BT & FC 求解N皇后问题

（BT：backtracking，FC：forward-checking）

- **尝试使用几种方法分别实现BT和FC，因此BT和FC分别有几个代码版本。**
  - **BT**
    - **使用树结点存储结点状态，求出解的路径。**
    - **优化存储结构，使用更简化的结构存储变量和域值，减少变量的使用。**
    - **分别设计求单个解和所有解的代码。**
  - **FC**
    - **使用树结点存储结点状态，求出解的路径。**
    - **优化存储结构，使用更简化的结构存储变量和域值，减少变量的使用。**
    - **分别设计求单个解和所有解的代码。**
    - **使用MRV优化探索路径，提高运行效率。**

虽然各个版本不一样，但是算法思想是一致的，不过由于存储结构和优化方法不一样，因此编写多个版本，同时由于求所有解和求单个解的写法有部分不同，因此也分开写，设置求单个解是为了更好地比较不同N时程序运行的效率，这是因为N越大，其所有解是非常庞大，因此没有办法在常规时间跑出来，因此选择求单个解进行比较。

经过优化，BT和FC效率都有明显的提高，特别是对存储结构的优化，空间复杂度的减少极大地减少了时间复杂度，同时，经过优化的FC再加上MRV优化有了质的提高，没有加入MRV前，FC跑N=22时，已经需要20.1s，加入MRV后，FC跑N=22只需要0.001s，同时可以轻松处理更大的N，当N=100时，只需要0.1019 s，可见效率非常高。

（1）算法原理

- 使用N个变量表示N个皇后，同时每个变量的域为[1:N]，目的是达到让所有的变量都赋予域范围内的值，并且满足N皇后的约束的结果。

1. BT
   - 对于每次递归调用，首先判断当前是否还有变量没有被赋值，如果没有，则结束递归，否则从未赋值变量中取出一个变量，遍历该变量所有值域。
   - 对于每一个取值，进行满足约束性检查，判断该变量是否与已经赋值的变量在同一行或同一列或一条斜线上，满足约束性则证明该变量可以取当前的值，进行下一次递归，一直找到解。

2. FC
   - 进入每次递归，首先将未赋值的变量作一个备份（为向上回溯时恢复），从中取出一个未赋值的变量，遍历该变量的所有合法取值，检查取值是否满足约束。
   - 如果当前变量为最后一个未赋值的变量，则每个合法取值都是作为解的最后一个位置，此时结束递归，否则先对剩下未赋值的变量的值域作一个备份，再对其值域进行更新，针对当前取值，对各个未赋值的变量进行约束性检查，将不合法的取值从值域中去掉。

- 如果有未赋值的变量的值域变为空，则证明当前探索路径失败，给当前变量更换另一个取值，进行上面相同的操作，如果所有未赋值变量的值域均不为空，则进行下一步探索，直到找到解。

3. MRV

FC检索变量是按照变量下标顺序选择的，然而实际上有可能在后面的变量其值域比前面的值域少，因此优先探索值域少的变量，能够更快确定是否有解，因此使用MRV算法，每次选择新未赋值变量探索时，都选择当前值域最少的变量进行扩展。

(2) 伪代码

1. BT

```
Algorithm Backtracking BT
  input: level  #表示变量的序号
  output: True & False  #返回找到与否
    #如果没有变量都被赋值，展示结果，返回True
    if all variables are assigned do
        display solution
        return True
    #取出未赋值的变量
    get var from variables[level]
    #遍历变量每个取值
    for value in Domain[var]
        let variable_list[var] <- value
        constrant_ok <- True
        #进行约束性检查，不满足约束，不能继续探索
        for v in assigned variables do
            if abs(variable_list[var] - variable_list[v]) == abs(var - v) or var == v or
variable_list[var] == variable_list[v] then
            constrant_ok <- False
        #满足约束条件，进行下一层的探索
        if constrant_ok == True do
            if BT(level+1) == True do
                return False

    return False
```

2. FC

```
Algorithm ForwardChecking FC
  input: level  #表示变量的序号
  output: True & False  #返回找到与否

    backup the unassigned variables  #将未赋值的变量集合备份
    #取出未赋值的变量
    get var from variables[level]
    #遍历变量每个取值
```

```
    for value in Domain[var]
        let variable_list[var] <- value
        #如果没有变量都被赋值，展示结果，返回True
        if all variables are assigned do
            display solution
            return True
        constrant_ok <- True
        #进行约束性检查，不满足约束，不能继续探索
        for v in assigned variables do
            if abs(variable_list[var] - variable_list[v]) == abs(var - v) or var == v or
variable_list[var] == variable_list[v] then
            constrant_ok <- False
        #满足约束条件，进行下一层的探索
        if constrant_ok == True do
            backup Domain  #将所有变量的值域备份
            #对剩下未赋值的变量的值域进行检查
            for v in un_assigned variables do
                #遍历所有取值
                for d in Domain[v] do
                    #如果有取值不满足约束条件，则从值域中去除该取值
                    if abs(variable_list[var] - d) == abs(var - v) or var == v or
variable_list[var] == d then
                        remove d from Domain[v]
                 #如果有变量值域为空，说明不能赋值，不能继续探索下去
                 if Domain[v] is empty then
                    recover Domain #将所有变量的值域恢复
                    continue
            #如果找到解，返回的是True，那直接返回True给上一级
            if BT(level+1) == True do
                return False
            recover Domain #将所有变量的值域恢复
    #没有找到解，返回False
    return False
```

（3） 关键代码

1）BT

1. **使用树结点存储结点状态，求出解的路径。**

- 使用Queen类存储结点，保存访问的顺序状态。

```
class Queen():
    def __init__(self,x,y,N,parent):
        # self.domain = [x for x in range(N)]
        self.index = x  #变量的序号
        self.solution = (x,y) #变量在棋盘的位置
        self.parent = parent  #变量的上一个父结点
```

- 一开始对第一个变量进行初始化，定义好访问和未访问队列，调用BT函数，根据返回值判断是否有解。

```
#开始递归调用BT函数
def start_BT(N):
    for i in range(N):
        start_node = Queen(0,i,N,None) #初始化起点
        is_visited = [] #存储已经访问的结点
        is_visited.append(start_node)
        #存储未访问的变量，元素为变量序号
        un_visited = [x for x in range(N)]
        un_visited.remove(0) #0已经访问，去除0号元素
         #调用BT函数
        dst_node = BT(start_node,N,is_visited,un_visited)
        #如果找到解，直接返回
        if dst_node != None:
            return dst_node

    return None
```

