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）

学　　号：16337113

姓　　名：劳马东

教学班级：教务 2 班

专　　业：超算

一、实验题目

使用 backtracking 和 forward-checking 求解 N 皇后问题，要求：

- 报告中需要包含对两个算法的原理解释；
- 需要包含两种方法在性能上的对比和分析；

二、实验内容

（一）算法原理

1、 N 皇后问题的 CSP 定义

将 N 个皇后放置在 $N \times N$ 的棋盘上，使得对于任意两个皇后，它们不在同一行、不在同一列、不在同一对角线上。图1是 8 皇后问题的一个例子。

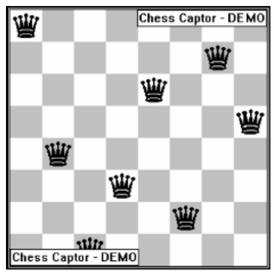

图 1. 8 皇后问题-Demo

（1） 变量集合

N 个皇后分别编号为 0、1、2...N-1。如图1的 8 个皇后从上到下分别编号为 0、1、2...6、7；

（2） 变量的值域

Q_i 表示第 i 个皇后的列号，则 $Q_i \in [0, N)$。如图1 每个皇后列号的取值范围为 $\{0, 1, 2, 3, 4, 5, 6, 7\}$；

（3） 约束集

- 列：$\forall\ i \neq j,\ Q_i \neq Q_j$
- 对角线：$\forall\ i \neq j,\ abs(Q_i - Q_j) \neq abs(i - j)$.

图1中 8 个皇后的列号分别为 $Q_0 = 0$、$Q_1 = 6$、$Q_2 = 4$、$Q_3 = 7$、$Q_4 = 1$、$Q_5 = 3$、$Q_6 = 5$、$Q_7 = 2$。

2、 backtracking

回溯思想简单而言就是“一条路走到黑，不撞南墙不回头”，在解决问题时不断尝试值域中的每个取值，往下递归直到可以确定不可能补全成正确解时放弃搜索，回溯并选择值域中的下一个值。算法的整体流程如下：

（1） 如果所有的变量都被赋值了，算法结束。

（2） 否则选择一个未赋值的变量 Q_i；

（3） 在 Q_i 的值域中选择一个值 x，测试 $Q_i = x$ 是否满足约束集，满足则回到步骤（1）。如果 Q_i 的所有取值都不满足约束，以搜索失败回溯。

图2是用 backtracking 算法求解 4 皇后问题的过程，黑色表示当前可放置（即满足约束）的位置，蓝色是不可放置的位置。大体过程如下：

- Q_0=0，无冲突；
- Q_1=0，与 Q_0 违反列约束；Q_1=1，与 Q_0 违反对角线约束；Q_1=2；
- Q_2=0，与 Q_0 违反列约束；Q_2=1 或 3，与 Q_1 违反对角线约束；Q_2=2，与 Q_1 违反列约束；无解，回溯到 Q_1；
- Q_1=3，无冲突；
- 以类似的过程继续，在右下角找到了一个解。

在第二层，Q0

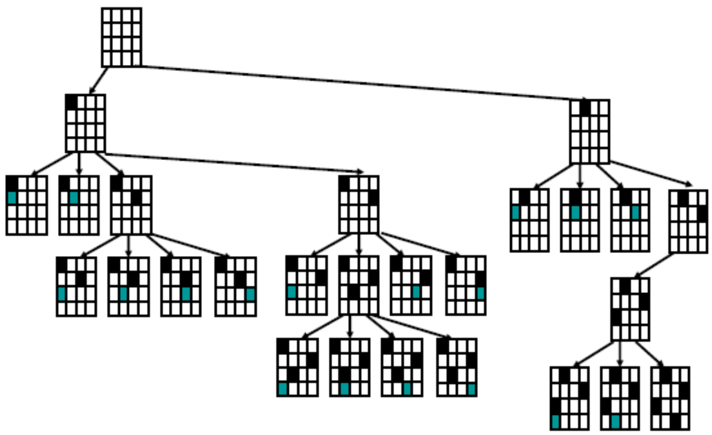

图 2. 4 皇后问题 backtracking 求解

算法 1 backtracking 搜索

输入: *Assigned*：N 个皇后的赋值状态，*Column*：N 个皇后的列号

function BACKTRACKING($Assigned, Column$)
    if all variables assigned **then**
        print all value of each variable
        return for more solutions **or exit** for only one solution
    else
        $V \leftarrow$ PickUnassignedVariable()
        $Assigned[V] \leftarrow$ TRUE
        for each $v \in$ Domain(V) **do**
            $Column[V] \leftarrow v$
            if CheckConstraint() == TRUE **then**
                backtracking($level + 1$)
            end if
        end for
        $Assigned[V] \leftarrow$ FALSE
        return
    end if
end function

3、 forward-checking

backtracking 的问题在于，（1）一个皇后的列号确定之后，不对其他皇后的值域做更新，实际上可以利用皇后之间的约束关系简化其他皇后的值域；（2）只有“撞了南墙才回头”，给所有皇后尽可能地赋值后，才知道是否找到解。

forward-checking 是一种约束传播（Constraint Propagation）算法，在算法的每一步都“向前看”，将当前变量的赋值结果“传播”到其他还没有赋值的变量，更新它们的值域，从而当某个变量不可取值时提前回溯。

