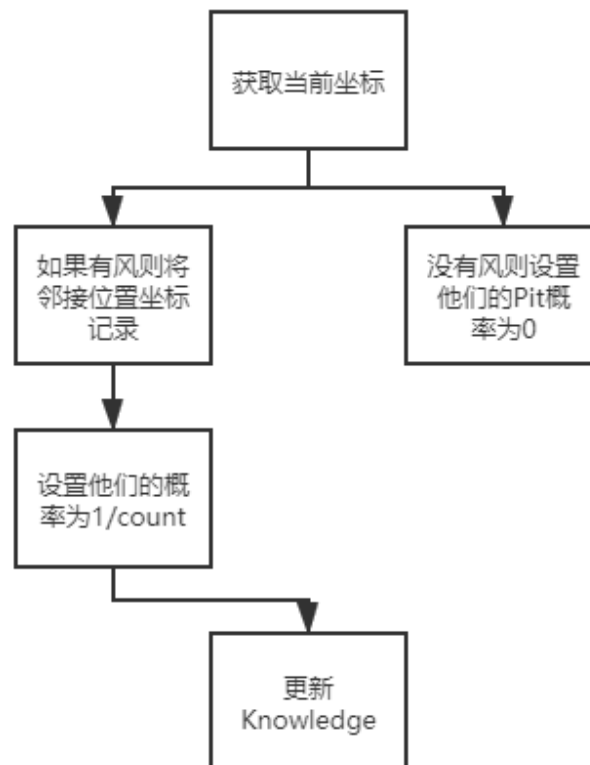
在这之后，要判断是否被 Wumpus 吃掉了或是掉进了洞里，如果两种情况都没有发生那么就要

根据当前位置距有的信息更新 Agent 的 Knowledge。

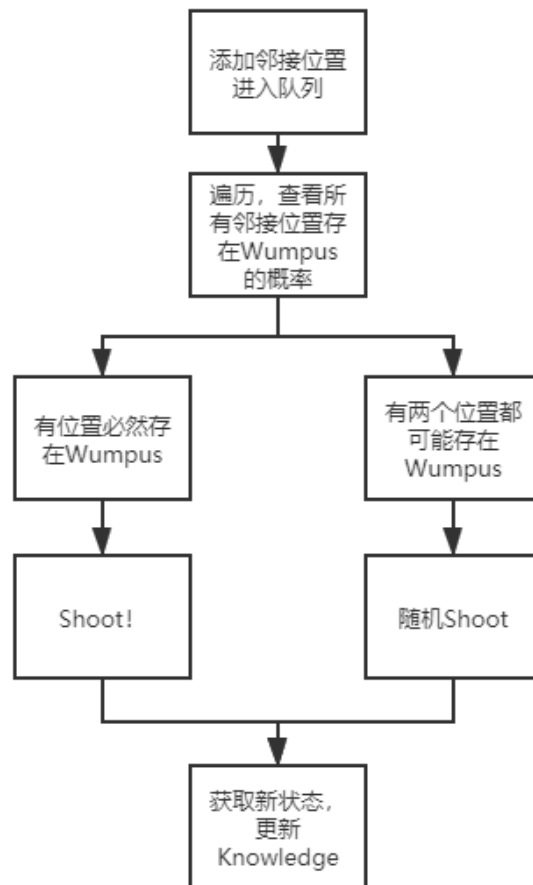
Wumpus 模块：



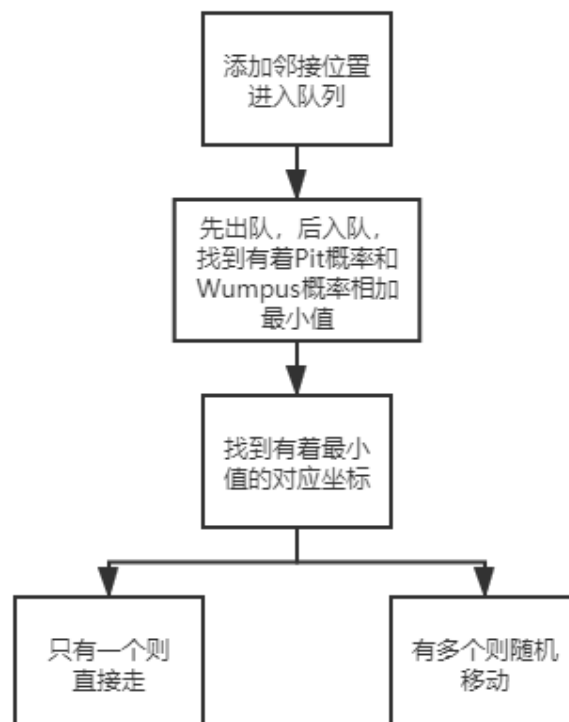
Pit 模块：



Shoot 模块:



Move 模块:



(4) 代码实现

```
public static void main(String[] args){
    int[][][] map = new int[4][4][5]; //create a 4X4 grid, where each point on the grid has 5 pieces
of data
    int AgentX = 0, AgentY = 3; //agent starts in bottom left corner of map

    Random r = new Random();

    //randomly place Wumpus
    int WumpusX;
    int WumpusY;
    do{
        WumpusX = r.nextInt(4);
        WumpusY = r.nextInt(4);
    }while(WumpusX == 0 && WumpusY == 3); //Wumpus cannot be in agent's starting position

    map[WumpusX][WumpusY][3] = 1;

    //randomly place gold
    int GoldX;
    int GoldY;
    do {
        GoldX = r.nextInt(4);
        GoldY = r.nextInt(4);
    } while((GoldX == 0 && GoldY == 3) | (GoldX == WumpusX && GoldY == WumpusY));
//Gold cannot be in agent's starting position, or where the Wumpus is
    map[GoldX][GoldY][2] = 1;
    /*
    Randomly add pits. Each location has a 0.2 chance of having a pit, except for location of
    Wumpus, location of gold, and agent's starting location
    */
    for(int x=0; x<4; x++){
        for(int y=0; y<4; y++){
            if(!(x == AgentX && y == AgentY)) && !(x == WumpusX && y == WumpusY)
&& !(x == GoldX && y == GoldY)){
                int pit = r.nextInt(10);
                if(pit<2){
                    map[x][y][4] = 1;
                }
            }
        }
    }
    //add a stench to squares directly adjacent to Wumpus
    for(int x=0; x<4; x++){
        for(int y=0; y<4; y++){

            if(x == WumpusX && y == WumpusY){
```

```

        if(WumpusX > 0){
            map[x-1][y][0] = 1;
        }
        if(WumpusX < 3){
            map[x+1][y][0] = 1;
        }
        if(WumpusY > 0){
            map[x][y-1][0] = 1;
        }
        if(WumpusY < 3){
            map[x][y+1][0] = 1;
        }
    }
}

//add a breeze to squares directly adjacent to pits
for(int x=0; x<4; x++){
    for(int y=0; y<4; y++){
        if(map[x][y][4] == 1){
            if(x > 0){
                map[x-1][y][1] = 1;
            }
            if(x < 3){
                map[x+1][y][1] = 1;
            }
            if(y > 0){
                map[x][y-1][1] = 1;
            }
            if(y < 3){
                map[x][y+1][1] = 1;
            }
        }
    }
}

//print the starting Wumpus Environment
System.out.print("    0    1    2    3");
for(int y=0; y<4; y++){
    System.out.print("\n" + y + " |");
    for(int x=0; x<4; x++){
        if(x == AgentX && y == AgentY){
            System.out.print(" A ");
        }else if(x == WumpusX && y == WumpusY){
            System.out.print(" W ");
        }else if(x == GoldX && y == GoldY){
            System.out.print(" G ");
        }else if(map[x][y][4] == 1){

```

```

        System.out.print(" P ");
    } else {
        System.out.print("   ");
    }
    System.out.print("|");
}
}
System.out.println("\n");

/*store all knowledge the agent has acquired. The knowledge for each square includes:
chance the square has a pit, the chance it has a Wumpus, and whether it has been visited by
the agent
*/
double[][][] knowledge = new double[4][4][3];

//the agent starts by assuming each square has a 0.2 chance of having a pit, and a 1/15 chance
of having the Wumpus
for(int x=0; x<4; x++){
    for(int y=0; y<4; y++){
        if(!(x==0 && y==3)){
            knowledge[x][y][0] = 0.2;
            knowledge[x][y][1] = 1d/15d;
        }
    }
}

boolean done = false; //true once the gold has been brought back to the starting point
boolean hasGold = false;
boolean hasArrow = true; //The agent starts with one arrow it can use to kill the wumpus
while(!done){
    knowledge[AgentX][AgentY][2] +=1; //mark the current location as visited
    System.out.println("\nCurrent location: (" + AgentX + ", " + AgentY + ")");

    if(map[AgentX][AgentY][3] == 1){
        System.out.println("The agent has been eaten by the Wumpus.");
        break;
    } else if(map[AgentX][AgentY][4] == 1){
        System.out.println("The agent has fallen in a pit.");
        break;
    }
    //update knowledge base based on whether or not the agent smells a stench
    knowledge = Wumpus.wumpus(knowledge, map, AgentX, AgentY,
map[AgentX][AgentY][0]);

    //update knowledge base based on whether or not the agent feels a breeze
    knowledge = Pit.pit(knowledge, map, AgentX, AgentY, map[AgentX][AgentY][1]);
    if(map[AgentX][AgentY][2] == 1){
        System.out.println("The Agent has picked up the gold.");
    }
}