- 每次进入BT函数，首先判断结束条件，取出未赋值变量进行探索。

```
def BT(node,N,is_visited,un_visited):
    #如果所有变量已经赋值，返回结果
    if len(is_visited) >= N:
        return node
    #取出未赋值的变量集合中第一个变量
    new_x = un_visited.pop(0)
```

- 对于每个取值，进行约束性检查，判断是否满足条件。

```
    #遍历所有取值范围
    for i in range(N):
        new_y = i
        #约束性检查
        constraintsOK = True
        for k in range(len(is_visited)):
            x = is_visited[k].solution[0]
            y = is_visited[k].solution[1]
            #如果不满足，则将条件设为False
            if x == new_x or y == new_y or (abs(x - new_x) == abs(y - new_y)):
                constraintsOK = False
                break
```

- 创建新的结点，如果满足递归结束条件，返回结果，否则递归调用BT函数，进入下一层，根据接收的结果判断是否找到解，并且对访问列表、未访问列表进行恢复。

```
        #如果满足约束性条件，进行下一步
        if constraintsOK == True:
            newNode = Queen(new_x,new_y,N,node) #创建新结点
            is_visited.append(newNode) #将该节点加入访问队列
            #如果所有变量已经被赋值，返回结果
            if len(is_visited) >= N:
                return newNode
            #递归调用下一层
            dstNode = BT(newNode,N,is_visited,un_visited)
            #找到解，返回上一级
            if dstNode != None:
                return dstNode
            #恢复访问队列
            is_visited.pop(-1)
    #恢复未访问队列
    un_visited.insert(0,new_x)
    return None
```

2. **优化存储结构，使用更简化的结构存储变量和域值，减少变量的使用。**

- 考虑到节省存储空间的需要和不需要保存路径状态，不再使用类结点存储结点，使用列表queen_list存储各个变量的状态，下标0至N-1为变量的序号，index表示当前变量的序号，从index=0开始，一层层遍历下去，当index>=N时，说明此时所有变量已经都赋值，探索完毕。

```
def BT(index,N,queen_list):
    #如果探索到最后一个变量，结束递归
    if index >= N:
        return True
    else:
        #遍历所有值域
        for k in range(N):
            queen_list[index] = k #为当前变量赋值
            #检查约束合法性
            if (valid(index,queen_list)):
                #进入下一层，如果返回True，说明找到解，继续返回上一级
                if BT(index+1,N,queen_list) == True:
                    return True
    #找不到解，返回False给上一级
    return False
```

- 约束合法性检查：算法思想相同，作了一些简化，考虑到k与index不会相同，因此去掉同一行的检测。

```
#约束性检查
def valid(index,queen_list):
    #检查前面已经赋值的变量
    for k in range(0,index):
        #如果在同一列，或者同一条斜线，返回False
        if abs(index - k) == abs(queen_list[index] - queen_list[k]) or queen_list[index] ==
queen_list[k]:
            return False
    #合法则返回True
    return True
```

3. **求出所有解。**

- 1和2都是基于求一个解设计，为了验证算法准确性，在2基础上稍作修改，找出所有解，从而证明N皇后的求解算法是正确的。
- 如果index>=N，则不会直接返回True，而是打印当前的解，同时使用全局变量num记录解数，queen_solution存储其中一个解，为了后面具体分析验证准确性，下面的BT递归调用也没有返回，函数结束也没有返回，因为如果返回，就会直接忽略其他可能存在的解，不返回有利于遍历所有解。

```
def BT(index,N,queen_list):
    #如果探索到最后一个变量，结束递归
    if index >= N:
        global num
        global queen_solution #记录其中一个解，方便后面具体分析
        num += 1
        if len(queen_solution) == 0:
            queen_solution = queen_list[:]
        #输出当前找到的解
        for i in range(N):
            print('(', i, ",", queen_list[i], ')', end='')
        print()
    else:
        #遍历所有值域
        for k in range(N):
            queen_list[index] = k #变量赋值
             #检查约束合法性
            if (valid(index,queen_list)):
                #进入下一层
                BT(index+1,N,queen_list)
```

2）FC

1. **使用树结点存储结点状态，求出解的路径。**

- 使用类存储结点的状态，记录路径的状态。

```
#类结点存储结点状态
class Queen():
    def __init__(self,x,y,N,parent):
        self.domain = [x for x in range(N)] #变量的域
        self.index = x #变量的序号
        self.solution = (x,y) #变量在棋盘的位置
        self.parent = parent #变量的上一个父母状态
```

- 初始化未赋值变量列表和已赋值变量列表，从第一个结点开始探索，调用FC函数。

```
def start_FC(N):
    #初始化为赋值的点的集合
    un_visited = []
    for i in range(N):
        newNode = Queen(i,0,N,None)
        un_visited.append(newNode)
    #从第一个结点开始
    for i in range(N):
        start_node = Queen(0,i,N,None) #初始化第一个结点
        is_visited = [] #初始化已赋值的列表
        #调用FC函数
        dst_node = FC(start_node,N,is_visited,un_visited)
        #找到解返回结果
        if dst_node != None:
            return dst_node
    return None
```

- 探索当前变量的各个取值，进行约束性检验，如果不符合，则放弃该取值。

```
def FC(node,N,is_visited,un_visited):

    new_x = node.index #变量的序号
    domain = node.domain #变量的值域
    #探索所有值域
    for i in domain:
        new_y = i
        #约束性条件检查
        constraintsOK = True
        for k in range(len(is_visited)):
            x = is_visited[k].solution[0]
            y = is_visited[k].solution[1]
            #如果在同一行、同一列或同一斜方向，不符合条件
            if x == new_x or y == new_y or (abs(x - new_x) == abs(y - new_y)):
                constraintsOK = False
                break
        if constraintsOK == False:

            continue
```

- 提前保存剩余变量的值域，进行FC_checking，更新其未赋值变量的值域。

```
        #将所有变量的值域保存备份
        save_domain = []
        for i in range(len(un_visited)):
            save_tmp = [x for x in un_visited[i].domain]
            save_domain.append(save_tmp)
        #检查更新剩余未访问的变量的值域
        DWOoccurred = False
        if FCcheck(new_x,new_y,un_visited) == "DWO":
            DWOoccurred = True
```

