中国科学院大学课程讲义

计算机算法设计与分析

中国科学院大学 计算机与控制学院

软件与理论教研室

马菲菲

2019 年 11 月 1 日

第一章 约束求解 ____ *2*
1 约束满足问题 ____ 2
2 约束传播 ____ 4
3 打破对称 ____ 10
第二章 SAT 求解算法及相关问题 ____ *16*
1 SAT 的相关定义 ____ 16
2 2-SAT 问题和相变 ____ 19
3 SAT 的求解算法概述 ____ 20
4 不完全求解器 ____ 21
5 DPLL 算法 ____ 22
6 MAX-SAT 问题 ____ 28
7 SMT 问题 ____ 29
第三章 启发式算法 ____ *33*
1 局部搜索 ____ 33
2 模拟退火 ____ 41
3 遗传算法 ____ 43

第一章 约束求解

约束是一个普遍存在的概念，代表了限制我们决策自由度的条件。人工智能和计算机科学其它领域中的大量组合问题都可以看成是约束满足问题(Constraint Satisfaction Problem，简称 CSP)，即寻求对问题变量的赋值,使之满足问题的约束。例如，N 皇后问题、数独、地图染色、航班调度、资源分配、路径规划、日程安排等很多问题都是 CSP 的特例，我们所熟悉的 SAT 等 NP 完全问题也属于 CSP。

从二十世纪六十年代起，科学家们开始系统地研究约束满足问题的建模、表示与求解方法，这一系列技术进而发展成为人工智能中的一个经典领域——Constraint Programming。Constraint Programming 的研究分为语言和算法两个流派。语言流派侧重于编程语言和系统实现，深受逻辑程序设计思想的影响，其代表性成果包括著名的 Prolog 语言。在本章中，我们将侧重于从算法角度介绍 Constraint Programming 的思想，因此也将其翻译为约束求解。

1 约束满足问题

约束在人们的生活中几乎处处都会遇到，例如会议要在 10 点之前开始，工资大于六千元，网络的流量不低于 50M 等等。相应的，约束满足问题在人工智能领域有着广泛的应用。例如一个学校在开学初期，要规划分配教室资源，以及安排课时；美国的 NBA 篮球赛，要在开赛之前安排赛程；一个繁忙的飞机场，要合理的调度飞机等。这些问题都涉及了一些约束条件。一些棋类游戏，可以被编码成约束问题；一些教科书中的经典问题，也属于约束求解问题，例如八皇后问题，数独问题等。

从数学意义上讲，约束满足问题是种数学的问题，其定义为一组对象(object)，而这些对象需要满足一些限制或条件。 CSPs 是人工智能和运筹学 的热门主题,因为它们公式中的规律，提供了共同基础来分析、解决很多看似不相关的问题。 CSPs 通常呈现高复杂性, 需要同时透过启发式搜索 和 联合搜索 的方法，来在合理的时间内解决问题。 布尔可满足性问题 (SAT), 可满足性的理论 (SMT)和回答集程序设计 (ASP) 可以算是某种程度上的约束满足问题。

下面给出约束满足问题的形式定义。

定义 1（约束满足问题） 一个约束满足问题 P 可以表示为一个三元组：P = <X, D, C>，其中，$X = \langle x_1, x_2, \ldots, x_n \rangle$是一组变量的集合，$D = \langle D_1, D_1, \ldots, D_n \rangle$代表每个变量的值域，也就是取值范围，即 $x_i \in D_i$ ，$C = \{c_1, c_2, \ldots, c_t\}$是变量集合 X 上的一组约束。

例 1. 课程时间表问题，该问题要求安排一个合理的课程表，其中的元素为：

- 变量：每个课程对应一个变量，例如语文、数学、外语等
- 值域：每个变量的值域是可能的上课时间，例如：上午九点、下午三点等
- 约束：对课程做的一些限制。例如：语文课不能在上午九点上课，两门课不能同时开课等
- 目标：对于每一门课，求一个上课时间，所形成的上课时间表要满足所有的约束

定义 2 （约束）一个约束 c 就是定义在一个变量序列 $X(c) = <x_{i1}, \ldots, x_{i|X(c)|}>$上的关系，也是这组变量取值的所有可能组合的一个子集。这组变量的个数$|X(c)|$，被称为约束的维度。给定了约束 c，检查对 X(c)的一个赋值元组是否满足该约束，被称为一个“约束检查”。

约束的表示方式有两种：

- 内涵（Intensional）方式：是描述约束性质的函数。例如，两个变量 x 与 y 相等的约束，可以用 x=y 来表示。
- 外延（Extensional）方式：一个约束本质上是一些变量的值域的笛卡尔集的子集， 外延方式是将这些子集的元素详细地写出来。例如，变量 x、y，它们的值域均为{1、2、3}，表示 x 与 y 相等的约束，可写为：{<1,1>,<2,2>,<3,3>}。

根据约束的维度，可以分为：

- 一元约束：即只包含一个变量的约束，一般这样的约束可以通过给定恰当的值域来描述。
- 二元约束：只包含两个变量的约束，例如不等于，大于，小于。
- 多元约束：包含三个或三个以上变量的约束，例如 alldifferent(x_1, x_2, x_3)。

如果约束满足问题中的所有约束的维度都不超过 2，则称为二元约束满足问题（Binary CSP）。二元约束满足问题是 NP 完全问题。

容易看出，多元约束 alldifferent(x_1, x_2, x_3) 可以转换为三个二元约束 $x_1 \neq x_2$, $x_1 \neq x_3$, $x_2 \neq x_3$。一般地，任何含有多元约束的约束满足问题都可以通过对偶图法或隐藏变量法转换为二元约束满足问题[1]。尽管在理论上无需多元约束，但在实际应用中多元约束是不可忽视的。将多元约束转换为二元约束有时会产生不自然的陈述，而多元约束在描述上更紧凑，有些多元约束有其特有的求解算法，从而在求解中效率更高。在本章中，由于篇幅所限，我们主要讨论二元约束满足问题，从中体会约束求解的思想。

二元约束满足问题可用约束网络来表示，网络中的每个顶点代表一个变量，每条边代表两个变量之间的约束。

例 2. 二元约束满足问题 P = <X, D, C>，其中，X= <x, y, z, t>，$D_x = D_y = D_z = D_t = \{1,2,3\}$，C={ x < y, y = z, t < z, x < t }。P 对应的约束网络如图 1.1-1 所示。

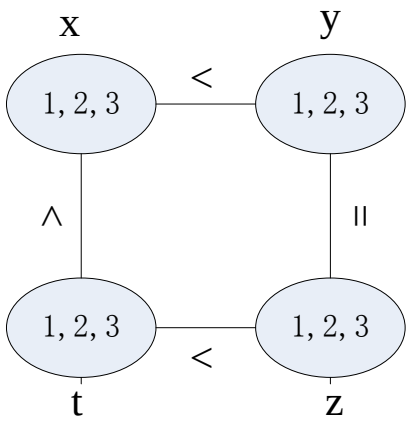


图 1.1-1 约束网络示例

2 约束传播

约束满足问题有两类主要的求解方式：系统的搜索技术和局部搜索技术。前者是完备搜索技术，无论是否有解，当搜索完成时都可以知道，也称为回溯搜索。后者是不完备搜索技术，如果有解，可能给出结果，也可能不给出结果。一般来说，局部搜索比系统搜索能求解更大规模的问题实例，但是如果问题不可满足，局部搜索是没有意义的。

回溯法是一种递归算法，在搜索过程中它保持部分变量的赋值。搜索开始的时候，所有的变量都还没被赋值。然后依次选取一个变量，并且将所有可能的值依次赋予该变量。对于每一个值，要对它的一致性(consistency)进行检查。如果具有一致性，则进行下一次赋值。如果所有的值都无法保持一致性，算法则回溯上层。在回溯中，可以采用某些技术通过回溯”一个以上的变量”来减少搜索空间。在搜索过程中，约束传递 (Constraint propagation)技术是用来检查一致性的一类重要技术，同时也是缩小变量的值域，从而减少之后的搜索空间的重要技术。常用的一致性是弧一致性，超弧一致性，和路径一致性。 最流行的方法是 AC-3 约束传播算法,该算法可以运行弧的一致性。

局部搜索方法是不完全满足的算法。如果问题有解，算法可能会找到解，也可能找不到。具体要看算法的终止策略。如果问题无解，则算法无法判断出这类情况。

简单的系统搜索的算法框架如下：

- 1. 给一个变量进行赋值
- 2. 其他的未被赋值的变量，在值域中可能有一些值已经不再满足某些约束，删除掉这些值
- 3. 如果某个变量的值域中所有的值都被删除，则证明该赋值无法导致问题的解，这时进行回溯，尝试其他的赋值

在系统搜索算法中的第二步，根据已有的赋值来消减未被赋值的变量的值域，就是传播。当一个变量的值域被消减后，可能会进一步影响其他变量，导致其他变量的值域也被消减，所以学术上将其形象的称为“传播”。传播技术在求解约束满足问题中占有重要作用。在约束传播中，约束网络不断收紧，使得系统搜索算法更早地发现局部的冲突，从而减少分支数。

一致性（consistency）是传播时需要加在约束满足问题上的重要性质。可以说，传播的本质是为整个约束满足问题维持某种一致性。原本问题是一致的，当某个值被消减掉之后，问题就变得不一致了，这个时候需要再进行值域的消减，以维护一致性。

定义 3 （弧一致性 Arc-consistency） 给定约束网络 R=(X, D, C)，其中 $R_{ij} \in C$，变量 x_i 相对于 x_j 是弧一致的当且仅当对于每一个 $a_i \in D_i$ 都存在一个 $a_j \in D_j$，使得$(a_i\ a_j) \in R_{ij}$ 。$\{x_i, x_j\}$对应的弧是一致的，当且仅当 x_i 相对于 x_j 是弧一致的，且 x_j 相对于 x_i 是弧一致的。如果约束网络中所有的弧都是一致的，则该约束网络是弧一致的。

例 3. 考虑例 2 中的约束满足问题 P = <X, D, C>，变量的值域皆为{1,2,3}， C={ x < y, y = z, t < z, x < t }。P 对应的约束网络在满足弧一致性时的状态如图 1.2-1 所示。

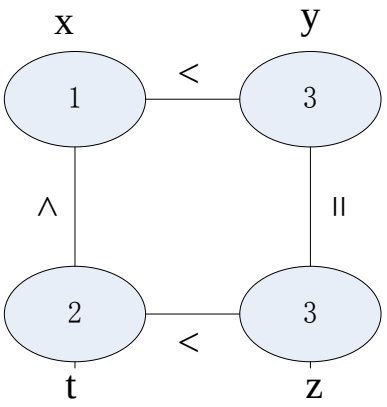


图 1.2-1 弧一致性示例

对于约束网络中的两个顶点（CSP 的两个变量）x_i, 和 x_j ，消减 x_i 的值域，使得 x_i 相对于 x_j 是弧一致的过程见算法 1。

算法 1 （弧一致性修正算法）

- REVISE((x_i), x_j)

input: *a subnetwork defined by two variables* $X=\{x_i, x_j\}$ *, domains:* D_i *and* D_j *, constraint* R_{ij}

output: D_i *, such that* x_i *arc-consistent relative to* x_j

for each $a_i \in D_i$

if there is no $a_j \in D_j$ such that $(a_i, a_j) \in R_{i,j}$

then delete a_i from D_i

endif

endfor

弧一致性修正算法可以用下面的数学式子来表示：

$$D_i \leftarrow D_i \cap \pi_i(R_{ij} \otimes D_j)$$

算法 1 中的弧一致性修正算法是保证一个变量相对于另一个变量的弧一致性。但我们希望整个约束网络都满足弧一致性，从而尽可能地消减所有变量的值域，减少搜索分支。由于当一个变量的值域被弧一致性修正算法消减后，可能会导致其它变量不能满足相对于此变量的弧一致性， 因此弧一致性修正算法需要被反复应用，直至整个约束网络是弧一致的。根据这一思路，我们可以写出基本的保证全局弧一致性的算法 AC-1，见下面算法 2。

算法 2 （AC-1）

- A network of constraint R=(X,D,C)
- **AC-1(R)**

```
repeat
    for every pair {xi, xj}∈C
        REVISE((xi), xj)
        REVISE((xj), xi);
    endfor
until no domain is changed
```

假设约束网络中，弧（约束）的数目为 e，顶点（变量）的数目为 n，每个变量的值域大小都为 k。每次迭代的复杂度为 ek^2，最坏情况下有 $n \times k$ 次迭代，所以算法的时间复杂度为 $O(enk^3)$。复杂度(Mackworth and Freuder, 1986): 当约束网络已经是弧一致时，复杂度为 $O(ek^2)$。

AC-1 中可能有很多无用的计算。显然，对于一条弧$\{x_i, x_j\}$来说，如果已经确保 x_i 相对于 x_j 是弧一致的，而之后的 REVISE 过程没对 x_i 的值域造成改变，那么 x_i 相对于 x_j 仍然是弧一致的，是不需要再检查的。因此，可以通过记录弧两端的变量的值域是否发生改变，来决定它是否需要重新修正，从而产生了 AC-3 算法。它也是最有名的弧一致性算法。

算法 3 （AC-3）

- **AC-3(R)**

```
for every pair {xi, xj}∈C
    queue= queue ∪ {(xi, xj ), (xj, xi)}
endfor
while  queue ≠ Φ
    select and delete (xi, xj ) from queue
    REVISE((xi), xj)
     if  REVISE((xi), xj) caused a change in Di
```

then queue= queue ∪ {(x_k, x_i)| i≠k };

endif

endwhile

AC-3 用队列存储了需要检查的变量对。初始化的时候，所有的变量对都要进入队列。一对变量(x_k, x_i)被再次放入队列的前提是，x_i 的值域 D_i 有了改变，x_k 中的某些值可能在 D_i 中找不到支持。在 AC-3 中，每条弧被放入队列中不超过 2k 次，每次处理 $O(k^2)$，因此算法的时间复杂度为 $O(ek^3)$。

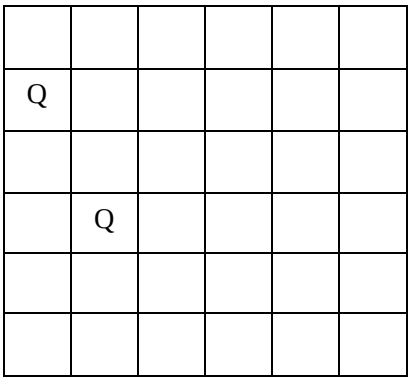


图 1.2-2　6 皇后问题

例 4. N 皇后问题是指在一个 N × N 的棋盘上放置 N 个皇后，使得每一行、每一列以及每一斜线上不能有两个以上的皇后。令 x_i（1≤i≤N）表示第 i 列的皇后放的行号，则 D_i ={1,…,N}，两个变量 x_i 与 x_j （0<i<j≤n）应满足约束 $x_i \neq x_j$， $|x_i-x_j| \neq |i-j|$。假设在求解 6 皇后问题时，算法已经在第 1 列和第 2 列分别放置了皇后，如图 1.2-2 所示。用 AC-3 维护弧一致性的过程如下：

1. D_3={1,6}，D_4={1,3}，D_5={3,5}，D_6={1,3,5,6}；Q={(x_3,x_4), (x_4,x_3), (x_3,x_5), (x_5,x_3), (x_3,x_6), (x_6,x_3), (x_4,x_5), (x_5,x_4), (x_4,x_6), (x_6,x_4), (x_5,x_6), (x_6,x_5)}

2. 依次检查变量对(x_3,x_4)、(x_4,x_3)、(x_3,x_5)、(x_5,x_3)、(x_3,x_6)、(x_6,x_3)、(x_4,x_5)、(x_5,x_4)、(x_4,x_6)，发现它们都是弧一致的，值域无变化。 Q={(x_6,x_4), (x_5,x_6), (x_6,x_5)}

3. 检查(x_6,x_4)的弧一致性，D_6 变为{5,6}， Q={(x_5,x_6), (x_6,x_5), (x_3,x_6), (x_4,x_6)}

4. 检查(x_5,x_6)的弧一致性，D_5 变为{3}，Q={(x_6,x_5), (x_3,x_6), (x_4,x_6), (x_3,x_5), (x_4,x_5)}