```



```

        hasGold = true;
    }
    //display information about the current location
    Information.information(AgentX, AgentY, knowledge, map);
    if(AgentX == 0 && AgentY == 3 && hasGold == true){
        done = true;
        System.out.println("\nThe agent has brought the gold back to the beginning!");
    }
    //decide whether or not to attempt to shoot the wumpus
    int[][][] m = map;
    if(hasArrow == true){
        map = Shoot.shoot(AgentX, AgentY, knowledge, map);
    }
    //if the knowledge base has changed, the arrow has been shot
    if(m != map){
        hasArrow = false;
    }
    //decide where to move. Agent will move to adjacent square with lowest chance of
    having a pit/wumpus
    int[] location = Move.move(AgentX, AgentY, knowledge, hasGold);
    AgentX = location[0]; AgentY = location[1];
}
}

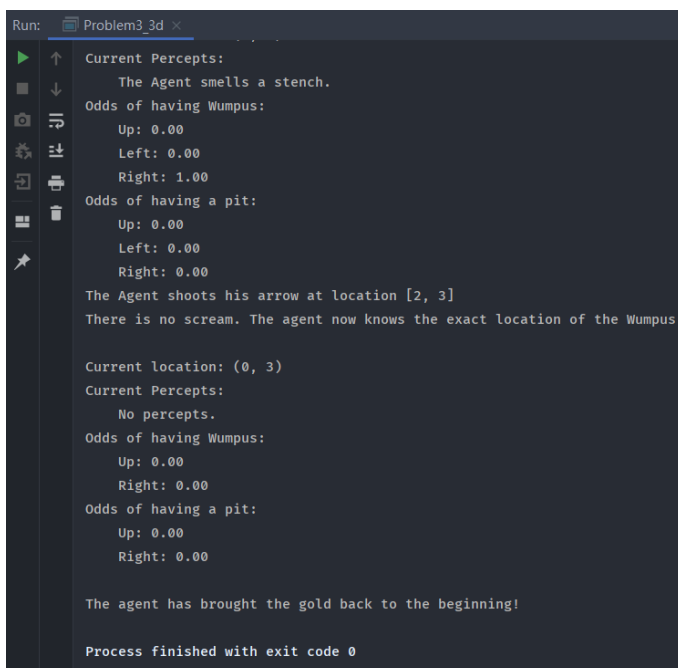
```

(5) 测试

通过对不同数据进行测试，程序可以正常运行，并且可以将解决一次 Wumpus 问题结果打印到控制台，经过对结果的检查，认为算法运行正确无误。

2.4 运行结果及分析

2.4.1 运行结果



```

Run: Problem3_3d x
Current Percepts:
  The Agent smells a stench.
Odds of having Wumpus:
  Up: 0.00
  Left: 0.00
  Right: 1.00
Odds of having a pit:
  Up: 0.00
  Left: 0.00
  Right: 0.00
The Agent shoots his arrow at location [2, 3]
There is no scream. The agent now knows the exact location of the Wumpus

Current location: (0, 3)
Current Percepts:
  No percepts.
Odds of having Wumpus:
  Up: 0.00
  Right: 0.00
Odds of having a pit:
  Up: 0.00
  Right: 0.00

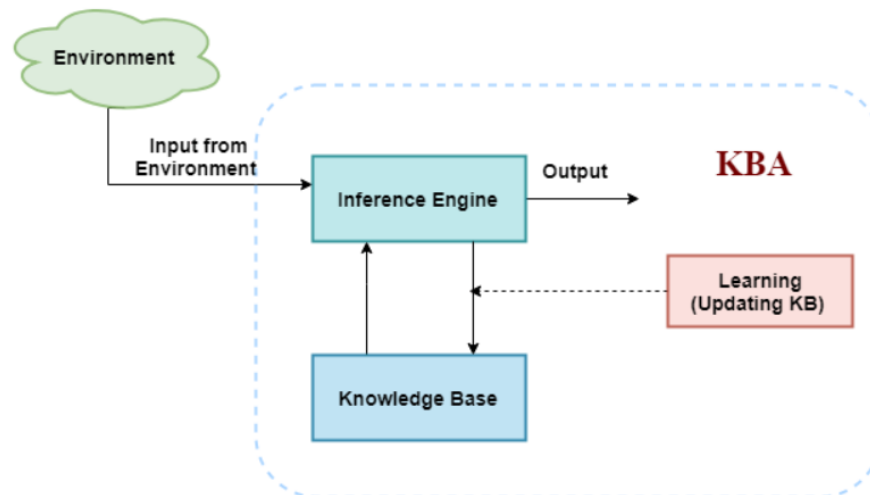
The agent has brought the gold back to the beginning!
Process finished with exit code 0

