约束满足问题(constraint satisfaction problem, CSP)

- CSP的形式化
- 回溯算法
- 向前检测算法

*Slides based on those of Sheila McIlraith

约束满足问题

- 我们目前所讨论的搜索算法把状态看作一个黑盒.
- 事实上，我们可以有一个适合于许多不同问题的通用状态表示。
- 我们可以构建专门的搜索算法，在这种一般的状态表示上有效地运行。
- 我们把可以用这种专门的表示法表示的问题称为CSPs —约束满足问题。

CSPs: 状态表示

其思想是：将状态表示为特征值的向量

- k个特征(或变量)的集合
- 每个变量都有一个具有不同值的论域, *e.g.*
  - 高度= {短, 平均, 高},
  - 重量= {轻, 平均, 重}
- 状态是通过为每个变量赋值来指定的。
- 部分状态是通过为某些变量赋值来指定的。
- 目标状态是通过特征值向量上的条件来指定的。

示例: 数独(Sudoku)

	2							
			6					3
	7	4		8				
					3			2
	8			4			1	
6			5					
				1		7	8	
5					9			
							4	

1	2	6	4	3	7	9	5	8
8	9	5	6	2	1	4	7	3
3	7	4	9	8	5	1	2	6
4	5	7	1	9	3	8	6	2
9	8	3	2	4	6	5	1	7
6	1	2	5	7	8	3	9	4
2	6	9	3	1	4	7	8	5
5	4	8	7	6	9	2	3	1
7	3	1	8	5	2	6	4	9

示例: 数独

- 81个变量, 每个变量表示一个单元格的值.
- 值的论域: 对于已经填充的单元格，论域是包含给定值的单元素集。其他变量的论域是集合{1-9}。
- 状态: 通过指定每个单元格中的值(1-9)给出的完整表格。
- 部分状态: 表格的不完整填充。
- 解: 满足以下约束条件的对每个单元格的赋值:
  - 同一列中的任何单元格都不能具有相同的值。
  - 同一行中的任何单元格都不能具有相同的值。
  - 同一子方格中的任何单元格都不能具有相同的值。

示例: 排时间表

想要为每个期末考试安排一个时间和一个场地，使得

- 没有一个学生同一时间需要参加多个期末考试。
- 所分配的场地必须在所设置的时间内可用。
- 场地必须足够大，以容纳所有参加考试的学生。

示例: 排时间表

我们使用下标i来唯一地标示考试 - "第i场考试"。

- T_i是一个表示第i场期末考试的预定时间的变量。
  - T_i的论域为{MonAm, MonPm, ⋯, FriAm, FriPm}.
- S_i是表示第i场期末考试的场地的变量。
  - S_i的论域是足够容纳第i场期末考试的所有教室。
- 对于每两场有一个学生都要参加的期末考试$i, j (i \neq j)$: $T_i \neq T_j$
- 对于每两场期末考试$i, j (i \neq j)$: $T_i \neq T_j$ 或$S_i \neq S_j$

CSP的形式化

- 一个CSP 包括
  - 一个变量的集合$V_1, \ldots, V_n$
  - 对于每个变量，可能值的一个（有穷）论域Dom$[V_i]$.
  - 一个约束的集合$C_1, \ldots, C_m$.
- CSP的一个解是一个满足每个约束的对所有变量的赋值。
- 如果不存在解，则一个CSP是不可满足的。

CSP的形式化

- 每个变量可以被赋予其论域中的任何值
  - $V_i = d$ 其中$d \in Dom[V_i]$
- 每个约束C
  - 作用在一个变量的集合上, 称为其辖域, *e.g.*, $C(V_1, V_2, V_4)$
  - 作为一个布尔函数，将这些变量的赋值映射为真/假，*e.g.*,
    - $C(V_1 = a, V_2 = b, V_4 = c) = True$
    - $C(V_1 = b, V_2 = c, V_4 = c) = False$

约束

- 一元约束(作用在一个变量上)
  - *e.g.*, $C(X) : X = 2$; $C(Y) : Y > 5$
- 二元约束(作用在两个变量上)
  - *e.g.*, $C(X, Y) : X + Y < 6$
- 高阶(n-元)约束: 作用在3个或更多变量上。

约束的表示

我们可以用一个表来表示约束，*e.g.*，$C(V_1, V_2, V_3)$，其中

- Dom$[V_1]$=$\{1,2,3\}$
- Dom$[V_2]$=Dom$[V_4]$=$\{1,2\}$

V1	V2	V4	C(V1,V2,V4)
1	1	1	False
1	1	2	False
1	2	1	False
1	2	2	False
2	1	1	True
2	1	2	False
2	2	1	False
2	2	2	False
3	1	1	False
3	1	2	True
3	2	1	True
3	2	2	False

约束的表示

通常，我们可以用一个表达式更简洁地表示约束：
e.g.，$C(V_1, V_2, V_3) = (V_1 = V_2 + V_4)$，

V1	V2	V4	C(V1,V2,V4)
1	1	1	False
1	1	2	False
1	2	1	False
1	2	2	False
2	1	1	True
2	1	2	False
2	2	1	False
2	2	2	False
3	1	1	False
3	1	2	True
3	2	1	True
3	2	2	False

示例：数独

- 变量：$V_{11}, V_{12}, \ldots, V_{21}, V_{22}, \ldots, V_{91}, \ldots, V_{99}$
- 论域：
  - Dom$[V_{ij}]$={1-9}，对空的单元格
  - Dom$[V_{ij}]$={k}，对填充了k的单元格
- 约束：
  - All-Diff$(V_{11}, V_{12}, V_{13}, \ldots, V_{19})$
  - All-Diff$(V_{21}, V_{22}, V_{23}, \ldots, V_{29})$
  - $\ldots$, All-Diff$(V_{91}, V_{92}, V_{93}, \ldots, V_{99})$
  - 列约束：
    - All-Diff$(V_{11}, V_{21}, V_{31}, \ldots, V_{91})$
    - All-Diff$(V_{21}, V_{22}, V_{32}, \ldots, V_{29})$
    - $\ldots$, All-Diff$(V_{19}, V_{29}, V_{93}, \ldots, V_{99})$
  - 子方格约束：
    - All-Diff$(V_{11}, V_{12}, V_{13}, V_{21}, V_{22}, V_{23}, V_{31}, V_{32}, V_{33})$
    - All-Diff$(V_{14}, V_{15}, V_{16}, \ldots, V_{34}, V_{35}, V_{36})$

8			4		6			7
						4		
	1					6	5	
5		9		3		7	8	
				7				
	4	8		2		1		3
	5	2					9	
		1						
3			9		2			5

求解CSPs

- 我们不关心达到一个目标状态的动作序列。
- 我们只关心找到一个满足目标的变量赋值。
- 因此，CSPs可以通过一种特殊的深度优先搜索来求解。
- 我们可以通过搜索部分赋值的空间来建立一个解。
- 原则上，我们对变量赋值的顺序并不重要—最终它们都必须被赋值。
- 如果我们在建立一个解的过程中违反了一个约束，我们可以立即拒绝当前的部分赋值。

CSP作为一个搜索问题

- 初始状态：空赋值
- 后续函数：将一个值赋给任何一个未赋值的变量使得不违反任何约束。
- 目标测试：赋值已完成

一个一般的回溯算法

- 选择一个变量*,
- 为它选择一个值*,
- 测试已赋值变量上的所有约束
- 如果有约束被违反，回溯
- 否则，对另一个变量赋值。
- 当所有的变量都被赋值，我们就找到了一个解。