5. (x_6,x_5)、(x_3,x_6)、(x_4,x_6)都是弧一致的，值域无变化，Q={(x_3,x_5), (x_4,x_5)}

6. 检查(x_3,x_5)的弧一致性，D_3 变为{6}， Q={(x_4,x_5), (x_4,x_3), (x_5,x_3), (x_6,x_3)}

7. 检查(x_4,x_5)的弧一致性，D_4 变为{1}，Q={(x_4,x_3), (x_5,x_3), (x_6,x_3), (x_3,x_4), (x_5,x_4), (x_6,x_4)}

8. (x_4,x_3)和(x_5,x_3)是弧一致的，值域无变化，Q={(x_6,x_3), (x_3,x_4), (x_5,x_4), (x_6,x_4)}

9. 检查(x_6,x_3)的弧一致性，D_6 变为{5}， Q={(x_3,x_4), (x_5,x_4), (x_6,x_4), (x_3,x_6), (x_4,x_6), (x_5,x_6)}

10. Q 中剩下的弧都是一致的。最终值域为：D_3={6}，D_4={1}，D_5={3}，D_6={5}

AC-3 的时间复杂度并非最优。AC-4 通过储存大量信息，避免在删除操作后的约束传播

中重复同样的约束检查，从而进一步改进了 AC-3 的时间复杂度。AC-4 的关键在于维护了两类数组: Counter[x_i, v_i, x_j]存储了 x_j 的值域中，支持 x_i 取值 v_i 的值的数目；S[x_i, v_i]存储了 $x_i=v_i$ 所支持的变量-值对的集合。一旦 Counter[x_i, v_i, x_j]变为 0，$x_i=v_i$ 在 x_j 的值域中没有支持，因此 v_i 应从 x_i 的值域中删除，同时把(x_i, v_i)加入队列 Q 中。Q 中保存了有待检查的被删掉的变量-值对。对 Q 中的任一对变量-值的组合(x_j, v_j)，如果其支持的变量-值对（即 S[x_j, v_j]中的元素）为(x_i, v_i)，若 v_i 仍在 x_i 的值域中，则令其在 x_j 中的 Counter 减去 1。重复以上操作，直至队列 Q 为空。

在 AC-4 中，共有 2ek 个 Counter，每个 Counter 的大小不超过 k，最坏情况下，是所有的 Counter 都减小为 0 时算法终止，因此复杂度为 $O(ek^2)$。

算法 4 （AC-4）

- AC-4(R)
- Q← Ø; S[x_j, v_j] =0; ∀ $v_j \in D(x_j)$, ∀$x_j \in X$

```
    foreach  x_i ∈ X,  c_ij ∈ C, v_i ∈ D(x_i) do
            initialize counter[x_i, v_i, x_j] to |{ v_j ∈ D(x_j)| (v_i, v_j) ∈ c_ij }|;
            if counter[x_i, v_i, x_j] =0 then remove v_i  from D(x_i) and add (x_i, v_i) to Q;
            add (x_i, v_i) to each S[x_j, v_j]  s.t. (v_i, v_j) ∈ c_ij ;
            if D(x_i)=Ø then return false;
    while Q ≠ Ø do
            select and remove (x_j, v_j) from Q;
            foreach (x_i, v_i) ∈ S[x_j, v_j]  do
                    if v_i ∈ D(x_i) then
                            counter[x_i, v_i, x_j] = counter[x_i, v_i, x_j] -1;
                            if counter[x_i, v_i, x_j] =0 then
                                    remove v_i  from D(x_i) ; add (x_i, v_i) to Q;
                                    if D(x_i)=Ø then return false;
     return true;
```

例 5. 图 1.2-3 的左图表示了一个 CSP 实例，含有三个变量{x, y, z}，D_x={2, 5}, D_y={2, 4}, D_z={2, 5}，约束为整除关系：z 整除 x 和 z 整除 y。

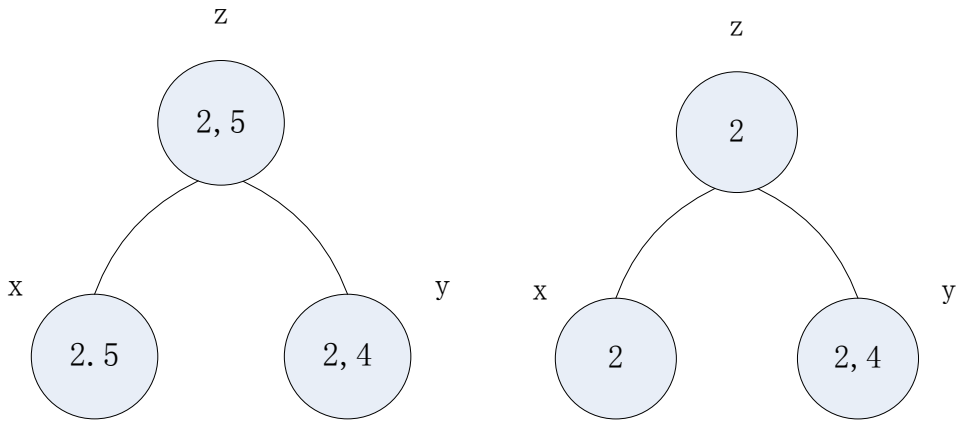


图 1.2-3 整除

AC4 算法是这样执行的：

初始化数组 S[x,a]（该数组包含所有(x,a)所支持的变量-值对），有：

S[z,2]={<x,2>,<y,2>,<y,4>}

S[z,5]={<x,5>}

S[x,2]={<z,2>}

S[x,5]={<z,5>}

S[y,2]={<z,2>}

S[y,4]={<z,2>}

初始化计数器 counter：

counter(x,2,z)=1

counter(x,5,z)=1

counter(y,2,z)=1

counter(y,4,z)=1

counter(z,2,x)=1

counter(z,2,y)=2

counter(z,5,x)=1

counter(z,5,y)=0

需要注意的是，如果变量间没有直接约束关系，则无需考虑，如 x 和 y。

在初始化过程中，由于 counter(z,5,y)=0，因此 D_z 由{2, 5}变为{2}，并将<z,5>放入队列 Q 中。初始化完成后，从 Q 中取出<z,5>，并根据 S[z,5]={<x,5>}，得知其支持的变量-值对只有<x,5>，由于 5 在 D_x 中，将 counter(x,5,z)减 1，得到 counter(x,5,z)=0。由于 counter(x,5,z)变为 0，x=5 在 z 中没有支持，故将 5 从其值域中删去，即 D_x={2}，同时将<x,5>放入 Q 中。现在从 Q 中取出<x,5>，由于 S[x,5]={<z,5>}，其支持的变量-值对只有<z,5>，而 D_z 的值域已不包含 5，故不需要继续传播。此时列表 Q 变为空，算法终止。结果见图 1.2-3 的右图。

我们对弧一致算法做总结如下：

AC-1 是一个基于强力搜索的算法，具有分布式特点，时间复杂度是 $O(enk^3)$；

AC-3 是基于序列的算法，时间复杂度是 $O(ek^3)$；

AC-4 是基于上下文的，优化的算法，时间复杂度是 $O(ek^2)$。

下面考虑这样一个问题，一个三角形的图，是否可用两种颜色对其顶点进行染色。这个问题的答案显然是否定的。不过，这个问题对应的约束网络是弧一致的，因此采用弧一致算法无法对其值域进行消减，也就是说弧一致算法对其无效。这时就需要考虑其他的一致性。除了弧一致外，路径一致也是一类重要的一致性概念。

定义 4 （路径一致性 Path-consistency） 给定约束网络 R=(X, D, C)，一组变量对$\{x_i, x_j\}$相对于变量 x_k 是路径一致的当且仅当对于每个一致的赋值($<x_i=a_i>$, $<x_j=a_j>$) 都存在一个 $a_k \in D_k$，使得($<x_i=a_i>$, $<x_k=a_k>$)和($<x_k=a_k>$, $<x_j=a_j>$)都是一致的。一个约束网络是路径一致的，当且仅当对于任意 R_{ij} 和任意 $x_{k\,(k\neq i,\,j)}$，$\{x_i, x_j\}$相对于 x_k 是路径一致的。

图 1.2-4 中展示了三角形的顶点染色问题的约束网络，左图是前面提到的 2 染色问题，不满足路径一致性，右图用了 3 种颜色，是路径一致的。

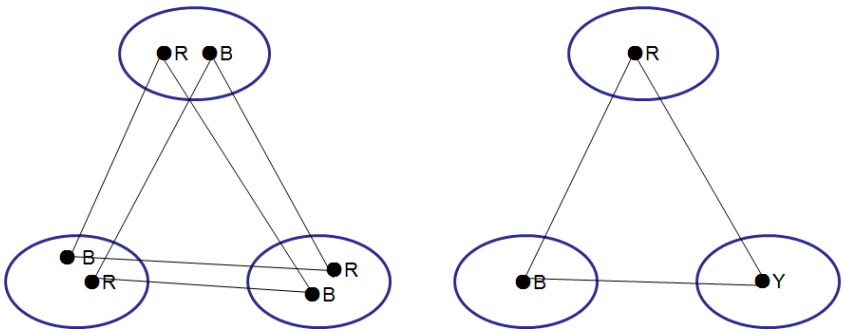


图 1.2-4 染色问题

3 打破对称

在 CSP 问题中对称性是一个常见的性质，因为对称性的存在，导致搜索的时候会不断搜索重复的空间，从而减低了搜索效率。

定义 5 （对称 Symmetry） 给定约束满足问题 *P*， (约束) 对称是对 *P* 的<变量-取值>对的一个置换，使得置换后 *P*（的约束）保持不变。

由于 *P* 的约束在对称变换中保持不变，*P* 的解集也保持不变。对称 σ 将任一<变量-取值>对 (x_i, j) 映射到另一个<变量-取值>对 $\sigma(x_i, j)$ 。

由于对称性的存在，一个解可能会和其它的解等价，搜索树的一个分支（部分解）也可能会和其它搜索分支等价，从而造成搜索空间的重复。有的问题中的对称的数目甚至随着问题规模呈指数增长。能否打破对称，避免搜索冗余的空间，对问题的求解效率往往有很大影

响。特别是当问题无解时，完备算法需要搜索完整个搜索空间，打破对称技术就至关重要。对称有时也被称为同构（Isomorphism），打破对称（Symmetry Breaking）也被称为消除同构（Isomorphism Elimination）。

打破对称可以从问题的三个阶段进行。第一，可以在问题建模的时候，尽量避免对称；第二，可以在问题求解开始的时候，通过添加约束来打破对称；第三，可以在求解过程中，动态地打破对称。

下面我们用例 5 来说明，如何在问题建模的时候，尽量避免对称。

例 5. 社交高尔夫问题：32 个高尔夫球手每周分成 8 组（每组 4 人）比赛，要求任何两人被分在同一组中最多一次。如何排日程，使得比赛的周数尽可能多。

假设问题的规模是 10 周。一个直接的建模方式，是引入变量 $x_{i,j,k}$，表示第 i 周（$1 \leqslant i \leqslant 10$）的第 j 组（$1 \leqslant j \leqslant 8$）中的第 k 个组员（$1 \leqslant k \leqslant 4$）。$x_{i,j,k}$ 的值域为 32 个高尔夫球手的集合。32 个高尔夫球手的置换有 32!个，每一个置换都保证了置换后的日程仍然是合法的，因此引入了 32!个对称。同理，10 周的置换引入 10！个对称，每一周的 8 个小组的置换引入 8!个对称，每个小组内部组员的置换引入了 4!的对称。这些对称复合起来，共将引入 $32!10!8!^{10}\ 4!^{80}$ 个对称。如果对建模方式做一些改进，采用“集合变量”建模。集合变量是那些值为集合的变量。例如，在高尔夫球手问题中，对每一周可以用 8 个集合变量，每个变量是一个四个元素的集合，对称可减少为 32!10!个。

在搜索中打破对称（Symmetry Breaking During Search）是一种重要的在求解过程中动态打破对称的技术，通过描述一个对称变换对变量赋值的影响，在搜索分支中避免一些赋值来消除对称。

下面我们用 n 皇后问题来说明如何在搜索中打破对称。n 皇后要求将 n 个皇后放在 n×n 棋盘上，使得每行每列有且仅有一个皇后，并且每条对角线上只有一个皇后。对 n 皇后问题，共有 7 种对称变换（第 8 种对称是恒等映射(identity)）。我们可以通过刻画这 7 种对称变换而消除所有的对称性。令 $r[i]$ 表示第 i 行皇后所在的位置（列号），7 种对称变换如下：

1. $x(r[i]=j) \rightarrow r[n-i+1]=j$
2. $y(r[i]=j) \rightarrow r[i]=n-j+1$
3. $d1(r[i]=j) \rightarrow r[j]=i$
4. $d2(r[i]=j) \rightarrow r[n-j+1]=n-i+1$
5. $r90(r[i]=j) \rightarrow r[j] = n-i+1$
6. $r180(r[i]=j) \rightarrow r[n-i+1]=n-j+1$
7. $r270(r[i]=j) \rightarrow r[n-j+1]=i$

现在，我们对 8 皇后问题进行求解，如图 1.3-1 所示。首先，把第 1 行的皇后放在第 2 列，即 r[1]=2；然后尝试将第 2 行的皇后放在第 4 列，即 r[2]=4；当{r[1]=2, r[2]=4}这个分支搜索完成后，进行回溯，回到 r[2]的位置，尝试{ r[1]=2，r[2]≠4}的分支。为了避免探索与{ r[1]=2, r[2]=4}同构的分支，我们来研究{ r[1]=2, r[2]=4}经过对称变换后得到的赋值情况。沿着 x 轴上下翻转，会将 r[1]=2 变为 r[8]=2，但 r[8]=2 不可能出现在{ r[1]=2，r[2]≠4}的分支中，因为它与 r[1]=2 冲突。同理可排除沿着 y 轴翻转，沿着对角线 d1 和 d2 翻转等三种对称，剩下右旋 90 度、180 度、270 度三种对称。只需要添加以下约束，即可使{ r[1]=2，r[2]

≠4}的子分支不与{r[1]=2, r[2]=4}的子分支同构：

- *r*90: *if r*[2]=8 *then r*[4] ≠ 7
- *r*180: *if r*[8]=7 *then r*[7] ≠ 5
- *r*270: *if r*[7]=1 *then r*[5] ≠ 2

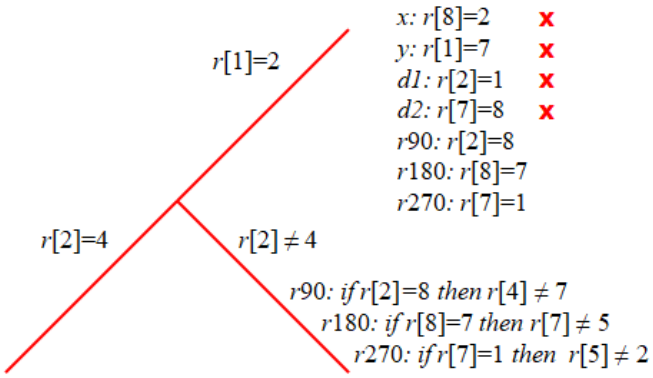


图 1.3-1　8 皇后问题的 SBDS 示例

实际上，由于 *r*[5] = 2 和 r[1]=2 冲突，*r*270 对应的打破对称约束也并不必要。

一般情况下，假设在搜索过程中，当前所有赋值的集合为 A，在当前变量 var 上进行分支搜索，在探索了 var = val 这个分支之后，回溯到 var，尝试 var ≠ val 的分支，对任意一个尚未被消除的对称 *g*，若 g(A)为真或者即将为真，则增加约束 g(var≠val)，如图 1.3-2 所示。

