中山大学数据科学与计算机学院

计算机科学与技术专业-人工智能

本科生实验报告

（2018-2019 学年秋季学期）

课程名称：**Artificial Intelligence**

教学班级	计科 2 班	专业（方向）	计算机科学与技术
学号	16337341	姓名	朱志儒

实验题目

约束满足性问题

实验内容

· 算法原理

1) Backtracking（回溯）算法

回溯算法类似枚举的搜索尝试过程，它就是在搜索尝试过程中寻找问题的解，当发现不满足约束时，就回溯返回，然后尝试新的路径。

在包含问题的所有解的解空间树中，按照深度优先的策略，从根节点出发深度优先探索空间树。当探索到某个节点时，判断该节点是否包含问题的解，如果包含，就从该节点出发深度优先探索其子节点，如果不包含，则逐层回溯至其祖先节点。如果使用回溯算法求解问

题的所有解，则需要探索根节点的所有子树。如果使用回溯算法求任一解时，则只要找到问题的解即可结束搜索。

算法步骤：

a) 定义一个解空间，它包含问题的所有解；

b) 将解空间组织成适合搜索算法的形式；

c) 使用深度优先搜索算法探索解空间；

d) 在搜索过程中使用剪枝函数避免无效搜索。

2) Forward-checking（向前检验）算法

向前检验算法是回溯算法的扩展，在向前检验算法中，每次为一个变量赋值后将影响之后变量的取值空间，即从它们的取值空间中删除不满足约束的值。如果某个变量的取值空间为空集，那么这就说明当前变量的取值是不可能得到解，则可以提前回溯。

算法步骤：

a) 选取一个未赋值的变量，将其标记为已赋值，进入步骤 b），如果不存在未赋值的变量，则说明找到问题的解，结束算法；

b) 从该变量的取值范围中，选取一个值赋予该变量，进入步骤 c），如果遍历整个取值范围都未找到解，则将该变量标记为为赋值，返回步骤 a）；

c) 检查所有可检验的约束，对于所有未赋值的变量，删除它们取值范围中不满足约束的值；

d) 如果存在某个变量的取值范围是空集，那么放弃 b）中的取值，恢复 c）中所有删除的值，返回步骤 b）；

e) 如果不存在取值范围是空集的变量，则执行步骤 a）。

· 流程图&伪代码

1) Backtracking（回溯）算法

```
function Backtracking(csp)
    return Recursive_Backtracking({}, csp)

function Recursive_Backtracking(assignment, csp)
    if assignment is complete:
        return assignment
    var := select_unassigned_variable(variables[csp], assignment, csp)
    for value in order_domain_values(var, assignment, csp):
        assignment.append({var = value})
        result := Recursive_Backtracking(assignment, csp)
        if result != failure:
            return result
        assignment.remove({var = value})
    return failure
```

2) Forward-checking（向前检验）算法

```
function FCCheck(C, x)
    for value in Domain[x]:
        if not_satisfied_constraint(C):
            Domain[x].remove(value)
    if Domain[x] is empty:
        return true
    return false

function Forward_checking(level)
    if unassigned_variables is empty:
        return true
    v := pick_an_unassigned_variable(unassigned_variables)
    unassigned_variables.remove(v)
    for value in Domain[v]:
        solution[v] := value
        DWO = false
        for Var in unassigned_variables:
            if (FCCheck(C, Var)):
                DWO = true
                break
        if not DWO:
            FC(level + 1)
```

```
        restore_changed_domains()
    unassigned_variables.append(v)
    return false
```

· 关键代码

约束性检测：

```
bool constraint_check(int board[30][30], int x, int y, int n) {
    //约束性检测，判断纵行和斜线上是否存在皇后
    int directions[3][2] = { {-1, -1}, {-1, 0}, {-1, 1} };
    for (int i = 0; i < 3; ++i)
        for (int nx = x, ny = y; nx < n && ny < n && nx >= 0 && ny >= 0; nx
+= directions[i][0], ny += directions[i][1])
            if (board[nx][ny])
                return false;
    return true;
}
```

