《人工智能》课程系列

约束满足问题*

Constraint Satisfaction Problem

武汉纺织大学数学与计算机学院

杜小勤

2020/07/31

Contents

*本系列文档属于讲义性质，仅用于学习目的。Last updated on: August 17, 2020。

1 概述

约束满足问题 (Constraint Satisfaction Problem, CSP) 指的是，针对给定的一组变量及其需要满足的一些限制或约束条件，找出满足这些约束关系的一个解、全部解或最优解。如果解集为空，则称该 CSP 问题是不满足的 (Unsatisfiable)；如果需要找出最优解或次优解，则 CSP 被称为约束优化问题 (Constraint Optimization Problem, COP)。

CSP 在许多领域都有着重要的应用，例如人工智能 (Artificial Intelligence)、运筹学 (Operations Reserch)、调度 (Scheduling)、供应链管理 (Supply Chain Management)、图算法 (Graph Algorithms)、计算机视觉 (Computer Vision) 及计算语言学 (Computational Linguistics) 等。

同其它大多数 AI 领域一样，CSP 也可以分为 (相互重叠的) 的 2 个子领域，即表示 (Representation) 与推理 (Reasoning)。前者分为通用的 (Generic) 及应用特定 (Application-Specific) 的方法；后者分为搜索 (Search) 和推断 (Inference) 方法。推断方法，指的是基于约束传播 (Constraint Propagation) 的一种特殊推理方法——在推理过程中，使用约束来减少一个变量的合法取值范围，并将这种影响通过 CSP 网络施加给与之存在约束关系的变量上。而搜索指的是，从变量的所有可能取值中，选取服从约束的取值。约束传播与搜索可以交替进行，或者，也可以将约束传播作为搜索前的预处理步骤——此时，约束传播可以消除不满足约束的子空间。在搜索过程中，使用约束传播，可以避免对不满足约束的子空间或子树进行搜索，从而大大提高了搜索效率。

在计算机科学与人工智能领域，CSP 中的回溯搜索研究，最早可以追溯至 1965 年，Golomb 和 Baumert 在 JACM 发表了论文“Backtrack Programming”。然而，实际上，回溯搜索是 19 世纪提出的基本算法，出现于当时的娱乐数学游戏中。CSP 中的玩具实例 8 皇后，据说在 1848 年由国际象棋选手 Max Bazzel 提出，也一直是早期约束满足问题的主要实验对象。在二次世界大战后，回溯搜索是计算机科学与运筹学的学术研究主题。随后，各种形式的约束满足与传播方法开始

出现在计算机科学领域中。

在特定类型的 CSP 问题中，最有影响力的早期工作是 Sketchpad 系统，用于求解图的几何约束问题，是现代图形程序与 CAD 的先驱。该项开创性工作由 Ivan Sutherland 在 1963 年的 MIT 博士论文“Sketchpad: A Man-machine Graphical Communication System”中，进行了详细的阐述。此外，CSP 发展早期的 2 个主流方向，即语言和算法，也都是由 Sutherland 提出来的。

在这 2 个主流方向的发展过程中，语言与算法的研究相互分离，且在一些特定的应用领域情况更甚。在语言研究领域，CSP 更多地受到逻辑程序设计 (Logic Programming)——某种形式的约束逻辑程序设计的影响，以程序设计语言与库的开发工作为主。在算法研究领域，CSP 主要受到人工智能领域中搜索方法的影响，它将主要精力集中在启发式方法与推理方法——基于约束的推理 (Constraint-based Reasoning) 与基于示例的推理 (Case-based Reasoning) 等的研究中。

实际上，CSP 是上述各种理论、技术与方法的综合与集成，是对更加具有通用性的一类方法的总称，具有众多的变体。

2 CSP 形式化

从研究对象的粒度来看，前面章节介绍的基于状态空间的搜索方法，仅仅把状态看作没有内部结构的黑盒子，这些方法通过领域特定的启发式对搜索进行引导与评估，并通过测试来确定状态是否为目标状态。

而 CSP 将状态进行进一步细分，利用一组变量及其值来表示状态。当每个变量的值满足所有关于变量的约束时，问题就得到了求解。CSP 算法充分利用了状态各变量及其内在约束结构，因而可以使用通用策略而不是问题特定的启发式方法来求解复杂问题。总体而言，CSP 具有如下优点：

- 结构化表示；
- 使用通用策略求解；
- 通过识别违反约束的变量值及其组合，可以迅速地消除大规模的搜索子空间，或避免对这些子空间进行搜索；

2.1 CSP 定义

CSP 可表示为三元组 $\mathcal{P} = \langle X, D, C \rangle$，其中 X 表示变量集合 $\{X_1, \cdots, X_n\}$；每个变量有自己的值域，D 是其值域集合 $\{D_1, \cdots, D_n\}$；C 是描述变量取值的约束集合。

值域 D_i 是由变量 X_i 的所有可能取值 $\{v_1, \cdots, v_{k_i}\}$ 组成的集合。每个约束 C_i 是有序对 $\langle scope, rel \rangle$，其中 $scope$ 表示约束中的变量组，rel 定义了这些变量取值应该满足的关系。对于关系的表示，既可以显式地列出所有关系元组，也可以使用支持如下两个操作的抽象关系来表示：测试一个元组是否为一个关系的成员；枚举所有关系成员。例如，如果 X_1、X_2 的值域均为 $\{A, B\}$，约束是二者不能取相同的值，那么关系可以描述如下：$\langle (X_1, X_2), \{(A, B), (B, A)\} \rangle$，或者 $\langle (X_1, X_2), X_1 \neq X_2 \rangle$。

有了以上基本概念，下面就可以来定义 CSP 的状态与解的概念了。状态，指的是对部分或全部变量的一个赋值，例如 $\{X_i = v_i, X_j = v_j, \cdots\}$。一个不违反任何约束条件的赋值被称为一致的或相容的 (Consistent)、合法的 (Legal)。完整的赋值 (Complete Assignment)，指的是每个变量都已经被赋值。CSP 问题的解，指的是相容且完整的赋值。部分的赋值 (Partial Assignment)，指的是仅有部分变量被赋值。

2.2 CSP 实例

8 皇后 (8-Queens) 问题是一个经典的 CSP 问题。该问题是，在国际象棋棋盘上，如何放置 8 个皇后，使得任意 2 个皇后都无法互相攻击对方。换句话说，同一行与同一列上只允许放置 1 个皇后，同一对角线上至多只能放置 1 个皇后。如图2-1所示，展示了 8 皇后问题的一个合法解。

另一个经典的 CSP 例子是，地图着色 (Map Coloring) 问题。在该问题中，算法需要为每个行政区域填充一种颜色，例如 $\{red, greed, blue\}$ 中的一种，但是要确保任意两个相邻的区域不能拥有相同的颜色。如图2-2所示，展示了澳大利亚地图的一个着色示例。