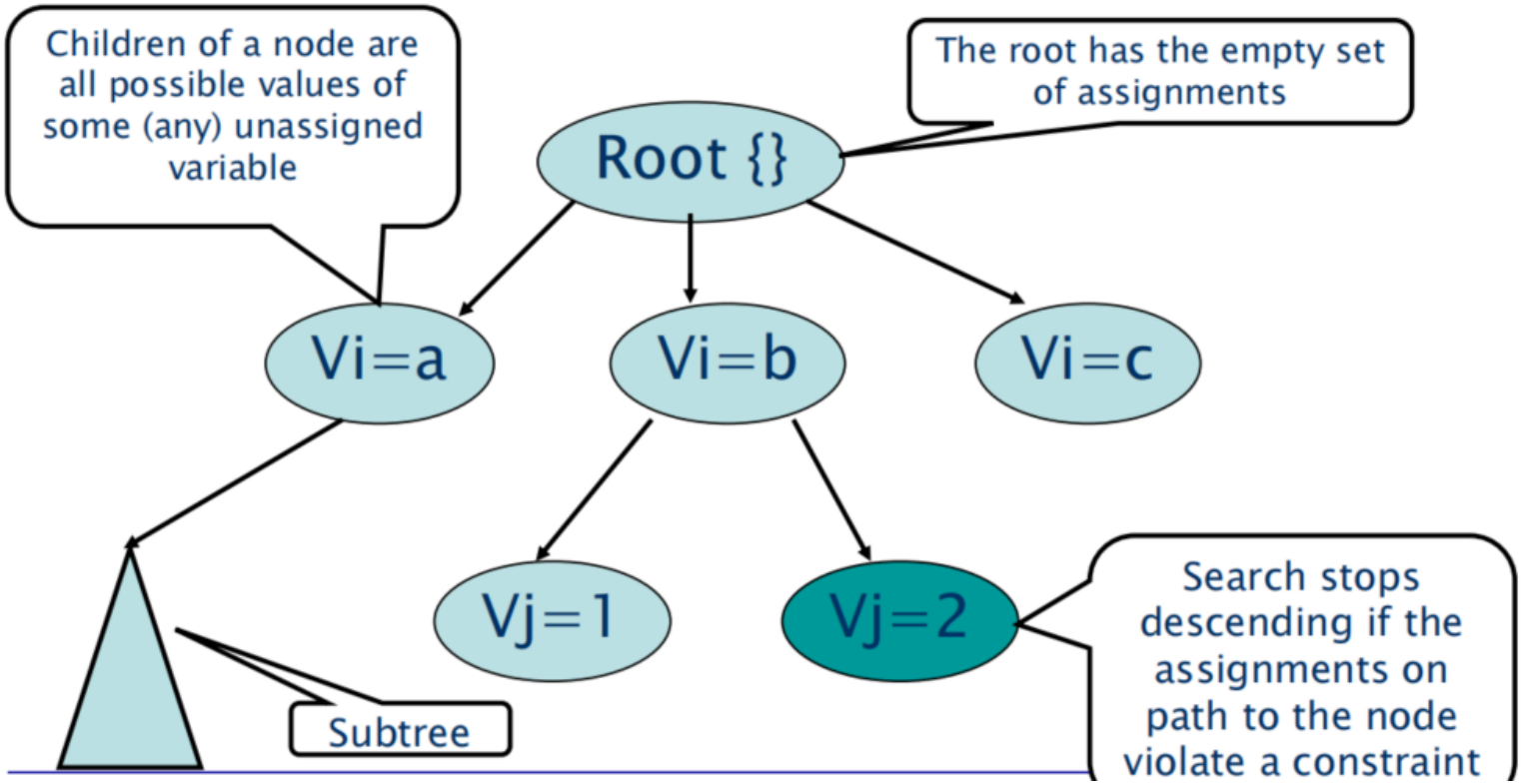
回溯搜索：算法BT

```
BT(Level)
  If all variables assigned
       PRINT Value of each Variable
       RETURN or EXIT (RETURN for more solutions)
                      (EXIT for only one solution)
  V := PickUnassignedVariable()
  Assigned[V] := TRUE
  for d := each member of Domain(V) (the domain values of V)
       Value[V] := d
       ConstraintsOK = TRUE
       for each constraint C such that
              a) V is a variable of C and
              b) all other variables of C are assigned:
                 ;(rarely the case initially high in the search tree)
            IF C is not satisfied by the set of current
               assignments:
               ConstraintsOK = FALSE
       If ConstraintsOk == TRUE:
           BT(Level+1)

  Assigned[V] := FALSE //UNDO as we have tried all of V's values
  return
```

回溯搜索

该算法搜索一个部分赋值的树。

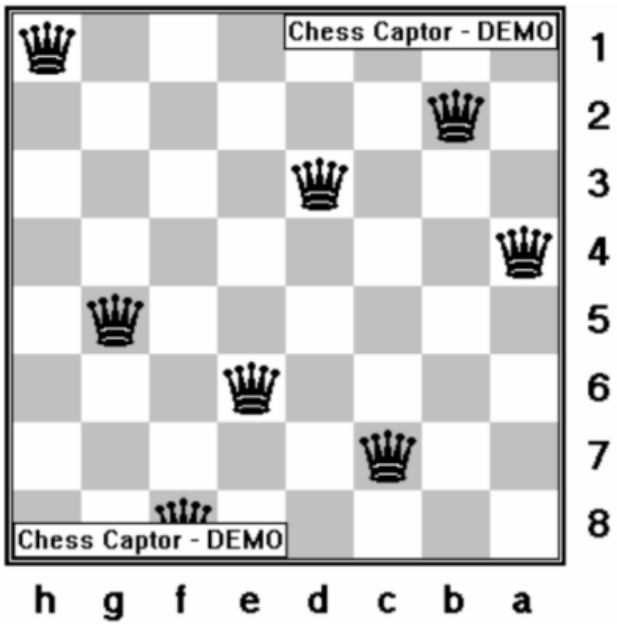


回溯搜索

- 启发式被用来确定
  - 变量赋值的顺序：
    PickUnassignedVariable()
  - 为每个变量所尝试的值的顺序。
- 下一个变量的选择可以因分支而异，e.g.
  - 在赋值$V_1 = a$下，我们可能选择下一步对V_4赋值，
    而在$V_1 = b$下，我们可能选择下一步对V_5赋值。
- 这种“动态”选择的变量顺序对算法性能有巨大的影响

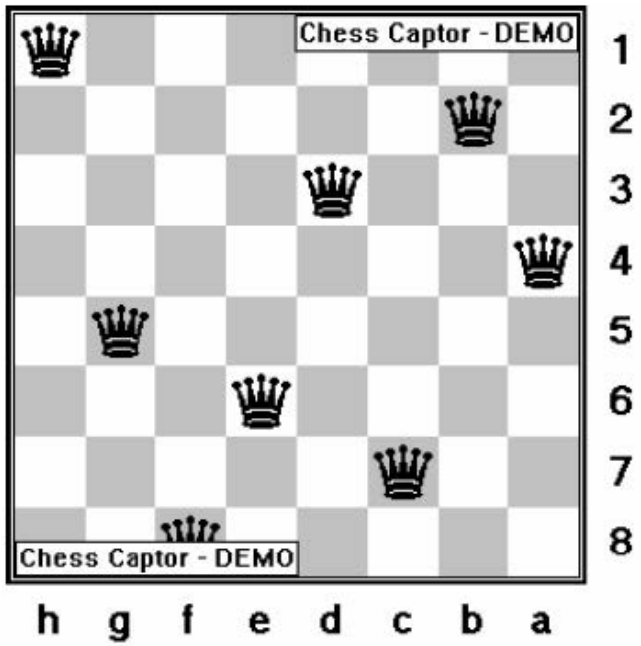
示例: N-Queens

将N个皇后放在NXN棋盘上，使得任何女王不能攻击任何其他女王。



形式化1

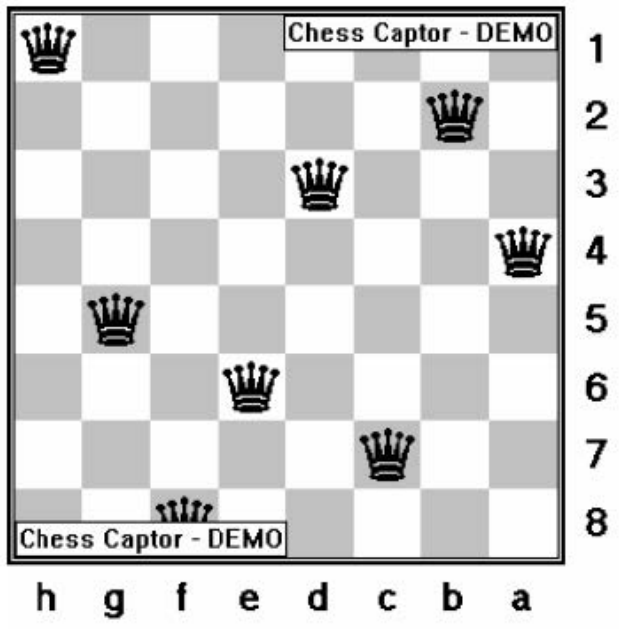
- N个变量, 每个变量的N^2个取值表示棋盘上的位置
- *e.g.*, Q1 = 1, Q2 = 15, Q3 = 21, Q4 = 32, Q5 = 34, Q6 = 44, Q7 = 54, Q8 = 59