```

2.4.2 分析

程序基于 Knowledge-based-agent 算法，能够成功在棋盘初始化后让 Agent 在游戏世界中移动并且在一定程度上避开 Wumpus 和 Pit，在这之后也可以拿到黄金，然后按照移动策略返回出发点。

The architecture of knowledge-based agent:



主要的分析应当集中在联机搜索的 Agent: Knowledge-based Agent 上。上面的图代表了一个 Knowledge-based 的 Agent 的通用架构。Knowledge-based Agent(KBA)通过感知环境来获取环境的输入。该输入由 Agent 的推理引擎获取，推理引擎也根据知识库中的知识与知识库通信来决定。KBA 的学习元素通过学习新知识定期更新知识库，这使得 Wumpus World 中的 Agent 能够不断学习到周围的每一个位置的信息。

Knowledge-base 是 Knowledge-based Agent 的核心组成部分，又称 KB。它是一个 sentence 的集合(这里的“sentence”是一个专业术语，它不等同于语言中的 sentence)。这些句子用一种叫做知识表示语言的语言来表达，KBA 的知识库存储关于世界的事实，本题代码中的 sentence 由一个具有三个元素的数组表示，其中每个元素都代表着在这个位置出现特定情况的概率。

2.5 小结

通过以 Wumpus 怪兽世界为例，我在各种资料和网站中查找资料，充分的了解到了联机搜索、Wumpus 怪兽世界问题与人工智能的联系等一系列知识，在这里面我挑选了 Knowledge-based Agent 作为整个游戏运行的一个策略，这不但让我将人工智能联系到了实际应用，更是拓展了眼界看到了更多的和人工智能相关的算法，我认为在今后的学习中更应该把理论和实际结合起来。

2.6 参考文献

- [1] <https://www.javatpoint.com/knowledge-based-agent-in-ai>
- [2] <https://www.javatpoint.com/ai-knowledge-base-for-wumpus-world>
- [3] https://github.com/mattkrikorian/WumpusWorld/tree/master/Problem3_3d