如果 CSP 中约束所含变量的个数不超过 n，那么我们称该问题为 n 元 CSP。例如，上述的 8 皇后问题与地图着色问题，都属于 2 元 CSP。可以证明，任意 n 元 CSP 都可以转换为 2 元 CSP，这使得 CSP 算法变得更简单通用些。因此，后

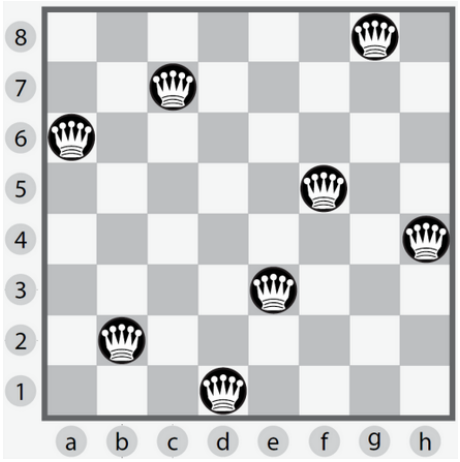

图 2-1: 8 皇后问题的一个解

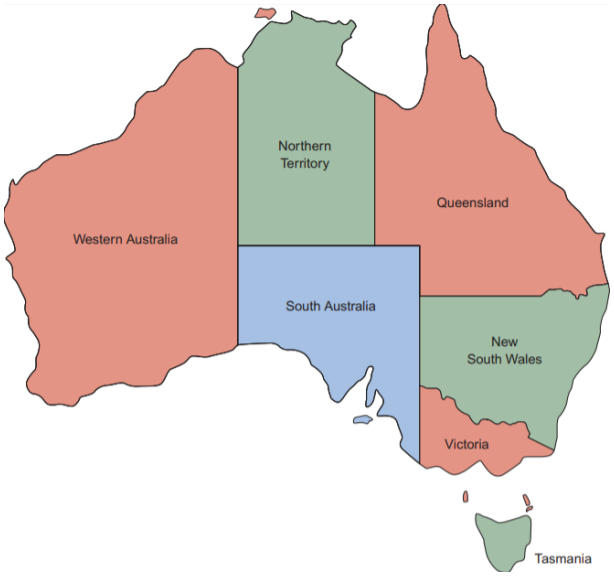

图 2-2: 澳大利亚地图着色示例

面的讨论，将以 2 元 CSP 为主要讨论对象。

依据前述的 CSP 形式化定义，可以将 8 皇后问题形式化如下：

- 使用变量 X_i 表示每一行中的皇后，其中 $1 \leqslant i \leqslant 8$，则 8 皇后问题中的变量集合为 $X = \{X_1, X_2, \cdots, X_8\}$，表示总共有 8 个皇后。
- 每个变量 X_i 的具体取值则表示该皇后所在列的位置，即值域 $D = \{D_1, D_2, \cdots, D_8\}$，其中 $D_1 = D_2 = \cdots = D_8 = \{1, 2, \cdots, 8\}$。
- 任意 2 个变量之间的约束关系可表示为 $\langle (X_i, X_j), R_{ij} \rangle$[1]，其中 $i \neq j$ 且 $1 \leqslant i, j \leqslant 8$。对于每个约束关系 R_{ij}，需要满足如下几个约束条件[2]：
  - 两个皇后不能在同一行上：此条件隐含于 $i \neq j$ 的约定中。
  - 两个皇后不能在同一列上：$X_i \neq X_j$。
  - 两个皇后不能位于对角线上：$|i - j| \neq |X_i - X_j|$。

  因此，约束关系可表示为：

$$\langle (X_i, X_j), i \neq j \text{ and } X_i \neq X_j \text{ and } |i - j| \neq |X_i - X_j| \rangle$$

类似地，可将澳大利亚地图着色问题形式化如下：

- 使用变量 X_i 表示澳大利亚的行政区或州，即 $X = \{X_1, X_2, \cdots, X_7\} = \{WA, NT, Q, NSW, V, SA, T\}$。
- 每个变量 X_i 的具体取值表示该行政区或州的着色,即值域 $D = \{D_1, D_2, \cdots, D_7\}$，其中 $D_i = \{red, green, blue\} (1 \leqslant i \leqslant 7)$。
- 约束关系表示为相邻的区域不能使用相同的颜色，其相邻关系可以使用约束图 (Constraint Graph) 来表示，如图2-3所示。图中，节点表示变量，边表示变量之间具有相邻关系，因而也表示着色的约束关系。

[1]使用简记符号 R_{ij} 表示 $R_{(x_i, x_j)}$。

[2]需要注意的是，一般情况下，CSP 的约束关系是双向的，例如 $\langle (X_i, X_j), R_{ij} \rangle \equiv \langle (X_j, X_i), R_{ji} \rangle$ 等。

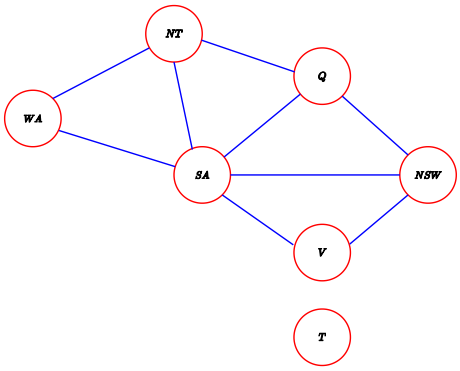

图 2-3: 澳大利亚着色问题的约束图

约束关系共有 9 个元组，枚举如下：

$$\begin{aligned} C = \{&\langle (X_6, X_1), R_{61}\rangle, \langle (X_6, X_2), R_{62}\rangle, \langle (X_6, X_3), R_{63}\rangle, \langle (X_6, X_4), R_{64}\rangle, \\ &\langle (X_6, X_5), R_{65}\rangle, \langle (X_1, X_2), R_{12}\rangle, \langle (X_2, X_3), R_{23}\rangle, \langle (X_3, X_4), R_{34}\rangle, \\ &\langle (X_4, X_5), R_{45}\rangle\} \end{aligned} \tag{1}$$

其中，R_{ij} 也可枚举如下：

$$\begin{aligned} &R_{ij} \\ &= \{X_i \neq X_j\} \\ &= \{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\} \end{aligned} \tag{2}$$

一般地，对于 1 元 CSP 和 2 元 CSP，可以使用约束图来表示：节点表示变量，边表示变量之间的约束关系。对于 n 元 CSP，可以使用约束超图 (Constraint Hypergraph) 来表示：方块节点表示 n 元约束节点，被称为超节点；圆形节点表示普通节点，即变量；边仍然表示约束关系。关于约束超图的具体内容，请阅读相关参考文献。

图2-4展示了 8 皇后问题的约束图，它是一个完全图 (Complete Graph)，因为每对变量之间都存在位置约束。

实际上，按约束中变量的个数对 CSP 进行分类的方法，也不是唯一的。例如，对于上述的 8 皇后问题，为了便于解的求解处理，可以将其归类为 2 元 CSP，从而可以方便地使用约束图来表示。但是，也可以将它归类为 8 元 CSP，因为它的所有变量必须取不同的值，即都存在约束关系。正如图2-4所展示的那样——任意

