北京大学

约束满足问题

Constraint Satisfaction Problems

·AIPKU·

内容来自：Artificial Intelligence A Modern Approach

Stuart Russel　Peter Norvig　Chapter 6

主讲人：李文新

引子

- 计算机产生的初衷是用来求解问题的
- 计算机的特点：数据是离散的（0-1），整数、实数
- 计算机解决问题的一种思路：枚举 + 验证 - 在解空间搜索
- 一些实际问题的例子：给出一些变量及其取值范围和约束关系，寻找一组变量赋值满足所有约束
  - 寄存器分配问题：将大量的程序变量合理分配给少量寄存 器，以提高程序执行效率。
  - 旅行商问题：给定城市列表和城市间距离，找到一条回 路访问每座城市恰好一次并返回起点，使得移动路线最短。
  - 组合拍卖的赢家确定问题：给定拍卖中的出价集合，将各拍卖品合理分配给出价者，使得拍卖者的收益最大化。
  - 绿色物流规划问题：根据供需关系、运输能力等，合理规划 物流活动中的运输、存储、配送等环节，降低物流成本，同时通过降低能源消 耗和废物排放来减少对环境的污染。
  - 地下水流速估计问题：给定地下水道网络结构，水流入口和出口等信息，估计每段区间水流的流速。
  - 基因组映射问题：给定一组基因片段及它们之间的距离约束，要求在一段 *DNA* 序列中定位这些基因片段。

内容提要

1. 定义约束满足问题（CSP）Constraint Satisfaction Problem
   - 地图着色问题、作业调度问题、CSP的形式化
2. 约束传播：CSP中的推理
   - 结点相容、弧相容、路径相容、k-相容、全局约束、数独
3. CSP的回溯搜索
   - 变量和取值顺序、搜索与推理交错进行、智能回溯：向后看
4. CSP局部搜索
5. 用PDL语言解决问题

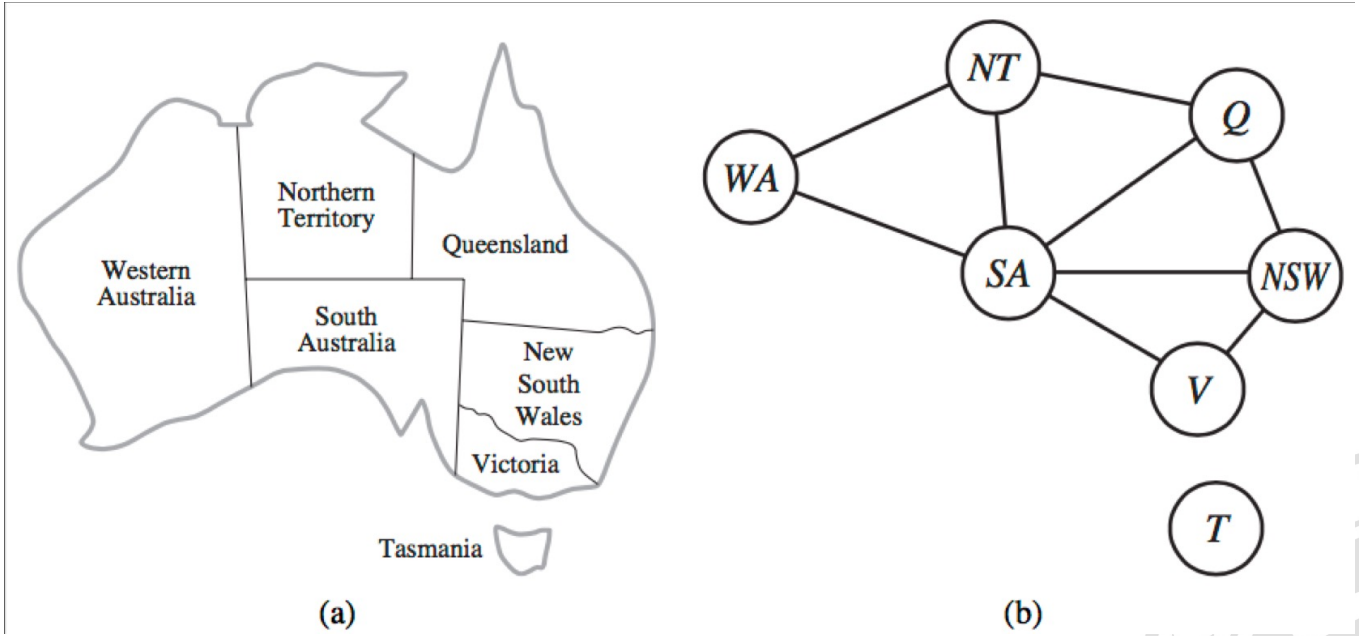
6.1 定义约束满足问题

- 一个约束满足问题有三个组成部分，X, D 和 C：
  - X 是一组变量，$\{X_1,\ldots,X_n\}$。
    D 是一组值域，$\{D_1, \ldots, D_n\}$，每个变量一个值域。
    C 是一组约束 用于指定变量取值之间的约束关系。
- 每个值域 D_i 包含一组对于变量X_i可行的取值 $\{v_1,\ldots,v_k\}$ 。每一个约束条件 C_i 包含一对数值$\langle scope, rel \rangle$，$scope$ 是约束条件中涉及的变量， rel 定义了变量的可行取值之间的关系。

6.1 定义约束满足问题

- 一个**关系**可以表示成明确列出的所有满足条件的取值组合，或者是一个抽象的关系。例如，如果X1 和 X2 的值域都是 {A,B}，而约束条件是 两个变量的取值不能相同。可以写作：⟨(X1, X2), [(A, B), (B, A)]⟩ 也可以写作⟨(X1, X2), X1 <> X2⟩。
- 为了解决一个约束满足问题，我们需要定义一个状态空间，以及一个解的表达。在CSP问题中，每个状态定义为 一些或全部变量的一组赋值，$\{X_i = v_i, X_j = v_j, \ldots\}$.
- 一组赋值如果不违背任何一个约束条件，则称为**一致的**或者是**合法赋值**。
  - 所有变量都被赋值了成为一个**完全赋值**，
  - 一个CSP问题的**解**是**一致的和完全的**。
  - **部分赋值**是指仅仅给一部分变量赋了值。

6.1.1 例1: 地图着色问题



- （a）澳大利亚简要地图 如何给这张地图着色可以看做一个约束满足问题，目标是相邻的州不使用同种颜色 （b）用一个抽象的约束图来描述这个地图着色问题

6.1.1 例1: 地图着色问题

- 我们的任务是用红、绿、蓝给地图的各个区域着色，要求是相邻的区域不能用相同颜色。把这个问题建模成一个约束满足问题（CSP） 我们先把区域定义成变量
- $X = \{WA, NT, Q, NSW, V, SA, T\}$.
- 每一个变量的值域是 $D_i = \{\text{red}, \text{green}, \text{blue}\}$. 约束要求相邻的区域使用不同颜色。因为有9条边连接相邻的区域，所以有9条约束 条件：

$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V,$$
$$WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\} .$$