- 这个表示有N^{2N}个状态
  - 对8-Queens: 64^8 = 281,474,976,710,656
- 有没有更好的方法来表示N-queens问题?
  - 我们知道我们不能把两个皇后放在同一行，我们可以利用这个事实

形式化2

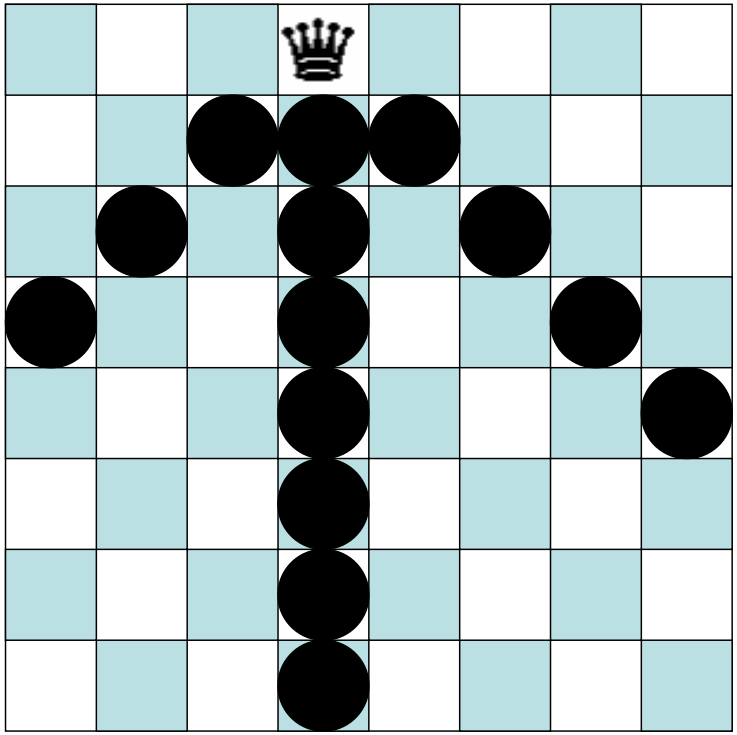
- N个变量Q_i, Q_i的值是第i行上的女王所在的列；可能的值{1, ⋯, N}.
- *e.g.*, Q1 = 1, Q2 = 7, Q3 = 5, Q4 = 8, Q5 = 2, Q6 = 4, Q7 = 6, Q8 = 3



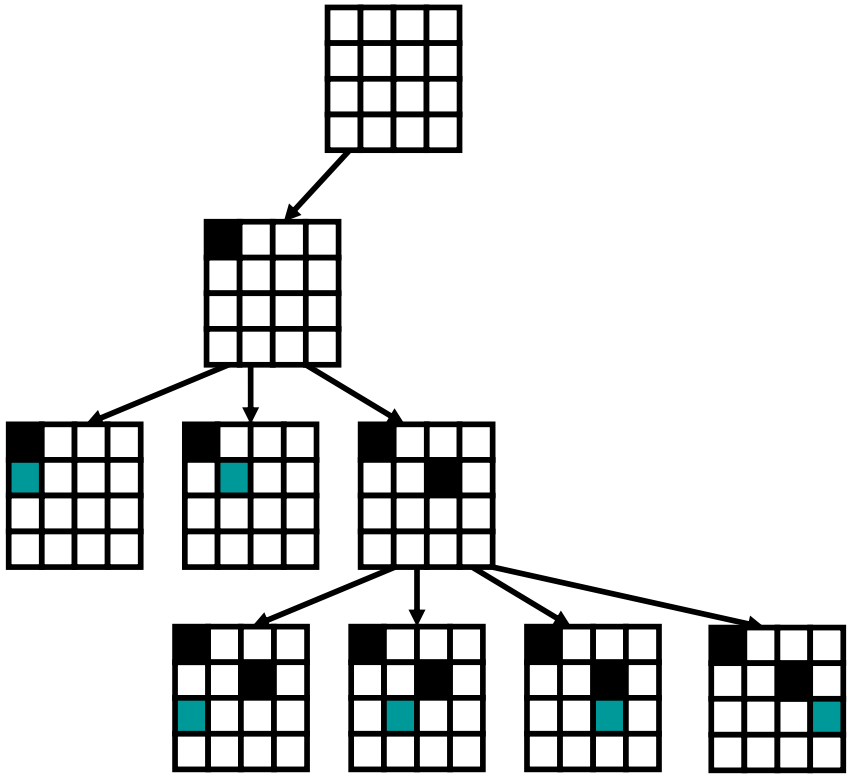
- 这个表示有N^N个状态
  - 对于8-Queens: 8^8 = 16,777,216
- 表示方式的选择可能决定问题是否能被求解!

约束

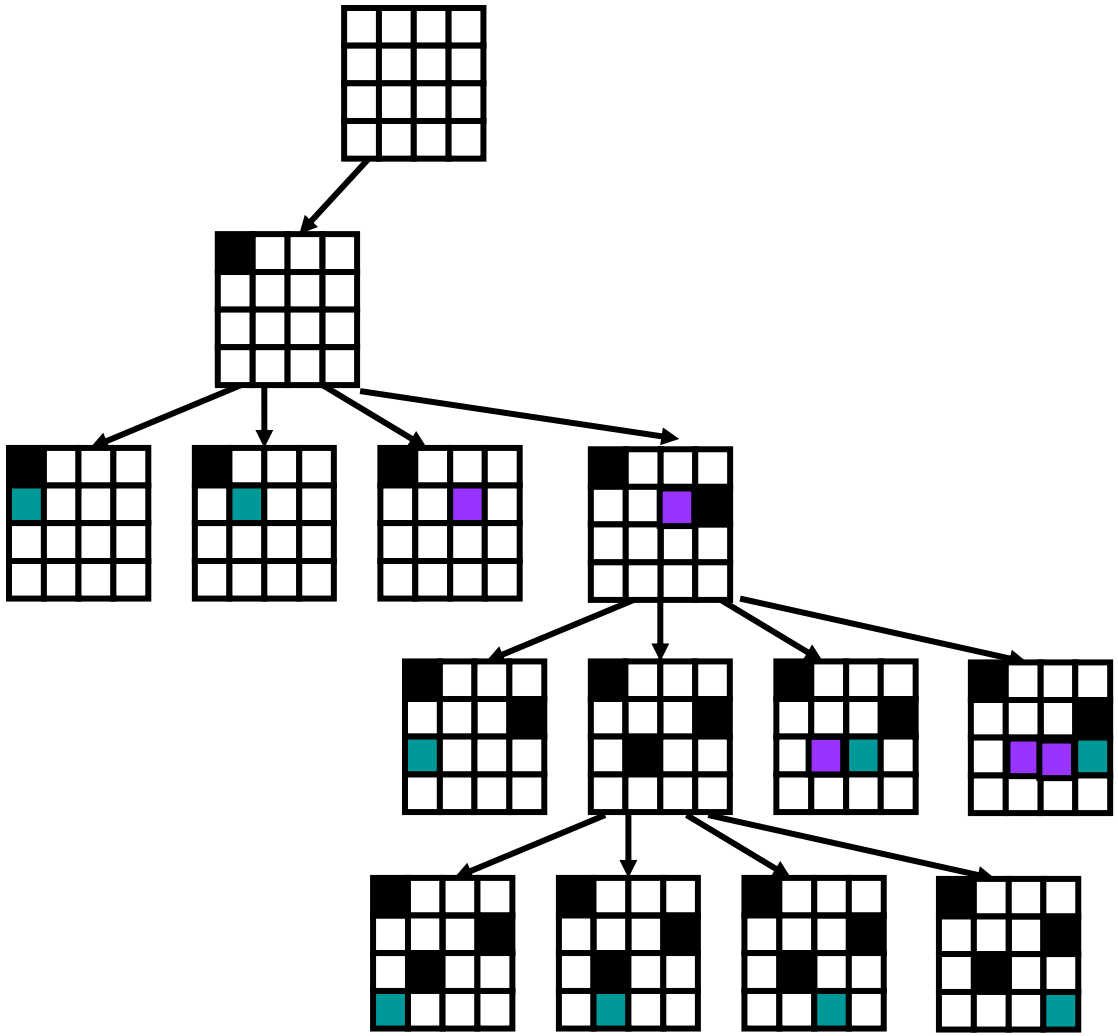
- 不能把两个女王放在同一列: $Q_i \neq Q_j$ for all $i \neq j$
- 对角线约束: $abs(Q_i - Q_j) \neq abs(i - j)$