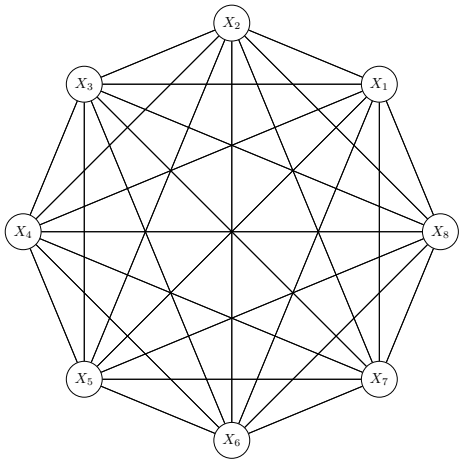

图 2-4: 8 皇后问题的约束图

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

Unsolved Sudoku

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

Solved Sudoku

图 2-5: 数独游戏的一个示例

一个节点与其余的所有节点之间都存在着约束边[3]。

对于变量个数任意的约束，我们称它为全局约束 (Global Constraint)。需要注意的是，这并不意味着它的每个约束都涉及到所有的变量。在全局约束中，最常见的是 Alldiff，它指的是约束中的所有变量之间都存在约束，必须取不同的值。例如，8 皇后问题可以看作是 Alldiff 约束类型。另一个经典的 Alldiff 例子是数独游戏，它的每一行或每一列或每个 3×3 方块内的所有变量必须取不同的值。如图2-5所示，展示了一个典型的数独游戏。左图表示数独游戏的初始状态，右图表示该数独游戏的目标状态或解状态。

将问题形式化为 CSP 以及表示为约束图，可以获得如下好处：CSP 表示问题

[3]这并不令人感到奇怪。前文也已提到过，可以将任何 n 元 CSP 转换为 2 元 CSP。

较为自然；可以使用 CSP 通用求解系统进行问题的求解，且比其它搜索技术求解要简单快捷得多。原因在于，CSP 求解能快速消除搜索空间中的一些无用子空间，或避免对这些子空间进行搜索。例如，在澳大利亚地图着色问题中，一旦选取 $X_6 = SA = red$，那么它的 5 个相邻节点都不能取值 red。在没有使用约束传播的情况下，搜索过程需要考虑 5 个相邻节点的 $3^5 = 243$ 种赋值组合；在使用约束传播时，可以消除 red 的冲突情形，使得总的赋值组合变为 $2^5 = 32$，减少了大约 87% 的赋值组合，这样就大大提升了搜索效率。

总体而言，CSP 属于 NP-Complete 问题：一方面，“不难”检测出一组给定的变量赋值是否满足约束，也“不难猜测”出一组合法的变量赋值，这表明 CSP 属于 NP 问题；另一方面，SAT(Satisfiability Problem) 属于 NP-Hard 问题，且可以被形式化为 CSP。这意味着，CSP 既属于 NP 问题，又属于 NP-Hard 问题，因而属于 NP-Complete 问题。

因此，不可能为 CSP 找到一个多项式时间级的算法。但是，在实际应用中，人们有极大的兴趣寻找实际表现较好的那些 CSP 算法。

3 求解 CSP 的通用方法

3.1 回溯法

回溯 (Backtracking，BT) 法是一种直接在解空间中按照深度优先搜索策略执行搜索的通用型算法，即它是一种求解问题的基本策略。为了能够在解空间执行有效的搜索，需要为解空间定义合适的结构，以便将解空间中的元素组织在一起，从而允许回溯法遍历整个解空间。一般情况下，可以将解空间组织成树或图的形式[4]。

早在 1960 年，Walker 就发表了回溯法论文“An Enumerative Technique for A Class of Combinatorial Problems”。在 1965 年，Golomb 和 Baumert 在 JACM 上发表了论文“Backtrack Programming”。实际上，一般认为，回溯算法被独立发明了若干次。

在本章，我们将讨论基本回溯法及结合约束传播的回溯法。基本回溯法是遍历解空间的一种基本框架。为提高它的搜索效率，避免无效的搜索，通常可以采

[4]通常情况下，回溯法涉及 2 种解空间：子集树与排列树。请参考《人工智能》课程系列之 Python 算法基础。

取 2 种措施[5]：

1. 使用约束函数剪去不满足约束条件的子树；

2. 使用限界函数剪去得不到最优解的子树；

关于与第 2 种方法的结合，已经在 α-β 博弈树搜索算法中进行了讨论。

从上面的描述可以看出，与其说回溯是一种算法，还不如说它是一种通用技术，可以用于各种递归深度搜索框架之中，例如 Minimax 与 α-β 博弈树搜索等算法。此外，其自身也是分支限界算法 (Branch and Bound Programming) 的必备“功能组件”。因此，从某种程度上看，回溯技术在人工智能的早期阶段发挥着非常重要的作用。尤其在 1970 年代，当 CSP 领域涌现出来后，回溯技术更是表现突出，取得了空前的成功。在 1980 年代早期，Haralick 和 Elliott 在 Artificial Intelligence Journal 上发表了论文“Increasing Tree Search Efficiency for Constraint Satisfaction Problems”，更是体现了回溯技术的价值。

在下面，我们将讨论约束函数与回溯框架的结合。在本章的最后，将讨论结合约束传播的混合回溯法。

通常情况下，回溯法使用深度优先搜索框架。为便于理解，将变量按层次表达为搜索树结构[6]：在搜索树中，每一层的所有节点表示某种变量；某个节点的所有分支表示该变量的所有可能取值。回溯法从树的根节点开始，每次选择一个分支执行，即选取该变量的某个具体取值 (此操作被称为“实例化变量”[7])。然后，算法检测新变量的取值与已有变量的取值是否满足约束：如果满足，表明新变量实例化成功，继续往下执行，到达下一层 (即选取另一个新变量)；否则，新变量实例化失败，需要回溯[8]，重新实例化一个新值。上述过程，递归进行：

- 如果所有变量都有合法值或满足约束的值，则找到问题的解；

[5]这 2 种措施通常可以结合在一起使用。此外，结合约束函数与限界函数的回溯法与分支限界法有些类似，但是两者之间还是有差别的，请参考《人工智能》课程系列之 Python 算法基础。

[6]需要注意的是，虽然此处按搜索树结构来表达回溯法的执行流程。但是在实现时，不一定按照显式的树结构来组织变量，正如下文回溯法伪代码所展示的那样。

[7]即，此处的实例化 (Instantiation) 意为选取该变量的一个具体取值。

[8]需要注意的是，存在两种回溯，一种是取值回溯或实例化回溯，另一种是变量 (间) 回溯。前者表示选取同一个变量的另一个取值；后者表示从当前变量切换到另一个变量，或选取一个新变量。此处，属于实例化回溯。