[4] <https://github.com/s-sandra/wumpus-adventurer>

[5] 人工智能：一种现代的方法（第三版）

三、采用 $\alpha - \beta$ 剪枝算法实现井字棋游戏

3.1 题目描述

采用 $\alpha - \beta$ 剪枝算法实现井字棋游戏。

- 图形化界面。
- 随机选取先手后手。

可以人-计算机或计算机-计算机

3.2 基本思想

根据倒推值的计算方法，或中取大，与中取小，在扩展和计算过程中，能剪掉不必要的分枝，提高效率。

3.3 软件设计

（1）需求分析

- ① 使用 alpha-beta 剪枝算法
- ② 游戏具有图形化界面
- ③ 游戏可以进行“人-计算机”对战和“计算机-计算机”对战

（2）总体设计

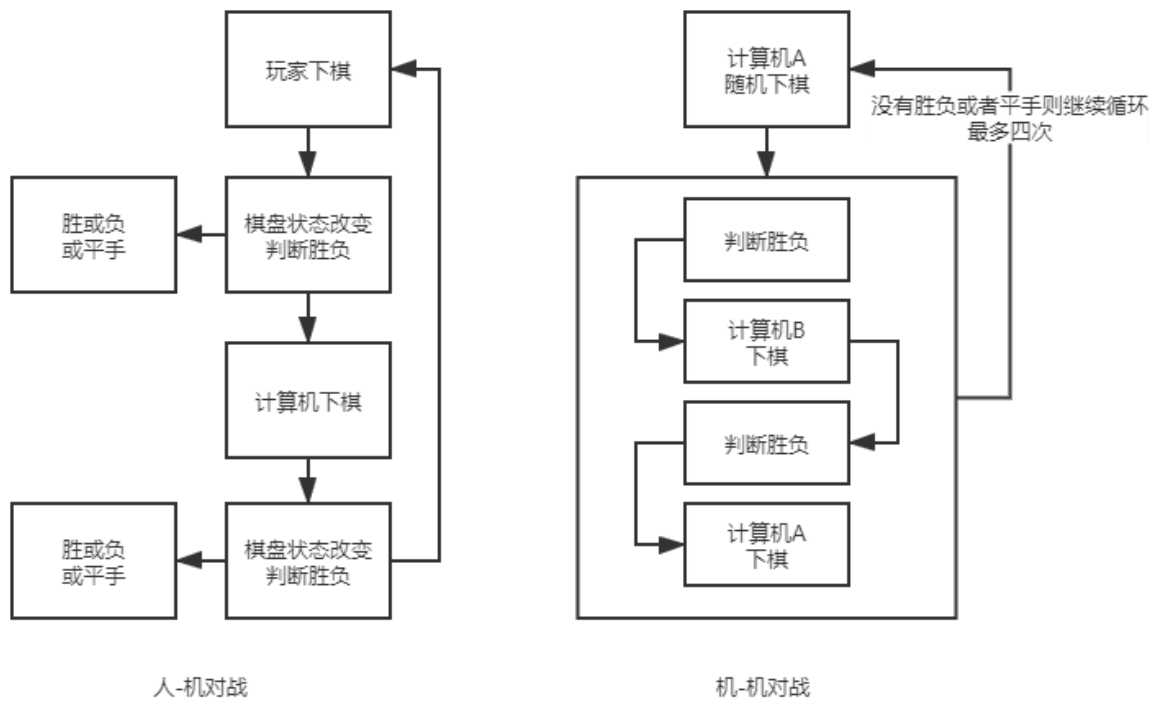
① 人机下棋

在整个棋盘中，每一个棋子落下的位置都是一个按钮，这样每一个按钮按下时触发了对应的槽函数，在这个槽函数中，首先就要将这个按钮的状态改变，然后，判断整个棋局是否下棋的一方胜利或者平局，如果两种情况均未发生，那么调用计算机下棋的函数，这里要注意会运用到 alpha-beta 剪枝算法。在每一次的下棋前后都要判断是否有一方胜利或是双方平手！

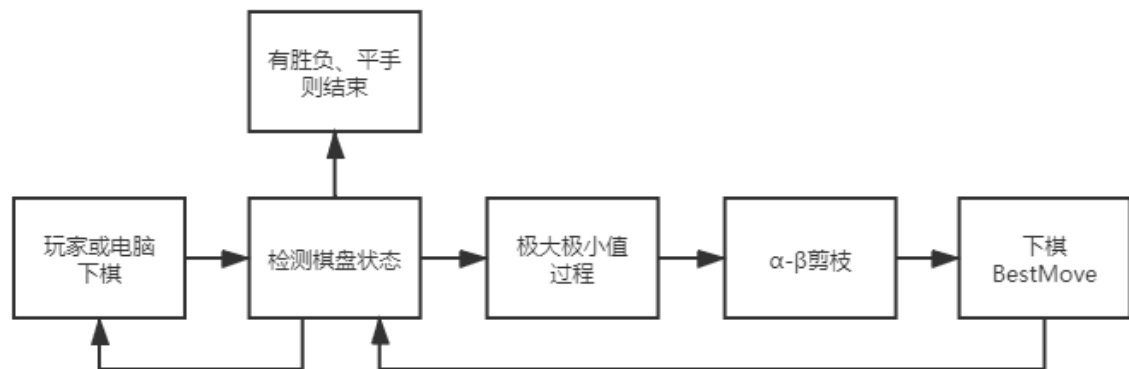
② 计算机自我下棋

计算机自我下棋要注意的是第一步需要随机选择一个位置下棋，我使用了一个随机函数选择 1-9 之内的任意一个数字，它对应的是整个棋盘中的九个位置，这样函数给出的数字就代表着第一手棋落在哪里。

这时，整个棋盘上还有八个空位置，因为只有“圈”与“叉”两方对战，那么使用一个四次的循环就可以把棋盘下满，同时，将判断胜负以及判断平手的函数穿插在双方下棋的交替过程中，就可以做到在某一方下棋的前后进行胜负的判断，在棋盘出现可以判定的结果时结束进程。