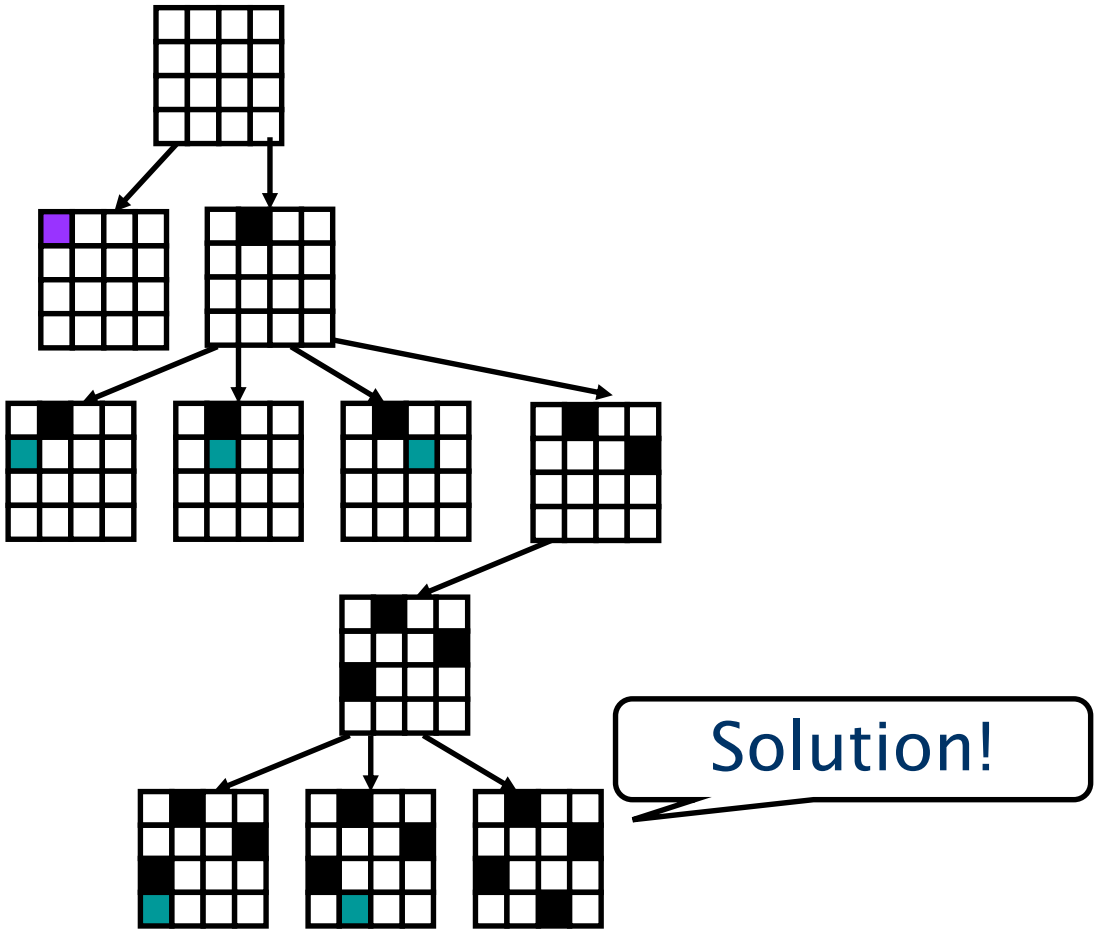
求解4-queens



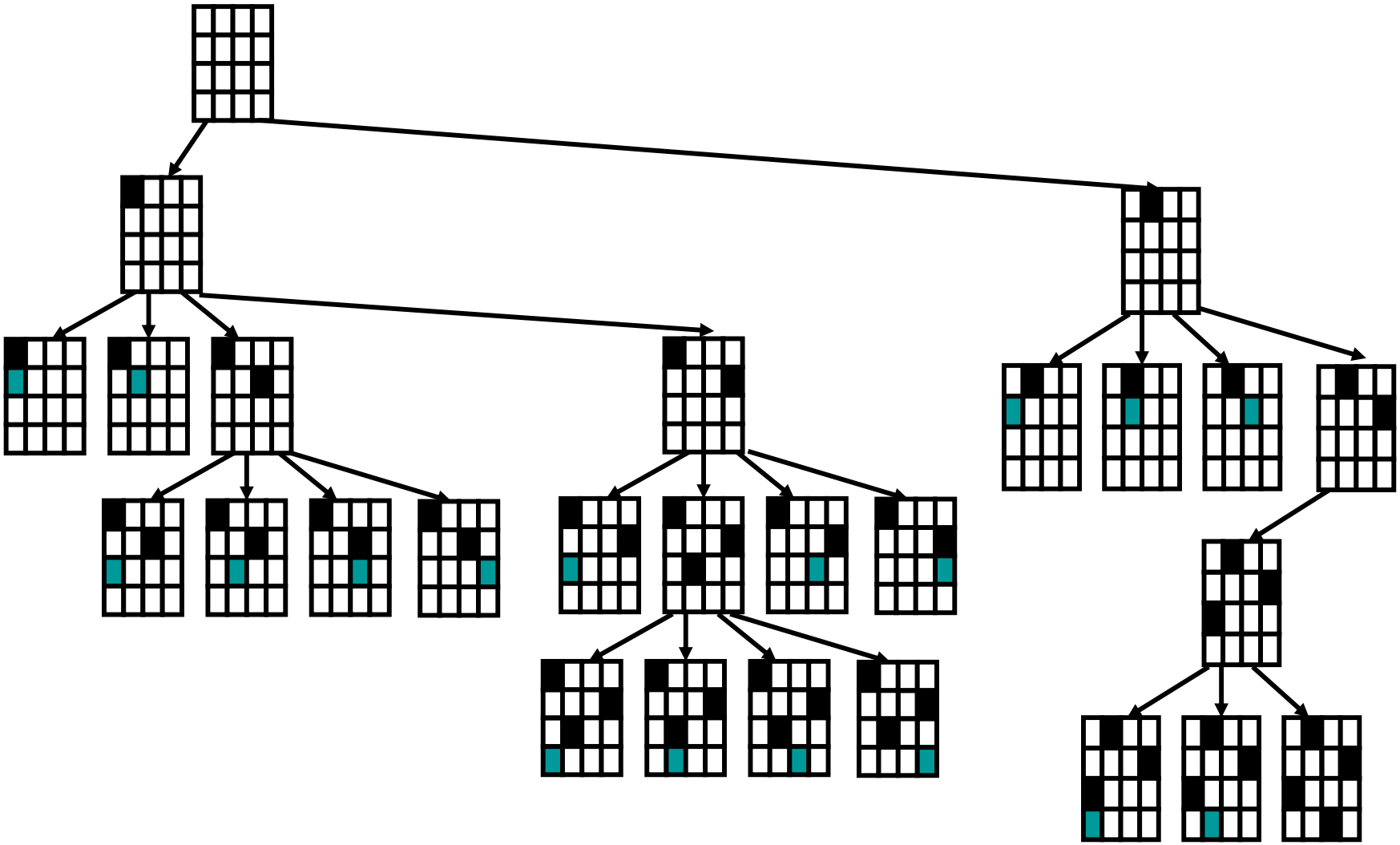
求解4-queens



求解4-queens



示例: 4-queens回溯搜索空间



CSP的形式化

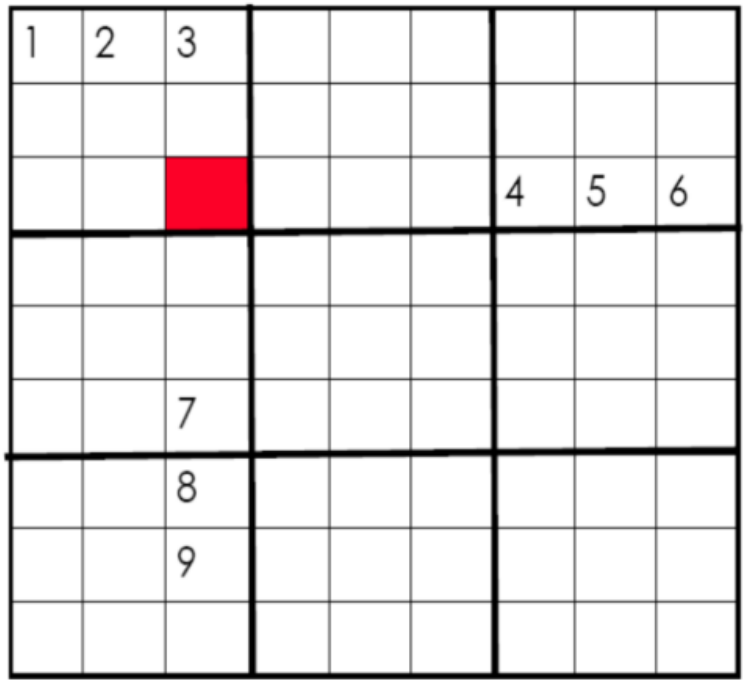
- 一个CSP包括
  - 变量的集合$V_1, \ldots, V_n$
  - 对于每个变量，可能值的一个（有穷）论域Dom$[V_i]$.
  - 约束的集合$C_1, \ldots, C_m$.
- CSP的一个解是一个满足每个约束的对所有变量的赋值。
- 如果不存在解，则CSP是不可满足的。

一个一般的回溯算法

- 选择一个变量*,
- 为它选择一个值*,
- 测试已赋值变量上的所有约束
- 如果有约束被违反，回溯
- 否则，对另一个变量赋值。
- 当所有的变量都被赋值，我们就找到了一个解。

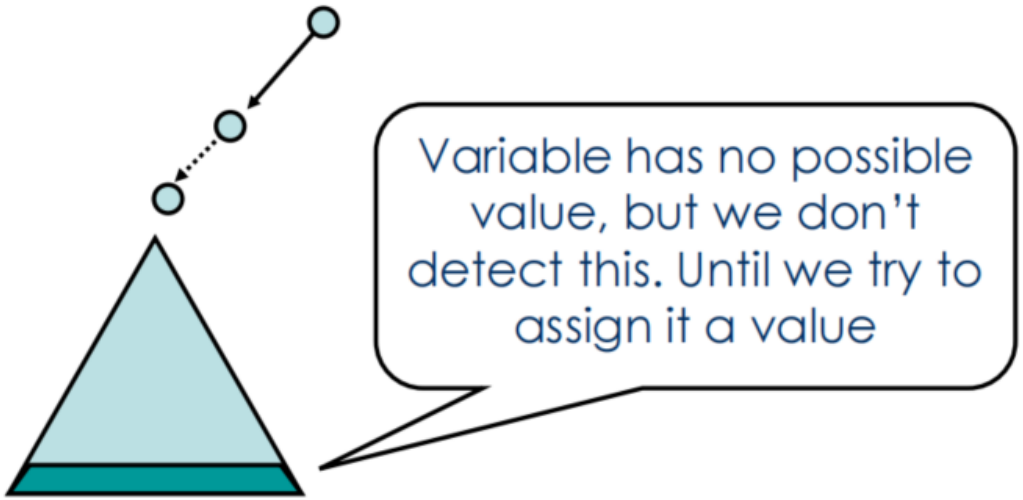
简单回溯的问题

$(3,3)$单元格没有可能的值.



简单回溯的问题

- 在回溯搜索中，我们不会检测到（3,3）单元格没有可能的值，直到第3行或第3列或第一个子方格的所有变量都被赋值。因此，我们有以下情形



- 这就引出了约束传播的思想

约束传播(推理)

- 约束传播是指在搜索中“展望(look ahead at)”尚未赋值的变量的技术。
- 试着检测明显的失败：“明显的”是指我们可以高效地检测。
- 即使我们没有检测到明显的失败，我们也可能对未来的搜索进行剪枝。

约束传播

- 在搜索过程中在搜索树的每个节点都应用传播。
- 它也可以在搜索开始之前应用！
- 传播本身是一个需要一些资源（特别是时间）的推理步骤。
  - 如果传播很慢，这可能会使搜索减慢从而使找到解需要更长的时间！
  - 因此存在以下二者之间的一个权衡：在搜索中搜索更少的节点，和拥有更高的节点/秒处理速率。
- 我们将介绍一种主要的传播类型：前向检测

前向检测(Forward checking)

- 前向检测是回溯搜索的一种扩展，它使用了“适度”的传播（look ahead）。
- 当一个变量被实例化时，我们将检测所有只剩下一个未实例化的变量的约束。
- 对于那个未实例化的变量，我们检测它的所有值，删除那些违反约束的值。

向前检测算法

For a single constraint C:

```
FCCheck(C,x)
   // C is a constraint with all its variables already
   // assigned, except for variable x.
   for d := each member of CurDom[x]
      IF making x = d together with previous assignments
         to variables in scope C falsifies C
      THEN remove d from CurDom[x]
 IF CurDom[x] = {} then return DWO (Domain Wipe Out)
 ELSE return ok
```