- 遍历各个剩余变量，对每个变量的取值进行约束性检查，如果不符合条件，去掉该取值，如果值域为空，则返回DWO，否则继续更新。

```
def FCcheck(new_x,new_y,un_visited):
    length = len(un_visited)
    #遍历剩余变量
    for i in range(new_x+1,length):
        node = un_visited[i]
        new_domain = []
        #遍历各种取值
        for y in node.domain:
            x = node.index
            #如果不符合条件，则去除该取值
            if x == new_x or y == new_y or (abs(x - new_x) == abs(y - new_y)):
                continue
            else:
                new_domain.append(y)
        #如果值域为空，说明不能再探索，返回DWO
        if len(new_domain) == 0:
            return "DWO"
        #更新值域
        node.domain = new_domain
    #更新成功
    return True
```

- 创建新的结点，判断是否所有变量已经赋值，如果是就结束递归，否则继续下一层调用，根据返回的结果判断是否结束函数，回溯时恢复访问的列表和剩余变量的值域。

```
        if DWOoccurred == False:
            node.solution = (new_x,new_y) #确定变量的位置
            is_visited.append(node) #添加变量到已访问的列表中

            #如果变量已经全部赋值，返回结果
```

```
            if len(is_visited) >= N:
                return node
            #要探索的新结点
            newNode = un_visited[new_x+1]
            newNode.parent = node
            #递归调用下一层
            dstNode = FC(newNode,N,is_visited,un_visited)
            #找到解直接返回上一级
            if dstNode != None:
                return dstNode
            #恢复访问的列表
            is_visited.pop(-1)
        #恢复未访问的变量之前的值域
        for i in range(len(save_domain)):
            un_visited[i].domain = [x for x in save_domain[i]]
    return None
```

2. **优化存储结构，使用更简化的结构存储变量和域值，减少变量的使用。**

- 用二维列表存储变量的值域和一维列表存储未赋值的变量，简化树结构存储结点，减少空间开销。

```
def start_FC(N):
    #二维列表表示变量的值域
    domain = []
    for i in range(N):
        domain.append([x for x in range(N)])
    #一维列表表示未赋值的变量
    un_assigned = [x for x in range(N)]
    solution = [0]*N
    #FC函数
    result = FC(un_assigned,domain,solution,N,0)

    if result == True:
        return solution
    return False
```

- 在探索变量的各个取值前，先将未赋值变量的列表做一个备份，简化后面的恢复操作，同理值域也是如此，可以节省不少时间开销。

```
def FC(un_assigned,domain,solution,N,index):
    #另开一个未赋值变量的副本，简化后面恢复的步骤
    cur_un_assigned = un_assigned[:]
    #将准备赋值的变量移出
    cur_un_assigned.remove(index)
    #遍历所有取值
    for value in domain[index]:
        solution[index] = value #赋值

        #如果当前所有变量已经访问完，则结束探索
```

```
        if len(cur_un_assigned) <= 0:
            return True
        #备份值域，简化后面恢复的步骤
        cur_domain = []
        for k in range(N):
            tmp = domain[k][:]
            cur_domain.append(tmp)
        #FC—check检查
        if FCcheck(index,value,cur_un_assigned,cur_domain) == "DWO":
            continue
        #调用下一层FC
        result = FC(cur_un_assigned,cur_domain,solution,N,index+1)
        if result != False:
            return result

    return False
```

- 由于各个变量的序号不同，因此肯定不在同一行，因此去掉行检测。

```
def FCcheck(index,value,cur_un_assigned,cur_domain):
    #遍历所有未赋值的变量
    for x in cur_un_assigned:
        new_domain = []
        #遍历各个取值
        for y in cur_domain[x]:
            #不符合条件，去掉该取值
            if y == value or (abs(x - index) == abs(y - value)):
                continue
            else:
                new_domain.append(y)
        #如果值域为空，返回DWO
        if len(new_domain) == 0:
            return "DWO"
        #更新值域
        cur_domain[x] = new_domain
    #更新成功
    return True
```

3. **使用MRV优化探索路径，提高运行效率。**
    - 在探索下一层前，找出剩余变量合法取值个数最少的变量，优先对该变量进行探索。

```
        min_index = cur_un_assigned[0] #存储当前取值最少的变量序号
        min_length = len(cur_domain[min_index]) #存储当前取值最少的个数
        #遍历，寻找取值最少的变量序号
        for k in range(1,len(cur_un_assigned)):
            #如果取值更少，则直接更新值域长度、变量序号
            if len(cur_domain[cur_un_assigned[k]]) < min_length:
                min_length = len(cur_domain[cur_un_assigned[k]])
                min_index = cur_un_assigned[k]
        #将取值最少的变量序号作为参数传递
        result = FC(cur_un_assigned,cur_domain,solution,N,min_index)
```

4. **求出所有解。**

- 在2的基础上进行修改，找到解后，不强制返回结束递归，而是统计解的个数，输出当前的解，并且让函数继续运行，其余部分也一样，不强制结束，让其遍历完变量的所有合法取值。

```
        #结束探索的条件
        if len(cur_un_assigned) <= 0:
            global num #统计解个数
            global queen_solution #存储其中一个解
            num += 1
            #存储其中一个解，方便后面验证合法性
            if len(queen_solution) == 0:
                queen_solution = solution[:]
            #输出当前找到的解
            for i in range(N):
                print('(', i, ",", solution[i], ')', end='')
            print()
            continue
```

（4）创新点&优化

- 恢复性优化

  初始设计FC时，一开始是先用列表保存变量的值域和未赋值变量的列表，探索后再将其恢复，但是N比较大时，这样会降低程序效率，因此最后改为每次探索前，提前给值域和未赋值变量列表创建一个副本，用副本进行下一步的探索，这样就不用对其进行恢复，可以提高程序运行效率。

- FC不做约束性检查

  一开始FC是先有对新探索的变量与已赋值变量的约束性检查，然后再进行FC_check，后来发现进行FC_check后，下一次探索的变量其值域中每一个取值实际上都是满足约束性检查，所以如果再进行一次约束性检查在N比较大时会大大增加时间开销，因此在FC中去掉约束性检查，保留FC_check。