- 如果当前变量的所有实例化回溯失败，则执行变量回溯，即回溯至上一个变量的某个取值；

直至找到一个解或所有解，或者已经遍历所有的变量取值组合，没有找到解，算法停止。

下面，给出针对 CSP 的回溯法伪代码。

算法 **3.1**（*CSP* 回溯法）

```
def Consistent(X_i,v^(i)):
    1. for each (X_j,v^(j))∈ Solution:
    2.     if R_ij ∈ R and (v^(i), v^(j)) ∉ R_ij:
    3.         return False
    4. return True

def Backtracking(Vars):
    1. Select a variable X_i ∈ Vars
    2. for each value v^(i) ∈ D_i:
    3.     if Consistent(X_i, v^(i)):
    4.         Solution ← Solution + (X_i, v^(i))
    5.         if X_i is the only variable in Vars:
    6.             return True
    7.         else:
    8.             if Backtracking(Vars\{X_i}):
    9.                 return True
   10.             else:
   11.                 Solution ← Solution − (X_i, v^(i))
   12. return False

def CSP-BT():
    1. Solution ← ∅
    2. return Backtracking(X)
```

需要注意的是，上述回溯算法只找出 1 个满足所有约束条件的解。如果需要找出所有解，对函数 Backtracking 的相应部分稍加修改即可。请试着对该函数进行修改，以实现此功能。关于修改后的算法描述，详见附录5.1。

图3-6展示了 4 皇后 CSP 回溯法的执行示意图。在该图中，被大圆圈框住的节点表示回溯法找到的一个解 $X = \{X_1 = 2, X_2 = 4, X_3 = 1, X_4 = 3\}$。此外，图中还标注了算法访问的总节点个数、约束检测函数的总执行次数以及每个节点的约束检测次数。它们都是反映算法执行性能的重要指标。

从回溯法的算法描述可以看出，回溯法使用约束函数来消除无效子树的搜索，相当于对解搜索树进行了剪枝处理，因而大大提高了效率。但是，回溯法本质上

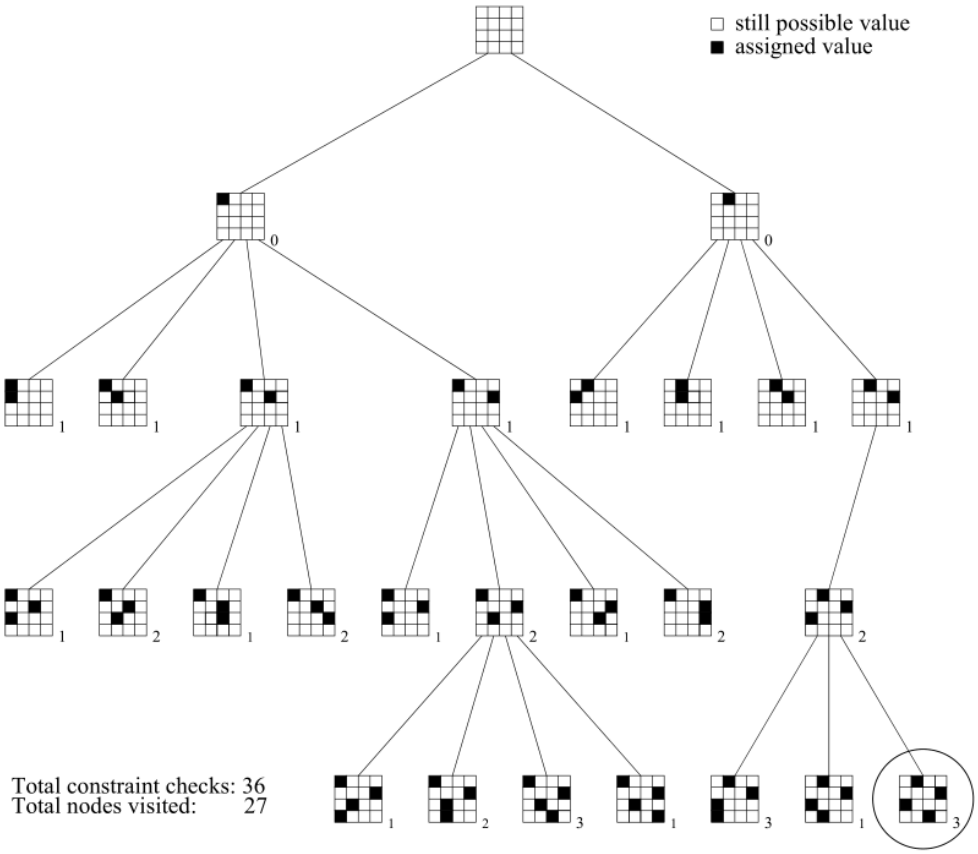

图 3-6: 4 皇后 CSP 回溯法执行示意图

还是解空间上的深度优先搜索，对大多数问题而言，其时间复杂度仍然是指数级的。回溯法也是一种完备的搜索算法，这意味着，如果问题有解的话，它一定能够找到解。

此外，回溯法还存在着 2 个主要问题：(反复) 抖动 (Thrashing) 和冗余的约束检测 (Redundant Constraint Check)。抖动指的是，算法在解空间的不同部分进行搜索时总是由于同样的原因而导致搜索失败。对于这个问题，算法可以直接回跳至导致失败的变量处进行回溯来避免，例如 Backjumping(BJ) 就是这样一种算法，它可以在多个层级之间进行回跳，以避免抖动现象的发生。而冲突导向的回跳 (Conflict-directed Backjumping,CBJ) 算法对 BJ 算法进行了改进，允许执行多个回跳。无论是 BJ 算法，还是 CBJ 算法，它们都可以减少无效节点的访问数目，从而间接地使得约束检测个数减少，因而大大提高了搜索效率。针对冗余约束检测问题，研究者提出了 Backmarking(BM) 算法，用来阻止相同的约束检测反复发生。然而，对于回溯法而言，抖动与冗余约束检测问题都无法完全避免。

从算法的通用性角度来看，CSP 的表示具有标准化与通用性的特点，这使得求解算法也具有通用性，即不需要针对特定应用设计特定的算法。因此，我们可以设计出求解 CSP 的通用算法。

3.2 约束传播

在常规的状态空间中，算法能够执行的唯一任务是搜索。但是，在 CSP 的解状态空间中，算法除了可以执行回溯搜索外 (即从变量的多种可能取值中选取新值)，还可以执行推断 (Inference)——这是一种利用约束关系进行推理的方法，被称为约束传播。具体而言，约束传播指的是，使用约束减少变量的合法或有效的取值范围，并将这种影响传播到与此变量存在约束关系的其它变量上，从而间接地压缩了状态空间，提高了求解效率。

约束传播可以作为回溯法的预处理步骤，也可以结合进回溯法的框架中：搜索与约束传播将会交替执行，可以进一步提高回溯法的执行效率。

约束传播的基本思想是，将一个 CSP 转换为等价的、更简单且易于求解的 CSP。其主要技术手段是，消去变量中无效冗余的取值，并将这种变化通过约束变量进行传播，从而也消去那些变量中的无效冗余取值。约束传播可以在不同的粒度上执行，从而形成了不同的约束传播方法。