6.1.1 例1: 地图着色问题

- 为什么要把问题建模成**约束满足问题CSP**呢？**原因一**是CSP问题可以很自然地表示很宽泛的类别的问题；如果你有一个通用的CSP求解器，你就可以无需设计一个专门的解决方案而很容易地得到问题的解。
- **此外**，CSP求解器会比基于状态空间的搜索算法更快，因为 CSP求解器可以迅速排除大量的无关搜索空间。例如，在地图着色问题上当我们选择 {***SA* = blue**}, 我们就可以肯定它的5个邻居都不能着蓝色。如果没有用到约束条件传播的好处，搜索需要为五个邻居考虑 $\mathbf{3^5 = 243}$ 种着色方案；但如果考虑了约束传播，我们只需要考虑 $\mathbf{2^5 = 32}$ 种着色方案减少了 87%。

6.1.1 例1: 地图着色问题

- 在一个传统的空间搜索问题中我们只能问： 这个状态是目标吗？不是? 那这个呢?
- 在CSP 问题中, 当我们发现一个部分赋值不是解，我们会立即放弃对这个部分赋值的后续探索。
- 不仅如此，我们还能够知道为什么某个赋值不是解 — 我们可以知道是哪个变量的赋值违背了某个约束 — 所以我们可以更加集中注意力在那些更关键的变量上。
- 总之，许多使用传统搜索方法不能出解的问题，建模成CSP问题可以很快出解。

6.1.2 例2: 作业规划问题

- 工厂需要解决每天规划要完成的任务的问题，其中有各种需要满足的约束条件。
- 实际中，许多这类问题是用 CSP 技术解决的。
- 让我们来看一个规划汽车装配任务的问题。整个任务包括许多个子任务，我们可以如下建模：
  - 每个子任务用一个变量表示, 每个变量记录的是该任务的起始时间，用一个整数表示，精确到分的时间
  - **约束 Constraints** 是某项任务必须在另一项任务之前完成。例如： 一个汽车轮子必须在安装轮毂[gǔ]罩之前装好 。约束也可以是某项任务的完成时长。

6.1.2 例2: 作业规划问题

- 我们考虑汽车装配的一小部分工作, 包含15个任务: 装车轴 install axles (front and back), 装4个轮子affix all four wheels (right and left, front and back), 拧紧每个轮子的螺丝帽 tighten nuts for each wheel,装轮毂罩affix hubcaps, 最后检查inspect the final assembly。我们用 15 个变量:
- $\boldsymbol{X} = \{Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF}, Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\}$.
- 变量的值是每项任务的起始时间。下面我们表示任务之间的前后顺序约束。当任务 $\boldsymbol{T_1}$ 必须在任务 $\boldsymbol{T_2}$ 之前完成, 并且任务 $\boldsymbol{T_1}$ 需要 $\boldsymbol{d_1}$ 分钟完成,时 我们增加一个约束条件 $\boldsymbol{T_1 + d_1 \leq T_2}$。

6.1.2 例2: 作业规划问题

- 在我们的例子中, 轴要在轮子之前安装, 装轴需要 10 分钟, 我们写成
  - $Axle_F + 10 \le Wheel_{RF}$; $Axle_F + 10 \le Wheel_{LF}$;
  - $Axle_B + 10 \le Wheel_{RB}$; $Axle_B + 10 \le Wheel_{LB}$.
- 接下来，对于每个车轮，我们必须装上轮子 (耗时 1 分钟), 拧紧螺母 (2 分钟), 最后装轮毂罩(1 分钟，没有表示出来):
  - $Wheel_{RF} + 1 \le Nuts_{RF}$; $Wheel_{LF} + 1 \le Nuts_{LF}$;
  - $Wheel_{RB} + 1 \le Nuts_{RB}$; $Wheel_{LB} + 1 \le Nuts_{LB}$;
  - $Nuts_{RF} + 2 \le Cap_{RF}$; $Nuts_{LF} + 2 \le Cap_{LF}$;
  - $Nuts_{RB} + 2 \le Cap_{RB}$; $Nuts_{LB} + 2 \le Cap_{LB}$.

6.1.2 例2: 作业规划问题

- 假设我们有4个工人装轮子，所以轮子可以同时进行。 但是他们需要共享一个工具来装轴。我们需要一个离散的约束来表示前轴和后轴的时间不能重叠：
- $(\mathbf{Axle_F} + 10 \leq \mathbf{Axle_B})$ or $(\mathbf{Axle_B} + 10 \leq \mathbf{Axle_F})$.
- 这个限制条件看上去有些复杂，结合了算术运算和逻辑运算。但是它还是有效减少了$\mathbf{Axle_B}$ 和$\mathbf{Axle_F}$的取值范围。

6.1.2 例2: 作业规划问题

- 我们也要声明检查（ **inspection** ）的工序在最后，持续3分钟。 对于除了检查以外的变量，我们加一个约束条件
- $X + d_X \leq Inspect$ 。
- 最后，我们还有一个要求，所有任务必须在***30 分钟***内完成。我们可以通过给出所有变量的可能取值的方式给出此约束：
- $D_i = \{1,2,3,\ldots,27\}$.
- 我们看到的只是一个很小的问题，事实上 CSP技术已经被用来解决像复杂作业调度这样的拥有数千个变量的复杂问题。

6.1.3 CSP模型的可变因素 - 变量的类型

- **CSP问题**是用**X、D、C**三元组表示的，X是变量，D是取值范围。

1. **变量X**，它的类型可以是**离散的**，也可以是**连续的**；
2. **值域D**，它可以是**有限的**，也可以是**无限的**；

- 最简单的CSP问题的**变量类型是离散的**、**值域有限的**。地图着色问题和有限时间工序安排问题都属于**简单CSP问题**。 8皇后问题也可以看做**有限值域**的 CSP问题, **Q1,…,Q8** 是每个皇后在各自列上所在行的位置。每个变量的取值范围是D_i = {**1,2,3,4,5,6,7,8**}。离散类型变量的取值也可以是**无限的**，例如整数，或者是字符串。
- **连续值域的约束满足问题的例子。例如，** 哈勃望远镜（**the Hubble Space Telescope**） 观测时间需要非常精确的；观测开始和结束的时间，是在连续值域上取值的变量，并且需要遵循一系列天体、先决条件和电力的限制。**连续值域的约束满足问题**在现实世界里是很常见的，也是被广泛研究的。

6.1.3 CSP模型的可变因素 - 约束条件的类型

3.约束条件C。

①用C所涉及到的**变量个数**来对它分类。

- **一元约束（unary constraint）**，对单个变量取值的约束。例如，在地图着色问题中，南澳大利亚不能使用绿色，我们可以用如下的一元约束表示 ⟨(SA),SA /= green⟩
- **二元约束**（**binary constraint**）涉及到两个变量。 例如, SA /= NSW 是一个二元约束。一个**二元CSP问题**只有二元约束，可以表示成约束关系图。
- **三元约束，** 例如，变量 Y 的取值在变量 X 和 Z 之间，表示成**三元约束**（ternary constraint），**Between(X, Y, Z)**.
- **全局约束**（**global constraint**）含有**任意多个变量**。最常见的**全局约束**是 **Alldiff**, 其含义是所有变量的取值必须互不相同。在 **Sudoku** 数独问题中, 在同一行或者同一列或者同一个小方块内的数字必须互不相同。