- 优化存储结构

最初BT和FC是使用树结构实现，存储结点的状态，并且记录了探索路径的次序，运行效率比较低，当N=8和10时，此时效率还是比较高，但是N=20时，BT需要15s左右，FC需要7s左右，N再增大，变得更加慢，时间增长速度呈指数级，到N=25时，在规定时间内，程序已经跑不出来；实际上在本题中不需要记录实际路径，我们最后得到的是一个解，因此只需要用一个列表存储每个变量的取值即可，所以我们不需要使用树结构，不需要跟踪其结点的父母，因此可以使用一个二维列表存储所有变量的值域，一个列表存储未赋值的变量，同时将之前版本的冗余实现进行简化，比如简化行检测、提前检测递归结束条件等，减少空间开销，从而提高了效率，经过简化，N=20时，FC只需要运行1.99s，N=25时，FC只需要运行0.55s，而未优化前FC是需要很长时间才能跑出。

- 使用MRV提高搜索效率

  虽然经过简化存储结构，FC的效率有了很大的提升，但是当N=30时，FC仍然要跑很久，因此再对FC进行优化，加入MRV，每次选择取值最小的变量优先探索，从而加快探索的效率，在此时，FC的运行效率有了质的提升，N=30时只需要0.006 s，甚至一直到N=100时，也只需要运行0.137 s，相比于之前的版本，FC_MRV有了不止一个数量级的提升。

（5）实验结果及分析

1）实验结果展示

1. BT

展示所有解的坐标，每一行表示一种解，对应N个坐标，表示皇后的位置，在下面展示总的解数，总运行时间，以及探索的总的访问状态数，为了验证正确性，输出其中一个解的坐标图。

N比较小时，展示全部的解数和其中具体的一个解，N比较大时，展示其中一个具体的解。

- N = 8

  8皇后的解数为92，探索状态数为2057，同时经验证，输出的结果正确。

```
Please input N = 8
Total solution:
( 0 , 0 )( 1 , 4 )( 2 , 7 )( 3 , 5 )( 4 , 2 )( 5 , 6 )( 6 , 1 )( 7 , 3 )
( 0 , 0 )( 1 , 5 )( 2 , 7 )( 3 , 2 )( 4 , 6 )( 5 , 3 )( 6 , 1 )( 7 , 4 )
( 0 , 0 )( 1 , 6 )( 2 , 3 )( 3 , 5 )( 4 , 7 )( 5 , 1 )( 6 , 4 )( 7 , 2 )
( 0 , 0 )( 1 , 6 )( 2 , 4 )( 3 , 7 )( 4 , 1 )( 5 , 3 )( 6 , 5 )( 7 , 2 )
( 0 , 1 )( 1 , 3 )( 2 , 5 )( 3 , 7 )( 4 , 2 )( 5 , 0 )( 6 , 6 )( 7 , 4 )
( 0 , 1 )( 1 , 4 )( 2 , 6 )( 3 , 0 )( 4 , 2 )( 5 , 7 )( 6 , 5 )( 7 , 3 )
...
( 0 , 6 )( 1 , 4 )( 2 , 2 )( 3 , 0 )( 4 , 5 )( 5 , 7 )( 6 , 1 )( 7 , 3 )
( 0 , 7 )( 1 , 1 )( 2 , 3 )( 3 , 0 )( 4 , 6 )( 5 , 4 )( 6 , 2 )( 7 , 5 )
( 0 , 7 )( 1 , 1 )( 2 , 4 )( 3 , 2 )( 4 , 0 )( 5 , 6 )( 6 , 3 )( 7 , 5 )
( 0 , 7 )( 1 , 2 )( 2 , 0 )( 3 , 5 )( 4 , 1 )( 5 , 4 )( 6 , 6 )( 7 , 3 )
( 0 , 7 )( 1 , 3 )( 2 , 0 )( 3 , 2 )( 4 , 5 )( 5 , 1 )( 6 , 6 )( 7 , 4 )
Total solution number:  92
Backtracking run time :  0.03468533728736059  s
visited state number:  2057
-----
One solution:

  X . . . . . .
```

```
  . . . . X . . .
  . . . . . . . X
  . . . . . X . .
  . . X . . . . .
  . . . . . . X .
  . X . . . . . .
  . . . X . . . .
```

- N = 10

  10皇后的解数为724，探索状态数为35539，其输出的结果验证正确。

```
Please input N = 10
Total solution:
( 0 , 0 )( 1 , 2 )( 2 , 5 )( 3 , 7 )( 4 , 9 )( 5 , 4 )( 6 , 8 )( 7 , 1 )( 8 , 3 )( 9 , 6 )
( 0 , 0 )( 1 , 2 )( 2 , 5 )( 3 , 8 )( 4 , 6 )( 5 , 9 )( 6 , 3 )( 7 , 1 )( 8 , 4 )( 9 , 7 )
( 0 , 0 )( 1 , 2 )( 2 , 5 )( 3 , 8 )( 4 , 6 )( 5 , 9 )( 6 , 3 )( 7 , 1 )( 8 , 7 )( 9 , 4 )
( 0 , 0 )( 1 , 2 )( 2 , 8 )( 3 , 6 )( 4 , 9 )( 5 , 3 )( 6 , 1 )( 7 , 4 )( 8 , 7 )( 9 , 5 )
( 0 , 0 )( 1 , 3 )( 2 , 5 )( 3 , 8 )( 4 , 2 )( 5 , 9 )( 6 , 7 )( 7 , 1 )( 8 , 4 )( 9 , 6 )
...
( 0 , 9 )( 1 , 7 )( 2 , 4 )( 3 , 1 )( 4 , 3 )( 5 , 0 )( 6 , 6 )( 7 , 8 )( 8 , 2 )( 9 , 5 )
( 0 , 9 )( 1 , 7 )( 2 , 4 )( 3 , 1 )( 4 , 3 )( 5 , 0 )( 6 , 6 )( 7 , 8 )( 8 , 5 )( 9 , 2 )
( 0 , 9 )( 1 , 7 )( 2 , 4 )( 3 , 2 )( 4 , 0 )( 5 , 5 )( 6 , 1 )( 7 , 8 )( 8 , 6 )( 9 , 3 )
Total solution number:  724
Backtracking run time :  0.932639943755548  s
visited state number:  35539
-----
One solution:
  X . . . . . . . .
  . . X . . . . . .
  . . . . . X . . .
  . . . . . . . X .
  . . . . . . . . X
  . . . . X . . . .
  . . . . . . . X .
  . X . . . . . . .
  . . . X . . . . .
  . . . . . . X . .
```