Backtracking（回溯）算法：

```
bool backtracking(int board[30][30], int level, int n) {
    if (level == n)
        //找到一个解，结束搜索
        return true;
    for (int i = 0; i < n; ++i)
        if (constraint_check(board, level, i, n)) {
            //满足约束条件，将皇后放置到该位置
            board[level][i] = 1;
            if (backtracking(board, level + 1, n))
                //深度优先搜索下一层位置
                return true;
            else
                //下层返回 false，表明不存在解，则将该位置的皇后移除
                board[level][i] = 0;
        }
    //未找到本层放置皇后的位置，返回 false
    return false;
}
```

FCCheck 函数用于删除或恢复未赋值变量的取值范围中不满足约束的值：

```
bool FCCheck(int domain[30][30], int level, int y, int n, bool recovery) {
    int directions[3][2] = { {1, -1}, {1, 0}, {1, 1} };
    for (int j = 0; j < 3; ++j)
        for (int nx = level + directions[j][0], ny = y + directions[j][1]; nx < n && ny < n && ny >= 0; nx += directions[j][0], ny += directions[j][1]) {
            if (!recovery)
                //在 domain 矩阵的相应位置+1，表示在取值范围中删除该值
                domain[nx][ny]++;
            else
                //在 domain 矩阵的相应位置-1，表值在取值范围中恢复该值
                domain[nx][ny]--;
        }
    if (recovery)
        //如果是恢复过程，则不需检测是否 DWO
        return true;
    for (int i = 0; i < n; ++i) {
        bool empty = true;
        for (int j = 0; j < n; ++j)
            if (domain[i][j] == 0) {
                empty = false;
                break;
            }
        if (empty)
            //存在某个变量的取值范围为空集，返回 DWO
            return false;
    }
    //不存在取值范围为空集的变量
    return true;
}
```

Forward-checking（向前检验）算法

```
void forwardchecking(int board[30][30], int n) {
    //声明 domain 矩阵，并将所有位置的值初始化为 0
    int domain[30][30];
    memset(domain, 0, sizeof(domain));
    //运行向前检测算法
    FC(board, domain, 0, n);
}
bool FC(int board[30][30], int domain[30][30], int level, int n) {
    if (level == n)
        //找到一个解，结束搜索
        return true;
```

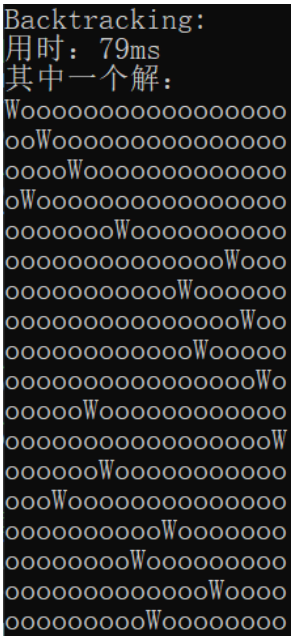
```
    for (int i = 0; i < n; ++i) {
        if (domain[level][i] == 0) {
            //如果 domain 矩阵中相应位置的上的值为 0，则可以放置皇后
            board[level][i] = 1;
            if (FCCheck(domain, level, i, n, false))
                //删除未赋值变量的取值范围中不满足约束的值，并检测 DWO
                if (FC(board, domain, level + 1, n))
                    //未检测到 DWO，则深度优先搜索下一层位置
                    return true;
            //检测到 DWO，则将该位置的皇后移除
            board[level][i] = 0;
            //复原 domain 矩阵
            FCCheck(domain, level, i, n, true);
        }
    }
    //未找到本层放置皇后的位置，返回 false
    return false;
}
```