6.1.3 CSP模型的可变因素 - 优先约束

②用C是否是强制性的来分类

- **绝对约束**，违背这种约束就意味着排除了成为解的可能性。
- **优先约束**，用来指定哪些解是优先选择的。例如，**R** 教授 可能会更倾向于上午。那么如果安排 R 教授 下午2点上课依旧是可行的，但不是最优选择。**优先约束（Preference constraints）**一般会实现成给每个变量的赋值增加一个花费（**costs**），例如，为**R**教授安排下午的课花费为 2，而安排上午的课花费为 1。这样，有优先约束的 CSP问题可以用优化搜索方法求解 - 全局或局部搜索。我们称这类问题为约束优化问题（**constraint optimization problem**），简称**COP**。线性规划问题属于这类问题。

内容提要

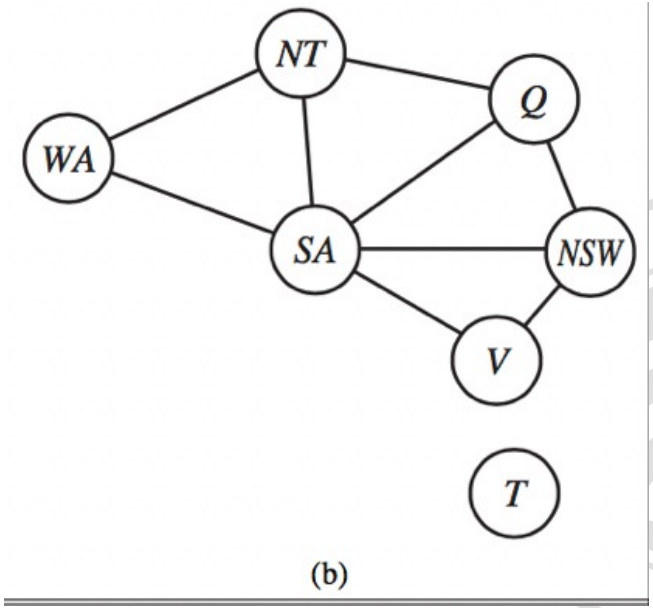
1. 定义约束满足问题（CSP）
   - 地图着色问题、作业调度问题、CSP的形式化
2. 约束传播：CSP中的推理
   - 结点相容、弧相容、路径相容、k-相容、全局约束、数独
3. CSP的回溯搜索
   - 变量和取值顺序、搜索与推理交错进行、智能回溯：向后看
4. CSP局部搜索
5. 用PDL语言解决问题

6.2 约束传播： CSP问题的推理

- **约束传播：CSP中的推理**
- 在普通的状态空间搜索中，我们能做的事情只有搜索。
- 在 CSP问题中，除了搜索还可以有一个选择就是推理（我们也称之为约束传播）： 使用约束减小一个变量的可能取值, 由此还可以减少其他变量的可能取值。约束传播可以和搜索相结合，也可以先用约束做预处理，再搜索。
- 有时预处理就可以解决全部问题，不用再搜索了。

6.2 约束传播： CSP问题的推理

- 主要思想是局部一致性。如果我们将每个变量表示成图上的一个点，每个二元关系是一条边, 那么在图上的每个局部保证一致性会使得全图中不一致的取值都被剔除。有不同种类的局部一致性，我们接下来会逐一讲解。



6.2.1 点一致

- 一个单独的变量(对应CSP网络中的一个点) 是点一致的（**node-consistent**）如果变量的所有取值都满足一元约束。
- **例如**，在变种的澳大利亚地图着色问题中，如果南澳大利亚不喜欢绿色，变量 **SA** 从取值域 {red, green, blue}出发, 我们可以通过删除绿色使它变成点一致的的，如此 **SA** 的取值域就缩减成 {red, blue}。如果所有点都是点一致的，我们称这个网络是**点一致**的。
- 在CSP问题中，我们可以通过**运行点一致性算法**去除所有的一元约束。

6.2.2 边一致

- 一个CSP中的变量是边一致的，如果该变量的所有取值都满足二元约束。变量 X_i 相对于变量 X_j 是边一致的，如果 D_i 中的所有取值都能在 D_j 中找到对应的值使得边 (X_i, X_j) 满足二元约束。
- **一个网络是边一致的，如果每个变量相对于其他任意变量都是边一致的。**
- 例如, 考虑约束 $Y = X^2$ ， **X** 和 **Y** 都是一位数字。 我们可以把约束明确地写成
- **⟨(X, Y), {(0, 0), (1, 1), (2, 4), (3, 9))}⟩** .
- 为了使 X 相对于 Y 边一致, 我们把 X的取值域缩小成 {0,1,2,3}。如果我们也使 Y 相对于X 边一致，则 Y 的值域变成 {0, 1, 4, 9} ，如此整个 **CSP 问题就变成边一致的了。**

·AIPKU·

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

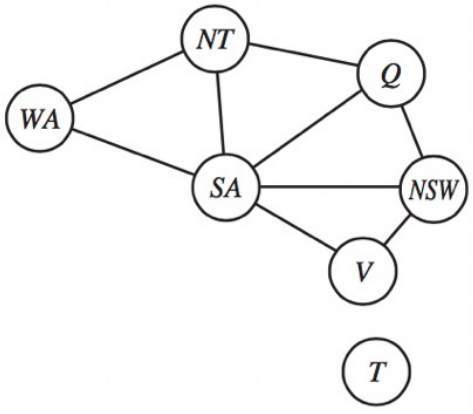
Figure 6.3 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it's the third version developed in the paper.

6.2.2 边一致

- **下面我们分析一下 AC-3 算法的复杂度**
- 假设 **CSP** 问题有 **n** 个变量, 每个变量的取值个数最多为 **d**, 一共有 **c** 个二元约束(边)，则每条边 (X_k,X_i) 被插入队列的次数为 **d** ， 因为 X_i 最多有 **d** 个取值能够被删除。检查一条边的一致性可以在 $O(d^2)$ 时间内完成, 所以在最坏情况下我们的时间复杂度是 $O(cd^3)$ 。
- 运行了**AC-3** 算法之后，我们就得到了一个和原问题等价的问题，也就是和原问题具有相同的解的问题，但是变量的取值范围被大大缩小了，所以解决起来会更快。

6.2.3 路径一致

- 再来考虑澳大利亚地图着色问题，但这次只有两个颜色，红和蓝。边一致性检查什么都不会做，因为无论我们给一个点指定什么颜色，另一个点都会去另外一种颜色，所以一个值都不会被删掉。这貌似问题是有解的。但是再进一步考虑，Western Australia, Northern Territory 和 South Australia 彼此都相连，WA选一种颜色，NT选另一种颜色，SA就没有颜色可选了。所以除了边一致，我们需要考虑更多。



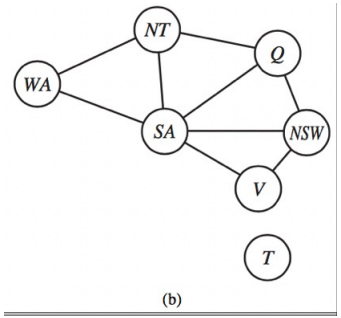
(b)

