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：

1. 定义一个解空间，它包含问题的所有解；
2. 将解空间组织成适合搜索算法的形式；
3. 使用深度优先搜索算法探索解空间；
4. 在搜索过程中使用剪枝函数避免无效搜索。
5. **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：

1. 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；
2. 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）；
3. 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；
4. 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）；
5. 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**
2. function Backtracking(csp)
3. **return** Recursive\_Backtracking({}, csp)
5. function Recursive\_Backtracking(assignment, csp)
6. **if** assignment **is** complete:
7. **return** assignment
8. var := select\_unassigned\_variable(variables[csp], assignment, csp)
9. **for** value **in** order\_domain\_values(var, assignment, csp):
10. assignment.append({var = value})
11. result := Recursive\_Backtracking(assignment, csp)
12. **if** result != failure:
13. **return** result
14. assignment.remove({var = value})
15. **return** failure
16. **Forward-checking（向前检验）算法**
17. function FCCheck(C, x)
18. **for** value **in** Domain[x]:
19. **if** not\_satisfied\_constraint(C):
20. Domain[x].remove(value)
21. **if** Domain[x] **is** empty:
22. **return** true
23. **return** false
25. function Forward\_checking(level)
26. **if** unassigned\_variables **is** empty:
27. **return** true
28. v := pick\_an\_unassigned\_variable(unassigned\_variables)
29. unassigned\_variables.remove(v)
30. **for** value **in** Domain[v]:
31. solution[v] := value
32. DWO = false
33. **for** Var **in** unassigned\_variables:
34. **if** (FCCheck(C, Var)):
35. DWO = true
36. **break**
37. **if** **not** DWO:
38. FC(level + 1)
39. restore\_changed\_domains()
40. unassigned\_variables.append(v)
41. **return** false

* **关键代码**

约束性检测：

1. **bool** constraint\_check(**int** board[30][30], **int** x, **int** y, **int** n) {
2. //约束性检测，判断纵行和斜线上是否存在皇后
3. **int** directions[3][2] = { {-1, -1}, {-1, 0}, {-1, 1} };
4. **for** (**int** i = 0; i < 3; ++i)
5. **for** (**int** nx = x, ny = y; nx < n && ny < n && nx >= 0 && ny >= 0; nx += directions[i][0], ny += directions[i][1])
6. **if** (board[nx][ny])
7. **return** **false**;
8. **return** **true**;
9. }

Backtracking（回溯）算法：

1. **bool** backtracking(**int** board[30][30], **int** level, **int** n) {
2. **if** (level == n)
3. //找到一个解，结束搜索
4. **return** **true**;
5. **for** (**int** i = 0; i < n; ++i)
6. **if** (constraint\_check(board, level, i, n)) {
7. //满足约束条件，将皇后放置到该位置
8. board[level][i] = 1;
9. **if** (backtracking(board, level + 1, n))
10. //深度优先搜索下一层位置
11. **return** **true**;
12. **else**
13. //下层返回false，表明不存在解，则将该位置的皇后移除
14. board[level][i] = 0;
15. }
16. //未找到本层放置皇后的位置，返回false
17. **return** **false**;
18. }

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：

1. **bool** FCCheck(**int** domain[30][30], **int** level, **int** y, **int** n, **bool** recovery) {
2. **int** directions[3][2] = { {1, -1}, {1, 0}, {1, 1} };
3. **for** (**int** j = 0; j < 3; ++j)
4. **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]) {
5. **if** (!recovery)
6. //在domain矩阵的相应位置+1，表示在取值范围中删除该值
7. domain[nx][ny]++;
8. **else**
9. //在domain矩阵的相应位置-1，表值在取值范围中恢复该值
10. domain[nx][ny]--;
11. }
12. **if** (recovery)
13. //如果是恢复过程，则不需检测是否DWO
14. **return** **true**;
15. **for** (**int** i = 0; i < n; ++i) {
16. **bool** empty = **true**;
17. **for** (**int** j = 0; j < n; ++j)
18. **if** (domain[i][j] == 0) {
19. empty = **false**;
20. **break**;
21. }
22. **if** (empty)
23. //存在某个变量的取值范围为空集，返回DWO
24. **return** **false**;
25. }
26. //不存在取值范围为空集的变量
27. **return** **true**;
28. }