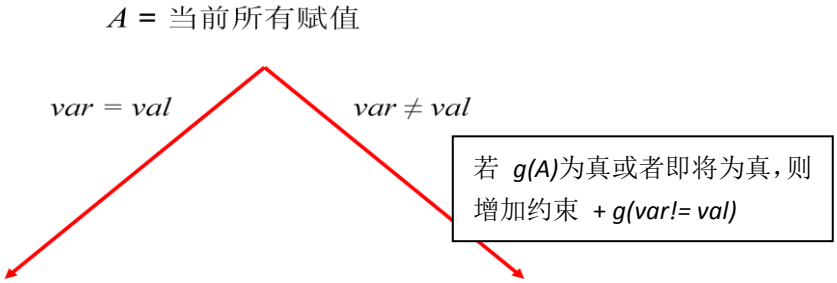


图 1.3-2

为了完全消除对称，SBDS 对每一个对称都需要给出打破对称的函数，这样做的问题是对称群可能及其庞大，从而带来巨大的额外开销。只有在少数特殊情况下，用较少的 SBDS 函数就可以完全打破对称。例如，在 n 个变量可互换的前提下，我们可以用针对变量两两交换的函数，就可完全消除对称，而不需要 *n*! 个 SBDS 函数。 介于上述情况，一般情况下，我们只消除一部分对称，即使这样，仍对搜索空间的剪枝非常有用。

除了上面两种方式，对 CSP 问题，还可通过添加约束，使得每一个等价类中只有一个（部分）解能满足此约束，从而消除冗余的（部分）解。这种方法的本质是建立了一个解更

少的新的 CSP 问题。它的缺点是容易引入错误和可能改变搜索顺序。第一点是人为因素，可以通过个人因素解决。对于第二点改变搜索顺序来说，则很可能会对效率产生影响。一般工业化的 CSP 问题大多有结构特点，针对该结构特点容易设计有效的搜索顺序，提高搜索效率。添加的打破对称约束可能改变了原问题的结构，进而改变了搜索顺序，有可能对搜索效率产生不利影响。这种方式的优点也是很明显的，它的剪枝效果好，更为关键的是，它实现起来非常简单，通过静态地添加约束，而不需要介入搜索过程。

Lex-Leader 约束是一种常用的打破对称约束。Crawford, Ginsberg, Luks 和 Roy 在 1996 年提出了添加 Lex-Leader 约束的方法，以打破 SAT 问题中的对称，后来也被用于打破变量对称。它为变量指定顺序，如 $x_1, x_2 \ldots, x_n$，变量的取值也应当有序，比如值域可能为整数、字母，或人为给值域中的元素赋以顺序。对变量的完整赋值，定义一个字典序，然后添加约束，使得每一个对称等价类中只有最小（大）的元素满足此约束，该元素称为 lex-leader。

现在我们讨论一种特殊的对称，叫作变量对称。它只作用于变量上，将 (x_i, j) 映射到 $(x_{\sigma(i)}, j)$ 。如果变量为 $x_1, x_2 \ldots, x_n$，则 σ 是对这些变量下标集合{1,2,…,*n*}的置换。变量对称可用 Lex-Leader 约束来破除。对每个变量对称 σ, 添加约束，使得对 $x_1, x_2 \ldots, x_n$ 的赋值应该字典序小于等于对 $x_{\sigma(1)}, x_{\sigma(2)}, \ldots, x_{\sigma(n)}$ 的赋值，即：

$x_1 x_2 \ldots x_n \leq_{lex} x_{\sigma(1)} x_{\sigma(2)} \ldots x_{\sigma(n)}$

该字典序约束等价于以下约束的集合：

$x_1 \leq x_{\sigma(1)}$；

if $x_1 = x_{\sigma(1)}$ then $x_2 \leq x_{\sigma(2)}$；

if $x_1 = x_{\sigma(1)}$ and $x_2 = x_{\sigma(2)}$ ， then $x_3 \leq x_{\sigma(3)}$;…..

if $x_i = x_{\sigma(i)}$ for $1 \leq i < n$， then $x_n \leq x_{\sigma(n)}$

例 6. 假设一个约束满足问题含有变量： $x_1, x_2 \ldots, x_6$，变量对称 σ 是{1,2,3,4,5,6}的置换 (1 3) (2 6 5)，设定变量序为 $x_1, x_2 \ldots, x_6$，σ 的 lex-leader 约束为:

$$x_1 x_2 \ldots x_6 \leq_{lex} x_{\sigma(1)} x_{\sigma(2)} \ldots x_{\sigma(6)}$$

由于 σ(1)=3, σ(3)=1, σ(2)=6, σ(6)=5, σ(5)=2, σ(4)=4，故有：

$$x_1 x_2 x_3 x_4 x_5 x_6 \leq_{lex} x_3 x_6 x_1 x_4 x_2 x_5$$

根据字典序约束的定义，只有在 $x_1 = x_3$ 且 $x_2 = x_6$ 的情况下，才会要求 $x_3 \leq x_1$，而此时后者是显然的，因此 x_3 和 x_1 可分别从$\leq_{lex}$的左右两边消去。同理，可在两边同时消去 x_4，以及分别消去 x_6 和 x_5。添加的 lex-leader 约束最终为：

$$x_1 x_2 x_5 \leq_{lex} x_3 x_6 x_2$$

对于字典序约束有一些理论结果，已经证明它是一致的和完备的。也就是说，如果我们对 CSP 问题的每一个对称都加上 lex-leader 约束，则有如下结论:

一致性: 在每个解的等价类中至少有一个解

完备性：在每个解的等价类中至多有一个解

如果对称群过于庞大，我们可以只加部分 lex-leader 约束，部分地打破对称，这样做可能丢失完备性。有时，一小组 lex-leader 约束也可完全打破对称。例如， 若变量 $x_1, x_2, \ldots, x_n$ 相互可交换, 只需添加 $x_1 \leq x_2 \leq \ldots \leq x_n$ 。

例 7. 假设一个约束满足问题 P = <X, D, C>，X={v_0,v_1,v_2}，D={0,1,2}，C={$v_0 \neq v_1$,$v_1 \neq$

$v_2, v_0 \neq v_2\}$。为打破对称，可以添加约束 $C'=\{v_0<v_1, v_1<v_2\}$。

现在我们考虑一种矩阵中常见的对称：行列对称。很多约束满足问题的变量可被建模为一个矩阵，比如覆盖数组（covering array）问题。

例 8. 覆盖数组 CA(N, $d_1\ d_2 \ldots d_k$,t) 是一个 N*k 的矩阵，第 j 列可取 d_j 个元素，对于任意 t 列，所有的元素组合都出现至少一次。图 1.3-3 展示了一个 CA(10, 2^5 ,3) ，可看到任意抽出 3 列，<0 0 0>, <0 0 1>,…,<1 1 1>的组合都出现了。给定参数 $d_1\ d_2 \ldots d_k$,t，生成最小行数 N 的 CA 是 NP 难问题。CA 中具有行列可交换的对称现象，交换 CA 的任意两行或任意两列，得到的矩阵仍是 CA。

0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

图 1.3-3 CA(10, 2^5 ,3)

对于一个 $m \times n$ 的矩阵， 若其行列可交换，则共有 $m!\ n!$ 种行列对称。此时如果对所有行列对称都引入 $\leq_{lex}$ 约束是不可行的。因此实际做法是仅对部分对称引入 $\leq_{lex}$ 约束。引入的时候要权衡利弊，寻找一个平衡：打破更多的对称可更好地剪枝搜索空间；但是添加更多约束又会增加约束传播的开销。

对于行列可交换对称，一种有效地打破对称方法是添加双字典序约束。双字典序约束禁止了相邻行/列交换引起的对称。假设一般情况下，一个 $m \times n$ 的矩阵如图 1.3-4 所示。

$x_{1,1}$	…	$x_{1,j}$	$x_{1,j+1}$	…	$x_{1,n}$
…	…	…	…	…	…
$x_{i,1}$	…	$x_{i,j}$	$x_{i,j+1}$	…	$x_{i,n}$
$x_{i+1,1}$	…	$x_{i+1,j}$	$x_{i+1,j+1}$	…	$x_{i+1,n}$
…	…	…	…	…	…
$x_{m,1}$	…	$x_{m,j}$	$x_{m,j+1}$	…	$x_{m,n}$

图 1.3-4

为了禁止交换第 i 行和第 i+1 行，添加：

$$x_{i1}\ldots x_{ij}\, x_{i,j+1} \ \ldots x_{in} \ \leq_{lex} \ x_{i+1,1}\ldots x_{i+1,j}\ x_{i+1,j+1}\ldots x_{i+1,n}$$

即，第 i 行 $\leq_{lex}$ 第 i+1 行

为了禁止交换第 j 列和第 j+1 列，添加：

$$x_{1j}\ldots x_{ij}\, x_{i+1,j}\ldots\, x_{m,j} \ \leq_{lex} \ x_{1,j+1}\ldots\, x_{i,j+1}\, x_{i+1,j+1}\ldots\, x_{m,,j+1}$$

即，第 j 列 $\leq_{lex}$ 第 j+1 列

需要注意，使用双字典序时，行、列都按字典序升序排列，或都按字典序降序排列。可以证明，任何一个矩阵，如果具有行列可交换对称，则总是可以通过交换行或交换列，调整到最终满足双字典序的形式。双字典序(lex^2) 约束被广泛应用于具有行/列对称的矩阵模型中。

字典序具有传递性，因此非相邻的行/列之间的打破对称约束是冗余的。双字典序(lex^2)约束消除了单独的行对称和列对称，但无法消除行和列同时交换引起的对称。

第二章 SAT 求解算法及相关问题

1 SAT 的相关定义

命题逻辑是最简单的一种形式逻辑系统。命题是它主要研究对象，每个命题有两种可能的值，一种为真一种为假。通常我们用 p,q,r,……这些符号代表任意的命题，用 T,F 或 1,0 来代表命题的真假值。这样的命题符号也被称为命题变量或布尔变量。

单个的命题变量，或者用连接符来连接起来的命题变量，就形成命题公式。常用的连接符是“与”(又称“合取”，记为∧或者&)，“或”(又称“析取”，记为∨或者|)，“非”(又称“否定”，记为┐或者-)，“蕴涵”(记为→)等等。这些连接符又被称为真值函数。它们的运算规则（真值表）是：

- 非:

P	┐P
0	1
1	0

- 合取

P	Q	P ∧ Q
0	0	0
0	1	0
1	0	0
1	1	1

- 析取

P	Q	P ∨ Q
0	0	0
0	1	1
1	0	1
1	1	1

- 蕴涵

P	Q	P→ Q
0	0	1
0	1	1
1	0	0
1	1	1

前文提到连接符连接命题变量就形成了命题逻辑公式，公式的正式定义是采用如下递归

形式定义的：

定义 1 （命题逻辑公式）

由命题逻辑变量和连接符按如下规则构成命题逻辑公式：

命题逻辑变量是公式，这样的公式被称为**原子公式**。

如果 φ 是公式，则(┐φ)也是公式。

如果 α，β 是公式，则(α*β)也是公式，其中*是任意的二元连接符，如∧，∨，→等等。

只有由上三条规则生成的表达式才是命题逻辑公式。

通常认为，判定一个命题逻辑公式是否有解的问题被称为 **SAT 问题**，不过因为命题逻辑公式的结构比较复杂，一般的 SAT 求解算法大都针对一个特殊结构的命题逻辑公式：合取范式，缩写为 CNF。该形式的公式结构简单。正式定义如下：

定义 2 （合取范式）

文字：原子公式或其否定形式为**文字**，原子公式本身被称为**正文字**，其否定被称为**负文字**;

子句：若干个文字的析取被称为**子句**，其**子句长度**是指其所含的文字的个数,只有一个文字的子句被称为**单子句**，没有文字的子句被称为**空子句**;

CNF：若干个子句的合取是一种特殊形式的公式，称为**合取范式(conjunctive normal form 简写为 CNF)**，其一般形式为：

$l_1 \vee l_2 \cdots \vee l_k$

这里的每个 l_i 是文字。

如前所述，命题变量是值域为 0,1 的变量，这里的 0 和 1 也可以看成假和真，或 F 和 T 等。给公式中的命题变量指派值就形成了赋值，关于赋值的一些正式定义是：

从命题变量集合到{TRUE,FALSE}的函数叫做**真值赋值**，简称**赋值（assignment）.**

如果一个赋值没有完全定义(即只有一部分变量被赋值)，那么称之为**部分赋值**。

给定一个赋值，由布尔运算的真值表，可以得到公式的值，使公式为真的赋值称为公式的一个**模型(model)**，同时称该公式为**可满足的(Satisfiable)**。如果对任意的赋值，该公式的值都为假，称该公式为**不可满足的(Unsatisfialbe)**。如果对任意的赋值，该公式的值都为真，称该公式为**永真的（Tautology）**。判断命题逻辑公式是否可满足的问题被称为**命题逻辑公式的可满足性问题(SATisfiablity problem,简称 SAT)**。因为当前的求解器大多接受 CNF 作为输入，一般也可以认为 SAT 问题是判定 CNF 公式是否可满足的问题，可见下文。

将赋值的概念应用到 CNF 公式上，下面的定义就比较直接明了：

赋值 t：变量到{0,1}的映射

- 对于文字 l： t(l) = 1 当且仅当：
  l 为正文字 x 且 t(x) = 1
  l 为负文字 ¬x 且 t(x) = 0
- 对于子句 $C=l_1 \vee \cdots \vee l_k$： t(C)=1 当且仅当
  对于某个 l_i 有 $t(l_i) = 1$
- 对于公式 $F= C_1 \wedge \cdots \wedge C_m$：t(F)=1 当且仅当
  对于所有的 C_i 有 $t(C_i) = 1$

可以看出，令一个子句为真，只需要使它的一个文字为真即可，令一个公式为假，只需要使它的一个子句为假即可。这种简单的结构，提供了算法设计上的便利性。因此大多数求解算法以 CNF 公式作为输入，而当前通用的可满足性问题的定义为：

定义 3 （SAT 问题）

给定一个 CNF 公式 F，判定它是否存在一个赋值 t，使得 t(F)=1。

从约束满足问题的角度来说，SAT 问题看做是一类特殊的 CSP 问题，即只包含命题逻辑变量的 CSP 问题。解决 SAT 问题的算法和工具被称为 SAT 算法和 SAT 工具或者叫 SAT 求解器。通常我们假定公式已经被转换为合取范式。一个 SAT 的完备求解算法能在有限的时间内，对于任意给定的 CNF 形式的命题逻辑公式(或者是任意的子句集合)判定它是否可满足。通常在可满足的情况下，SAT 算法往往会给出使子句集满足的一个赋值。

如果 SAT 问题中每个子句都最多包含 n 个文字，这样的 SAT 问题为 **n-SAT**。

SAT 问题是一个重要的问题，在理论意义上，它是最早被证明为 NP 完全的问题；对 SAT 问题算法复杂性的研究有助于解决计算领域最重要的问题：P 是否等于 NP 问题。早期的计算复杂性的研究，大多数是通过利用 SAT 进行归约进行的。

回顾一下 P 与 NP 的概念。NP 完全理论是为了确定不同问题的难度，检验各个问题在难易程度上的相互关系而建立的理论。P 和 NP 是该理论中的两个基本概念。其中 P 是指所有可以在多项式时间内用确定图灵机求解的判定问题的集合。而 NP 是指所有可以在多项式时间内用非确定图灵机求解的判定问题的集合。

如果一个问题属于 P，则它是易解的。如果一个问题属于 NP 但不属于 P，则它不一定存在多项式时间的算法。P 是否等于 NP 是在计算机科学领域中至今尚未解决的一个著名难题。人们一般认为 P 不等于 NP，也就是说，很可能存在一些 NP 问题，它们无法在多项式时间内用确定图灵机求解。

1971 年，Stephen Cook 在 ACM 计算理论年会(STOC)上发表的论文“The Complexity of Theorem-proving Procedures”奠定了 NP 完全性理论的基础。他在该文中证明了所有 NP 问题都可以在多项式时间内归约为 SAT 问题。他的证明是利用 SAT 问题对图灵机的运算过程进行编码来完成的。一般认为 SAT 问题是第一个被证明为 NP 完全的问题，早期研究中确立的其他的一些 NP 完全问题，大都是利用 SAT 进行归约得到。