6.2.3 路径一致

- 为了解决图着色问题，我们需要更强的一致性检查。路径一致使用**三元约束。**
- 我们称二元约束{X_i,X_j} 是相对于变量X_m路径一致的，如果 对于任意一个满足二元约束{X_i , X_j } 的赋值 {X_i = a, X_j = b} 都能找到 X_m 的一个赋值满足约束 {X_i , X_m } 和 {X_m , X_j }。这叫做路径一致是因为我们可以想象成寻找一条从 X_i 中间经过 X_m 去到 X_j 的路径。

6.2.3 路径一致

- 让我们再来看看澳大利亚地图着色问题，我们设置 {**WA, SA**} 相对于 **NT** 是路径一致的。
- 我们来枚举一致的赋值情况。上述例子中只有两种情况: {**WA = red, SA = blue**} 和 {**WA = blue, SA = red**}.
- 对于上述两种赋值 **NT** 都不能是 red 或者 blue （否则它就与 **WA** 或者 **SA** 冲突）。 因为没有合法的取值给 **NT**，我们就把上述两种情况都删除，并得到 {**WA, SA**}没有合法赋值的结论。所以我们得知此问题无解的结论。
- **PC-2** 算法 (Mackworth, 1977) 实现了**路径一致性**检查，其做法与 **AC-3** 实现**边一致性**检查类似。



6.2.4 *K*-一致性

- 更强的约束传播是 k-**一致性检查。**
- 一个 CSP问题是 **k-一致的，如果**对于任意 **k − 1** 个变量，以及对这些变量的满足约束的赋值，我们总能找到一个一致的赋值给第 **k**个变量。
- **1-一致性** 即是**点一致性。**
- **2-一致性** 即是**边一致性。**
- **3-一致性** 即是**路径一致性。**
- 一个 CSP问题是**强 k-一致的**，如果它是 k-一致的，也是 (k − 1)-一致的, (k − 2)-一致的, . . . 直到1-一致的。

6.2.4 *K*-一致性

- 现在我们假设有个CSP问题有 **n** 个点，并且是**强n-一致的** 。我们就可以按照如下方法求解它：首先，我们为$\mathbf{X_1}$ 选择一个一致的赋值。然后我们为$\mathbf{X_2}$ 选择一个一致的赋值。我们一定能够找到因为这个问题是 2-一致的, 类似地 $\mathbf{X_3}$ …… 对于每个变量 $\mathbf{X_i}$, 我们需要在 **d** 个取值里找到一个与 $\mathbf{X_1, \ldots, X_{i-1}}$ 一致的取值。我们可以在$\mathbf{O(n^2d)}$ 时间内找到问题的解.
- 当然了，没有免费的午餐: 所有要使问题成为 **n-一致的** 算法在最坏情况下都是指数时间的。**更糟的是** n-一致性检查算法需要 相对于**n** 的指数空间。 空间问题比时间问题更严重。在实际问题中，一致性检查进行到什么程度是一门实验科学。通常实践者检查到 **2-一致** ，少数时间检查到 **3-一致**.

6.2.5 全局约束

- **全局约束**（**global constraint**）是包含**任意多个**变量的约束 (不一定要包含所有变量).
- 全局约束在现实问题中很常见，一般使用专用算法比通用算法更有效。
- 例如，约束**Alldiff** 表示所有变量必须具有不同的取值 。
- 一个简单的检查 **Alldiff** 约束的不一致性的算法是：如果约束中包含 **m 个变量**, 并且他们总共有 **n** 个可能取值, 并且 **m > n,** 那么就可以下结论这个约束不能被满足。

6.2.5 全局约束 - Alldiff

- 这就衍生了如下算法:
- 首先, 去除所有只有单一取值的变量, 在余下的值域里去除该变量的取值；
- 重复这一过程直到没有单一取值的变量了；
- 如果遇到值域为空，或者变量数多于取值数时，我们就检测到了不一致。

6.2.5 全局约束- Alldiff

- 这个方法可以检测到前面例子 {**WA = red , NSW = red** } 中的不一致。 变量 **SA, NT,** 和 **Q** 之间存在**Alldiff** 约束，因为两两之间要用不同的颜色。
- 在使用 **AC-3** 进行部分赋值后，值域缩减成 {**green, blue**}. 此时我们有三个变量，却只有 2 种颜色，所以 **Alldiff** 约束被打破。
- **所以一个简单的高阶一致性检查程序要比使用AC系列算法对等价的一组二元约束作检查高效得多。**

6.2.5 全局约束 - Atmost

- 另一个高阶约束是资源约束，也称为**atmost** 约束；
- 例如，在调度问题中，$\mathbf{P_1, \ldots, P_4}$ 表示分派给4个任务的人员数。如果分派人员的总数不超过 **10** 个，可以写成如下约束：**Atmost(10,P_1,P_2,P_3,P_4)**。我们可以通过简单地把当前各变量取值域的最小值加起来就可以检测不一致性；
- 例如，如果每个变量的值域是 {**3,4,5,6**}, 则 Atmost 约束不能被满足。如果一个变量的值域和另一个变量的最小值冲突，我们可以通过删除值域里的最大值来强行保证一致性。在上面的例子里，如果每个变量的值域都是 {**2, 3, 4, 5, 6**}, 那么 **5** 和 **6** 可以从所有值域里去掉。

6.2.5 全局约束 – 边界传播

- 对于整数数值型大规模资源受限问题 — 例如把上千人分派到上百辆车里的问题 — 通常我们不太可能为每个变量列出一大批整数，然后逐渐去减少它们。
- 于是，我们把值域描述成上界和下界，然后使用边界传播。
- 例如，在一个航班调度问题中，假设有两架飞机 **F1** 和 **F2**, 分别能装 **165人** 和 **385人**。每架飞机上能装的人数初始为
- **D1 = [0,165] 和 D2 = [0,385]** .

6.2.5 全局约束 – 边界传播

- 现在假设我们有了一条新的约束：两架飞机一共至少要装 **420** 人。**F1 + F2 = 420**。
- 利用边界约束的传播, 我们把两个域缩减成 **D1 = [35, 165] 和 D2 = [255, 385]** . 因为如果**F1**装**34**人以下，**F2**即使装满总人数也到不了**420**，
- 我们称一个 CSP 问题是边界一致的，如果对于所有变量 **X** 的上界和下届，都能找到相应的 **Y**，满足 **X** 和 **Y** 之间的约束；同样对于所有 **Y** 的取值，也有同样的性质。在实际的约束满足问题中， 这种边界传播很常用。

6.2.6 例3：Sudoku 数独问题

- 数独问题向千百万人介绍了约束满足问题，尽管有些人还没有意识到。一个数独问题有 81 个小方格，一些方格在初始时已经填上了1 到 9 的数字。问题的目标是要在所有空着的格子里填上 1 到 9 的数字。使得任何行、列和 3 × 3 的方格中的数字都不重复。一个行、列或者3 × 3小方格称为一个单元。

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Figure 6.4 (a) A Sudoku puzzle and (b) its solution.