- N = 20

  N比较大时，由于解数比较多，因此结果展示的是求一个解的情况，探索状态数也是针对一个解情况。

```
Please input N = 20
( 0 , 0 )( 1 , 2 )( 2 , 4 )( 3 , 1 )( 4 , 3 )( 5 , 12 )( 6 , 14 )( 7 , 11 )( 8 , 17 )( 9 , 19 )(
10 , 16 )( 11 , 8 )( 12 , 15 )( 13 , 18 )( 14 , 7 )( 15 , 9 )( 16 , 6 )( 17 , 13 )( 18 , 5 )( 19
, 10 )
Backtracking run time :  13.155356041804248  s
visited state number:  199636
  X . . . . . . . . . . . . . . . . . . .
  . . X . . . . . . . . . . . . . . . . .
```

```
. . . . X . . . . . . . . . . . . . . .
. X . . . . . . . . . . . . . . . . . .
. . . X . . . . . . . . . . . . . . . .
. . . . . . . . . . . . X . . . . . . .
. . . . . . . . . . . . . . X . . . . .
. . . . . . . . . . . X . . . . . . . .
. . . . . . . . . . . . . . . . . . X . .
. . . . . . . . . . . . . . . . . . . . X
. . . . . . . . . . . . . . . . X . . .
. . . . . . . . X . . . . . . . . . . .
. . . . . . . . . . . . . . . . X . . . .
. . . . . . . . . . . . . . . . . . . X .
. . . . . . . X . . . . . . . . . . . .
. . . . . . . . . X . . . . . . . . . .
. . . . . . X . . . . . . . . . . . . .
. . . . . . . . . . . . . X . . . . . .
. . . . . X . . . . . . . . . . . . . .
. . . . . . . . . . X . . . . . . . . .
```

2. FC

N=8和N=10情况同BT一致。

- N = 8

  FC+MRV进行优化，其探索的结点数只有977，比BT 2057的一半还要少。

```
Please input N = 8
Total solution:
( 0 , 0 )( 1 , 4 )( 2 , 7 )( 3 , 5 )( 4 , 2 )( 5 , 6 )( 6 , 1 )( 7 , 3 )
( 0 , 0 )( 1 , 5 )( 2 , 7 )( 3 , 2 )( 4 , 6 )( 5 , 3 )( 6 , 1 )( 7 , 4 )
( 0 , 0 )( 1 , 6 )( 2 , 3 )( 3 , 5 )( 4 , 7 )( 5 , 1 )( 6 , 4 )( 7 , 2 )
...
( 0 , 7 )( 1 , 2 )( 2 , 0 )( 3 , 5 )( 4 , 1 )( 5 , 4 )( 6 , 6 )( 7 , 3 )
( 0 , 7 )( 1 , 3 )( 2 , 0 )( 3 , 2 )( 4 , 5 )( 5 , 1 )( 6 , 6 )( 7 , 4 )
Total solution number:  92
Forwardchecking run time :  0.015159873419776707  s
visited state number:  977
-----
One solution:
  X . . . . . .
  . . . . X . . .
  . . . . . . . X
  . . . . . X . .
  . . X . . . . .
  . . . . . . X .
  . X . . . . . .
  . . . X . . . .
```

- N = 10

  解数与BT的一致，探索状态数只有14029，比BT的35539效率提高一半以上。

```
Please input N = 10
Total solution:
( 0 , 0 )( 1 , 2 )( 2 , 5 )( 3 , 8 )( 4 , 6 )( 5 , 9 )( 6 , 3 )( 7 , 1 )( 8 , 4 )( 9 , 7 )
( 0 , 0 )( 1 , 2 )( 2 , 5 )( 3 , 8 )( 4 , 6 )( 5 , 9 )( 6 , 3 )( 7 , 1 )( 8 , 7 )( 9 , 4 )
...
( 0 , 9 )( 1 , 7 )( 2 , 4 )( 3 , 1 )( 4 , 3 )( 5 , 0 )( 6 , 6 )( 7 , 8 )( 8 , 2 )( 9 , 5 )
( 0 , 9 )( 1 , 7 )( 2 , 4 )( 3 , 1 )( 4 , 3 )( 5 , 0 )( 6 , 6 )( 7 , 8 )( 8 , 5 )( 9 , 2 )
Total solution number:  724
Forwardchecking run time :  0.20889021797368185  s
visited state number:  14029
-----
One solution:
  X . . . . . . . . .
  . . X . . . . . . .
  . . . . . X . . . .
  . . . . . . . . X .
  . . . . . X . . . .
  . . . . . . . . . X
  . . . X . . . . . .
  . X . . . . . . . .
  . . . . X . . . . .
  . . . . . . . X . .
```

- N = 50

  解数较多，输出其中一个解的皇后坐标表示和图表示，由此可见，虽然N=50时，BT需要跑很长时间，但是PC_MRV能够在0.066s内运行完毕，可见其优越性。

  对于求一个解的情况，即使N=50，FC_MRV也只需要探索1556个状态就能找到解。

```
Please input N = 50
( 0 , 0 )( 1 , 2 )( 2 , 4 )( 3 , 21 )( 4 , 40 )( 5 , 3 )( 6 , 33 )( 7 , 6 )( 8 , 32 )( 9 , 41 )(
10 , 48 )( 11 , 45 )( 12 , 5 )( 13 , 30 )( 14 , 35 )( 15 , 27 )( 16 , 7 )( 17 , 28 )( 18 , 34 )(
19 , 29 )( 20 , 26 )( 21 , 13 )( 22 , 8 )( 23 , 36 )( 24 , 31 )( 25 , 12 )( 26 , 46 )( 27 , 49 )
( 28 , 23 )( 29 , 9 )( 30 , 44 )( 31 , 39 )( 32 , 47 )( 33 , 38 )( 34 , 43 )( 35 , 1 )( 36 , 18
)( 37 , 10 )( 38 , 42 )( 39 , 14 )( 40 , 24 )( 41 , 37 )( 42 , 19 )( 43 , 22 )( 44 , 25 )( 45 ,
15 )( 46 , 11 )( 47 , 16 )( 48 , 20 )( 49 , 17 )
Forwardchecking run time :  0.06671628079785866  s
visited state number:  1556
  X................................................
  ..X.............................................
  ....X...........................................
  ..................X.............................
  ....................................X.........
  ...X............................................
  ..............................X.................
  ......X.........................................
  ............................X.................

  ....................................X.........
```

```
.........................................x.
.......................................x....
.....x..................................
.........................x..............
................................x.........
.....................x..................
.......x................................
........................x...............
..............................x.........
.........................x..............
.....................x..................
............x..............................
........x..................................
..............................x.........
.........................x..............
...........x..............................
.....................................x...
.....................................x
...................x......................
........x..................................
.....................................x.....
..................................x........
.....................................x..
..................................x.........
.....................................x......
x........................................
................x.........................
.........x................................
.....................................x......
............x..............................
....................x......................
..................................x.........
................x.........................
.....................x......................
.....................x.....................
...........x..............................
.........x................................
...............x..........................
.................x........................
...............x..........................
```