实验结果及分析

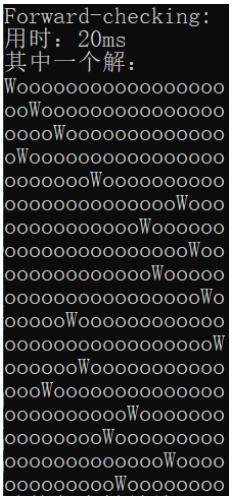
· 实验结果展示

1) 求解 18 皇后问题：

Backtracking（回溯）算法，用时 79ms：

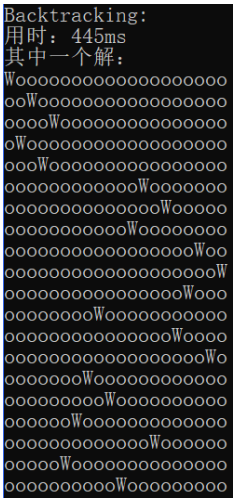


Forward-checking（向前检验）算法，用时 20ms：

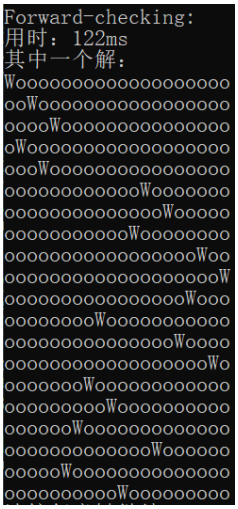


2)　求解 20 皇后问题：

Backtracking（回溯）算法，用时 445ms：

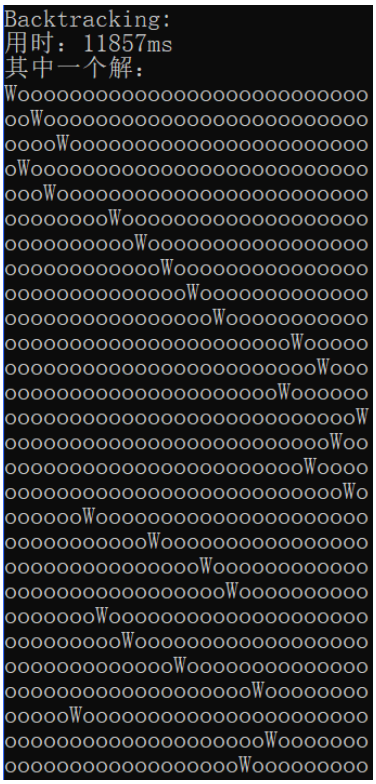


Forward-checking（向前检验）算法，用时 122ms：

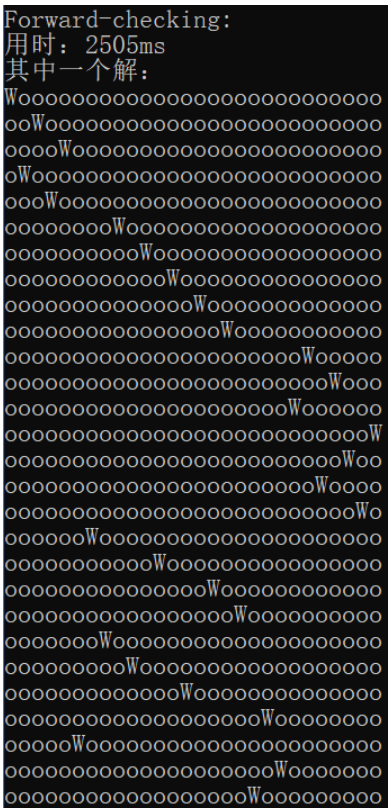


3) 求解 28 皇后问题：

Backtracking（回溯）算法，用时 11857ms：



Forward-checking（向前检验）算法，用时 2505ms：



· 评测指标展示

对于 N 皇后问题，从实验结果展示中，我们可得到：

	Backtracking 算法	Forward-checking 算法
18 皇后	79ms	20ms
20 皇后	445ms	122ms
28 皇后	11857ms	2505ms

从上表中，我们可以看出 Forward-checking（向前检验）算法比 Backtracking（回溯）算法效率较高。N 越大，FC 比 BT 的效率越高，因为 FC 算法中探索的节点比 BT 的少，每当给一个变量赋值时，将减小其他变量的取值范围，从而减少探索的节点数，如果出现某个变量的取值范围为空集，则将提前回溯，而不像 BT 算法继续探索。