6.2.6 例3：Sudoku 数独问题

- 印在报纸和各类杂志上的数独问题都是有唯一解的问题。
- 一些问题可以通过数十分钟的手算得到答案，然而最难的问题通过一个CSP求解器也可以在0.1秒内出解。
- 数独问题可以表示成有81个变量（每个小方格一个变量）的CSP问题。我们用 **A1** 到 **A9** 表示最上边一行从左到右的方格， **I1** 到 **I9** 表示最下面的方格，中间的以此类推。
- 空白的方格的取值域是 **{1, 2, 3, 4, 5, 6, 7, 8, 9}** ，已填充的方格的取值域只有一个数值。
- 另外，有27 个不同的 **Alldiff** 约束: 每行、每列、每个3*3的方格各有一个。

6.2.6 例3：Sudoku 数独问题

- Alldiff (A1, A2, A3, A4, A5, A6, A7, A8, A9)
- Alldiff (B1, B2, B3, B4, B5, B6, B7, B8, B9)
- …
- Alldiff(A1,B1,C1,D1,E1,F1,G1,H1,I1)
- Alldiff(A2,B2,C2,D2,E2,F2,G2,H2,I2)
- …
- Alldiff (A1, A2, A3, B1, B2, B3, C1, C2, C3)
- Alldiff (A4, A5, A6, B4, B5, B6, C4, C5, C6)
- …

6.2.6 例3：Sudoku 数独问题

- 考虑下图中变量**E6**，2 和 8 之间的空格。根据在3*3方格内的约束，我们不仅可以排除 **2** 和 **8** ，还可排除 **1** 和 **7** 。根据列约束，我们可以排除 **5, 6, 2, 8, 9, 3**。这样 E6 的值域变为 {**4**}; 由此我们已经找到了 E6 的解。
- 现在考虑变量 **I6** 最下面一行中间被**1, 3, 3**围绕的空格。在它所在的列进行边一致性检查，可以排除**5, 6, 2, 4** (我们已经知道 E6 是 4), **8, 9, 3**。根据**I5** 的取值我们排除 **1**, 现在**I6** 的取值就只剩下**7** 了。

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Figure 6.4 (a) A Sudoku puzzle and (b) its solution.

6.2.6 例3：Sudoku 数独问题

- 现在在第6列我们已经知道了8个数字, 根据边一致性我们可以推知 **A6** 的取值为 **1**。如此推理下去，AC-3 可以解决整个问题 — 所有变量都有了唯一的取值。如下如所示：

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Figure 6.4 (a) A Sudoku puzzle and (b) its solution.

6.2.6 例3：Sudoku 数独问题

- **AC-3 只能解决简单的数独问题**
- 如果所有数独问题都能用AC-3 轻松解决，那么数独问题就不那么有趣了。事实上**AC-3 只能解决最简单的数独问题。**稍微难一点儿的问题可以用 **PC-2** 算法解决，但是需要大量计算开销: 在一个数独问题中能有 255,960 个不同的路径约束。
- 为了解决最难的数独问题，我们需要变得更加聪明。

6.2.6 例3：Sudoku 数独问题

- **CSP 问题形式化的威力**
- 现在我们尝试不涉及数独问题独有的性质，
- 我们仍旧把问题表述成有 81 个变量, 每个变量的取值域是 1 到 9, 有 27 个 Alldiff 约束。
- 除此之外, 所有的策略 — 边一致, 路径一致等 — 适用于所有CSP问题, 而不仅仅是Sudoku问题。
- 这就是 **CSP 问题形式化的威力**： 对于每一个问题领域, 我们只需要把问题定义成约束；然后就能够用**通用的约束问题求解器**来解决。

内容提要

1. 定义约束满足问题（CSP）
   - 地图着色问题、作业调度问题、CSP的形式化
2. 约束传播：CSP中的推理
   - 结点相容、弧相容、路径相容、k-相容、全局约束、数独
3. CSP的回溯搜索
   - 变量和取值顺序、搜索与推理交错进行、智能回溯：向后看
4. CSP局部搜索
5. 用PDL语言解决问题

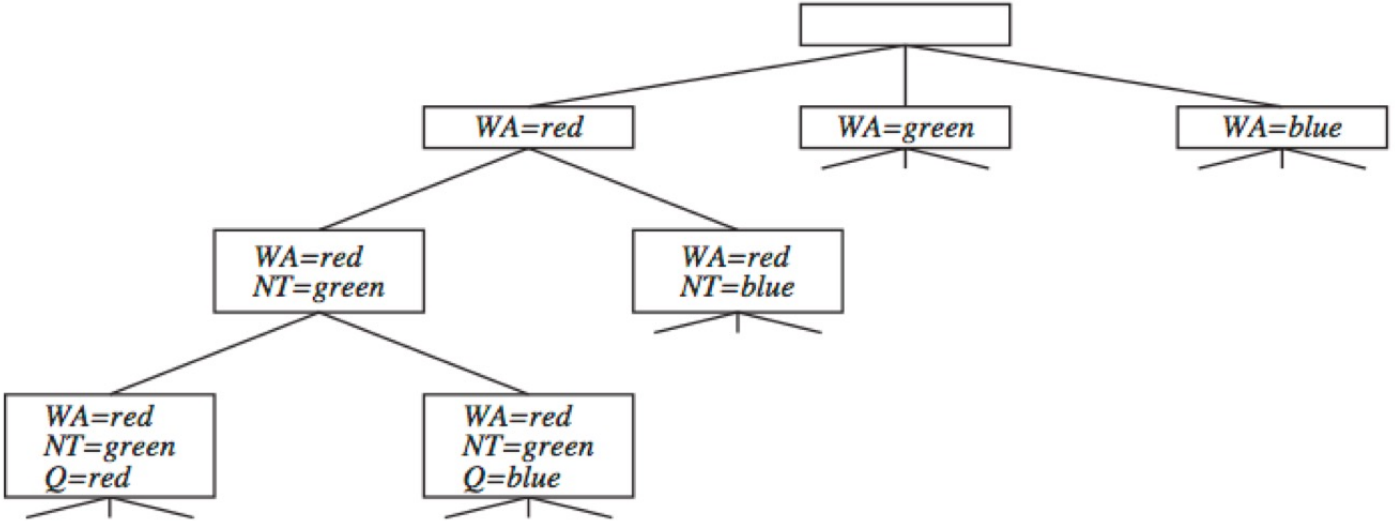
6.3 用回溯搜索方法解 CSP 问题

- 一个问题是**可交换的（commutative）**如果**赋值的顺序**对求解没有影响。
- CSP问题是**可交换的**，因为赋值的顺序对解没有影响。
- 所以在搜索树的一个节点处，我们只需要考虑一个变量的赋值。
- 例如，在澳大利亚地图着色问题的搜索树的根节点, 我们可以在**SA=red, SA=green, SA=blue** 三者中选择, 但是我们不用在 **SA=red** 和 **WA=blue** 中做选择。在这样的限制下，叶子节点的个数为 $\mathbf{d^n}$