(3) 详细设计



(4) 代码实现

```
int MainWindow::a_isWin() {
    // 判断横向输赢
    for (int i = 0; i < 3; i++)
    {
        if (board[i][0] + board[i][1] + board[i][2] == 3)
            return 1;
        else if (board[i][0] + board[i][1] + board[i][2] == -3)
            return -1;
    }
}
```

```

// 判断竖向输赢

for (int j = 0; j < 3; j++)

{

    if (board[0][j] + board[1][j] + board[2][j] == 3)

        return 1;

    else if (board[0][j] + board[1][j] + board[2][j] == -3)

        return -1;

}

// 判断斜向输赢

if (board[0][0] + board[1][1] + board[2][2] == 3 || board[0][2] + board[1][1] + board[2][0] == 3)

    return 1;

else if (board[0][0] + board[1][1] + board[2][2] == -3 || board[0][2] + board[1][1] + board[2][0]

== -3)

    return -1;

else    return 0;

}

// 人机对战的平局

bool MainWindow::a_isDraw()

{

    int times = 0;

    for (int i = 0; i < 3; i++) {

        for (int j = 0; j < 3; j++) {

            if (board[i][j] != 0) {

                times ++;

            }

        }

    }

}

// 平局

if (times == 9) {

    QMessageBox::information(this, tr("井字棋"), tr("平局! "), QMessageBox::Ok);

```

```

        ending();

        return true;
    }

    else

    {

        return false;
    }
}

// 评估函数

int MainWindow::a_evaluteMap() {

    int i, j;

    //如果计算机赢了，返回最大值

    if (a_isWin() == COM)

        return MAX_NUM;

    //如果计算机输了，返回最小值

    if (a_isWin() == MAN)

        return -MAX_NUM;

    //该变量用来表示评估函数的值

    int count = 0;

    //将棋盘中的空格填满自己的棋子，既将棋盘数组中的 0 变为 1

    for (i = 0; i < 3; i++)

        for (j = 0; j < 3; j++)

            {

                if (board[i][j] == 0)

                    tempBoard[i][j] = COM;

                else

                    tempBoard[i][j] = board[i][j];

            }

    //电脑一方

    //计算每一行中有多少行的棋子连成 3 个的

    for (i = 0; i < 3; i++)

```

```

        count += (tempBoard[i][0] + tempBoard[i][1] + tempBoard[i][2]) / 3;
    for (i = 0; i < 3; i++)

        count += (tempBoard[0][i] + tempBoard[1][i] + tempBoard[2][i]) / 3;
    count += (tempBoard[0][0] + tempBoard[1][1] + tempBoard[2][2]) / 3;
    count += (tempBoard[2][0] + tempBoard[1][1] + tempBoard[0][2]) / 3;
    //将棋盘中的空格填满对方的棋子，既将棋盘数组中的 0 变为-1
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
        {
            if (board[i][j] == 0)
                tempBoard[i][j] = MAN;
            else tempBoard[i][j] = board[i][j];
        }
    //对方
    //计算每一行中有多少行的棋子连成 3 个的
    for (i = 0; i < 3; i++)
        count += (tempBoard[i][0] + tempBoard[i][1] + tempBoard[i][2]) / 3;
    for (i = 0; i < 3; i++)
        count += (tempBoard[0][i] + tempBoard[1][i] + tempBoard[2][i]) / 3;
    count += (tempBoard[0][0] + tempBoard[1][1] + tempBoard[2][2]) / 3;
    count += (tempBoard[2][0] + tempBoard[1][1] + tempBoard[0][2]) / 3;
    // 返回的数因为包括了负数和整数，所以不会太大
    return count;
}

void MainWindow::a_makeMove(Move curMove) {
    board[curMove.x][curMove.y] = player;
    player = (player == COM) ? MAN : COM;
}

void MainWindow::a_unMakeMove(Move curMove) {

```

```
        board[curMove.x][curMove.y] = 0;

        player = (player == COM) ? MAN : COM;
    }
}
```

// 获取棋盘上一共还剩多少步

```
int MainWindow::a_getMoveList(Move moveList[]) {

    int moveCount = 0;

    int i, j;

    for (i = 0; i < COL; i++)

    {

        for (j = 0; j < ROW; j++)

        {

            if (board[i][j] == 0)

            {

                moveList[moveCount].x = i;

                moveList[moveCount].y = j;

                moveCount++;

            }

        }

    }

    //返回一共多少个空的位置

    return moveCount;

}
```

// 极小极大过程

```
int MainWindow::a_miniMaxsearch(int depth) {

    //估值

    int value;

    //最好的估值

    int bestValue = 0;

    int moveCount = 0;
```



```

int i;

//保存可以下子的位置

Move moveList[9];

// 如果在深度未耗尽之前赢了

if (a_isWin() == COM || a_isWin() == MAN)
{
    // 这里返回的评估函数值是给递归用的

    return a_evaluateMap();
}

//如果搜索深度耗尽

if (depth == 0)
{
    // 这里返回的评估函数值是给递归用的

    return a_evaluateMap();
}

// 如果在深度未耗尽并且都没赢。

// 先给一个初始值

if (COM == player) {
    bestValue = -MAX_NUM;
}

else if (MAN == player)
{
    bestValue = MAX_NUM;
}

//深度未耗尽并且都没赢的情况下，电脑需要获取到棋盘剩余的位置，并且找到某一个位置
下子

// 获取棋盘上一共还剩多少步

moveCount = a_getMoveList(moveList);

// 根据难度等级降低查找的步数

moveCount -= levelType;

if(moveCount < 1)