根据 NP 完全问题的性质，只要解决了 SAT 问题，也就解决了所有的 NP 问题，因此可以认为 SAT 问题是 NP 问题中“最难”的问题之一。以目前的研究情况看，不太可能有多项式时间复杂度的算法。不过，对于一些特定形式的公式，已经找到了多项式时间的 SAT 算法。比如，2SAT 问题和 Horn 子句集的 SAT 问题。这些我们在后面章节中介绍。下面谈一下 SAT 在实际应用中的意义。

当前的 SAT 求解器发展迅猛，所能求解的问题的规模达到百万个变量，千万个子句的规模。SAT 求解器强大的能力使得其在实际中有很大的用途。例如它可以编码验证和测试电路图案问题，两个电路的等价性检测问题，软件的可达性分析以及程序的形式验证问题等。近年来 SAT 领域每年都举行一个国际学术会议，该会议上会举办 SAT 求解器求解效率的比赛，比赛所采用的例子很多来自工业届。这些例子是将一些实际中的工业问题编码成 SAT 问题进行求解。

2 2-SAT 问题和相变

SAT 是 NP 完全问题，但是如果对其添加一些约束，则有可能形成一些具有多项式时间判定算法的子问题，2-SAT 就是这样一类问题。

2-SAT 公式是每个子句最多只包含两个文字的 CNF 公式，判定 2-SAT 是否可满足是可以在多项式时间内完成的（事实上可以在线性时间内完成）。下面我们用一个比较简单的方式来说明这点。

消解(resolution，又称为归结)是 SAT 求解中的一类技术，它是将这样的子句 A∨p, B∨┐p 合成一个子句 A∨B，后者被称为消解结果，或者结语。这里 A 和 B 是一些文字的析取。对于消解，有这样的一个重要结论：

公式 1：F∧（A∨p）∧（B∨┐p）

公式 2：F∧（A∨p）∧（B∨┐p）∧（A∨B）

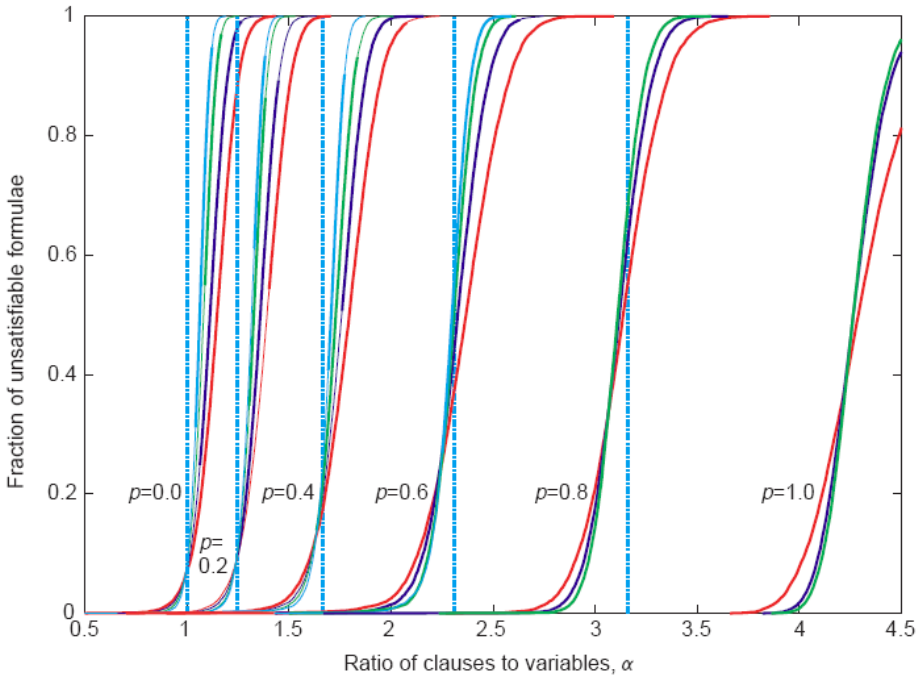
可以看出公式 2 是公式 1 加上消解结果构成，关于消解的重要结论就是公式 1 和公式 2 **可满足性等价**。这里的可满足性等价是指的两个公式具有相同的可满足性。如果消解出空子句，则证明该公式不可满足，例如公式中包含两个单元子句 p，┐p，消解后就得到空子句。

消解运算对于任意 SAT 公式都有效，对于 2-SAT 公式意义尤其重大，可以看出，对于 2-SAT 公式而言，其消解结果一定是最多只包含两个问题，也就是一个 2-SAT 公式添加上消解结果后，仍然是 2-SAT 公式。一个具有 N 个变量的 2-SAT 公式，所包含的不同的子句最多是多少呢？答案是 N（N-1）/2。这说明，如果一个公式是不可满足的，最多只需要 N（N-1）/2 次，就能得到空子句。因此 2-SAT 公式的可满足性判定是多项式时间的。

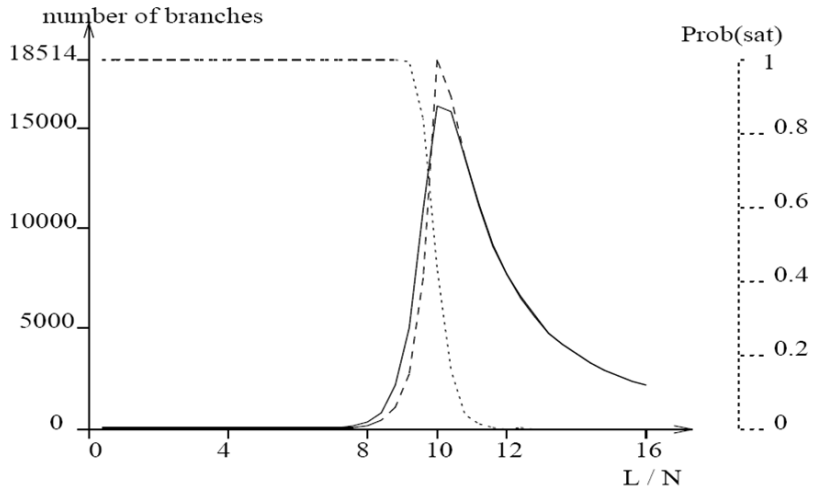
对于 2-SAT 公式的判定，还有其他的一些方法，大家可以参考一下相关资料。

对于普通的 SAT 问题，就无法用消解来设计多项式时间算法了，因为消解结果的长度无法限制。

对于 SAT 问题，相变是一个有意思的现象，也引起了一些学者的关注。对于一个随机 SAT 问题，变量和子句的数目对于公式的可满足性问题有何影响，是否有相变现象，是学者们所感兴趣的问题。在 SAT 问题上，相变有点类似突变的意思，例如，随机公式里变量和子句数目的关系中，变量多子句少使得公式倾向于可满足；反之倾向于不可满足。这种趋势不是渐进的，而是突变的。这个比例数值有一个比较窄的区间范围，这个范围的一侧，公式绝大多数可满足，另一侧的公式绝大多数不可满足，这种现象就是 SAT 中的相变现象。对相变的研究主要是寻找相变现象以及寻找导致相变现象的区间范围。下图是两个有意思的相变。



2-3SAT，p 为 2 子句的比例，不同颜色的线条表示不同数目的公式



相变现象能够揭示一些 SAT 问题中的规律，在理论上和实践上都有一定的意义。

3 SAT 的求解算法概述

最直接的 SAT 算法就是穷举法，事实上就是构造公式的真值表，由于命题变量有限，且每个变量只能取真和假两个值，所以这个算法在理论上是可行的，但是在命题变量数比较多时，它的效率就不足以满足实际需要了。可以想象的出，对于一个具有有 n 个变量时，

需要尝试 2^n 个可能的组合，这势必引起组合爆炸问题。因此对于实际上 SAT 问题的求解，往往应用一些启发式策略来生成比较可能满足布尔公式的赋值。目前没有多项式时间的 SAT 算法，现有的算法都不能说是很有效，一般来说，没有一个算法能在任何情况下都比其他算法效率更高，我们所能作的就是找到在特定条件下效率更高的算法。

求解 SAT 问题的算法一般被称为 SAT 求解器，它以一个 CNF 公式作为输入，以 0,1 作为输出，表示该公式是否可满足。有的求解器还可以给出满足的解，或者不满足的原因。目前也出现了一些以非 CNF 公式作为输入的 SAT 求解器，但是没形成主流。

SAT 算法主要可以划分为两大类：完全算法和非完全算法。完全算法理论上能保证正确判断任意的实例的可满足性，特别对于 UNSAT 实例的判断是非完全算法不可替代的。非完全算法不能证明 UNSAT，但是这类算法采用启发式策略指导搜索，在实例可满足的情况下，很多时候快于完全算法。他们的特点如下：

完全求解器：穷尽搜索空间，因此只要有足够的时间，一定能够给出答案。当前实际应用中的问题主要是采用完全求解器。

不完全求解器：大多采用局部搜索方法，也就是说随机给出一个赋值，然后在邻域内移动测试；因此当问题可满足的时候，有可能给出答案，也有可能给不出答案；当问题不可满足的时候，无法给出答案；不完全求解器比较适用于随机产生的 CNF 公式。

4 不完全求解器

我们本节简单介绍一下不完全求解器。这些求解器大都采用局部搜索算法。这类算法是一种非完全算法。它在算法中引入随机因素，以牺牲求解的成功概率，达到每次求解的高效。在 SAT 问题研究中，随机算法已成为高效算法设计的一种重要的途径。

20 世纪 80 年代末 90 年代初，非完全算法取得了突破性进展，Selman 和顾钧（Jun Gu）分别提出了 GSAT 算法和 GU 算法，这些局部搜索算法当时可以求解数千个变量规模的 SAT 问题，而当时最好的完全算法只能求解到变量数为 400 左右规模的问题。

后来的 GenSAT 算法、WalkSAT 算法、Novelty 算法等几乎所有的局部搜索法都是随机算法。在上个世纪末的时候，它们的求解效率是相当高的。但是在本世纪没有突出的进展，因此，相应的 SAT 求解器被很多完全算法的求解器超过。近几年国内的学者研究出格局检测技术，极大的提升了不完全求解器的效率。下面较为详细的介绍一下 SAT 的不完全求解器。

SAT 求解的不完全算法大都采用局部搜索，局部搜索是解决很多 NP-hard 问题（特别是优化问题）的一种常见的方法。相对于一个一个地确定变量的值，逐步地将部分解扩充为完整的解的回溯算法，局部搜索法的基本思想是，先任意地给每个变量取一个值，得到一个赋值，然后反复对现有的赋值作局部调整（如将一个或者几个变量的取值翻转等等），使得被满足的子句越来越多，在理想的情况下，最终将得到一个解，它使得所有子句被满足。

局部搜索法是非完全算法的代表。这类方法求解可满足实例普遍快于完全算法，求解的

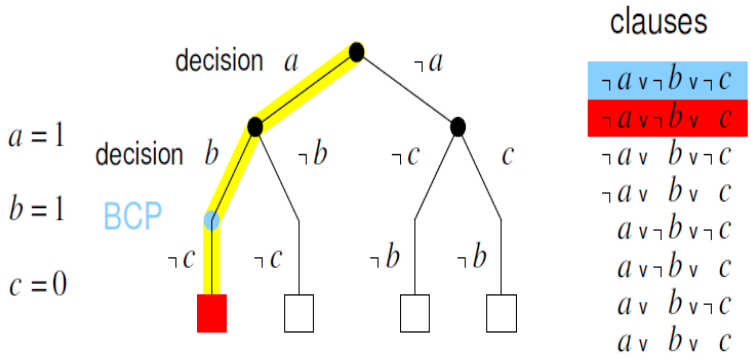
规模也比完全算法大的多，受到理论界和工程界的重视。局部搜索法翻转变量的策略主要有两种：一是优先翻转未满足子句中的变量，二是以满足子句数为爬山目标函数。但是作为非完全算法的局部搜索法，缺少完全算法的完备性，无法给出不满足实例的证明，导致该类算法在不可满足实例上计算失败。另外，由于有可能在运行过程中陷入局部最小，导致长时间找不到满足的解，也使得这类算法有一定的局限性。

5 DPLL 算法

DPLL 是一种回溯搜索算法，属于完全算法。当前的主流 SAT 求解器大都基于 DPLL。1960 年，Davis 和 Putnam 提出了解决 SAT 问题的一个重要的完全算法，后来被称为 Davis-Putnam 算法，简称 DP。Davis，Logemann 和 Loveland 对它作了进一步的描述，所以它有时也被称为 DPLL。DP 算法目前仍然是最有效的完全算法之一，现在完全算法中有不少是该方法的改进或者变种。

我们首先从一个简单的 DPLL 算法过程讲起，然后再探讨对该技术的一些改进以及关键问题。

DPLL 算法是一个回溯搜索算法,它的基本思想是每次选中一个未被赋值的变量进行赋值，然后判断该赋值是否满足整个公式：如果满足则结束搜索；如果导致冲突（某个子句为 0）则回溯；如果既不满足也没有导致冲突，则对下一个变量进行赋值。



图示是一个 DPLL 算法执行的例子，是一个树状结构。分别给变量 a,b,c 赋值，给变量赋值为 1 用变量本身表示，赋值为 0 用变量的非表示。当给变量 a,b,c 分别赋值为 1,1,0 的时候，遇到了冲突，这时需要回溯到 b，重新给 b 赋值。

可以看出，DPLL 算法通过隐式的枚举问题变量的取值空间中的可能赋值来实现的。它要作的就是找到一个满足公式的赋值。从一个空的赋值开始，搜索算法通过搜索决策树来寻找可能满足公式的赋值。

下面给出 DPLL 算法框架：

算法 1 （DPLL）

- **Sat_solve( )**

```
if  preprocess( ) = CONFLICT  then return UNSAT
while  TRUE  do
    if not decide-next-branch( )  then return SAT
    while  deduce( ) = CONFLICT do
        blevel ← analyze-conflict( )
        if blevel=0 then return UNSAT
        backtrack(blevel)
    endwhile
endwhile
```

以上算法框架是目前主流 SAT 求解器所采用的算法框架，这些求解器的主要区别是一些细节技术不同。算法主要分为四个部分：

decide-next-branch() 是选择一个没有被赋值的变量，并给定一个赋值；

deduce() 要进行单元值传播（unit propagation），也就是对字句中那些必须进行赋值的文字进行赋值。例如当有一个子句变成单子句的时候，必须将剩下的那个文字赋值为真。这样的操作会一直到不能进行下去为止（因此叫传播）。这种传播有时也被称为 BCP (Boolean Constraint Propagation)

analyze-conflict()是遇到冲突时找出导致冲突（一个变量既被赋值为真又被赋值为假 ）的原因，有的时候还会添加一些新的子句来约束后面的搜索。

backtrack()进行回溯，并取消一些赋值和他们诱导出的赋值。

下面的例子说明这个算法的执行过程：

- f = a (b + c + d) (b’ + c) (b’ + d) (x’ + y’) (x + z’) (x’ + b’ + y) (x + b’ + z) (c + d + y’ + z’).
  - a = 1.
- f = (b + c + d) (b’ + c) (b’ + d) (x’ + y’) (x + z’) (x’ + b’ + y) (x + b’ + z) (c + d + y’ + z’).
  - 决策(分支)变量: b = 1.
- f = c d (x’ + y’) (x + z’) (x’ + y) (x + z) (c + d + y’ + z’).
  - c = 1, d = 1.
- f = (x’ + y’) (x + z’) (x’ + y) (x + z).
  - 决策变量: x = 1.
- f = y’ y .
  - 冲突
- f = (x’ + y’) (x + z’) (x’ + y) (x + z).
  - 翻转最近的决策变量赋值，i.e., x = 0.
- f = z’ z.
  - 还是冲突，翻转再上一个决策变量赋值， i.e., b = 0.

- f = (c + d) (x’ + y’) (x + z’) (c + d + y’ + z’).
  - 决策变量: x = 1.
- f = (c + d) y’ (c + d + y’ + z’).
  - y = 0.
- f = (c + d).
  - 决策变量: c = 1.
- f = 1.