6.3 用回溯搜索方法解 CSP 问题的伪代码

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, value)
            if inferences ≠ failure then
                add inferences to assignment
                result ← BACKTRACK(assignment, csp)
                if result ≠ failure then
                    return result
        remove {var = value} and inferences from assignment
    return failure
```



- 回溯搜索描述的是深度优先搜索每次为一个变量选择赋值，当变量没有值可选时就回溯。它不停地选择没被赋值的变量，依次选择这个变量的可能取值，试图发现一个解。如果有冲突被检测到，就返回失败，引发上一层的赋值尝试另外一种赋值。 上图是一个部分赋值了的搜索树。每层依次尝试 WA, NT , Q, ...
- 注意回溯搜索只保留状态的一种赋值，然后不停地修改它。

6.3 用回溯搜索方法解 CSP 问题

- 在前面的经典搜索中，我们提到用启发式函数加入一些问题领域相关的知识帮助搜索更快找到解。在这一章里，我们不使用问题领域相关的知识来解决CSP问题。我们可以使用一些高级的方法：
  - 1. 下一变量应该选哪个(**SELECT-UNASSIGNED-VARIABLE**), 它的值的选择顺序是什么(**ORDER-DOMAIN-VALUES**)?
  - 2. 在每一步应该采用哪种方法进行推理 (**INFERENCE**)?
  - 3. 当我们检测到赋值冲突时，如何避免重复尝试这种情况?

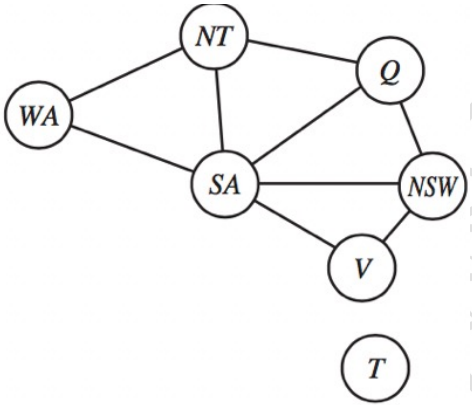
6.3.1 变量和取值的顺序

- 在回溯算法里有这样一行

  var ← SELECT-UNASSIGNED-VARIABLE(csp) .

- SELECT-UNASSIGNED-VARIABLE 最简单的策略是按顺序选择下一个未被赋值的变量, {**X1, X2, . . .**}. 这种静态的选择变量的顺序很少能够得到高效的搜索效果。

- 例如, 下图中在做了 **WA = red** ， **NT = green 之后**, **SA** 只有一种可能的取值 **SA = blue** 显然此时选择**SA** 好过选择 **Q**。事实上，在 **SA** 被选择后, **Q, NSW** , **V** 的选择都是固定的。

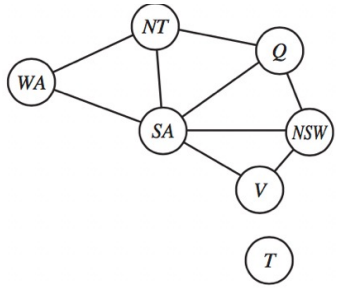


6.3.1 变量和取值的顺序

- **变量的顺序 - minimum-remaining-values (MRV)最少余值**
- 这种直觉，选择剩余的合法取值个数最小的变量的方法称作最少余值启发法 **minimum- remaining-values** (MRV) 。
- 它也被称作最受限变量“**most constrained variable**” 或者 最先失败“**fail-first**” 启发法, 后者是因为我们选择了一个更像是会很快引起冲突的变量，所以剪枝效果更好。
- 如果有些变量X没有合法取值了, 最少余值法会选择X，然后冲突就会立即被检测到，从而避免了无谓地尝试给其他变量。
- 最少余值法通常比随机或者固定顺序搜索高效。 尽管效率提高取决于具体问题，不过通常情况下效率会提高1,000倍。

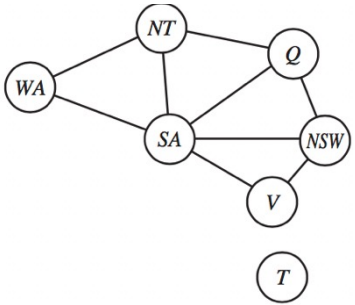
6.3.1 变量和取值的顺序

- **变量顺序 – 程度启发 degree heuristic**
- 最小余值法并不能帮助地图着色问题选择第一个点，因为所有点都有三种颜色可以选择。
- 这种情况下，我们使用 **程度启发（degree heuristic）**。这种方法通过选择与尽量多的未被赋值的变量存在约束关系的变量，试图减少未来选择中的分支数。
- 在上面的例子中，**SA** 的度数最高,为 **5**; 其他变量的度数为 **2** 或者 **3**, **T** 的度数为 **0**。事实上，当我们应用程度启发而选择了 **SA** 后，我们可以没有任何回溯地找到问题的解 — 在任何节点选择任何一致的赋值都能找到解。
- 最少余值法是个更有效的方法，但是程度启发有利于打破平局的情况。



6.3.1 变量和取值的顺序

- **取值顺序 – 最少受限值least-constraining-value**
- 当一个变量被选择后，算法就要决定以何种顺序尝试它可能的取值。
- 在一些情况下，最少受限值（ **least-constraining-value** ）方法是有效的。 它倾向于为它的邻居保留更多的选择的取值。例如，当我们已经有 **WA = red** ， **NT = green** 时，我们接下来要为**Q** 选值。**Blue** 是一个坏的选择，因为它使得它的邻居**SA**没有值可选了。最少受限值法于是推荐 **red** ，而不是 **blue**.
- 一般地，本方法倾向于为后续的赋值留有最大的自由度。
- 当然，如果我们要找到所有可行解，而不仅仅是第一个， 顺序是无关紧要的，因为我们需要考虑所有的赋值可能。如果问题无解，顺序也是无关紧要的，因为在下结论前，我们走到了所有可能赋值。



6.3.1 变量和取值的顺序

- **Why should variable selection be fail-first, but value selection be fail-last? 为什么变量顺序用最早失败法，而取值用最晚失败法呢？**
- 经验表明，对于大多数问题，选择具有最少取值数目的变量有助于通过尽早地剪枝掉树的大部分而减少搜索树的节点数目
- 对于取值顺序，关键在于我们只需要一个解，所以优先搜索那些貌似可以得到解的路径会帮助我们尽快地找到一个解。
- **如果我们需要找到所有解，则取值顺序是无关紧要的。**

6.3.2 交替使用搜索和推理

- 到目前为止，我们已经看到 AC-3 等算法可以在搜索之前通过推理缩小变量的取值范围。但是如果在搜索过程中进行推理往往会更有效。 每次我们为一个变量选择一个取值，就有机会通过推理缩减邻居的取值范围。
- 最简单的推理是向前检查。当一个变量 **X** 被赋值，向前检查会启动 边一致性检查: 对于所有与 **X** 相邻的尚未被赋值的 **Y** ，删除所有与 **X** 的选值相冲突的 **Y** 的取值。因为向前检查只做边一致性检查，所以如果我们已经在预处理阶段做了边一致性检查，我们就没有必要在搜索中再做向前检查。

6.3.2 交替使用搜索和推理

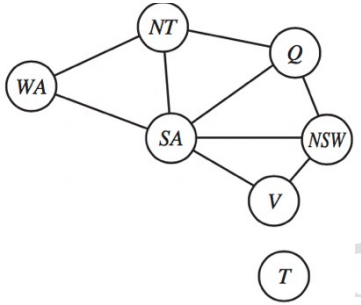
	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

- 上图给出了在澳大利亚地图着色CSP问题中加入了向前检查的回溯搜索的执行过程。
- 有两点注意：
- **第一**, 在**WA = red** 和 **Q = green** 赋值后, **NT** 和 **SA** 的取值个数降为 **1**；我们通过从**WA** 和 **Q** 的推理减少了其他变量的取值范围。
- **第二**，取 **V =blue**后， **SA** 的值域为空。所以向前检查已经发现了部分赋值 {**WA=red, Q=green,V =blue**} 与问题的约束条件相冲突，所以算法立即回溯。

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

- 对于很多问题，如果我们把最少余值法**MRV**与向前检查结合起来，搜索会更加高效。考虑上图在 {WA=red}后，看上去它的邻居 **NT** 和 **SA** 受到了限制, 所以我们接下来优先考虑这两个变量, 然后再据此考虑后面的变量。这也是最小余值法MRV所建议的: **NT** 和 **SA** 有两个合法取值,