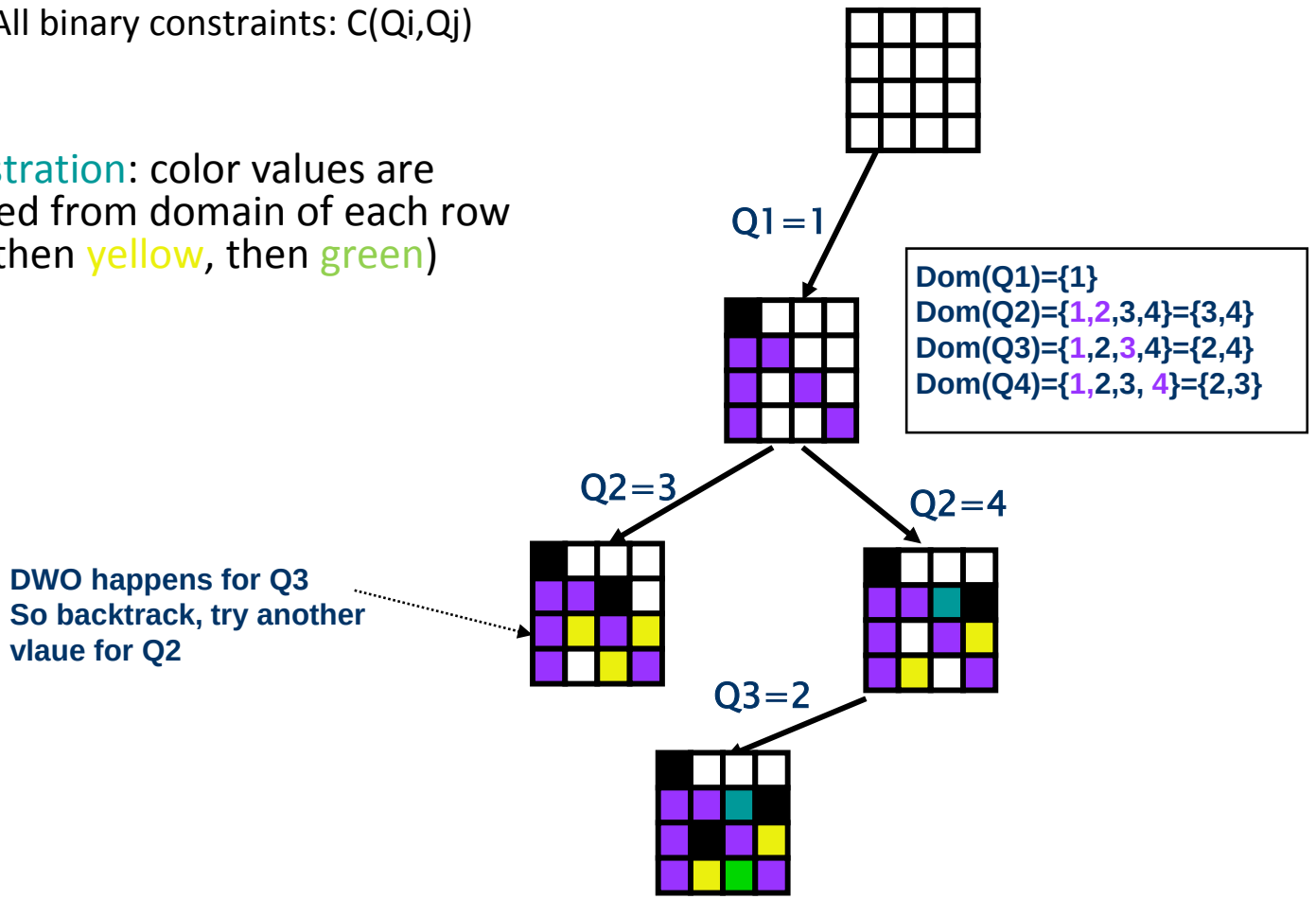
向前检测算法

```
FC(Level) /*Forward Checking Algorithm */
   If all variables are assigned
        PRINT Value of each Variable
        RETURN or EXIT (RETURN for more solutions)
                       (EXIT for only one solution)
   V := PickAnUnassignedVariable()
   Assigned[V] := TRUE
   for d := each member of CurDom(V)
        Value[V] := d
        DWOoccured:= False
        for each constraint C over V such that
              a) C has only one unassigned variable X in its scope
           if(FCCheck(C,X) == DWO)    /* X domain becomes empty*/
                DWOoccurred:= True
                break /* stop checking constraints */
        if(not DWOoccured) /*all constraints were ok*/
           FC(Level+1)
        RestoreAllValuesPrunedByFCCheck()
   Assigned[V] := FALSE //undo since we have tried all of V's values
   return;
```

FC 4-queens

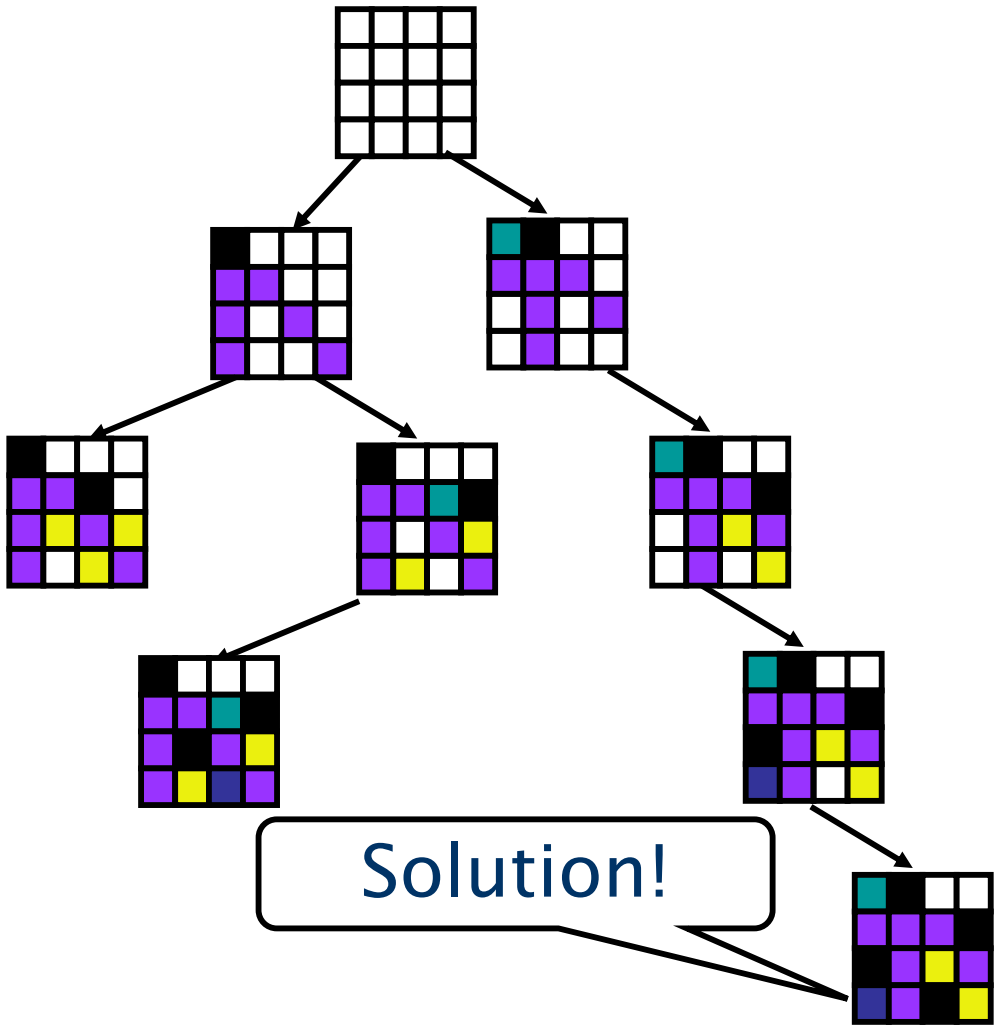
- 4X4 Queens
  - Q1,Q2,Q3,Q4 with domain {1..4}
  - All binary constraints: C(Qi,Qj)

- FC illustration: color values are removed from domain of each row (blue, then yellow, then green)