Forward-checking（向前检验）算法

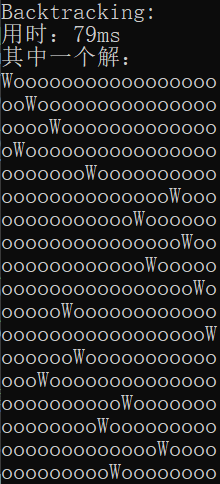
1. **void** forwardchecking(**int** board[30][30], **int** n) {
2. //声明domain矩阵，并将所有位置的值初始化为0
3. **int** domain[30][30];
4. memset(domain, 0, **sizeof**(domain));
5. //运行向前检测算法
6. FC(board, domain, 0, n);
7. }
8. **bool** FC(**int** board[30][30], **int** domain[30][30], **int** level, **int** n) {
9. **if** (level == n)
10. //找到一个解，结束搜索
11. **return** **true**;
12. **for** (**int** i = 0; i < n; ++i) {
13. **if** (domain[level][i] == 0) {
14. //如果domain矩阵中相应位置的上的值为0，则可以放置皇后
15. board[level][i] = 1;
16. **if** (FCCheck(domain, level, i, n, **false**))
17. //删除未赋值变量的取值范围中不满足约束的值，并检测DWO
18. **if** (FC(board, domain, level + 1, n))
19. //未检测到DWO，则深度优先搜索下一层位置
20. **return** **true**;
21. //检测到DWO，则将该位置的皇后移除
22. board[level][i] = 0;
23. //复原domain矩阵
24. FCCheck(domain, level, i, n, **true**);
25. }
26. }
27. //未找到本层放置皇后的位置，返回false
28. **return** **false**;
29. }

## 实验结果及分析

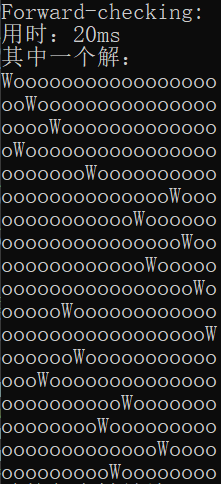
* **实验结果展示**

1. **求解18皇后问题：**

Backtracking（回溯）算法，用时79ms：

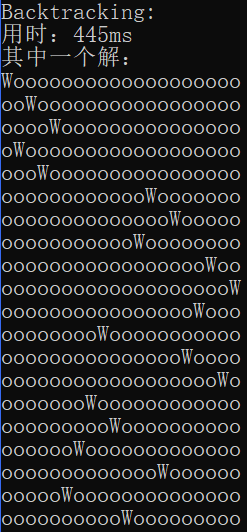


Forward-checking（向前检验）算法，用时20ms：

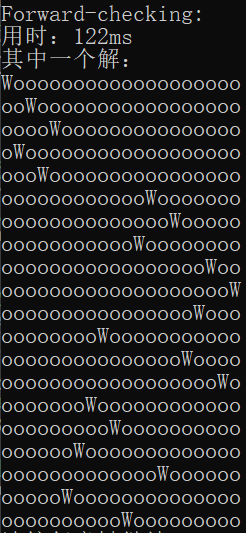


1. **求解20皇后问题：**

Backtracking（回溯）算法，用时445ms：

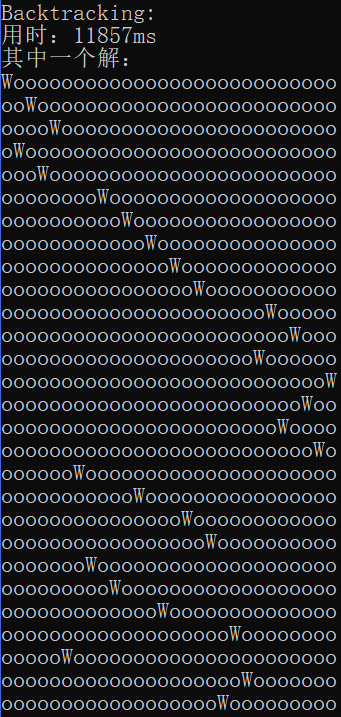


Forward-checking（向前检验）算法，用时122ms：

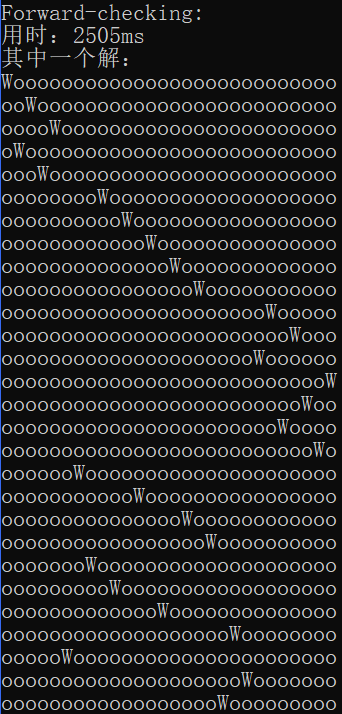


1. **求解28皇后问题：**

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。