- N = 100

  使用FC_MRV跑出的结果，由于图太大，只展示皇后坐标，运行效率仍然比较高，只需要0.1 s左右。

  对于N=100，FC_MRV只需要探索163个状态就找到解。

```
Please input N = 100
( 0 , 0 )( 1 , 2 )( 2 , 4 )( 3 , 56 )( 4 , 58 )( 5 , 3 )( 6 , 63 )( 7 , 6 )( 8 , 57 )( 9 , 70 )(
10 , 80 )( 11 , 59 )( 12 , 5 )( 13 , 90 )( 14 , 81 )( 15 , 89 )( 16 , 7 )( 17 , 82 )( 18 , 76 )(
19 , 64 )( 20 , 72 )( 21 , 25 )( 22 , 8 )( 23 , 44 )( 24 , 36 )( 25 , 62 )( 26 , 65 )( 27 , 61 )
( 28 , 43 )( 29 , 9 )( 30 , 47 )( 31 , 53 )( 32 , 42 )( 33 , 68 )( 34 , 41 )( 35 , 46 )( 36 , 17
)( 37 , 10 )( 38 , 71 )( 39 , 67 )( 40 , 49 )( 41 , 55 )( 42 , 60 )( 43 , 35 )( 44 , 32 )( 45 ,
16 )( 46 , 11 )( 47 , 50 )( 48 , 99 )( 49 , 92 )( 50 , 96 )( 51 , 87 )( 52 , 34 )( 53 , 83 )( 54
, 77 )( 55 , 18 )( 56 , 12 )( 57 , 98 )( 58 , 66 )( 59 , 75 )( 60 , 91 )( 61 , 74 )( 62 , 86 )(
63 , 95 )( 64 , 93 )( 65 , 84 )( 66 , 19 )( 67 , 13 )( 68 , 94 )( 69 , 31 )( 70 , 97 )( 71 , 54
)( 72 , 39 )( 73 , 79 )( 74 , 48 )( 75 , 51 )( 76 , 45 )( 77 , 52 )( 78 , 20 )( 79 , 14 )( 80 ,
40 )( 81 , 1 )( 82 , 26 )( 83 , 33 )( 84 , 21 )( 85 , 69 )( 86 , 73 )( 87 , 28 )( 88 , 24 )( 89
, 29 )( 90 , 37 )( 91 , 85 )( 92 , 15 )( 93 , 78 )( 94 , 23 )( 95 , 38 )( 96 , 27 )( 97 , 22 )(
98 , 30 )( 99 , 88 )
Forwardchecking run time :  0.10351607116493904  s
visited state number:  163
```

结果显示BT和FC对N=8和N=10时的解个数一致，并且分别验证了其解的正确性，证明算法正确，同时通过时间比较，FC的效率高于BT，带MRV的FC的效率又远高于FC。