对 DPLL 算法的改进，主要是集中在三个方面：分支策略 decide-next-branch()，传播 BCP 和冲突分析 analyze-conflict()。

分支策略是选择下一个将要被赋值的变量和所进行的赋值，分支决策变量的选取显著的影响了搜索树的深度。分支策略在本质上就是约束搜索的方向，尽量使得当前选择的决策变量的赋值能诱导出最多的变量赋值。主要是依赖下面的一些直观的思想：

出现频繁的文字似乎是自然的选择；

搜索应该局部化，即，冲突分析后，应该选择那些导致冲突的文字做为决策文字；

每一个变量都分配一个计数器，子句中出现一次变量，该变量就累加一。我们可以选择没有赋值的变量中，该值最高的变量做为决策变量。

根据上述思路，形成了一下几类重要的分支策略：

MOM（Maximum Occurences in clauses of Minimum size）策略优先考虑在最短子句中出现次数最多的文字，统计变元出现次数时，已被满足的子句忽略不计。也可以优先考虑最短正子句中出现次数最多的文字，即最短正子句优先策略。

VSIDS（Variable State Independent Decaying Sum）是一类重要的决策策略，也是当前主流的分支策略。在早期 GRASP 的 DLIS 策略基础上，作者提出了基于文字计数的启发式策略，这些计数器记录了每个变量的真假取值会导致多少子句成假。这些计数器是状态相关的（State-dependent），因为不同的赋值状态下，计数器的值可能会有不同的变化。因此，这些计数器的值必须在每一个变量赋值和取消赋值的时候进行更新。后来 Chaff 提出的 VSIDS 策略给每一个文字设置一个计数器，初始状态下，这些计数器的值等于输入的 CNF 公式中该文字出现的次数。每当冲突分析产生了一个新的学习子句的时候，这些计数器的值就会被更新，在学习子句中出现的文字的计数器的值会被增加 1。而且，周期性的，在进行了大量的决策后，所有的计数器的值都会减半。因此，VSIDS 的基于文字的记数实际上加大了近期的学习子句中出现的文字的权重。当需要选择决策变量的时候，VSIDS 选择没有被赋值的变量中分数最高的变量作为决策变量，如果该变量的正文字的分数比负文字的高，则赋值为真，反之亦然。VSIDS 提供了一个类统计变量序，而且强制求解器优先在最近的那些冲突局部搜索。VSIDS 的统计开销相对较小，这使得 VSIDS 成为了一个开销较小的决策策略。

大量的实验表明，增加决策的局部性，有利于显著的消减搜索空间。常见的两种增加决策局部性的策略如下：

Variable Ordering Scheme for VSIDS（VSIDS 基础上的变量序模式）：这个是 zChaff 默认的决策策略。一个增加 VSIDS 局部性的方法就是增加计数器的值衰减的频率。

BerkMin Type Decision Heuristic（类 BerkMin 的启发式决策）：VSIDS + Conflict clause resolution, 它在最近的一个没有被满足的冲突子句中寻找决策变量，这是 BerkMIn 采用的一种费效比很低的好途径。

Conflict Clause Based Assignment Stack Shrinking（基于冲突子句的赋值栈收缩）：这个策略是在当最新得到的 UIP 学习子句超过了一个固定的长度时采用的。UIP 学习子句的含义在下文冲突学习机制中有介绍。

下面再看一下对 BCP 的改进。实验表明，BCP 占据了求解的大量时间（大概 80%），因此该技术是提高求解效率的关键。一般采用的方式是避免检查所有的子句是否变成了单子句。对每一个子句选择 2 个文字作为观察文字（watching literals ），仅当有一个观察文字为假的时候这个子句才可能变成单子句，因此算法只检查这样的子句，这样就节省了大量的时间。下面就 BCP 的细节技术和发展历史做下介绍。

早期的 BCP 过程实现会给每一个子句一个已经被赋值的文字数的计数器，用来标记子句是单子句还是冲突子句（conflicting clause）。后来有学者提出了一个应用首尾链表的 BCP 机制，从而大幅度提高了 BCP 的效率。但是在基于计数的模式和首尾链表的模式下，取消一个变量的赋值都是一个开销巨大的操作，它的复杂度甚至可以和给一个变量赋值相比。

随后学者提出了 Two Literal Watching 模式。每一个子句都是由它包含的文字构成的数组。初始状态下，每个子句中两个没有被赋值的文字被标记为观察文字(Watching Literal)。每一个文字都有一个子句列表，列表中的子句都是以该文字作为观察文字的子句。如果不将至少一个观察文字赋值为假，一个子句是不可能变成单子句或者冲突子句的。因此，当一个文字被赋值为假的时候，BCP 过程只需要访问以该文字作为观察文字的子句就可以了。

Two Literal Watching 模式的最关键的好处就是，在回溯的时候，不需要修改子句数据库中的观察文字。撤销一个变量的赋值只需要简单的将这个变量的值赋值为 Unknown 就行了。更近一步，将一个曾经赋值过的变量重新赋值的相关操作会比它第一次赋值的时候要快。这是因为一般情况下将这个变量作为观察文字的子句会比上一次赋值的时候少。因此，这将减少内存总的访问量。下面图示是一个具体例子：

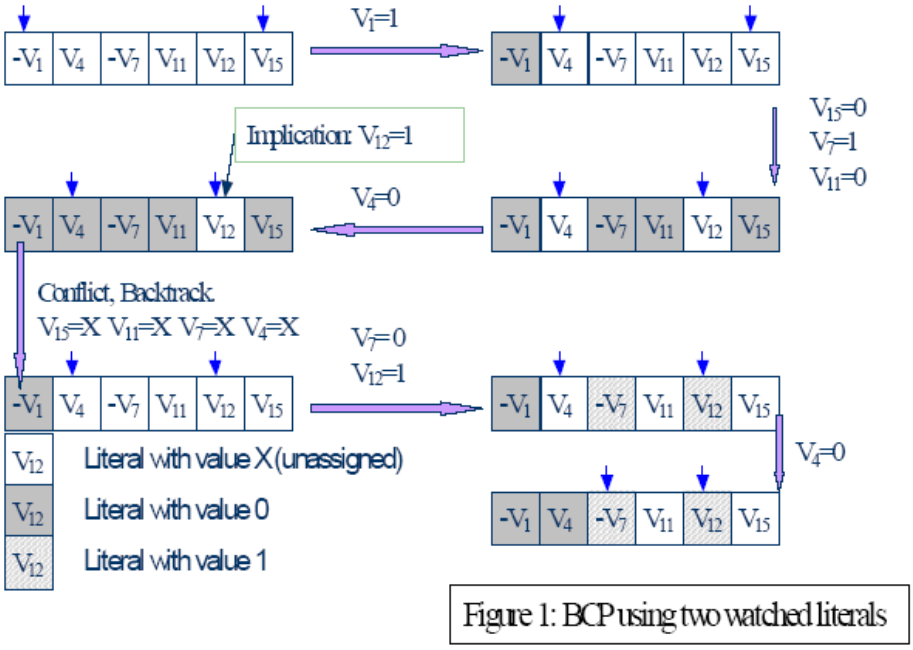


Figure 1: BCP using two watched literals

冲突分析技术相对来说比较复杂，其目的主要是避免重复空间的搜索。例如一个公式，如果按照 a, b, c, d, e 的顺序搜索，那么如果冲突分析得出是 a 导致的冲突，那么我们不需要按照年代顺序一步步回溯回去，直接跳到 a 即可，这样就避免了重复的子空间搜索。冲突分析主要是通过添加一些子句来避免重复的子空间搜索。一个简单的例子：假设有两个互相冲突的单子句 P 和 Q。 考虑 P 和 Q 中那些导致他们成为单子句的变量。 如果那些变量是 a, b, c (他们的赋值为 1)和 d, e, f (他们的赋值为 0)，那么我们将添加子句 (a' + b' + c' + d + e + f)。下文详细的介绍一个经典的冲突分析技术。

冲突分析发生在 BCP 过程侦测到在当前赋值下产生了冲突，这时求解器需要回溯。冲突分析过程会得到一个当前赋值集合的子集，正是这个子集中的变量赋值导致了冲突的产生。求解器记录这些赋值并构造一个子句（Conflicting clause）使得当这个子集中的赋值出现时，这个子句为假。这个子句就能对搜索空间进行剪枝，而且避免这个冲突再次出现。

新的学习子句（Conflicting clause）又被称为引理（lemma），它可以是通过两个等价的途径得到，一个是归约（Resolution），另一个是蕴含图的切（a cut in the implication graph）。更多细节请阅读[7]。由于学习子句是由 2 个已有的子句作 resolution 得到的，所以，它在逻辑上是冗余的，也就说说，添加这个子句并不影响问题的可满足与否。但是，添加这些 lemma 会极大地影响 BCP 过程，从而剪枝掉一些搜索空间。

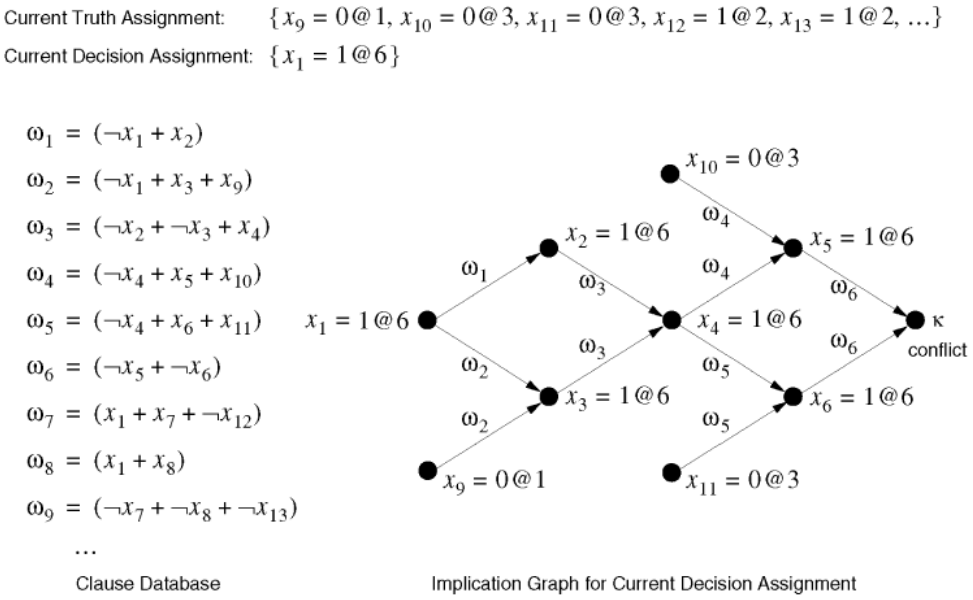
一个很根本的问题是，应该添加哪些子句归约出来的学习子句。这个问题有很多答案。在冲突驱动的子句学习中，求解器的搜索引擎可以发现一组适合做归约的子句，上面提到了，每一个非决策变量的赋值都是有一个祖先子句（antecedent clause）的。如果一个变量 v 通过 BCP 过程被赋值为 1，那么它的祖先子句将包含 v 这个正文字而且其他文字都被赋值为假。学习子句 Cf 仅包括这些为假的文字。因此，Cf 可以通过 v=0 和 v=1 的两个祖先子句进行归约得到。

在下图这个例子中，我们给出了 CNF 公式的一个子集，我们假设当前进行到第 6 次赋值操作，称为决策等级（decision level）是 6，对应的决策赋值是 x1 = 1。如图所示，这个赋值导致了一个涉及到 w6 的冲突。通过检查蕴含图（Implication Graph），我们很容易就可以看出导致这个冲突的充分条件是：

(x10 = 0) & (x11 = 0) & (x9 = 0) & (x1 = 1)

通过添加子句 w10 = (x10 + x11 + x9 + not x1)，我们避免了在将来的搜索中重复这个冲突。

再后来有学者进一步提出了 1UIP 或者叫 first UIP（UIP 是 Unique Implication Point 的缩写，英文解释是 vertex that dominates vertexes leading to conflict）的概念，从而使得学习得到的子句更加短小精悍。在下面的例子中 x1 是 UIP (导致冲突的决策变量当然也是一个 UIP)，同时 x4 也是 UIP。按照 1UIP 的算法，添加的子句为 w11 = (x10 + not x4 + x11)。和 w10 比起来要短。



一个冲突可以通过分析得到很多不同的冲突子句，而全部保留的话，程序的空间存贮开销就过于庞大了，因此，选择什么样的冲突子句进行存贮就是一个必须要解决的问题。一个自然的想法就是，尽量保留长度较短的子句（也即包含较少的文字）。越短的学习子句剪枝的搜索空间越大，从而能导致更快的 BCP 过程和更快的检测到冲突。此外还有一些其方法，例如一些学者推荐仅保留最靠近冲突的。

除了上述提到的问题外，从程序的鲁棒性考虑，有人提出了重启（Restart）的概念。其目的就是避免搜索陷入到一个需要接近指数时间搜索的子空间。但是，判断一个搜索是不是陷入了一个难解的子空间是很难的，因此，很多求解器采用的是近似的判定方法，这些方法都是基于冲突的计数。最初级的方法就是，冲突出现 n 次后就 Restart。再进一步就是，Restart 的阈值 n，在每次重启后都乘以一个系数 m。

Restart 的另一个问题就是，保留多少 lemma，保留什么样的 lemma。因为现在主流的

SAT 求解器都是确定性算法了，所以，如果不保留 lemma，每次的搜索过程就会一样，得不到变化。显然，保留尽量少的 lemma，同时保证约束比较强，这样的思路是很正确的。因此，很多求解器就是保留长度最短的那些 lemma。关于保留什么样的 lemma，这个问题的研究还是比较初步，有很多尝试的余地。

对于一个 SAT 求解器来说，存储公式的数据结构对于软件的效率是有很大影响的。而且，数据结构决定了 BCP 过程，冲突分析等技术的实现的具体手段和效率。有大量的 SAT 程序采用链表或者类似的结构表示子句以及子句集，这种想法比较自然，但是对于应用上有一些障碍，比如，子句间的关系不明显等等。因此，有一些其他的数据结构被提出，比如采用稀疏矩阵存储，采用 trie 结构存储等。采用 trie 结构的好处是，节省空间，并且能有效地查出单子句和多余的子句。另外，由于多个子句可能共享几条边，而且边上的标记是从小到大排序，所以能比较简单地对多个子句实施单子句规则。不过，由于 trie 结构上的操作比较复杂，这给编程增加了难度。自从 two literal watching 策略的提出并给出相应的数据结构后，此后的大多数确定性求解器都采用了和这个本质上一样的数据结构。

关于并行处理，在 SAT 领域也有人做了尝试。对于 NP-hard 问题，采用并行处理并不能从根本上解决问题，但是在有些情况下可能能较快的得到结果。实际上就是在多个分支同时搜索时，有可能在某个分支上较快的找到解。一种处理的途径是，把问题实例分解成几个小的实例交给不同的进程同时搜索；另一种是，把搜索空间划分成几个不相交的子空间，再交给不同的进程同时搜索。

6 MAX-SAT 问题

MAX-SAT 是对 SAT 问题的一个扩展。SAT 问题是判定公式是否可满足，MAX-SAT 是求得一个满足公式里最多子句数目的解。例如：

$$(x1 \vee \neg x2) \wedge (\neg x1 \vee x2 \vee x3) \wedge (x2 \vee \neg x3) \wedge (x1 \vee x3)$$
$$\wedge (\neg x1 \vee \neg x2 \vee \neg x3) \wedge (\neg x1 \vee x3)$$

这个例子，令

x1=1,x2=1,x3=1，将会满足 5 个子句，是该 MAX-SAT 问题的最优解。

MaxSAT 是 SAT 的优化版本，是 NP 难问题。 它的求解基本上是由基础的 SAT 算法而来。

MAX-SAT 还有一些变种或者扩展，Weighted MaxSAT 问题是带权的 MAX-SAT 问题，在这列问题中，每个子句都被赋予一个数字作为权值。求解的最终目的也就由求出满足最多的子句的赋值，变为求出所满足的子句的权值最大的赋值。