在本章的开始，我们引入了约束图的概念，它使用图结构表示二元约束关系：

节点表示变量，节点之间的边表示 2 个变量之间存在约束关系；如果节点存在一条导向自己的边，则表明该节点所表示的变量存在 1 元约束。显然，约束传播算法可以方便地使用各种图算法来实现。

在“回溯法”一节中，我们提到过，回溯法会呈现出病态的抖动行为 (Pathological Thrashing Behavior)——算法反复对无效子树进行搜索，而这些无效子树在本质上都是一样的，只是在具体的变量赋值上有所区别。通常情况下，抖动对回溯算法的执行效率有很大的影响，需要极力避免。

研究表明，各种一致性或相容性检测算法 (Consistency Check Algorithm) 通过将约束变量集的规模与交互层次限制在局部范围，使用效率更高的多项式级算法，使得约束变得更加紧凑，或者使得隐式的约束显式化，从而可以识别并消除许多抖动行为。

在下面的讨论中，假设 CSP 为 $\mathcal{P} = \langle X, D, C \rangle$，一元约束为 $C_i = \left\langle (x_i), R_{(x_i)} \right\rangle$，使用简记符号 R_i 表示 $R_{(x_i)}$。类似地，二元约束可表示为 $C_s = \left\langle (x_i, x_j), R_{(x_i,x_j)} \right\rangle$，其中 $i \neq j$，且使用简记符号 R_{ij} 表示 $R_{(x_i,x_j)}$[9]。

3.2.1 节点一致性

节点一致性算法 (Node Consistency Algorithm) 是最简单的一致性算法。当且仅当 $D_i \subseteq R_i$ 时，我们称值域 D_i 的变量或顶点 X_i 具有节点一致性。如果节点 X_i 不具有节点一致性，那么执行如下操作：

$$\begin{aligned} D_i' &= D_i \cap R_i \\ D_i &\leftarrow D_i' \end{aligned} \tag{3}$$

可以使得节点 X_i 成为节点一致性的。

因此，一次遍历约束图中所有的节点，并进行一致性处理，可以使约束图具有节点一致性：$\mathcal{P}' = \langle X, D', C \rangle$，$D' = \langle D_1', D_2', \cdots, D_n' \rangle$，我们称上述操作为 $\mathcal{P}' = NC(\mathcal{P})$。显然，$sol(\mathcal{P}) = sol(\mathcal{P}')$。令 $\Omega' = \bowtie D_i'$，则 $|\Omega'| \leqslant |\Omega|$。其中，$\bowtie$ 表示关系代数中的联合 (Join) 操作。

节点一致性算法仅仅关注约束图中的 1 元约束。如果 CSP 约束图不满足节点一致性，那么意味着至少存在一个节点 X_i，它的某个实例化值 (不妨令其为 $v^{(i)}$)

[9]为方便起见，我们仅仅讨论有限值域、具有 1 元和 2 元约束关系的 CSP 算法，并且假定对图论 (Graph Theory)、集合论 (Set Theory) 与关系代数 (Relational Algebra) 有一定的认识。

总是会导致回溯失败。换句话说，节点 X_i 的实例化值 $v^{(i)}$ 是冗余的，不会出现在任何解中。因此，可以安全地将 $v^{(i)}$ 去除。

总之，通过运行节点一致性算法，总能消除约束图中的所有 1 元约束。另外，可以将 n 元约束转换成 2 元约束。因此，一般情况下，只定义 2 元约束 CSP 求解器。本章所进行的讨论，都是基于这一认识。

3.2.2 弧一致性

弧一致性算法 (Arc Consistency Algorithm) 使用二元约束进一步对变量值域进行压缩处理。

设节点 X_i 与 X_j 的值域分别为 D_i 与 D_j，在 X_i 与 X_j 之间存在关系 R_{ij}。考虑弧 (X_i, X_j)，我们称弧 (X_i, X_j) 具有一致性，当且仅当：

$$D_i \subset \pi_i \left(R_{ij} \bowtie D_j\right) \tag{4}$$

其中，π 是投影 (Porjection) 算子。它表达的含义是，对于 D_i 的每个元素，在 D_j 中存在相应的元素，它们之间满足关系 R_{ij}。当弧 (X_i, X_j) 不满足一致性时，使用下面的操作可以使弧 (X_i, X_j) 取得一致性：

$$\begin{aligned} D_i' &= D_i \cap \pi_i \left(R_{ij} \bowtie D_j\right) \\ D_i &\leftarrow D_i' \end{aligned} \tag{5}$$

显然，上述操作将删除掉 D_i 中与 D_j 不满足关系 R_{ij} 的那些元素。类似地，对弧 (X_j, X_i) 也可以执行弧一致性检测算法，且在不满足弧一致性约束时执行弧一致性操作，以进一步压缩变量 X_j 的值域 D_j。

从上述讨论可以看出，弧一致性具有方向性，即弧 (X_i, X_j) 满足一致性，并不意味着弧 (X_j, X_i) 也满足一致性。例如，如图3-7所示，弧 $\{NT, Q\}$ 具有一致性，因为其中的一个具体取值 $\{NT = red, Q = green\}$ 满足约束；而弧 $\{Q, NT\}$ 不具有一致性，因为它的具体取值 $\{Q = red, NT = red\}$ 不满足约束。因此，在对节点 Q 执行弧一致性操作时，将删除掉 Q 中的冗余值 red，从而使其值域缩减为 $\{green\}$。

在弧一致性算法中，如果对约束图中的变量 X_i 执行弧一致性操作，那么其值域 D_i 将会发生变化，此时必须重新检测所有弧是否满足一致性：如果不满足，则需要执行弧一致性操作；上述过程，持续进行，直至任意节点的值域没有发生缩

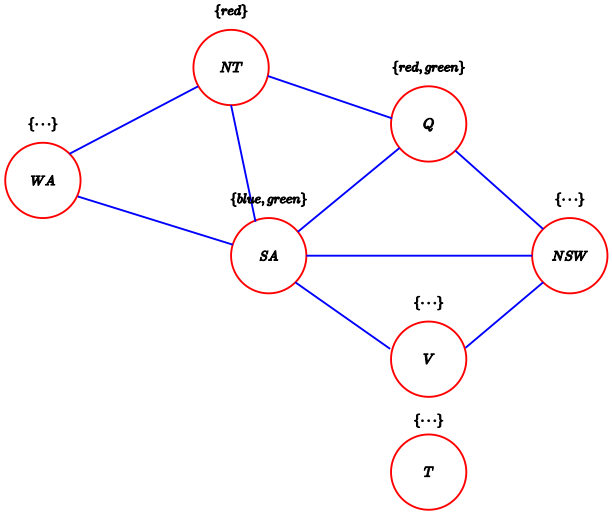

图 3-7: 弧一致性的方向性

减为止，此时约束图达到稳定状态。如果约束图中所有的弧满足一致性，我们称约束图满足弧一致性。例如，在图3-7中，由于 Q 节点的值域缩减为 $\{green\}$，发生了变化，使得弧 $\{SA, Q\}$ 不再具备弧一致性，需要删除节点 SA 中的值 $green$，从而使其值域缩减为 $\{blue\}$。因此，在执行弧一致性操作后，必须要将这种值域缩减变化进行传播，确保整个约束图仍然维持弧一致性。