2）评测指标展示

由于上面展示已经展示N皇后不同N的全部的解数，而N越大，全部求解耗费时间较大，为了更好地比较BT和FC，下面展示的均为求一个解的比较。

- 展示各个版本的BT和FC求解不同的N皇后问题的运行时间。

A 树形结构BT

B 树形结构FC

C 树形结构FC + MRV

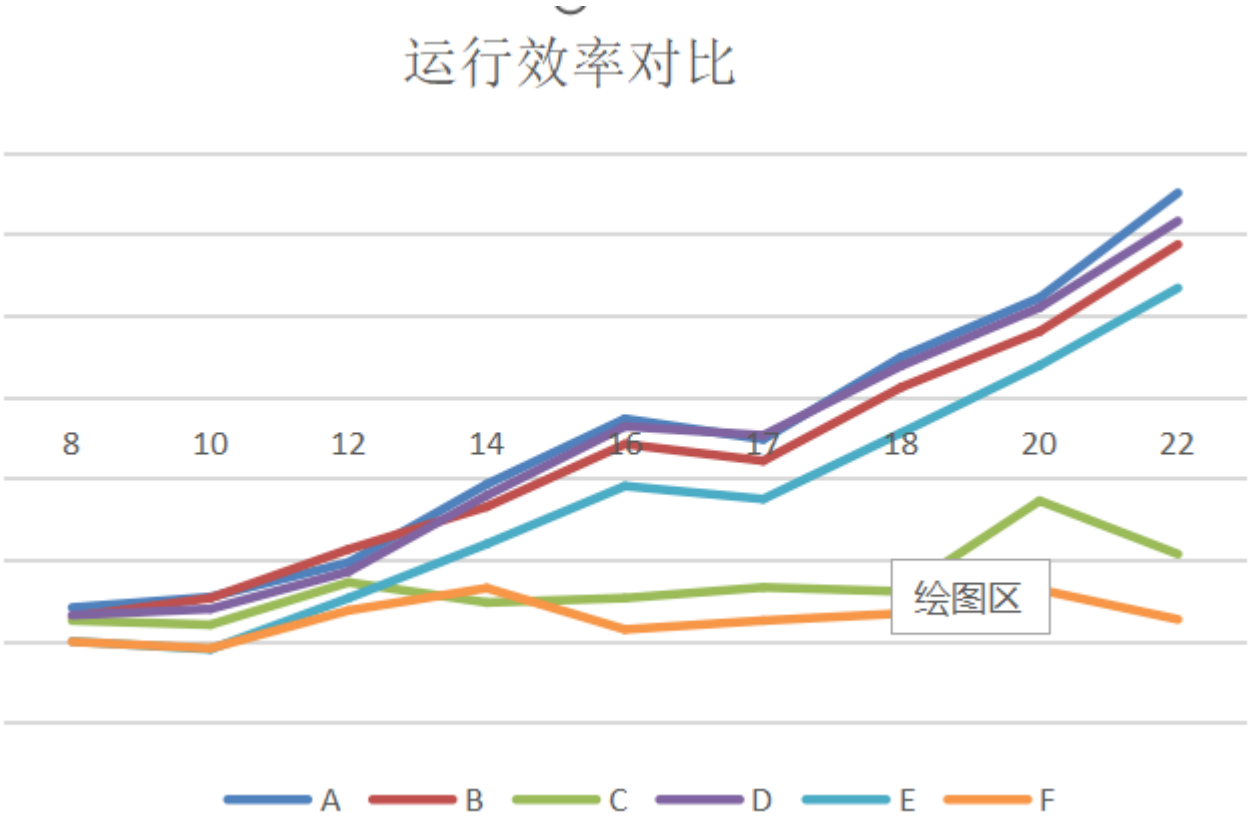
D 存储结构优化的BT

E 存储结构优化的FC

F 存储结构优化的FC + MRV

单位：s

N	8	10	12	14	16	17	18	20	22	30	40
A	0.0026	0.0035	0.0093	0.0849	0.539	0.3019	3.103	16.538	319.39	None	None
B	0.0020	0.0034	0.0135	0.0450	0.2634	0.1644	1.3192	6.3810	74.3440	None	None
C	0.0018	0.0016	0.0053	0.0030	0.0034	0.0046	0.0041	0.0531	0.0118	3.042	None
D	0.0021	0.0025	0.0072	0.0619	0.4353	0.3378	2.421	12.548	144.089	None	None
E	0.0010	0.0008	0.0034	0.0157	0.0806	0.0558	0.3640	2.4399	21.7171	None	None
F	0.00099	0.00082	0.0024	0.0045	0.0014	0.0018	0.0022	0.0043	0.00186	0.0075	0.0108



总体上，N越大，状态空间更复杂，其程序运行时间越来越多。

首先对A B C 和D E F进行比较，A B C是采用树结构，存储状态比较麻烦，经过状态访问优化和数据处理后，D E F在运行时间上有了明显的提升，特别是N=20时，可以发现优化后的BT和FC效率都有明显的提高，分别从16、6降低到12，2。

对BT和FC，由于BT采用一层层探索，基本上每个未赋值的变量都要遍历所有的取值，N比较大时，复杂度更高，因此时间效率降低，而FC每次给一个变量赋值，都将剩余变量的值域更新一次，减少不必要的探索，因此可以提高效率，因此当N=22时，BT需要144s，而FC只需要21s。

无论是BT，还是FC，N越大，探索的状态数目会急剧增加，因此当N到30时，基本跑不出结果，因此此时MRV起到很好的效果，在FC基础上，每次优先选取值域数少的变量进行探索，可以提早一步确定是否有解，从而大大缩短了探索的分支，因此N=30时，可以见到C和F都能跑出，同时经过存储优化的FC+MRV在N=40时是仍然能够以0.01s的效率跑出。

- 由于N比较大时，BT和不加MRV的FC基本跑不出，因此对于比较大的N，选择版本F进行测试。

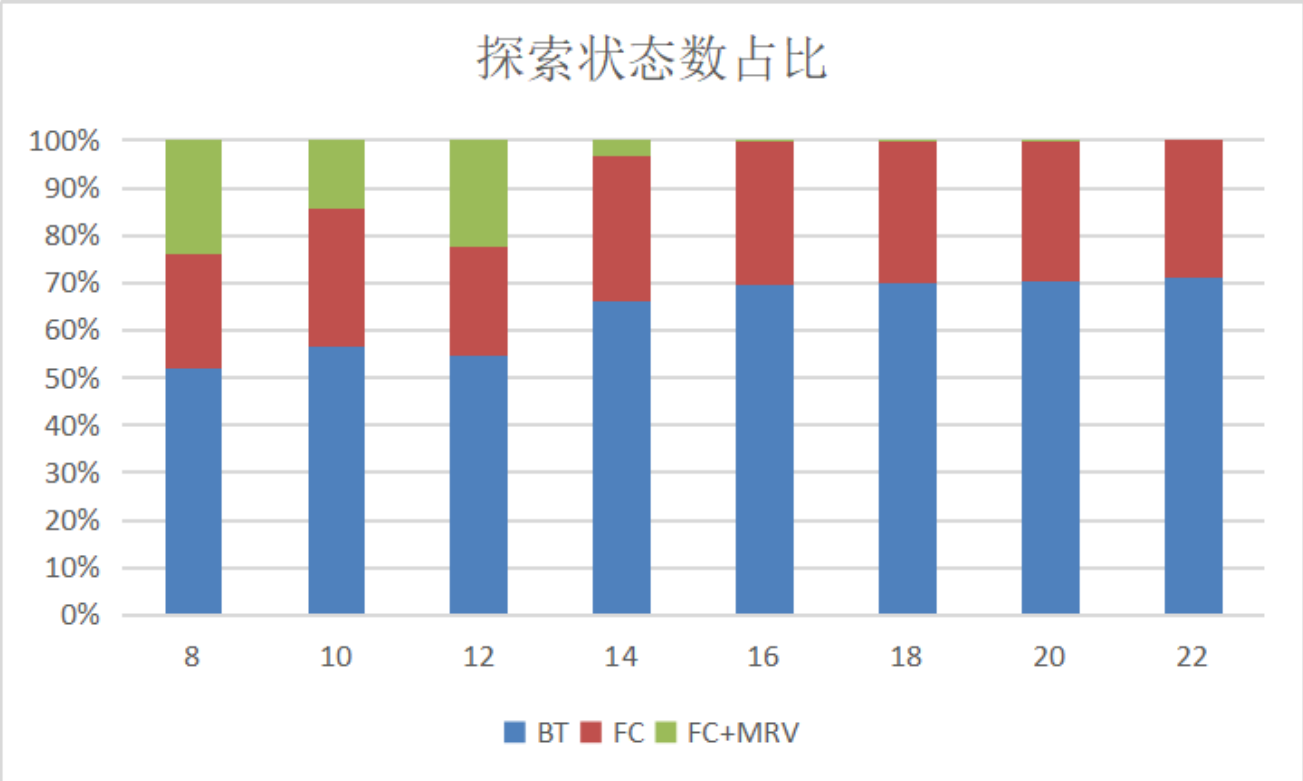
N	50	60	70	80	90	100	105
FC+MRV	0.0668	0.074	0.0363	0.0821	0.1511	0.1050	12.8423