Partial MaxSAT 是 MAX-SAT 的另一个扩展，它把一个公式的句子分为硬子句和软子句，所求得的解需要满足所有的硬子句，并且满足尽可能多的软子句。

Weighted Partial MaxSAT 是前面两者的结合，公式的子句分为硬子句和软子句，并且子句附有权值，所求得解要求满足所有硬子句，并且使软子句的权值尽可能的大。

7 SMT 问题

随着研究的深入，人们发现 SAT 在表达能力上有很大的局限性，许多应用用 SAT 进行编码并不是很明智的选择，它们需要比 SAT 更强的表达方式。在这种形势下，人们将 SAT 问题进行了扩展，扩展为 SMT，经过扩充，SMT 能比 SAT 更好地表达一些人工智能和形式化方法领域内的问题，比如在资源的规划，时序推理，编译器优化等方面的应用很多用到了 SMT。本节介绍 SMT 以及相关求解技术。

SMT 的全称是 Satisfiability Modulo Theories，可被翻译为“可满足性模理论”、“多理论下的可满足性问题”或者“特定（背景）理论下的可满足性问题”。其判定算法被称为 SMT 求解器。简单地说，一个 SMT 公式是结合有理论背景的逻辑公式，其中的命题变量可以代表理论公式。可以这样说，SMT 公式是结合了背景理论的一阶逻辑公式，这些理论包括一些数学理论和计算机领域内用到的数据结构理论等。这种结合的体现是在 SMT 公式中，命题变量有时会被解释为背景理论公式。例如下面的公式：

例 1： $x+y<3 \wedge y>2$

这是一个 SMT 公式，它的逻辑形式是 A∧B，其中命题变量 A 和 B 分别被解释为数学公式 x+y<3 和 y>2。

给定一个 SMT 公式，在通常的逻辑解释和背景理论解释下，如果存在一个赋值使该公式为真，那么我们称该公式是可满足的；否则就称该公式是不可满足的。这样的赋值被称为模型。

由于使用了背景理论，SMT 有着比 SAT 更灵活的表达方式，可以方便地表示一些工业上和学术上的问题。目前的 SMT 求解器能处理的理论主要有一些数学的理论，一些数据结构的理论，还有未解释函数。如果细分的话，主要有以下几个：

(1)**UF**：这个理论的全称是 Uninterpreted Function，未解释函数。它主要包含一些没有经过解释的函数符号和它们的参数，比如 f(x), g(y+1,z)等。

通常这个理论带有等号，因此它也被称为 EUF。下面是一个 EUF 的例子：

例 2：$\{a=b, b=f(c), \neg(g(a)=g(f(c)))\}$

在这个例子中，大括号里的公式属于合取关系（如前文所述）。这个例子是不可满足的。因为如果将¬(g(a)=g(f(c)))中的 a 用 b 取代，再将 b 用 f(c)取代，即可得到¬(g(f(c))=g(f(c)))，这个公式显然是不成立的。

(2)**LRA** 和 **LIA**：分别是 Linear Real Arithmetic 和 Linear Integer Arithmetic, 线性实数演算和线性整数演算。这两类理论的公式形如：

$$a_1x_1+\ldots+a_nx_n \odot c$$

这里⊙可以是=、<=、>=、≠, c 是一个常数，$a_1 \ldots a_n$是常系数，$x_1 \ldots x_n$是实数变量（在 LRA 中）或者整数变量（在 LIA 中）。下面是一个例子：

例 3：$\{x-2y=3, 4y-2z<9, x-z>7\}$

这组公式可通过变量替换化为{x-2y=3,4y-2z<9,2y-z>4}，并进一步变换为{x-2y=3,8<4y-2z<9}。在 LRA 理论中，8<4y-2z<9 是有解的，因而这个例子是可满足的。但

8<4y-2z<9 没有整数解，所以在 LIA 理论中，这个例子是不可满足的。这两类理论统称为 LA。

⑶**NRA** 和 **NIA**：分别是 Non-Linear Real Arithmetic 和 Non-Linear Integer Arithmetic, 非线性实数演算和非线性整数演算。公式的形式为任意的数学表达式。

⑷**RDL** 和 **IDL**：分别是 Difference Logic over the Reals 和 Difference Logic over the Integers, 即实数和整数的差分逻辑。它的一般形式为：

$$x\text{-}y \odot c$$

这里⊙可以是=，〈=, 〉=, ≠, c 是一个整数或者布尔值，x 和 y 可以是实数（在 RDL 中）或者整数（在 IDL 中）。这两类理论总称为 DL。

为了更好的表示应用中的问题，SMT 公式经常包含两种以上的理论，例如 UFBV 指的是包含未解释函数和位向量的公式，UFLIA 指的是包含未解释函数和线性整数演算的公式

SMT 的判定算法大致可分为积极（eager）类算法和惰性（lazy）类算法。前者是将 SMT 公式转换为可满足性等价的 SAT 公式，然后求解该 SAT 公式，后者是结合了 SAT 求解和理论求解，先得出一个对命题变量的赋值，然后再判断该赋值是否理论一致。其中 SAT 求解用的是 DPLL 算法(Davis–Putnam–Logemann–Loveland algorithm)。

积极算法是早期的 SMT 求解器采用的算法，它是将 SMT 公式转换成一个 CNF 型（conjunctive normal form，合取范式型）的命题公式，然后用 SAT 求解器求解。这种方法的好处是它可以利用高效的 SAT 求解器，它的求解效率依赖于 SAT 求解器的效率。早期 SAT 求解技术取得的重大进步推动了积极算法的发展。对于不同的理论，积极算法需要用不同的转换方法以及改进方法，这样有助于提高转换和求解的效率。例如对于 EUF 用到了 per-constraint 编码，对于 DL 通常用 small-domain 编码等。

积极算法的正确性依赖于编码的正确性和 SAT 求解器的正确性，而且对于一些大的例子来说，编码成 CNF 公式很容易引起组合爆炸，也就是公式的长度指数级增长，因此这类方法在实际应用中效果不是很好，解决不了很大的工业界例子，目前主流的 SMT 求解器大都采用惰性类算法。

惰性算法是目前采用的主流方法，也是被研究得最多的算法。这种算法是先不考虑理论，将一个 SMT 公式看做是 SAT 公式求解，然后再用理论求解器判定 SAT 公式的解所表示的理论公式是否一致。目前主流的 SMT 求解器大都采用了惰性算法。可以看出这类算法结合了 SAT 求解和理论求解。在介绍该算法之前，首先介绍一个术语。前文说过 SMT 是逻辑公式和背景理论公式的结合，我们称一个 SMT 公式中的逻辑公式为它的**命题框架**，比如这样的 SMT 公式：

例 6：$x<2 \wedge x+y>3 \rightarrow y>2$

这个公式的命题框架是 A∧B→C, 其中 A, B, C 分别代表理论公式 x<2，x+y>3 和 y>2。对于一个 SMT 公式 F，我们用 PS(F)来表示它的命题框架。

我们首先介绍判定命题公式的 **DPLL 算法**。算法如下：

输入：一个 CNF 公式 F

输出：真，假

1. 对 F 进行预处理，如果公式为假，返回假

2.选择下一个没有被赋值的变量进行赋值

3. 根据赋值进行推导

4. 如果推导出公式为真，则返回真

5. 如果推导出冲突，则

6. 分析冲突，如果能回溯，则回溯

7. 如果不能回溯，则返回假

8. 如果没有推导出冲突，返回 2

算法 1：DPLL 算法

第 1 行的预处理是检查公式的结构，看看是否存在冲突的子句，这个检查消耗的代价很小。如果在预处理阶段没有发现冲突，则按照一定的策略对公式中的变量进行赋值。其中第 3 行的推导，主要指这样一个操作：对某个变量赋值后，某些子句中的变量就会被强迫赋值，例如子句 a∨b,如果给 a 赋值成 0（假），那么 b 就必须被赋值成 1，这个步骤可以不断进行下去。推导的作用主要是找出这样的文字，并扩充赋值。第 5 行中的冲突指的是推导出不可能的赋值，比如一个公式中存在这样两个子句 a∨b 和 a∨﹁b，如果给 a 赋值成 0，那么前一个子句要求 b 为 1，后一个子句要求 b 为 0，这时就产生了冲突。产生冲突后，用回溯算法返回前面的变量，重新进行赋值。如果没有冲突，而且公式已经被满足，这时就可以返回真，否则要对进行下一个变量的赋值。

惰性算法通常又被称为 **DPLL（T）算法**，T 代表理论求解器。这个算法的一般形式是：

输入：一个 SMT 公式 F

输出：真，假

1.得到 F 的命题框架 PS(F)

2.如果 PS(F)是不可满足的，返回假

3.否则对于 PS(F)的每一个模型 M，检查 M 所代表的理论是否一致

4. 如果存在一个模型是理论一致的，返回真

5. 如果所有的模型都不是理论一致的，返回假

算法 2：DPLL（T）算法

上面算法中的模型 M 指的是 PS(F)的一个解。第 2 行用到的是 SAT 求解器，第 3 行中取得模型的方法也是用 SAT 求解器，检查模型的理论一致性用相应的理论求解器。SAT 求解器一般用到 DPLL 算法。一个逻辑公式的解（模型）可以看做是一组文字的合取，检查模型的理论一致性就是检查这组文字代表的理论公式是否有解，**理论一致**指的是这组文字代表的理论公式有解。下面用一个例子来说明一下算法的执行过程。给定 SMT 公式：

例 7：$F=(x>2\vee y<5)\wedge x<1$

算法首先取得它的命题框架 PS(F):(A∨B)∧C，其中 A,B,C 分别代表 x>2，y<5，x<1。然后算法发现 PS(F)是可满足的，因此检查它的模型的一致性。首先会检查模型{A,C}，也就是检查 x>2 和 x<1 是否有解，这种检查是通过理论求解器来进行的。理论求解器发现模型{A,C}无解，然后算法检查第二个模型{B,C}，也就是 y<5 和 x<1。这个模型有解，因此算法返回真，说明公式 F 是可满足的。

早先的 DPLL(T)算法是将 SAT 求解器（DPLL 部分）当做黑盒，也就是说理论求解器检查模型的一致性要等到 DPLL 算法给出一个解之后才进行，这种算法有时也被称为 offline 方

法。这种方法的缺点是理论求解器很少参与 DPLL 的求解过程。后来出现了 online 方法对其做了改进，使得理论求解器参与 DPLL，从而提高了求解效率。这些改进主要有以下几种。

理论预处理：这种改进是对 SMT 公式进行预处理，对其进行简化和标准化，必要的时候可以修改模型中的理论公式。理论预处理可以和 DPLL 预处理相结合。

选择分支：在 DPLL 中选择下一个被赋值的命题变量是很重要的。在 DPLL(T)中，可以使用理论求解器来帮助选择下一个被赋值的命题变量。

理论推导：这种方法主要是通过理论的帮助来推导出一些文字。在 DPLL 中推导是一个提高效率的重要手段。理论推导可以导致三种结果：第一是推导出一个文字，该文字和目前的部分模型冲突，这个时候需要回溯；第二是推导出一个可满足的模型，这时就可以返回模型，结束求解；第三是推导终止后，既没有得到可满足的模型，又没有发生冲突，这时需要进行下一步的赋值。

理论冲突分析：在 DPLL 中，冲突分析有助于回溯到早期的变量，从而提高求解效率。在 DPLL(T)中这种分析是通过理论求解器来进行的。

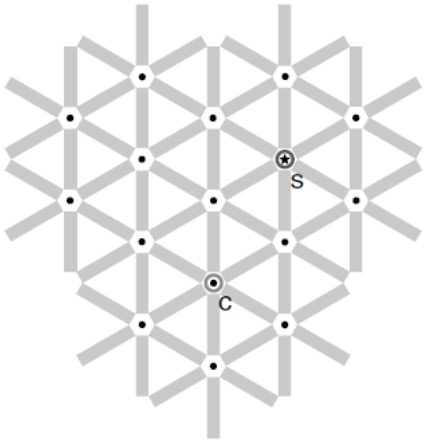
第三章 启发式算法

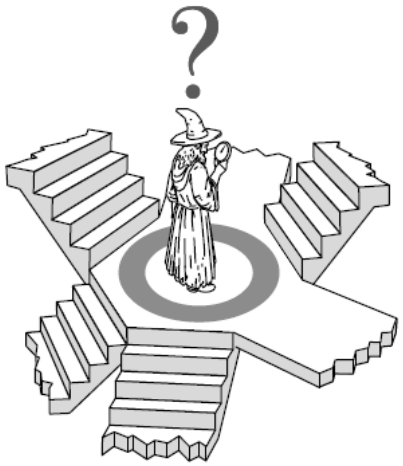
本章介绍启发式算法。对于某些问题，当普通的完全算法效率低下时，可以采用启发式算法进行求解。启发式算法不像完全算法一样能得到解，但是具有高效性，所付出的代价一般是解的精度。启发式算法又分为很多类，本节介绍主要的三类启发式算法，分别是局部搜索算法，模拟退火算法和遗传算法。这三类算法也是在实际应用是使用较多的算法。

1 局部搜索

我们首先看一下局部搜索基本概念。局部搜索是一类用于解决困难的组合问题的方法，目前应用的最为广泛。局部搜索更像是一类算法框架，而不仅仅是特定的算法。对于不同的组合问题，人们往往会设计相应的局部搜索策略，或者算法。我们这一种介绍一些主要的局部搜索方法以及重要的组成部分，这一部分使用的例子主要是 SAT 问题和旅行商问题（TSP）

局部搜索算法框架：局部搜索的过程大致如下，给定了需要解决的组合问题之后，搜索发生在候选解空间内，也就是在这些所有的候选解中寻找一个最优解。它首先选择一个初始的候选解，当然该解的形式要根据上下文决定，例如可满足问题，初始解将是一组变量的赋值（候选解也是类似的结构）。局部搜索算法通过不停的由一个候选解移动到和该候选解相邻的解来进行迭代，直到满足一定的终止条件为止。满足终止条件后将结束算法。局部搜索中的一些选择，例如初始解的构建，移动目的等，都可以是随机的，也可以使用一些策略。





局部搜索具有简单和高效两个特点，主要用于处理组合优化问题。因为很多问题都可以看成某种形式的优化问题。所以局部搜索的使用范围很广。对于不同的问题，只需要定义解空间的邻域结构就可以采用局部搜索算法来求解了。局部搜索的实现也是相当简单。

局部搜索的要素主要由：

1 搜索空间：候选解的集合。有时这些候选解也称被称为位置，点，状态等。

2 可行解的集合：搜索空间的一个子集，包含问题的所有可行解。可行解不一定是最优解，但是最优解必须是可行解。

3 候选解的邻域关系：规定了什么样的两个解是邻居

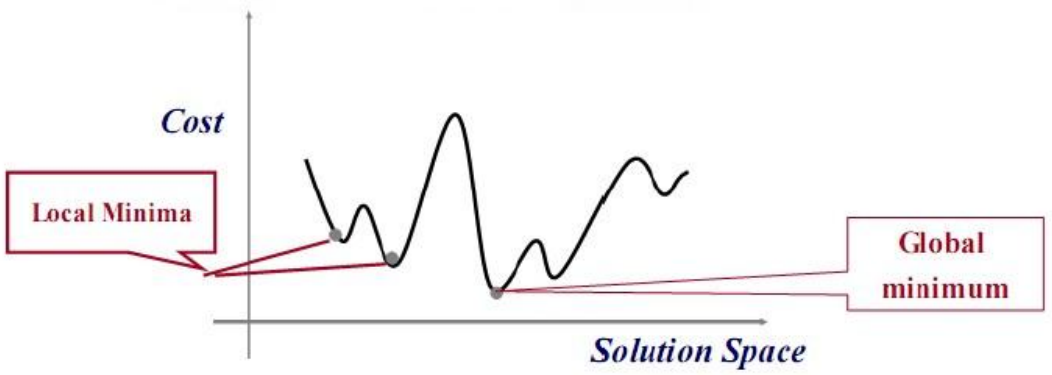
4 初始化函数：用于生初始化解

5 转移函数：用于从一个候选解到另一个候选解。一次转移就是一个搜索步

上面的定义只是基于单个候选解的，也就是每个搜索步只访问候选解。局部搜索算法现在并不局限于单个候选解，也可以每一步维护多个候选解。

一般来说，局部搜索算法大都会用到一个评估函数，是一个解集合到实数的函数，用于衡量解的质量的好坏。评估函数是针对算法而言的，和所处理的优化问题中的目标函数不一定一样。但是设计算法的一般，一般这两个函数趋势要一致，也就是说，评估函数较小的时候，目标函数也要较小；反之亦然。这样才能把搜索往正确的方向上引导。