基本的弧一致性算法简单地检测所有的弧，直至没有进一步的 (变量) 值域缩减，该算法被称为 AC-1 算法。Waltz 意识到，重新检测弧是否满足一致性，只需要检测那些已经受到影响的弧，于是对基本算法进行了改进，得到了 AC-2 算法：将某个变量值域的修改通过弧传播出去。AC-3 算法由 Mackworth 提出，是对 AC-2 算法的泛化与简化，它仍然是广泛使用的、有效的一致性算法。对于弧一致性算法，设 $\mathcal{P}' = AC(\mathcal{P})$，显然有 $sol(\mathcal{P}') = sol(\mathcal{P})$ 和 $|\Omega'| = |\Omega|$ 成立。这表明，一致性操作前后，约束图具有等价性。

我们可以将弧一致性算法看作局部一致性的消解与传播算法：随着消解与传播的迭代，整个约束图的一致性将逐渐地以单调的方式达到一个固定点。此时，约束图取得了一致性，算法即终止。

下面给出 AC-1 算法的伪代码。

算法 **3.2**（*AC-1* 算法）

```
def ArcConsistent(X_i,X_j):
    1. Changed ← False
    2. for each v^(i) ∈ D_i:
    3.     Found ← False
    4.     for each v^(j) ∈ D_j:
    5.         if (v^(i), v^(j)) ∈ R_ij:
    6.             Found ← True
    7.             break
    8.     if not Found:
    9.         D_i ← D_i − v^(i)
   10.         Changed ← True
   11. return Changed

def AC-1():
    1. Arcs ← ∅
    2. for each variable X_i ∈ X:
    3.     for each variable X_j ∈ X such that i ≠ j:
    4.         if R_ij ∈ R:
    5.             Arcs ← Arcs ∪ (X_i, X_j)
    6. while Changed:
    7.     Changed ← False
    8.     for each (X_i, X_j) ∈ Arcs:
    9.         if ArcConsistent(X_i,X_j):
   10.             if D_i = ∅:
   11.                 return False
   12.             Changed ← True
   13. return True
```

AC-1 算法比较低效，每次值域更新后，需要重新检查所有的弧。实际上，只需要重新检查所有与之相关的弧。下面给出 AC-3 算法的伪代码。为简便，只列出算法的改动部分。

算法 **3.3**（*AC-3* 算法）

```
def AC-3():
    1. ⋮
    2. while Arcs ≠ ∅:
    3.     Select and remove (X_i, X_j) from Arcs
    4.     if ArcConsisten(X_i,X_j):
    5.         if D_i = ∅:
    6.             return False
    7.         else:
    8.             for each variable X_k ∈ X such that k ≠ j:
    9.                 if R_ki ∈ R:
   10.                     Arcs ← Arcs ∪ (X_k, X_i)
   11. return True
```

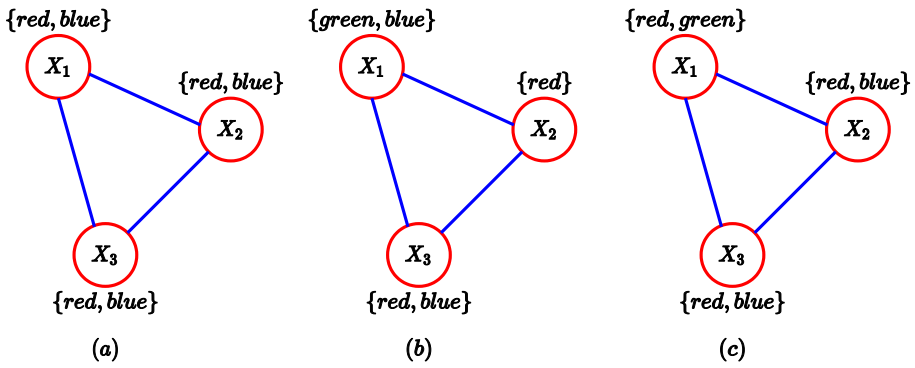

图 3-8: 弧一致性约束图的几种解示例

一般情况下，执行弧一致性算法不会直接得到 CSP 的解，除非下列几种情况：

1. 得到的约束图中存在值域为 $\emptyset$ 的节点，则 CSP 无解；
2. 得到的约束图中每个节点刚好只有 1 个值，则 CSP 仅有 1 个解；
3. 得到的约束图中共有 N 个节点，其中的 $N-1$ 个节点，每个节点刚好只有 1 个值，剩下的 1 个节点有 k 个值，则 CSP 有 k 个解；

在其它情况下，即便得到的约束图满足弧一致性，CSP 可能有解，也可能无解。如图3-8所示，分别展示了几种解示例情形：(a) 图中的 CSP 无解；(b) 图中的 CSP 有 1 个解；(c) 图中的 CSP 有 2 个解。

3.2.3 路径一致性

路径一致性算法 (Path Consistency Algorithm) 进一步增强了变量的局部约束检测能力，将二元约束检测升级为三元约束检测。

设长度为 2 的路径，以节点 X_i 为起点，X_m 为中间节点，X_j 为终节点，则路径 (X_i, X_m, X_j) 是路径一致性的，当且仅当：

$$R_{ij} \subset \pi_{ij}\left(R_{im} \bowtie D_m \bowtie R_{mj}\right) \tag{6}$$

即对于关系 R_{ij} 允许的每对值 (a, b)，在变量 X_m 中，都存在对应的值 c，使得 R_{im} 允许的值对 (a, c) 和 R_{mj} 允许的值对 (c, b) 都存在。

如果路径 (X_i, X_m, X_j) 不具有路径一致性，那么可以通过如下操作使之达到路径一致性：

$$\begin{aligned} R'_{ij} &= R_{ij} \cap \pi_{ij}\left(R_{im} \bowtie D_m \bowtie R_{mj}\right) \\ R_{ij} &\leftarrow R'_{ij} \end{aligned} \tag{7}$$

与弧一致性算法一样，只要约束图中一致性做了修改，那么就需要重新检测整个约束图的一致性，这被称为 PC-1 算法。PC-2 在 PC-1 的基础上做了效率上的改进：只在受到影响的地方进行重新检测。PC-2 算法与 AC-3 算法类似，在此不再赘述。