```

```

{
    bestMove = moveList[0];

    return bestValue;
}

// 遍历棋盘上剩余的每一步，找到最优点
for (i = 0; i < moveCount; i++)
{
    // 拿到棋盘剩余棋格中的一个棋格

    Move curMove = moveList[i];

    // 假装下个棋子

    a_makeMove(curMove);

    // 假装下子完成后，调用 miniMax。

    // 调用完成后，获取返回值 2

    value = a_miniMaxsearch(depth - 1);

    // 把假装下子的棋格清空

    a_unMakeMove(curMove);

    if (player == COM)
    {
        if (value > bestValue)
        {
            bestValue = value;

            // 防止出现递归未完成时，也调用了最优点

            // 当递归 return 到最初开启递归那层时，赋值最优点

            if (depth == currentDepth)
            {
                bestMove = curMove;
            }
        }
    }

    else if (player == MAN)

```

```

    {
        if (value < bestValue)
        {
            bestValue = value;

            if (depth == currentDepth)
            {
                bestMove = curMove;
            }
        }
    }

    return bestValue;
}

```

// 电脑下子

```

int MainWindow::a_com_play() {
    // 可以不需要接收返回的最好值，因为已经直接改掉了 bestMove
    a_miniMaxsearch(currentDepth);

    board[bestMove.x][bestMove.y] = COM;

    int place = (bestMove.x * 3) + bestMove.y;

    for (int i = 0; i < btnList.size(); i++) {
        if (i == place) {
            btnList[i]->setStyleSheet("border-image: url(/fork.png);");
            btnList[i]->setEnabled(false);
        }
    }

    // 普通下子

    return 1;
}

```

```

int MainWindow::b_com_play()

```

```

{
    a_miniMaxsearch(currentDepth);

    board[bestMove.x][bestMove.y] = MAN;

    int place = (bestMove.x * 3) + bestMove.y;

    for (int i = 0; i < btnList.size(); i++) {

        if (i == place) {

            btnList[i]->setStyleSheet("border-image: url(:/ring.png);");

            btnList[i]->setEnabled(false);

        }

    }

    // 普通下子

    return 1;

}

```

```

// 人类下子

int MainWindow::a_man_play(QPushButton *btn)
{
    // 人类没有下子

    if(!btn)

        return -1;

    int x = btn->pos().x() / 100;

    int y = btn->pos().y() / 100;

    board[y][x] = MAN;

    btn->setStyleSheet("border-image: url(:/ring.png);");

    btn->setEnabled(false);

    // 普通下子

    return 1;

}

```

```

void MainWindow::on_reset_clicked()

{
    ending();
}

// =====machineVSmachine=====

//生成随机数，先手落子，这个是随机落子

int MainWindow::rand_X(int x)

{
    return rand()%x;
}

//先手落子

int MainWindow::randomInitial()

{
    int x = rand_X(3);
    int y = rand_X(3);
    cout<<board[x][y]<<endl;
    board[x][y] = COM;
    cout<<board[x][y]<<endl;
    int place = (x * 3) + y;
    //srand((int)time(0));//使用系统时间作为随机种子
    for (int i = 0; i < btnList.size(); i++) {
        if (i == place) {
            btnList[i]->setStyleSheet("border-image: url(:/fork.png);");
            btnList[i]->setEnabled(false);
        }
    }
}

int MainWindow::randomPlayer()

```

```

{
    int a;

    srand(time(0)); //初始化种子

    a = rand()%2; //产生 0、1 之间的随机数

    if(a==1){
        return a;
    }

    else if(a==0){
        a--;
        return a;
    }
}

```

```

void MainWindow::on_machineFight_clicked()

```

```

{
    cout<<"MACHINE VS MACHINE"<<endl;

    int i=0;

    randomInitial();

    // 深度减一

    currentDepth--;

    while(i<4){
        if(player == a_isWin()){
            QMessageBox::information(this, tr("井字棋"), tr("A 赢了! "), QMessageBox::Ok);

            return;
        }

        else if(a_isDraw()){
            return;
        }

        b_com_play();

        if(player == a_isWin()){
            QMessageBox::information(this, tr("井字棋"), tr("B 赢了! "), QMessageBox::Ok);