局部最优：局部最优是局部搜索算法中的一个重要概念。假设说我们要处理的优化问题是，某类最小化问题；如果对于一个候选解 s 而言，它的所有的邻居的目标函数的值均大于它的目标值，那么 s 就是一个局部最优解。也叫局部最优解。如果所有的其他候选解的目标函数也都大于 s 的目标函数，那么 s 被称为全局最优解。很明显，全局最优解是我们所要求的解。而布局最优解并不是。



下面我们给出一个局部搜索的算法框架：

输入：问题 P
输出：解 s，或者空
s:=init（P）
while 没有到达终止条件 do
    s:=step（s,P）
end
if s 是候选解
    return s
else
    return 空集

对上面的算法做一个简单的解释。给定问题 P，init（P）先得到一个初始的解 s。在没有到达终止条件的时候，这个条件一般是时间条件，step（s,P）不断的从 s 转移到它的邻域中的一个解。最后，如果得到的解是可行解，就返回该解，否则就认为算法没得到什么有用的解。

各类局部搜索算法的差异很大程度在转移函数上，如何从当前解到下一个候选解，也就是上图中的 step（s,P）。这也是局部搜索算法最本质的地方。

一般来说，转移方法有四类简单的迭代方法。通过这四类简单的迭代方法又可以组合生成一些复杂的迭代方法。四类简单的迭代方法分别是：
1 迭代改进（爬山法）：每一步从当前解中邻域中选择一个最优的进行转移，直到达到一个局部最优解
2 随机游走：每一步从邻域中随机选择一个进行转移
3 随机迭代：每一步先产生一个随机概率，然后根据这个该剧进行随机游走
4 简单禁忌搜索：在算法的过程中会对邻域中的一些解做禁忌；每一步从没有被禁忌的解中选择一个最优的候选解进行转移

通过这四种简单的方法，可以构造出一些复杂的转移方法，例如迭代局部搜索，是在没有达到局部最优的时候，一直进行迭代改进；达到局部最优后，改为随机游走。贪心随机自适应搜索（GRASP）由很多大迭代组成，每一步先是用贪心法构造一个初始解，然后用某类局部搜索算法找到局部最优解；然后进入下一个大迭代。

局部搜索的一个比较重要的地方是进入局部最优后如何跳出的问题。这个也是局部搜索算法需要进行设计的地方。

局部搜索算法的优点是简单和高效；它的缺点主要是不能保证解的质量。而且，局部搜索算法的优劣目前一般只能通过实验来检验和分析；而不能通过理论来进行说明。

局部搜索的设计思想主要是集中性和多样性的平衡。集中性是搜索主要集中在一小块区域，多样性指的是搜索要尽可能分散。好的搜索算法要在搜索过程中对这两种搜索方式切换得当。实施上很多局部搜索策略都是集中性搜索和多样性搜索的某种体现。

局部搜索容易遇到的另一个问题是循环问题。为了提高搜索效率，尽可能的搜索更多的候选解，一般局部搜索算法不会花费资源保存更多的信息，它通常只保留当前的局部信息。这时算法会重复访问最近已经访问过的候选解，这就是循环现象。循环问题不仅会浪费大量的算法使劲按，也使得蒜贩常陷入局部最优，导致算法性能下降。因为局部搜索算法的性质，循环问题是不能被消除的，只能被缓解。

我们下面通过两个例子来深入理解一下语句搜索算法，一个是可满足性问题，另一个是旅行商问题。

首先回顾一下可满足性问题的定义：

文字：布尔变量或其否定形式为文字

子句：若干个文字的析取被称为子句，没有文字的子句被称为空子句;

CNF：若干个子句的合取是一种特殊形式的公式，称为合取范式(conjunctive normal form 简写为 CNF)

```
输入：CNF 公式 F
输出：解 s，或者空
s:=init（F）
while 没有到达终止条件 do
    s:=step（s,F）
    if s 是候选解
        return s
end
return 空集
```

这个例子和前文的通用框架似乎比较类似，稍微有所不同，我们对其做详细的解释。首先，输入的公式 F 是一个 CNF 公式；s:=init（F）得到的解记作 s;这里的 s 是对 F 中的每个布尔变量的赋值，也就是说 s 是一个布尔 F 中的布尔变量到 0，1 的映射。

终止条件我们假定为时间因素，或者迭代次数。如果没有到时间，或者没有到迭代次数，

不停的运行 s:=step（s,F）；直到得到一个可满足的解为止。

s:=step（s,F）是从一个解到另一个解的操作，也是局部搜索的关键。后者一定要是前者的邻居。

对于一个候选解 s，什么样的解是它的邻居是设计局部搜索的一个重要部分。对于可满足问题，惯用的做法是对 s 进行的一个反转得到的解做为它的邻居。例如对于四个布尔变量；当前的解是 1101,也就是对第一个布尔变量赋值为 1，第二个为 1，第三个为 0，第四个为 1. 这个解有四个邻居，分别是反转其中的一个变量得到的：0101,1001,1111,1100；step 函数就是从当前的 1101 根据某中规则，转移到这四个邻居之一。

当然，也可以设置其他的邻居规则，例如，可以把反转两个变量做位邻居；这个是设计者自己设定的。目前来看，对于可满足问题，反转一个变量是比较好的邻域规则。

设置好邻域后，step 要根据不同的算法规则来进行转移；这要依据解的评价机制，也就是解的评估函数。对于一个可满足性问题，我们的输入是一个 CNF 公式，可以将该解满足的子句的个数作为评估函数，满足越多的子句的解，质量越好。

假设我们当前使用的迭代改进法，那么，它就会选择邻域中的一个满足最多子句的解；如果是随机游走法，那么它就会随机选择一个解。

这里可以简单的设置一个禁忌搜索规则，假定当前的解是由某个解 s’得到，明显 s’是当前解的邻居。这时候，如果再返回到 s’是没有意义的，所以我们可以把 s’作为当前解的一个禁忌对象，采用简单禁忌搜索的方式。也就是说，在当前的解中选择转移对象的时候，要在 s’之外的候选解中进行选择。不同的问题有不同的禁忌方法。

如果当前的解所满足的子句的数目，大于了它所有的邻域中的解所满足的子句的数目，我们就说它进入了一个局部极小状态（准确说是局部极大）。

如果这时候，该解是一个可行解，就可以终止程序。对于可满足性问题，我们可以设置可行解为满足了该公式的解。因此对于可满足性问题，只要找个一个可行解就可以。

如果进入局部最优值，也就是当前解满足了比所有邻域中的解都要多的子句，但是还不是可行解。这个时候要采用某种方式跳出这个局部极值，搜索其他的部分。跳出方法也是局部搜索算法的一个重要方面。可以进行随机游走，或者重新启动（重新从另一个初始值开始迭代）等。

可满足问题是一个判定问题，下面用一个优化问题的例子再描述一下局部搜索算法的各个要素。旅行商问题是这样一个问题：

对于一个无向带权图，寻找一个遍历该图的全部顶点仅一次的，并且权值最小的回路。于此密切相关的一个问题是汉密尔顿图问题：对于一个无向图，遍历它的所有顶点仅一次的回路被称为汉密尔顿回路。旅行商问题的非正式说法，就是在一个加权图中，寻找一个权值和最小的汉密尔顿回路。

旅行商问题的局部搜索算法框架和可满足问题差不多，为了便于阅读，我们写在下面：

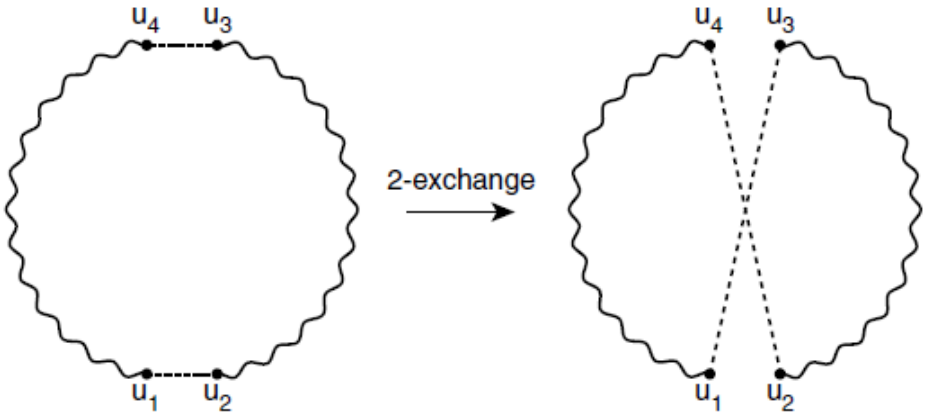
输入：旅行商问题的实例 G
输出：解，或者空
s:=init（G）

```
while 没有到达终止条件 do
    s:=step（s,G）
    if s 是可行解并且比 s'更优
        s'←s
end
return  s'
```

旅行商问题的输入是一个带权图，它的候选解可以看作是一个顶点的序列；如果这个序列能构成一个汉密尔顿回路，那么它就是一个可行解；算法的目的要返回一个权值最小的可行解；局部搜索算法是随机算法的一种，对于旅行商问题，它首选要保证解是一个汉密尔顿回路，然后权值和尽可能的小。

初始化函数采用某种方式生成一个初始的解；在没有达到终止条件的时候进行迭代；达到终止条件后，就返回所找到的那个权值最小的汉密尔顿回路，或者为空（在没有找到任何汉密尔顿回路的情况下），

如何构造邻域是旅行商问题的关键；对于 SAT 问题，反转一个变量是较为常用的邻居构造方法。对于旅行商问题，交换两条边是一个常用的方法，也就是同时交换两个顶点。



每一次迭代就是通过前文讲到的四类简单方法，或者一些复合方法，从一个候选解到另一个候选解。并且要记录当前找到的最好的那个可行解，也就是权值最小的汉密尔顿回路。直到满足终止条件，就返回该回路。

下面介绍几个复杂的局部搜索框架。局部搜索作为一种启发式搜索技术，里面涉及的大都是技巧性技术，而且很多是针对具体问题的，当然也有一些技术用途比较广泛。这类技术大都没有理论上的保证，效果主要是依靠实验。本部分给大家介绍两个较有影响的随机搜索框架，一个是影响较大的 Greedy Randomized Adaptive Search Procedure，贪婪随机自适应搜索（GRASP），另一个是最近出现的三阶段搜索技术（TPS）。

```
输入：问题 P
输出：解 s，或者空
while 没有到达终止条件 do
    s:=init（P）
    如果 s 不是可行解
      修复 s
    s:=step（s,P）
end
return  s
```

GRASP 和普通局部搜索框架的主要区别在于它更重视初始解的构造；它的初始解是在迭代内部而不是外部。它的每次迭代都是先构造一个初始解；构造通过贪心加随机的方法，其中贪心和随机的部分由一个随机函数来控制。如果所构造出来的解不是可行解，还要进行修复，将其变为可行解。然后再进行邻域搜索，达到局部最优后，重新构造初始解。

该算法的框架的关键在于如何构造初始的解。它首先把所有的解按照优劣（例如按照引起目标函数的变化情况）进行排序，一般是最优解排在前面，然后选取其中的 k 个放入一个名为 RCL 的表中。然后在 RCL 表中随机选择一个作为初始解。所以它的构造初始解的方法可以说是贪心加随机的组合。算法通过控制 RCL 表的长度来确定贪心部分和随机部分的分配。如果该表包含了所有候选解，则是纯随机；如果该表长度为 1，则是纯贪心。

近年来提出了一些其他的局部搜索框架，这里给出三阶段搜索框架，算法如下：

```
输入：问题 P
输出：解 s，或者空
s:=init（P）
while 没有到达终止条件 do
    s:=step1（s,P）
    s:=step2（s,P）
    如果 s 是历史最优可行解
      记录 s
    s:=step3（s,P）
end
return  s
```

这个算法将搜索分为三个阶段，我们为了简化，直接用 step1，step2，step3 来表示。其中算法的执行方式是：第一阶段 step1 一般用梯度下降的方式达到一个局部最优解；然后 step2 继续进行邻域内搜索，力图接受一些较差的候选解，跳出局部最优，搜索附近的解空间；step3 是采用扰动的方式，将搜索定位到更远的地方。三阶段搜索是比较新的搜索框架，已经在一些领域内取得较好的效果。

最后我们介绍一些传统的经过检验的技巧，以及一些较新的局部搜索技巧，这些技巧或者技术分别在某个或者多个领域内取得了一定的成效。

邻域策略：有的时候，一个过大的邻域会影响搜索效率，这时可以通过一些限制适当减少邻域规模。考虑到这样一个问题，在一个带权完全图中寻找一个划分，使得所有划分的权值和最小，这个问题叫 CPP 问题。对于 CPP 问题，常用的候选解形式是一个顶点的划分：G1,…,Gk.常用的邻居规则是将一个顶点 v 从一个划分部分 Gi 转移到另一个部分 Gj。这时，如果有图 n 个顶点，划分有 k 个，邻域的大小为 n*k。一个转移需要在这些邻居中选择。如果对其做出规定，规定只有那个产生最大的收益（评估函数的增量最大）的转移才属于邻域，这样对于每个变量只有一个选择，邻域的大小将变为 n,减少一个量级。这个策略在 CPP 问题中被证明是有效的。在其他问题中，也可以考虑类似的技术。

多重选优策略。这个也是一个邻域策略的变种，是一个简单而有效的策略。这一策略在选择待执行移动时只随机采样一部分可选移动来比较，而并非比较所有的候选点。这一策略虽然并不能给出所有候选移动中最好的那个，然而却能给出一个可以接受的较优移动。例如，对总移动数为 10000 的样例，采样 200 个点，求得的最优移动在总移动中排前 5%的概率为 1-(1-5%)200=99.99%，选点的时间复杂度 O(V)却从降低到了常数级别。

随机扰动策略：局部搜索算法的一个关键问题是如何在算法陷入局部最优解以后跳出这个局部最优，去搜索其他的解空间。传统的扰动方法是使用的随机扰动的方式，即在扰动部分时，算法随机选择一些合法的移动去执行，其对于移动的选择完全基于随机。这时候可以通过增加一些贪心策略来提高扰动效果，例如可以采用 GRAS 的思想，用一个随机函数来规定随机和贪心的执行权重，通过 RCL 表来确定扰动范围。

格局检测：我们最后介绍格局检测技术，这是近年来随机搜索的一个重要的策略，在很多问题上取得了优异的成果，有的学者称其为里程碑式的工作。这项工作是蔡少伟完成的，前文的多重选优策略也是其提出的。在决策候选解每一步邻域内动作的选择时不仅候选解本身的评价，还取决于它所处的环境，如其所属社区以及与全局的关系等。格局检测是针对不同的问题，定义了格局的概念。用对格局的检测来替代传统禁忌搜索中的禁忌策略。通过对格局未发生改变的顶点的禁忌，达到对无用动作的禁忌，使得搜索能够更高效的进行。该方法对于局部搜索算法领域有很大的推进作用，很多常用算法都能通过该方法，准确的禁忌掉无用操作，从而达到高效的求解。针对于 SAT 问题和 MAXSAT 问题，在定义邻域动作为翻转一个变量的赋值的基础上，针对每个变元 var 的格局定义为与当前变元在同一子句中的其他变元赋值情况，如果它们发生改变则判定 var 的格局发生变化，否则判定 var 的格局未发生变化且禁止翻转 var 的值。针对最小顶点覆盖问题，格局检测方法是定义每个顶点的相邻顶点为当前顶点的格局，如果相邻顶点的状态都没有发生过改变则判定当前顶点的格局未发生改变，从而禁止掉一些会形成循环的动作来避免算法陷入循环。很多学者也针对其他的问题采用了各种不同的格局定义和格局检测策略，大都取得了不错的结果。