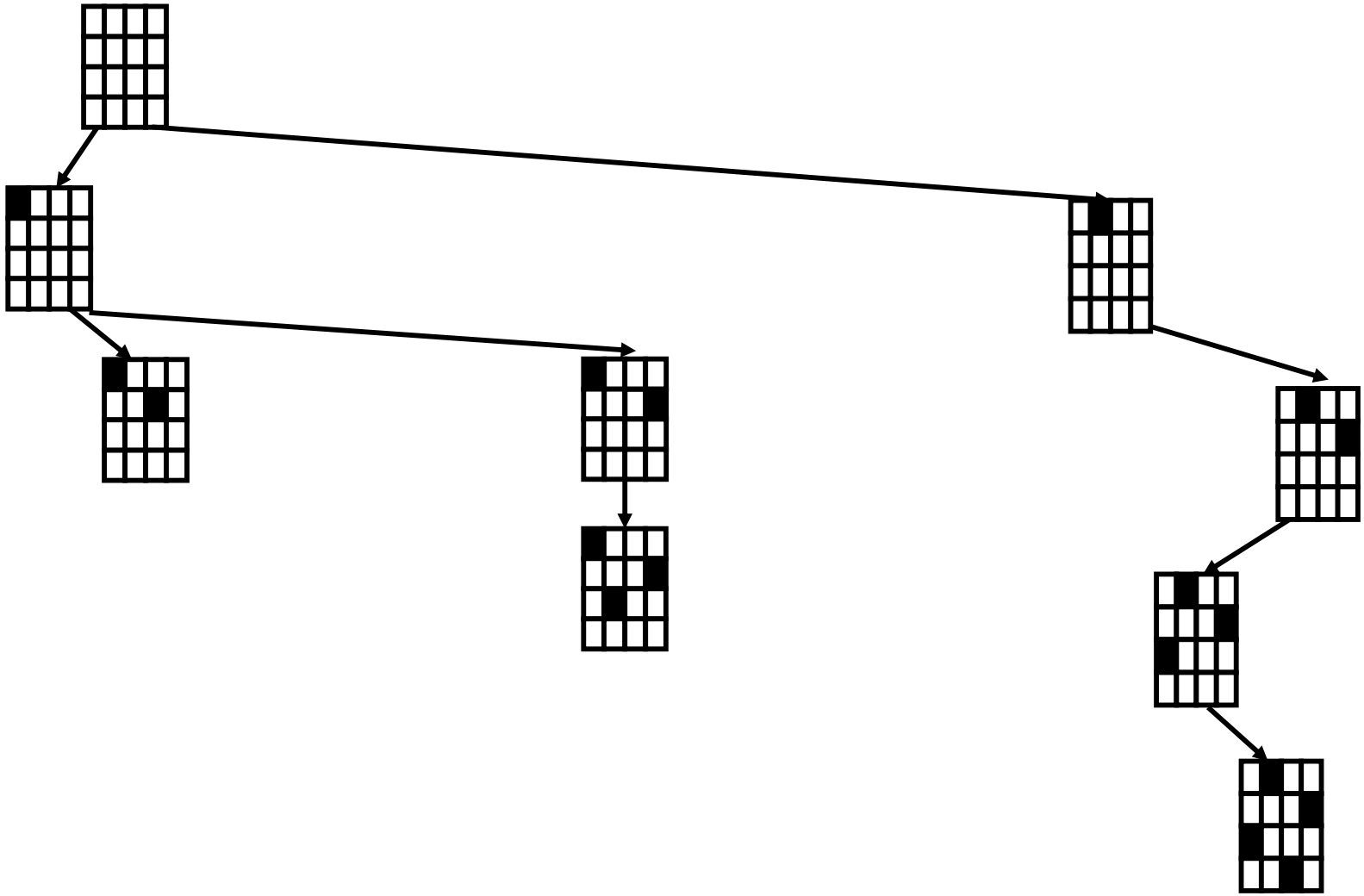


FC 4-queens

- 4X4 Queens continue...



FC 4-queens 搜索空间



FC: 恢复值

- 在我们从当前赋值（在for循环中）回溯之后，我们必须恢复由于该赋值而删除的值。
- 需要进行一些记录操作(bookkeeping)，因为我们必须记住哪些值是youyu哪个赋值删除的（每次递归调用FC时都会调用FCCheck）。

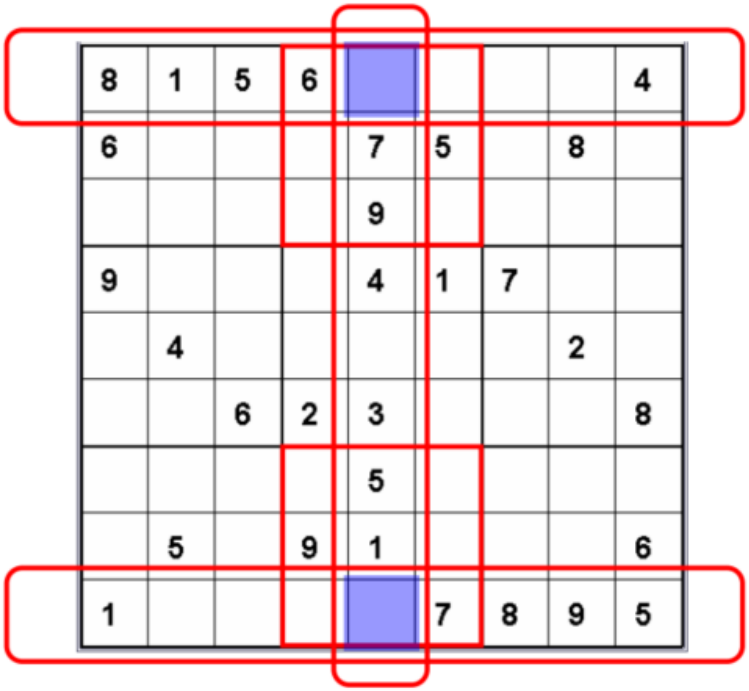
FC: 最小剩余值启发式(MRV)

FC还免费为我们提供了一个用于确定下一步要尝试的变量的非常强大的启发式：

- 始终选择一个具有最小剩余值（最小CurDom）的变量。
- 如果一个变量只剩下一个值，那么这个值必须被选择，所以我们应该立即传播它的效果。
- 这种启发式方法倾向于生成更小的搜索树。
- 这意味着可以用更少的搜索实例化更多的变量，因此会发生更多的约束传播/DWO失败
- 我们可以更快地发现不一致

MRV Heuristic: Human Analogy

- 你会先尝试哪些变量？

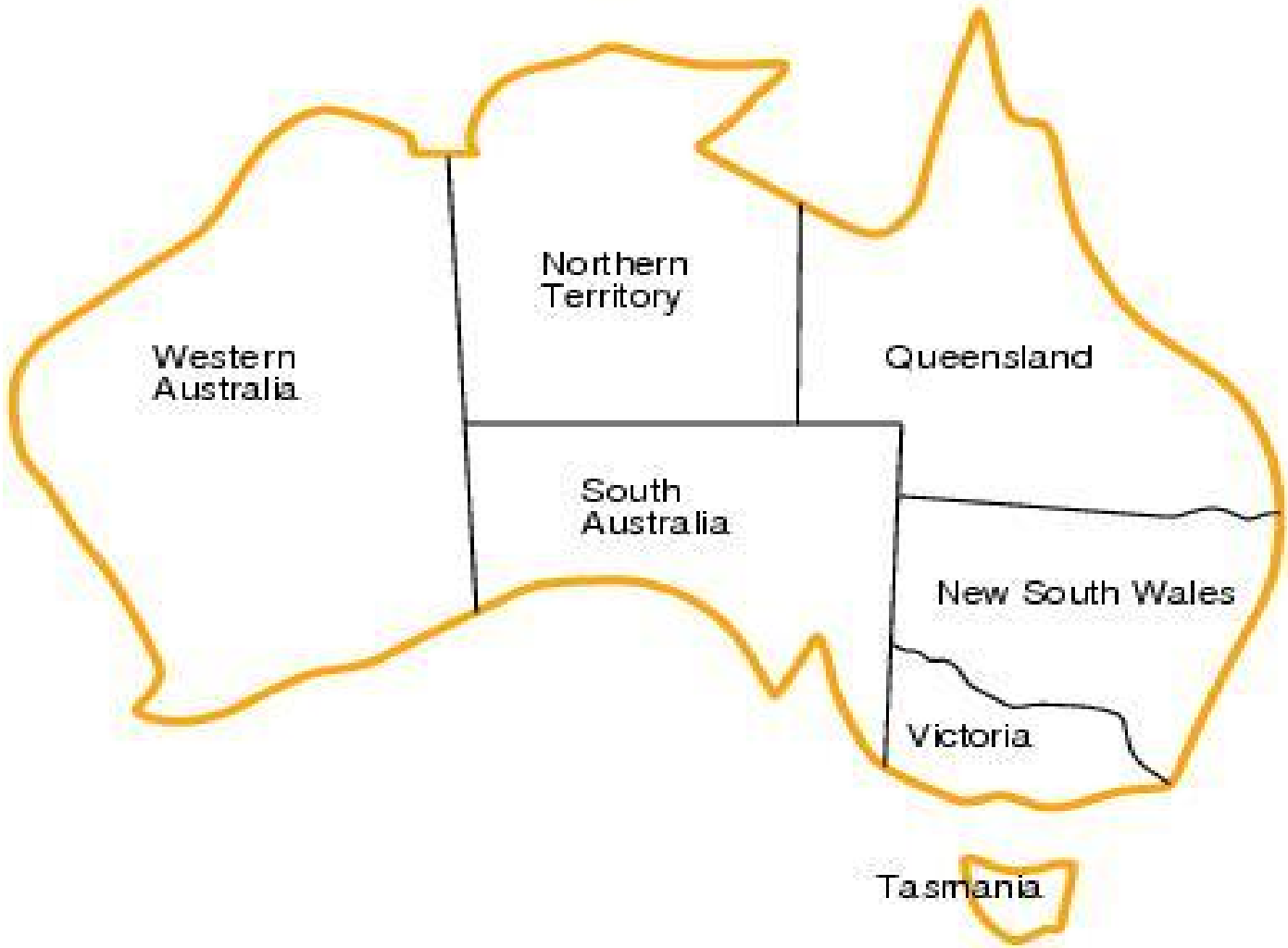


- 每个变量的论域：$\{1, \ldots, 9\}$
- (1,5)不可能的值：
  行：{1,4,5,6,8}
  列：{1,3,4,5,7,9,}
  子方格：{5,6,7,9}
  → Domain={2}
- (9,5)不可能的值：
  行：{1,5,7,8,9}
  列：{1,3,4,5,7,9,}
  子方格：{1,5,7,9}
  → Domain={2,6}
- 赋值2到单元格(1,5)之后：Domain={6}