然而，需要注意的是，与弧一致性算法类似，路径一致性并不能确保直接能够找到 CSP 的解。

3.2.4 k 一致性

k 一致性 (k-Consistency) 可以定义更强的约束传播形式。如果对于 $k-1$ 个变量的任何一致性赋值，第 k 个变量中总存在着 1 个取值，它与这前 $k-1$ 个赋值是一致的，那么我们称 CSP 具有 k 一致性。

由该定义可知，1 一致性指的是空集情况下每个单变量形成的集合是一致性的。因此，1 一致性就是前述的节点一致性。依此类推，2 一致性对应弧一致性，3 一致性对应路径一致性。

如果约束图具有 k 一致性，且同时也具有 $k-1$ 一致性、$k-2$ 一致性、$\cdots$、1 一致性，那么我们称 CSP 具有强 k 一致性 (Strong k-Consistency)。节点一致性与弧一致性，分别对应强 1 一致性与强 2 一致性。

实际上，这给我们提供了一种 CSP 的“求解思路”：如果已知 CSP 具有强 k 一致性，且 $k = n$(其中 n 表示约束图中变量或节点的个数)，那么对变量 X_1，可以选择 1 个一致或相容值；类似地，强 k 一致性可以确保分别为变量 X_2、X_3、$\cdots$、X_n 选择 1 个相容值。由此，就得到了 CSP 的解。但是，上述求解方法的代价太大：建立强 k 一致性约束图的时间与空间代价均为指数级。

因此，在实际应用中，考虑到性能与效率问题，一般只考虑使用弧一致性。少数情况下，也可以考虑使用路径一致性。但是，正如前面指出的那样，单独使用这 2 种类型的约束传播方法，不足以求解出 CSP。另一方面，回溯法具有完备性。因此，可以考虑将回溯法与约束传播结合在一起使用。

3.3 回溯法与约束传播的结合

在前面，分别讨论了回溯法与约束传播及其存在的问题。总体上来说，有 2 种结合方法：

1. 将约束传播作为回溯法的预处理步骤：利用约束传播对约束图中节点的值域进行压缩处理 (例如使其满足弧一致性)，然后再在预处理后的约束图上执行回溯搜索；

2. 将约束传播作为算法组件，嵌入到回溯法框架中；

本节主要讨论第 2 种方法。具体地，将两者结合在一起的基本想法是，在搜索树上，每当访问一个节点时，需要执行一次约束传播算法：压缩相关节点或变量的值域。一旦发现约束传播路径上任何节点的值域为空 (即意味着不可能产生有效的解)，则该节点将被裁减掉。通过执行上述步骤，可以提早发现搜索路径上的死胡同或死路 (Dead End)，因而可以减少前述抖动现象的发生，也能够有效地压缩搜索树的规模，从而降低无效搜索的次数。

FC(Forward Checking) 和 MAC(Maintaining Arc Consistency) 算法都属于这种方法。前者，在引入启发式信息后，被认为是求解 CSP 的最佳算法之一；而后者据认为在大规模且难以求解的 CSP 上表现更佳。两者的主要差别在于约束传播的范围：FC 算法只执行部分的弧一致性操作；而 MAC 算法执行完全的弧一致性操作，且将弧一致性操作作为回溯搜索前的预处理步骤。下面，分别给出 FC-3 算法与 MAC-3 算法。

算法 **3.4**（*FC-3* 算法）

```
def DomainWipedOut(X_i):
    1. for each value v^(i) ∈ D_i:
    2.     if v^(i) is not marked:
    3.         return False
    4. return True

def RestoreDomainAtLevel(Vars, Level):
    1. for each variable X_i ∈ Vars:
    2.     for each value v^(i) ∈ D_i:
    3.         if v^(i) is marked at Level:
    4.             Set v^(i) as unmarked

def ForwardChecking(Vars, Level, X_i, v^(i)):
    1. for each variable X_j ∈ Vars:
    2.     if R_ij ∈ R:
    3.         for each value v^(j) ∈ D_j such that v^(j) is not marked:
    4.             if (v^(i), v^(j)) ∉ R_ij:
    5.                 Set v^(j) as marked at Level
    6.         if DomainWipedOut(X_j):
    7.             return False
    8. return True

def BacktrackingWithForwardChecking(Vars, Level):
    1. Select a variable X_i ∈ Vars
    2. for each value v^(i) ∈ D_i such that v^(i) is not marked:
    3.     Solution ← Solution + (X_i, v^(i))
    4.     if X_i is the only variable in Vars:
    5.         return True
    6.     else:
    7.         if ForwardChecking(Vars\{X_i}, Level, X_i, v^(i)) and
    8.            BacktrackingWithForwardChecking(Vars\{X_i}, Level + 1)
    9.             return True
   10.         else:
   11.             Solution ← Solution − (X_i, v^(i))
   12.             RestoreDomainAtLevel(Vars\{X_i}, Level)
   13. return False

def FC-3():
    1. for each variable X_i ∈ X:
    2.     for each value v^(i) ∈ D_i:
    3.         Set v^(i) as unmarked
    4. Solution ← ∅
    5. return BacktrackingWithForwardChecking(X, 1)
```

下面介绍 FC-3 算法中各部分的作用：

- 函数 DomainWipedOut 用于判断给定节点的值域是否为空。如果节点的某

个取值被标记了 (Marked)，意味着该取值因不满足约束而被“删除”。需要注意的是，使用二维数组来保存每个变量的每个取值的标记状态；程序伊始，将其初始化为“无标记”或“非删除”状态，表示取值有效，即取值可能出现在解 (Solution) 中。另外，除了做普通标记外，还可以使用递归层次 *Level* 对取值做标记，参见函数 ForwardChecking；清除标记时，也将依据 *Level* 清除指定递归层次的标记，参见函数 RestoreDomainAtLevel。

- 函数 RestoreDomainAtLevel 用于清除指定变量集或节点集[10]中位于层次 *Level* 的那些具体取值的标记，即去掉“删除”标记，恢复为正常标记。

  从全局来看，当变量实例化失败时，该函数将抹去本次约束传播算法对变量取值作标记时产生的痕迹，确保后续回溯与约束传播仍然可以正常地工作。

- 函数 ForwardChecking 从指定的节点 X_i 出发，按取值 $v^{(i)}$ 执行边约束检查[11]：一旦发现某个节点的值域为空，表明该节点为不相容节点，因而进一步表明此条路径不可能产生解，则直接返回 $False$，提醒主调函数，已经到达死胡同，需要执行回溯；否则，所有节点检测完毕，均为相容节点，则对所有这些相容节点的不相容取值做上标记，最后返回 $True$，表明可进一步递归搜索。

- 递归函数 BacktrackingWithForwardChecking 以回溯法作为基本框架，结合函数 ForwardChecking(简单的约束传播)，在搜索树上执行较为高效的搜索，完成 CSP 的求解。

- 函数 FC-3 将所有变量的所有取值初始化为“无标记”(表示这些取值都被“保留”或“不删除”)，并将解集初始化为空，最后调用递归函数 BacktrackingWithForwardChecking，求解 CSP。