算法2是该过程的伪代码,其中大部分与 backtracking 相同,增加了两个过程:一是当前变量赋值（$Column[V] = v$）后，更新未赋值皇后的可取值范围（$updateDomain$），并在回溯时恢复更新前的值域；二是判断某个被更新的值域是否为空，为空则说明 $Column$ 现在的取值方式找不到解，回溯。

算法 2 forward-checking 搜索

输入: $Assigned$：N 个皇后的赋值状态，$Column$：N 个皇后的列号, $CurDom$：N 个皇后当前值域

```
function FORWARD-CHECKING(Assigned, Column, CurDom)
    if all variables assigned then
        print all value of each variable
        return for more solutions or exit for only one solution
    else
        V ← PickUnassignedVariable()
        Assigned[V] ← TRUE
        for each v ∈ CurDom[V] do
            Column[V] ← v
            DWO ← updateDomain(CurDom, V, v, Assigned)
            if DWO happen then
                return DWO
            else
                forward-checking(Assigned, Column, CurDom)
            end if
            Restore(CurDomain, V, v, Assigned)
        end for
        Assigned[V] ← FALSE
    end if
end function
```

三、关键代码

1、 backtracking

```
def n_queens_backtracking(chess_board, row=0):
    n = chess_board.shape[1]
    if row == n:    # n个皇后都被赋值了
        print(board)
        return 1    # 找到一个解
    else:
        cnt = 0
        for i in range(n):    # 皇后列号Qk的值域是[0, n)
            if csp(chess_board, (row, i)):  # Qk=i满足约束
                # 将皇后防放置在棋盘上
                chess_board[row][i] = 1
                # 往下递归，选择第row+1个皇后
                cnt += n_queens_backtracking(chess_board, row + 1)
                # 回溯，恢复棋盘状态
                chess_board[row][i] = 0
        return cnt
```

2、 forward-checking 值域更新

```
for x in curdom_copy:
    restore = []     # 用于后面恢复值域
    ...
    for i in range(min_dom_index + 1, n):   # 所有未更新皇后在[min_dom_index + 1, n)
        other_queen_id, domain = domains[i]   # domains存了皇后id和它的列值域set
        restore.append(set())
        if x in domain:     # 同一列
            restore[-1].add(x)    # 被删除的元素放入restore
            domain.remove(x)
        for j in domain.copy():   # 同一对角线
            if abs(j - x) == abs(other_queen_id - queen_id):
                restore[-1].add(j)
                domain.remove(j)
        ...
```

3、 MRV 选择变量

```
for x in curdom_copy:
    ...
    min_dom_size = float('inf')
    min_dom_i = None
    # 找出被更新值域中，取值个数最少的一个
    for i in range(min_dom_index + 1, n):
        ...
        length = len(domain)
        if length < min_remain:
            min_dom_size = length
            min_dom_i = i
    # min_dom_size为0说明发生DWO
    if min_dom_i is not None and min_dom_size != 0:
        swap(domains, min_dom_index + 1, min_dom_i)
        cnt += n_queens_fc(board, index + 1, cnt + 1, domains)
```

4、 forward-checking 值域恢复

```
if min_dom_i is not None and min_dom_size != 0:
    swap(domains, min_dom_index + 1, min_dom_i)
for j, d in enumerate(restore):
    domains[min_dom_index + j + 1][1].update(d)
```

四、实验结果及分析

实验中测试了 10 皇后和 12 皇后问题，统计找出问题的所有解的运行时间，如表1。10 皇后问题解的个数为 724 个，使用 backtracking 算法时，需要 6.39 秒找出所有解，而使用 forward-checking 只需 0.23 秒，只有 backtracking 时间的 $\frac{1}{28}$ 左右；12 皇后问题解的个数为 14200，forward-checking 算法只有 backtracking 算法的 $\frac{3.70}{231.03} \approx \frac{1}{62.4}$，更突显了 forward-checking 算法的优越性。

n	解个数	backtracking 时间/s	forward-checking 时间/s
10	724	6.39	0.23
12	14200	231.03	3.70

表 1. 算法运行时间

从定性的角度分析，backtracking 算法每次递归都认为皇后的值域是 [0, N)，并且对每个值都运行一遍约束检测；而 forward-checking 算法每个皇后的值域最多有 N 个，并且由于约束传播更新了值域，会发生很多的剪枝，递归次数和层数都比 backtracking 少，自然就能在比 backtracking 短的时间内找出所有结果。

从定量的角度分析，在 backtracking 算法中，易得时间 $T(n) = n \times T(n-1) + n^2$，其中 $O(n^2)$ 是检查是否满足 csp 的时间，从这个递推式可以得出 backtracking 算法的时间复杂度是 $O(n!)$；forward-checking 算法的时间复杂度是 $O(nsd)$，其中 n 是变量个数，d 是初始值域的大小，s 是每个变量的最大约束个数，因此在 N 皇后问题中，时间复杂度就是 $O(n^2)$。对比二者，一个是阶乘级的时间复杂度，一个是多项式级的时间复杂度，相差近 $\frac{n!}{n^2} = \frac{(n-1)!}{n}$ 倍。