最受限制的变量！= MRV

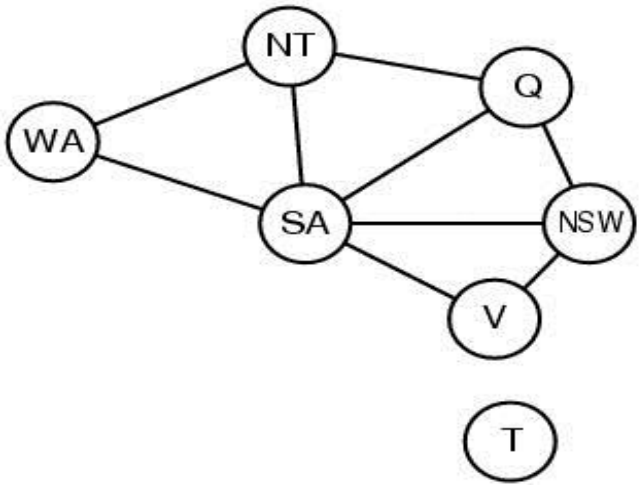
示例– 地图着色

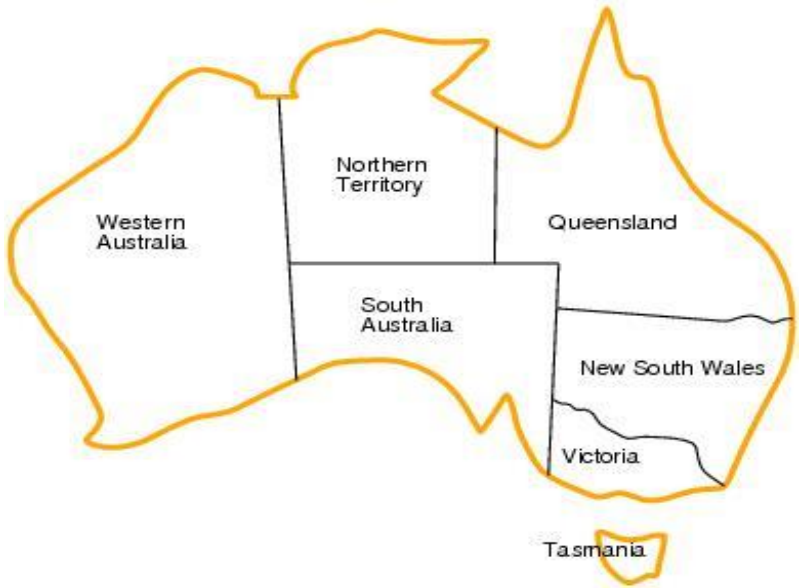
使用红色、绿色和蓝色给下面的地图着色，使得相邻的区域有不同的颜色



形式化

- 变量: WA, NT, Q, NSW, V, SA, T
- 论域: $D_i = \{red, green, blue\}$
- 约束: 相邻的区域必须有不同的颜色, e.g., $WA \neq NT$

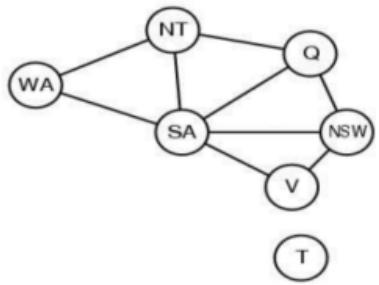




示例– 地图着色

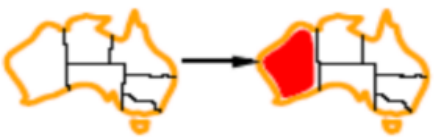
向前检测思想：

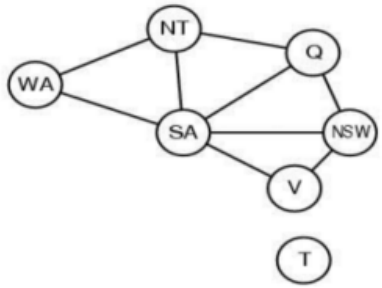
- 跟踪未赋值变量的剩余合法值。
- 当任何一个变量没有合法值时，回溯或终止搜索。

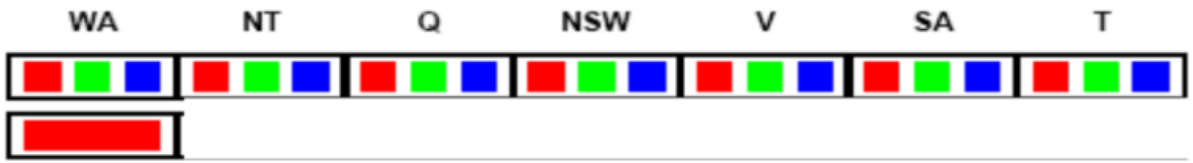


示例– 地图着色

- 赋值{WA=red}
- 对通过约束与WA相关的其他变量的效果。
  - NT不能再是红色了
  - SA不能再是红色的了

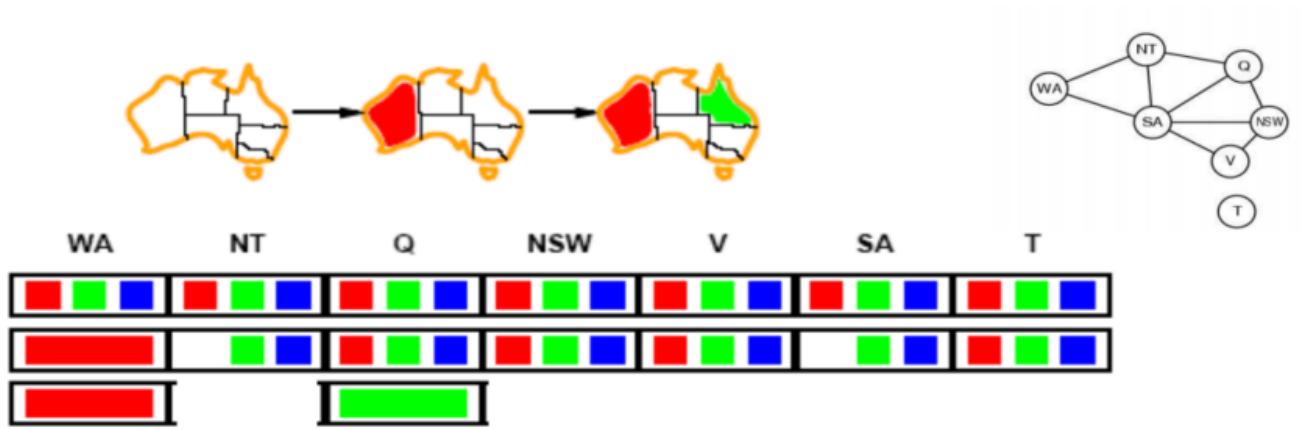






示例– 地图着色

- 赋值{Q=green}（注意：没有使用MRV）
- 对与Q相关的其他变量的效果。
  - NT不能再是绿色了
  - NSW不能再是绿色了
  - SA不能再是绿色的了
- MRV启发式将自动选择NT或SA



示例– 地图着色

- 赋值{V=blue}（注意：没有使用MRV）
- 对与V相关的其他变量的效果。
  - NSW不能再是蓝色了
  - SA是空的(DWO - Domain Wipe Out!)
- FC检测到部分赋值与约束不一致，发生回溯。

Exercise

使用MRV求解前面的地图着色示例

Empirically

- FC通常比BT快100倍左右
- 使用MRV的FC通常要快10000倍。
- 但在一些问题上，加速效果可能会大得多
  - 从而将无法解决的问题转换为可解决的问题
- FC仍然没有那么强大。
- 在实践中会使用其他更强大的约束传播形式，如Generalized Arc Consistency。