```

```

        return;
    }

    else if(a_isDraw()){

        return;

    }

    a_com_play();

    i++;

}

if(COM == a_isWin()){

    QMessageBox::information(this, tr("井字棋"), tr("A 赢了! "), QMessageBox::Ok);

}

else if(MAN == a_isWin()){

    QMessageBox::information(this, tr("井字棋"), tr("B 赢了! "), QMessageBox::Ok);

}

else{

    QMessageBox::information(this, tr("井字棋"), tr("平局! "), QMessageBox::Ok);

}

}

```

(5) 测试

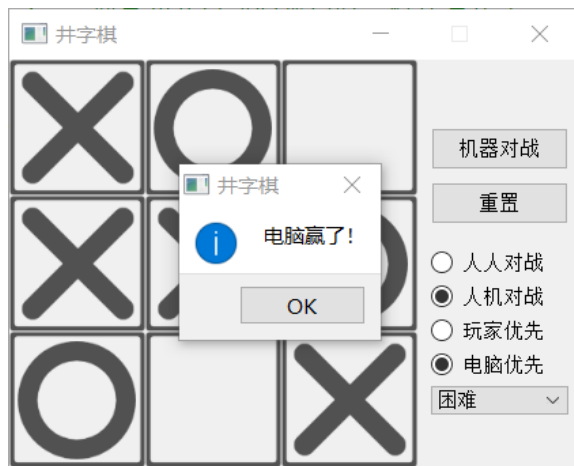
通过对不同数据进行测试，程序可以正常运行，并且可以与玩家正常进行井字棋游戏，经过对结果的检查，认为算法运行正确无误。

3.4 运行结果及分析

(1) 运行结果

① 人机对战运行结果





② 机器自我对战运行结果



(2) 分析

在人机博弈中，双方回合制地进行走棋，己方考虑当自己在所有可行的走法中做出某一特定选择后，对方可能会采取的走法，从而选择最有利于自己的走法。这种对弈过程就构成了一颗博弈树，双方在博弈树中不断搜索，选择对自己最为有利的子节点走棋。在搜索的过程中，将取极大值的一方称为 \max ，取极小值的一方称为 \min 。 \max 总是会选择价值最大的子节点走棋，而 \min 则相反。这就是极小化极大算法的核心思想。

极小化极大算法最大的缺点就是会造成数据冗余，而这种冗余有两种情况：①极大值冗余；②极小值冗余。相对应地，**alpha** 剪枝用来解决极大值冗余问题，**beta** 剪枝则用来解决极小值冗余问题，这就构成了完整的 **Alpha-beta** 剪枝算法。

Alpha-beta 剪枝本质是 **alpha** 剪枝和 **beta** 剪枝的结合，这两种剪枝的发生条件不同，因此在博弈中总是首先需要区分取极小值和取极大值方，这在一定程度上让算法的效率打了折扣。在具体的博弈中，结合博弈的特定规则进行优化，比如说，将一些先验知识 (**prior knowledge**) 纳入剪枝条件中，这种基于具体应用的优化将是 **alpha-beta** 剪枝的重要发展方向。

3.5 小结

这道题目让我重新复习了 $\alpha - \beta$ 剪枝算法，同时在复习的过程中初步了解了一些井字棋的实现方式，算法的本身很重要，但是同样不能忽略研究问题，把理论结合到实际应用的方法， $\alpha - \beta$ 剪枝算法一定是人工智能这门课中最精彩的部分之一。

总之，本次课设让我有了提高，编程过程中也有难题，不过在自己查阅资料，和老师的帮助下走都得以解决。再次感谢老师的帮助。

3.6 参考文献

- [1] <https://www.youtube.com/watch?v=xBXHtz4Gbdo>
- [2] https://www.youtube.com/watch?v=_i-lZcbWkps
- [3] <https://www.youtube.com/watch?v=l-hh51ncgDI>
- [4] <https://zhuanlan.zhihu.com/p/65108398>
- [5] <https://blog.csdn.net/baixiaozhe/article/details/51872495>
- [6] https://blog.csdn.net/qq_31615919/article/details/79681063
- [7] <https://www.cnblogs.com/tk55/articles/6012314.html>
- [8] <https://recomm.cnblogs.com/blogpost/4100059?page=3>
- [9] <https://www.jiqizhixin.com/graph/technologies/56dbb21e-c3f9-4e06-b16a-2e28f25b26c8>
- [10] <https://juejin.im/entry/6844903474887393287>