- 所以它们中的一个先被选择，然后是另一个，然后 **Q**, **NSW** 和 **V** 依次被赋值。 最后 **T** 还有三个取值, 任何一个都构成解。我们可以把**向前检查**看做是为**最少余值法**做准备的。

	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B



- 尽管向前检查法可以检测到很多不一致，但是它并不能检测到全部的冲突。问题在于它只检查当前变量的边一致性，并不会向前看以检测所有变量的边一致性。例如， 上图第三行。 **WA** 是 **red** 同时 **Q** 是 **green**，**NT** 和 **SA** 都被强制成 blue。 向前检查并不能向前检查出此处的不一致性: **NT** 和 **SA** 相邻所以不能同时为blue。
- 算法MAC (for **Maintaining Arc Consistency (MAC)**) 会检查上述定位不一致性

6.3.2 交替使用搜索和推理

- **保持边一致性 Maintaining Arc Consistency (MAC)**
- 当一个变量 **Xi** 被赋值时, 推理过程调用 **AC-3**, 但是它并不检查**CSP**问题中的所有边，我们只检查 (**Xj,Xi**) 这里 **Xj** 是待赋值的与**Xi** 相邻的变量。从这里开始, **AC-3** 像往常一样做约束传播，如果有任何变量的值域为空，**AC-3** 调用失败，我们就立即回溯。
- 我们可以看到 **MAC** 一定比向前检查更高效，因为向前检查做了 **MAC** 检查中的第一步; 但是不像 **MAC**, 向前检查没有递归地传播约束。

内容提要

1. 定义约束满足问题（CSP）
   - 地图着色问题、作业调度问题、CSP的形式化
2. 约束传播：CSP中的推理
   - 结点相容、弧相容、路径相容、k-相容、全局约束、数独
3. CSP的回溯搜索
   - 变量和取值顺序、搜索与推理交错进行、智能回溯：向后看
4. CSP局部搜索
5. 用PDL语言解决问题

6.4 CSP问题的局部搜索算法

- 局部搜索算法对许多 CSP 问题都有很好的效率。局部算法用全部变量赋值的形式表示状态: 初始时为每个变量指定一个值，搜索时每次改变一个变量的取值。
- 例如，在 8皇后问题中，初始状态是8个皇后分别在8列，但是在列内的位置是任意的。我们每次在某一列中移动一个皇后。一般情况下，初始状况会和某些约束相冲突。局部搜索的核心思想是每次减少冲突。
- 在为一个变量选择一个新的取值时，最显然的考虑是选择一个值使得它与其他的变量的冲突尽量的少，即最小冲突（**min-conflicts）启发式。** 后面两页给出了算法及算法作用在8皇后问题上相应的解题步骤。

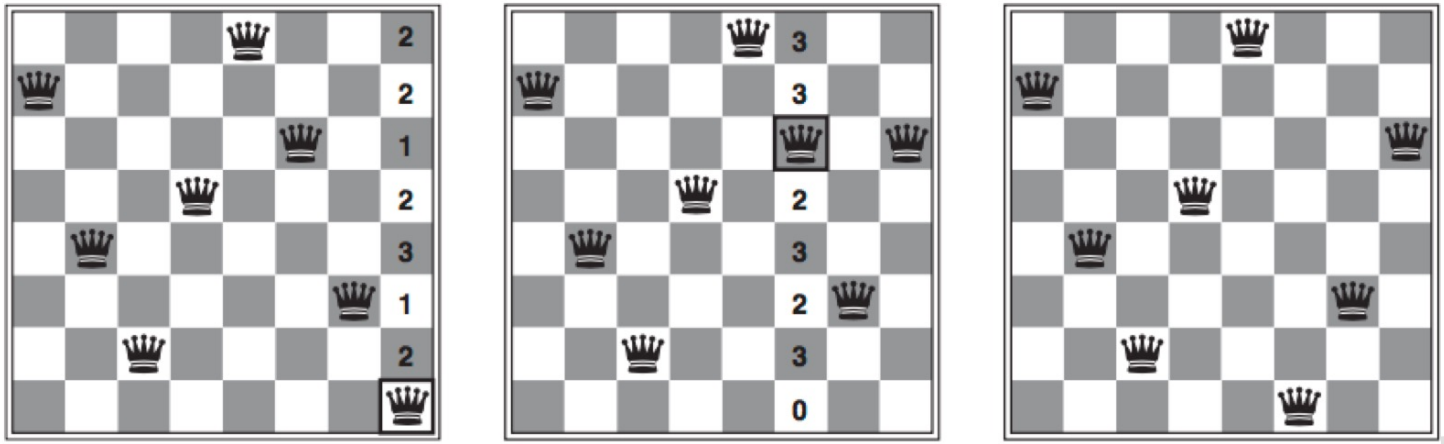
6.4 CSP问题的局部搜索算法

function MIN-CONFLICTS(*csp*, *max_steps*) **returns** a solution or failure
  inputs: *csp*, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for *csp*
  for *i* = 1 to *max_steps* **do**
    if *current* is a solution for *csp* **then return** *current*
    var ← a randomly chosen conflicted variable from *csp*.VARIABLES
    value ← the value *v* for *var* that minimizes CONFLICTS(*var*, *v*, *current*, *csp*)
    set *var* = *value* in *current*
  return *failure*

- 本页图示给出了使用最小冲突法对CSP问题进行局部搜索的算法。初始状态可以是随机的。或者用贪心法为每个变量选一个最小冲突的赋值。在给定当前的赋值下，CONFLICTS函数计算当前被违背了的约束的个数。

6.4 CSP问题的局部搜索算法



- 本页图中给出了用最小冲突法仅用2步就解出八皇后问题的例子。在每个阶段，选一个皇后来改变她的赋值。每个格子显示出冲突的个数。算法使皇后移动到拥有最小的冲突数的位置，有些时候为了打破僵局，也会选择随机平移到冲突数相同的地方。

6.4 CSP问题的局部搜索算法

- **对于很多CSP问题，最小冲突法令人吃惊地高效。**
- 令人惊奇地, 在 **n-皇后** 问题中, 如果忽略初始状态的赋值过程，最小冲突法的效率几乎是与问题规模无关的。它解决100万皇后问题平均只需要50步。
- 这个令人惊奇的现象引发了1990年代局部搜索方法针对解决简单与困难问题的区别的研究浪潮。简单地说, 对于局部搜索方法，n-皇后问题是简单问题，因为可行解密集地分布在解空间中。最小冲突法在解决困难问题中也表现良好。

- 例如，该方法被用来解决哈勃望远镜（**Hubble Space Telescope**）的日程安排问题，成功地将安排一周的观测时间表的时间从3周降至10分钟。

6.4 CSP问题的局部搜索算法

- 第4章讲的所有局部搜索算法都适用于CSP问题，一些方法被证明相当高效。
- CSP问题在最小冲突 **min-conflicts** 启发式方法下的状态空间有一系列平台。有上百万的变量赋值组合距离解只有一个冲突。
- **平台搜索 Plateau search**— 允许平移，即移动到相同分数的赋值 — 可以帮助局部搜索跳出平台。
- 在平台上移动可以配合一个查找表， 保存一个小的最近访问过的状态列表，并且避免再次进入这些访问过的状态。模拟退火方法也可以帮助逃离平台。

6.4 CSP问题的局部搜索算法

- 局部搜索的另一个好处是可以用于解决在线系统的改变问题。
- 这对调度问题尤为重要。
- 一周的航班安排可能包含几千个航班和几万个人员安排，但是机场的坏天气可能会使这些安排不能付诸实施。
- 我们需要用最小的改动来修复航班安排。这个任务很容易用局部搜索方法实现。
- 以最新约束使用回溯搜索算法可能会花费更多时间，并且很有可能会找到一个与原计划相比有很多改动的新日程。

小结 1/2

- **约束满足问题 Constraint satisfaction problems** (CSPs) 表示一个状态由一组变量/取值对构成，一个解表明变量赋值满足一组约束条件。 很多重要的实际问题都可以描述成 **CSP问题。**

小结 2/2

- **推理技术**用约束来推断变量的赋值是否是一致的。这些技术包括点、边、路径和 k-一致性。
- **回溯搜索（Backtracking search）**，是一种深度优先搜索，广泛用来解决CSP问题。推理和搜索可以交替进行。
- **最小余值法**（The **minimum-remaining-values）和程度启发式**都是领域无关的在回溯搜索中用来决定下一个变量选哪个的方法 。回溯发生在当发现一个变量没有可选的合法值时。
- **局部搜索（Local search）**使用最小冲突启发式，对解决CSP问题被证明非常有效。
- CSP问题的解决方法的复杂度与约束图的结构密切相关。

通用求解器的内部实现结构

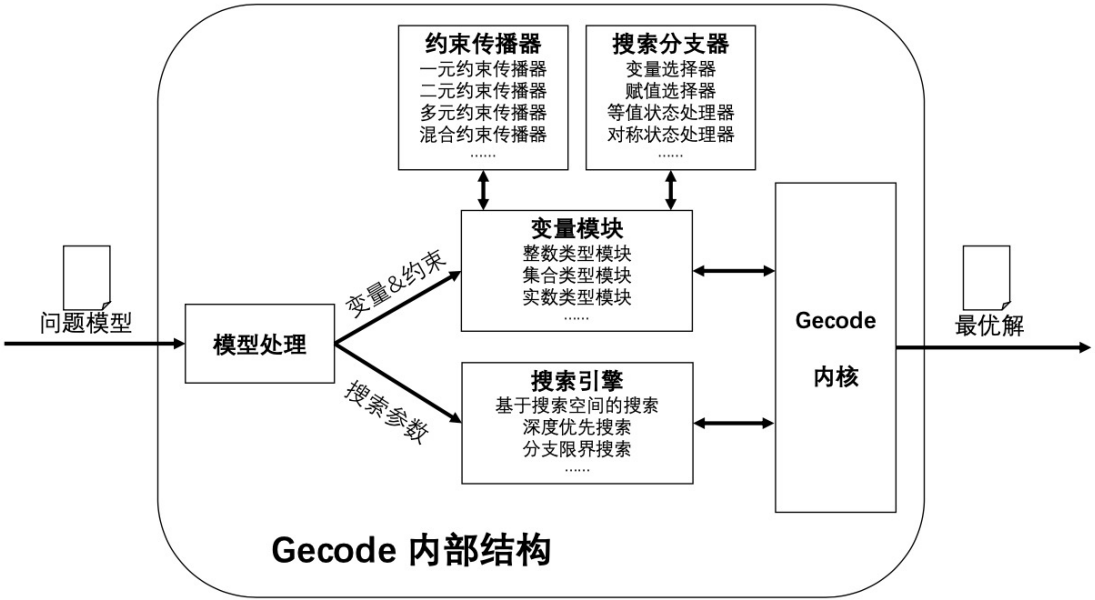


图 1.2　Gecode 内部实现结构

- MiniZink 语言描述
- % 求A和B的最大公约数

- 1..1000000: A;
- 1..1000000: B;

- var int: G;
- var int: X;
- var int: Y;

- constraint A = G * X;
- constraint B = G * Y;
- solve maximize G;

PDL2C 实现流程

1. 通过词法分析和语法分析创建标识符表;
2. 选取独立变量
   ① 识别变量之间的依赖关系;
   ② 将所有变量按照取值范围从小到大排序;
   ③ 采用启发式搜索的方式找到最小的独立变量集;
3. 根据独立变量的类型情况，生成基于循环的搜索程序框架 或 基于递归的搜索程序框架;
4. 进行搜索剪枝优化和动态规划优化(此处为难点，单 列于第二个创新点);
5. 生成 C 代码。

PDL语言的问题描述