可以见到，经过存储优化，FC+MRV可以处理较大N皇后的问题，提前缩短不必要的分支，减少没找到解的分支，明显提高效率，当N=100时，只需要0.1s就可以找到解，此后N继续增大，需要更多的时间，说明此后的状态访问次数更多。

- 比较BT、FC、FC_MRV求解不同N皇后的访问状态数，选择版本D、E和F比对。

  *只求一个解比较

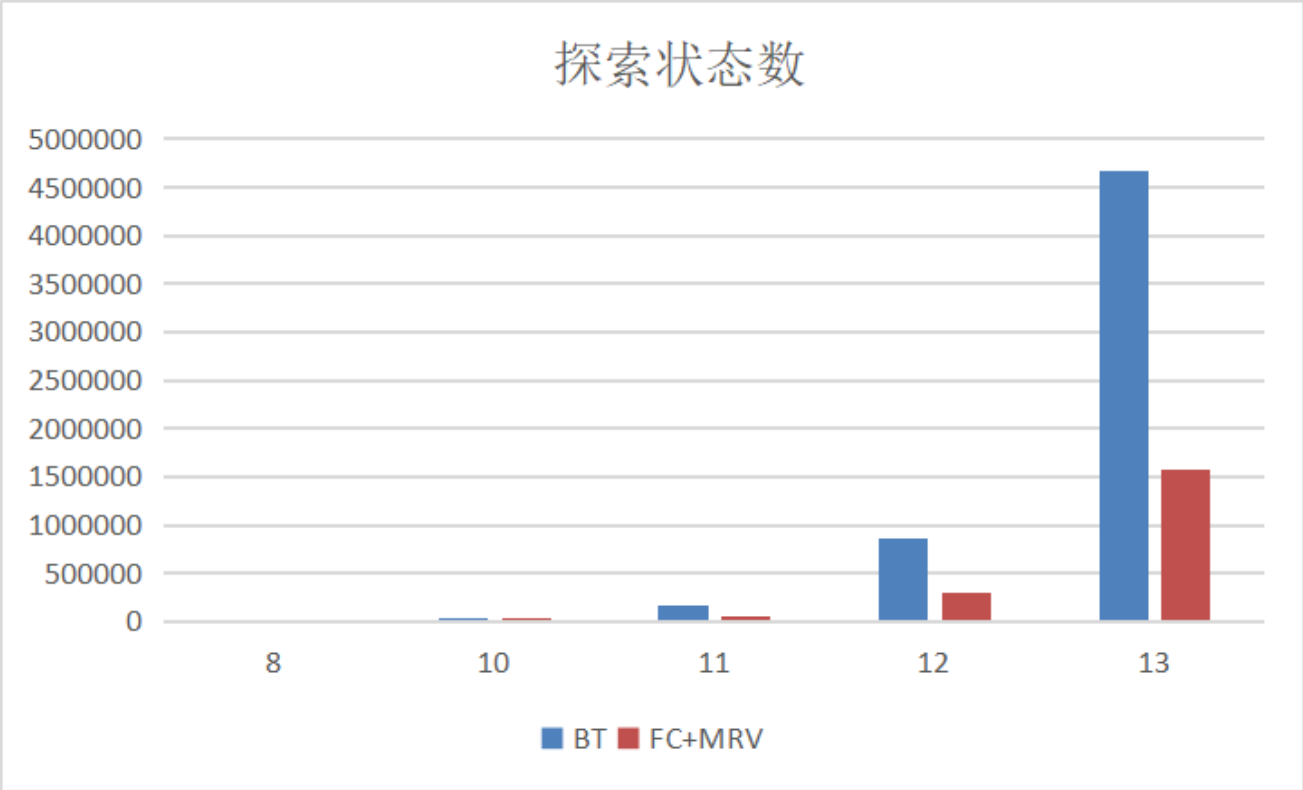
N	8	10	12	14	16	18	20	22
BT	114	103	262	1900	10053	41300	199636	1737189
FC	53	53	111	875	4378	17625	84400	703200
FC+MRV	52	26	107	91	37	44	112	23



图表表示对每个N皇后问题，三种策略探索状态数目的对比，当N较小时，FC_MRV占比较少，但是差距不是很大，当N到22时，此时FC_MRV探索的状态数目占比可以忽略不计，只剩下FC和BT，并且BT占主要部分，说明BT搜索的低效性。

* 求全部解比较

N	8	10	11	12	13
BT	2057	35539	166926	856189	4674890
FC+MRV	977	14029	61464	300747	1565896



由图可知，BT搜索全部解的状态数比FC+MRV高出一个数量级，并且当N=13时，差距已经非常明显。

总体上，当N比较小时，探索结点数：FC+MRV < FC < BT，但差距不是很明显；当N增大时，探索结点数FC+MRV << FC << BT，此时FC+MRV远小于FC，FC远小于BT。

综上比较，BT一层层对每个变量所有取值遍历，因此访问状态数比较多，当N比较大时，访问状态数剧增，从而影响程序运行时间；而FC提前对剩余的变量的取值进行筛选，将不符合约束性条件的取值去除，从而不用探索这样的分支，减少不必要的时间开销；在FC基础上加上MRV，在探索过程中，许多分支是找不到解的，因此选择取值少的优先探索能够更快判断是否有解，从而避免探索更多的分支。

总结

1. Forwardchecking在Backtracking基础上，提前对剩余没有赋值的变量进行检测，进而更新其值域，目的是提前删除不符合条件的值，这能够有效地提高探索的效率，提前预知探索的路径是否能够找到最终解，有效地减少探索的次数，同时节省大量的存储空间。
2. Forwardchecking在处理N大于30时仍然有局限性，这是由于搜索是按照变量序号顺序探索，N越大时，结点数越多，因此需要更多时间，而MRV加入可以使得Forwardchecking优先考虑取值选择较少的变量进行探索，有效地缩短探索的分支，从而运行时间经过实验验证有了质的提高。
3. 通过Backtracking和Forwardchecking的学习和性能比对可以得知优化搜索分支的重要性，尤其在处理结点比较多的问题上，同时对存储结构进行优化也是一个重要策略，程序运行过程中，空间占用过多往往导致更高的时间开销，将存储结构进行简化，尽量避免过多的存储开销，减少内存的开销，往往能够明显提高程序效率。