算法 **3.5**（*MAC-3* 算法）

[10]注意，这些节点属于还未被访问或实例化的节点。在递归函数 BacktrackingWithForwardChecking 中，对函数 RestoreDomainAtLevel 的调用参数进行简单的分析，就可以得出这一结论。

[11]与算法3.1对比，可以发现，在 CSP 回溯法中，函数 Consistent 在当前变量与所有已访问变量之间检测约束是否满足。而在本算法 (FC-3) 中，函数 ForwardChecking 在当前变量与所有未被访问 (相邻) 变量之间检测约束是否满足。下文即将介绍的 MAC-3 算法也有类似特点——只不过，MAC-3 执行双向的弧一致性检测，而 FC-3 只执行单向的弧一致性检测。

```
def Revise(X_i, X_j, Level):
    Deleted ← False
    for each v^(i) ∈ D_i such that v^(i) is not marked:
        Found ← False
        for each v^(j) ∈ D_j such that v^(j) is not marked:
            if (v^(i), v^(j)) ∈ R_ij:
                Found ← True
                break
        if not Found:
            Set v^(i) as marked at Level
            Deleted ← True
    return Deleted

def MaintainingArcConsistency(Vars, Level):
    while Arcs ≠ ∅:
        Select and remove (X_i, X_j) from Arcs
        if Revise(X_i, X_j, Level):
            if DomainWipedOut(X_i):
                return False
            else:
                for each variable X_k ∈ Vars such that k ≠ j:
                    if R_ki ∈ R:
                        Arcs ← Arcs ∪ (X_k, X_i)
    return True

def BacktrackingWithMAC(Vars, Level):
    Select a variable X_i ∈ Vars:
    for each value v^(i) ∈ D_i such that v^(i) is not marked:
        Solution ← Solution + (X_i, v^(i))
        if X_i is the only variable in Vars:
            return True
        else:
            % Eliminate all the other values of X_i
            for each value v_*^(i) ∈ D_i\{v^(i)} such that v_*^(i) is not marked:
                Set v_*^(i) as marked at Level
            for each variable X_j ∈ Vars:
                if R_ji ∈ R:
                    Arcs ← Arcs ∪ (X_j, X_i)
            if MaintainingArcConsistency(Vars\{X_i}, Level) and
              BacktrackingWithMAC(Vars\{X_i}, Level + 1)
                return True
            else:
                Solution ← Solution − (X_i, v^(i))
                RestoreDomainAtLevel(Vars, Level)
    return False
```

```
def MAC-3():
  1. for each variable X_i ∈ X:
  2.     for each value v^(i) ∈ D_i:
  3.         Set v^(i) as unmarked
  4. Solution ← ∅
  5. Arcs ← ∅
  6. for each variable X_i ∈ X:
  7.     for each variable X_j ∈ X such that i ≠ j:
  8.         if R_ij ∈ R:
  9.             Arcs ← Arcs ∪ (X_i, X_j)
 10. if MaintainingArcConsistency(X, 0)
 11.     return BacktrackingWithMAC(X, 1)
 12. else:
 13.     return False
```

下面介绍 MAC-3 算法中各部分的作用：

- 函数 Revise 对节点对 (弧) 中的第 1 个节点进行标记处理：如果两节点不一致，则将第 1 个节点的相应取值，标记为无效。

- 函数 MaintainingArcConsistency 执行完全且双向的弧一致性检测，对变量的无效取值做标记。

- 递归函数 BacktrackingWithMAC 在回溯法框架的基础上，嵌入了 MAC 约束传播组件。在第 8-9 行，仅仅考虑 X_i 的当前取值，不考虑其它取值：它们被标记为无效。如果当前取值无法找到解 (第 13-14 行返回 $False$)，则第 18 行将恢复其它取值，继续搜索。

- MAC-3 函数首先将所有变量的所有取值形成的二维标记数组初始化为“无标记”，即全部有效；然后，根据节点间关系，构建 (双向) 弧集合，用于后期的弧一致性检测；接着，调用函数 MaintainingArcConsistency 执行弧一致性预处理，如果有解，则继续调用递归函数 BacktrackingWithMAC，求解 CSP；否则，返回 $False$，CSP 无解。

表3-1展示了 Backtracking、FC-3 与 MAC-3 算法在 4 皇后问题上的表现。从该表可以看出，FC-3 算法访问的节点要多于 MAC-3，而 MAC-3 算法在每个节点上所执行的约束检测要远远多于 FC-3 算法。实际上，MAC-3 所执行的约束检测更完全，可以提早发现搜索路径上的无效节点，即节点值域为空的情形 (Domain

表 3-1: 算法性能比较：BT、FC-3 与 MAC-3

Algorithm	Constraint Checks	Nodes Visited	Checks per Node
BT	36	27	1.33
FC	38	9	4.22
MAC	138	6	23.00

Wiped Out)，因而访问的节点个数要少于 FC-3 算法。这就解释了为什么 MAC-3 算法在大规模且难以求解的 CSP 上表现要比 FC-3 算法更好。

然而，尽管如此，将回溯法与约束传播相结合的混合算法，必须在访问节点的个数与节点上执行的约束检测次数之间取得某种平衡——需要考虑两者整体上的时间代价。关于混合算法的最新进展，请阅读相关论文。

4 练习

1. 编程实现 8 皇后问题；

2. 编程实现图2-5所展示的 9×9 数独游戏；

5 附录

5.1 回溯法：找出所有解

算法 **5.1**（回溯法：找出所有解）

```
def Consistent(X_i,v^(i)):
    1. for each (X_j,v^(j))∈ Solution:
    2.     if R_ij ∈ R and (v^(i), v^(j)) ∉ R_ij:
    3.         return False
    4. return True

def Backtracking(Vars):
    1. Select a variable X_i ∈ Vars
    2. for each value v^(i) ∈ D_i:
    3.     if Consistent(X_i, v^(i)):
    4.         Solution ← Solution + (X_i, v^(i))
    5.         if X_i is the only variable in Vars:
    6.             Solutions ← Solutions + Solution
    7.         else:
    8.             Backtracking(Vars\{X_i})
    9.         Solution ← Solution − (X_i, v^(i))

def CSP-BT():
    1. Solutions ← ∅
    2. Solution ← ∅
    3. return Backtracking(X)
```

6 参考文献

1. Constraint Satisfaction Problem. https://en.wikipedia.org/wiki/Constraint_satisfaction_problem.

2. Stuart J. Russell, Peter Norvig, 殷建平等译。《人工智能：一种现代的方法》，第 3 版，清华大学出版社。

3. Constraint Satisfaction: An Emerging Paradigm. https://www.cs.ubc.ca/ mack/Publications/HCP-FM-06.pdf. Handbook of Constraint Programming.

4. Zhe Liu. Algorithms for Constraint Satisfaction Problems, A Thesis for Master of Mathematics in Computer Science, University of Waterloo, 1998. http://www.cs.toronto.edu/~fbacchus/Papers/liu.pdf.