```
// 求A和B的最大公约数
#input
    A of int in [1,10^6]
    B of int in [1,10^6]
#required
    G of int
    X of int
    Y of int
    A = G * X
    B = G * Y
#output
    max (G)
```

```
int A, B;
int _best, _cur;
bool _checkConstraints(int X) {
    if (X == 0 || A % X > 0)
        return false;
    int G = A / X;
    if (G == 0 || B % G > 0)
        return false;
    return true;
}
void _updateAnswer(int X) {
    _cur = A / X;
    if (_cur > _best)
        _best = _cur;
}
int _enumFunc() {
    _best = -1000000 - 1;
        for (int X = -1000000; X <= 1000000;
                if (_checkConstraints(X))
            _updateAnswer(X);
}
int main() {
        //TODO read input value(s): A, B
        _enumFunc();
        //TODO print answer(s)
    return 0;
}
```

PDL - Problem Description Language

求A和B的最大公约数。

```
#input
        A of int in [1,10^6];
        B of int in [1,10^6];

#required
        G of int in [0,?];
        X of int;
        Y of int;
        A = G * X;
        B = G * Y;

#objective
        maximize G;
```

PDL - Problem Description Language

0-1背包问题

选取若干物品放入背包，在物品总重量和不超过背包容量的条件下，物品总价值和尽可能大。

```
#input
        N of int in [1,100];
        capacity of int in [1,10000];
        profits of (int in [1,1000])[1~N];
        weights of (int in [1,1000])[1~N];

#required
        knapsack of (int in [1,N]){};
        summation
            [weights[i] : forall i (i in knapsack)]
            <= capacity;

#objective
        maximize summation
            [profits[i] : forall i (i in knapsack)];

#output
        knapsack;
```