2 模拟退火

模拟退火算法(Simulated Annealing，SA)最早的思想出现于 1953 年提。1983 年,S. Kirkpatrick 将退火思想引入到组合优化领域。它是基于 Monte-Carlo 迭代求解策略的一种随机寻优算法，其出发点是基于物理中固体物质的退火过程与一般组合优化问题之间的相似性。模拟退火算法从某一较高初温出发，伴随温度参数的不断下降,结合概率突跳特性在解空间中随机寻找目标函数的全局最优解，即在局部最优解能概率性地跳出并最终趋于全局最优。模拟退火算法是一种通用的优化算法，理论上算法具有概率的全局优化性能,目前已在工程中得到了广泛应用，诸如 VLSI、生产调度、控制工程、机器学习、神经网络、信号处理等领域。

首先看一下固体退火过程。固体退火是将固体加热到足够高的温度，使分子呈随机排列状态，然后逐步降温使之冷却，最后分子以低能状态排列，固体达到某种稳定状态（规整晶体）的热力学过程。过程分为三个阶段：

加温过程：增强粒子的热运动，使其偏离平衡位置，目的是消除系统原先可能存在的非均匀态；

等温过程：退火过程中要让温度慢慢降低，在每一个温度下要达到热平衡状态，对于与环境换热而温度不变的封闭系统满足自由能较少定律，系统状态的自发变化总是朝自由能减少的方向进行，当自由能达到最小时，系统达到平衡态；

冷却过程：使粒子热运动减弱并渐趋有序，系统能量逐渐下降，从而得到低能的晶体结构。当液体凝固为固体的晶态时退火过程完成。

模拟退火算法（SA）基本思想是：SA 开始于一个很高的初始温度（随机行走），伴随温度参数的不断下降，在每一温度下（算法的状态转移）慢慢达到热平衡（爬山法），最终达到物理基态（全局最优）。 在系统向着能量减小的趋势变化过程中，允许系统按一定概率（以温度为参数）跳到能量较高的状态，以避开局部最优，最终稳定在全局最优。

模拟退火算法的基本要素是：

状态空间与状态产生函数：搜索空间也称为状态空间，它由经过编码的可行解的集合组成。状态产生函数（邻域函数）应尽可能保证产生的候选解遍布全部解空间。通常由两部分组成，即产生候选解的方式和候选解产生的概率分布。候选解一般采用按照某一概率密度函数对解空间进行随机采样来获得。概率分布可以是均匀分布、正态分布、指数分布等。

状态转移概率：状态转移概率是指从一个状态向另一个状态的转移概率。通俗的理解是接受一个新解为当前解的概率。它与当前的温度参数 T 有关，随温度下降而减小。一般采用 Metropolis 准则如下：

Metropolis 准则 (以概率接受新状态) ：

在温度 Tk，当前状态 i → 新状态 j

若 Ej<Ei，则接受 j 为当前状态；

否则，若概率 p=exp[-(Ej-Ei)/T] 大于[0,1)区间的随机数，则仍接受状态 j 为当前状态；若不成立则保留状态 i 为当前状态。

内循环终止准则：也称 Metropolis 抽样稳定准则，用于决定在各温度下产生候选解的数目。常用的抽样稳定准则包括：

1）检验目标函数的均值是否稳定。

2）连续若干步的目标值变化较小。

3）按一定的步数抽样。

外循环终止准则：即算法终止准则，常用的包括：

1）设置终止温度的阈值。

2）设置外循环迭代次数。

3）算法搜索到的最优值连续若干步保持不变。

4）检验系统熵是否稳定。

模拟退火算法新解的产生和接受可分为如下四个步骤：

第一步是由一个产生函数从当前解产生一个位于解空间的新解；为便于后续的计算和接受，减少算法耗时，通常选择由当前新解经过简单地变换即可产生新解的方法，如对构成新解的全部或部分元素进行置换、互换等，注意到产生新解的变换方法决定了当前新解的邻域结构，因而对冷却进度表的选取有一定的影响。

第二步是计算与新解所对应的目标函数差。因为目标函数差仅由变换部分产生，所以目标函数差的计算最好按增量计算。事实表明，对大多数应用而言，这是计算目标函数差的最快方法。

第三步是判断新解是否被接受，判断的依据是一个接受准则，最常用的接受准则是 Metropolis 准则: 若$\Delta t' < 0$则接受 S′作为新的当前解 S，否则以概率 $\exp(-\Delta t'/T)$接受 S′作为新的当前解 S。

第四步是当新解被确定接受时，用新解代替当前解，这只需将当前解中对应于产生新解时的变换部分予以实现，同时修正目标函数值即可。此时，当前解实现了一次迭代。可在此基础上开始下一轮试验。而当新解被判定为舍弃时，则在原当前解的基础上继续下一轮试验。

模拟退火算法与初始值无关，算法求得的解与初始解状态 S(是算法迭代的起点)无关；模拟退火算法具有渐近收敛性，已在理论上被证明是一种以概率收敛于全局最优解的全局优化算法；模拟退火算法具有并行性。

模拟退火算法的应用很广泛，可以求解 NP 完全问题，但其参数难以控制，其主要问题有以下三点：

(1) 温度 T 的初始值设置问题。

温度 T 的初始值设置是影响模拟退火算法全局搜索性能的重要因素之一、初始温度高，则搜索到全局最优解的可能性大，但因此要花费大量的计算时间；反之，则可节约计算时间，但全局搜索性能可能受到影响。实际应用过程中，初始温度一般需要依据实结果进行若干次调整。

(2) 退火速度问题。

模拟退火算法的全局搜索性能也与退火速度密切相关。一般来说，同一温度下的“充分”搜索(退火)是相当必要的，但这需要计算时间。实际应用中，要针对具体问题的性质和特征

设置合理的退火平衡条件。

(3) 温度管理问题。

温度管理问题也是模拟退火算法难以处理的问题之一。实际应用中，由于必须考虑计算复杂度的切实可行性等问题。

3 遗传算法

遗传算法的思想来自达尔文的进化理论。生命自从在地球上诞生以来，就开始了漫长的生物演化历程，低级、简单的生物类型逐渐发展为高级、复杂的生物类型。这一过程已经由古生物学、胚胎学和比较解剖学等方面的研究工作所证实。生物进化的原因自古至今有着各种不同的解释，其中被人们广泛接受的是达尔文的自然选择学说。

自然选择学说认为，生物要生存下去，就必须进行生存斗争。生存斗争包括种内斗争、种间斗争以及生物与无机环境之间的斗争三个方面。在生存斗争中，具有有利变异(mutation)的个体容易存活下来，并且有更多的机会将有利变异传给后代，具有不利变异的个体容易被淘汰，产生后代的机会也少得多。因此，凡是在生存斗争中获胜的个体都是对环境适应性比较强的。达尔文把这种在生存斗争中适者生存、不适者淘汰的过程叫做自然选择。达尔文的自然选择学说表明，遗传和变异是决定生物进化的内在因素。

遗传算法是借鉴生物界自然选择原理和自然遗传机制而形成的一种迭代式自适应概率性全局优化搜索算法。 它模拟达尔文“优胜劣汰、适者生存”的原理激励好的结构，模拟孟德尔遗传变异理论在迭代过程中保持已有结构，同时寻找更好的结构。该算法在 70 年代初期由美国 Michigan 大学的 Holland 教授发展起来的。以 1975 年 Holland 的专著《Adaptation in natural and Artificial systems》出版为标志。

算法的基本思想是从一组解的初值开始进行搜索，这组解称为一个种群，种群由一定数量、通过基因编码的个体组成，其中每一个个体称为染色体。不同个体通过染色体的复制、交叉和变异又生成新的个体，依照适者生存的规则，个体也在一代一代进化，通过若干代的进化最终得出条件最优的个体。

个体（Individual）就是模拟生物个体而对问题中的候选解的一种称呼，一个个体也就是搜索空间中的一个点。

种群（ population ）：是模拟生物种群而由若干个体组成的群体, 它一般是整个搜索空间的一个很小的子集。种群中含有的个体的数量叫做种群的规模(population size)。

考虑如下优化问题：

$$\max\{f(x)\,|\,x\in X\}$$

这里 f 是 X 上的一个正值函数，即对任意 $x\in X$，$f(x)>0$。X 是问题的解空间，即问题的所有可能解全体。它可以是一个有限集合(如组合优化问题)，也可以是实空间 R^n 的一个子集(如连续优化问题)等。

对以上问题，我们通常用两类方法来求解之，一类是解析法，通过一定的手段直接计算出问题的解，另一类是迭代法，给出问题的一个初始猜测，然后从此初始点出发，逐步迭代，直至达到停机条件。遗传算法也是一种迭代法，但它有别于传统的迭代法。

设计一个遗传算法要涉及到以下步骤：

确定编码方案

确定适应值函数

选择策略的确定

遗传算子的设计

确定算法的终止准则

控制参数的选取

我们看一个例子：

$$\max_{0\le x\le 1} f(x),\ f(x)=\sin x+2e^{-x}-1$$

在这个例子中，变量是演化算法的表现型形式。编码是将优化问题解空间中可行解（个体），即设计变量映射到 GA 中的基因型数据结构。我们这里采用二进制编码，将某个变量值代表的个体表示为一个{0,1}二进制串。

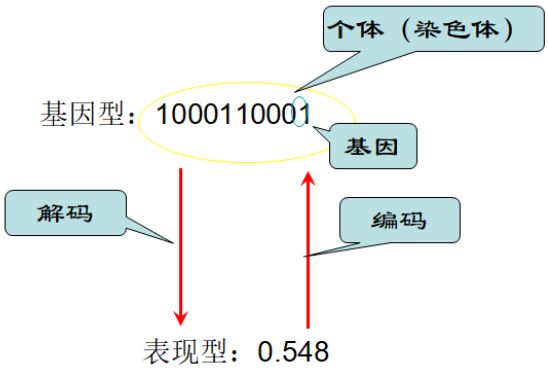
串长取决于求解的精度。如果确定求解精度到 3 位小数，由于区间长度为 1，编码的二进制串长至少需要 10 位。

二进制串转化为十进制：

$$<b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0>=(\sum_{i=0}^{9}b_i\cdot 2^i)=x' \qquad x=x'\cdot\frac{1}{2^{10}-1}$$

如：s=<1000110001>　x'=561　x=0.548

<0000000000>与<1111111111>表示区间端点0和1。



随机生成初始种群：

S_1=<1110001110>, x1 = 0.890
S_2=<0101000111> , x2 = 0.320
S_3=<0011001101>, x3 = 0.200
S_4=<1011110011>, x4 = 0.738
S_5=<1001101101>, x5 = 0.607
S_6=<0111011000>, x6 = 0.461
S_7=<1111001100>, x7 = 0.950
S_8=<0000100101>, x8 = 0.036

构造适应值函数：直接将目标函数作为适应值函数: f(x) =sinx+2e^{-x} -1 ，有了适应值函数，可以对初始种群进行评估。

$S_1^{(0)}$=<1110001110> ,$f(x_1)$= 0.598
$S_2^{(0)}$=<0101000111> , $f(x_2)$=0.767
$S_3^{(0)}$=<0011001101> , $f(x_3)$=0.836
$S_4^{(0)}$=<1011110011> , $f(x_4)$=0.629
$S_5^{(0)}$=<1001101101> , $f(x_5)$=0.660
$S_6^{(0)}$=<0111011000> , $f(x_6)$=0.706
$S_7^{(0)}$=<1111001100> , $f(x_7)$=0.578
$S_8^{(0)}$=<0000100101> , $f(x_8)$=0.965

在遗传算法中，需要做出选择：决定哪些解被保留并被允许繁殖，哪些解应当消亡的过程。 选择要保持种群规模，优胜劣汰。 并且保证优解被复制多份，消除劣解。

繁殖池(breeding pool)选择是这样的，它是一种基于适应值比例的选择。

首先按下式计算其相对适应值：

$$rel_i = \frac{f_i}{\sum_{i=1}^{N} f_i}$$

其中 f_i 是群体中第 i 个个体的适应值，N 是群体的规模。

每个个体的繁殖量为：$N_i=round(rel_i \cdot N)$，其中 $round(x)$ 表示与 x 距离最小的整数。

将每个个体复制 N_i 个生成一个临时群体，即繁殖池（中间代，Intermediate Generation）。

轮盘赌选择又称比例选择算子，它的基本思想是：各个个体被选中的概率与其适应度函数值大小成正比。设群体大小为 n ，个体 i 的适应度为 F_i，则个体 i 被选中遗传到下一代群体的概率为：

$$P_i = F_i / \sum_{i=1}^{n} F_i$$

比武选择（Tournament Selection）是另外一种选择方式，它随机从原来的种群中采样出两个个体，将优胜者放入中间代中。 如果种群大小为 N，则进行 N 次，从而生成大小为 N 的中间代。 比武选择没有出现在早期 Holland 风格的遗传算法中，但现在较为常用。

遗传算子的设计

a. 杂交

单点杂交：杂交概率 p_c =0.75，6 个个体参加杂交

$s_2^{(0)}$和 $s_6^{(0)}$, $s_1^{(0)}$和 $s_5^{(0)}$, $s_3^{(0)}$和 $s_4^{(0)}$

$s_2^{(0)}$ =<0101|000111>　　s'$_2$=<0101| 011000>, f(s'$_2$)=0.644

$s_6^{(0)}$ =<0111|011000>　　s'$_6$=<0111| 000111>, f(s'$_6$)=0.712

$s_1^{(0)}$ =<111|0001110>　　s'$_1$=<111| 1101101>, f(s'$_1$)=0.580

$s_5^{(0)}$ =<100|1101101>　　s'$_5$=<100| 0001110>, f(s'$_5$)=0.687

$s_3^{(0)}$ =<00110011|01>　　s'$_3$=< 00110011| 11>, f(s'$_3$)=0.834

$s_4^{(0)}$ =<10111100|11>　　s'$_4$=< 10111100|01>, f(s'$_4$)=0.629

b. 变异

变异概率 p_m=0.02，10*8*0.02=1.6，

随机选取 2 个基因位进行变异

$s_6^{(0)}$=<0111011000>　　s'$_6$=<0111011100 >　　f(s'$_6$)=0.704

$s_3^{(0)}$ =<0011001101>　　s'$_3$=< 0010001101 >　　f(s'$_2$)=0.880

新群体：所有子代和父代中最好的 8 个个体：

$s_1^{(1)}$ =< 0000100101 >，$f(s_1^{(1)}) = 0.965$

$s_2^{(1)}$ =< 0010001101 >，$f(s_2^{(1)}) = 0.880$

$s_3^{(1)}$ =< 0011001101 >，$f(s_3^{(1)}) = 0.836$

$s_4^{(1)}$ =< 0011001111 >，$f(s_4^{(1)}) = 0.834$

$s_5^{(1)}$ =< 0101000111 >，$f(s_5^{(1)}) = 0.767$

$s_6^{(1)}$ =< 0111000111 >，$f(s_6^{(1)}) = 0.712$

$s_7^{(1)}$ =< 0111011000 >，$f(s_7^{(1)}) = 0.706$

$s_8^{(1)}$ =< 0111011100 >，$f(s_8^{(1)}) = 0.704$

$$\sum_{i=1}^{8} f(s_i^{(1)}) = 6.404$$

确定算法的终止准则：

a. 遗传运算的终止进化代数 T，一般取为 100~500

b. 最好个体在若干代内无改变

控制参数的选取：

a. 种群规模 N，一般取为 20~100

b.杂交概率 p_c, 一般取为 0.4~0.9

c. 变异概率 p_m, 一般取为 0.001~0.1

遗传算法的正式形式描述为 GA＝($P(0), N, l, s, g, p, f, t$)。

初始种群：$P(0)=(a_1(0), a_2(0),\ldots,a_N(0)) \in I^N$ ；位串空间：$I=\{0,1\}^l$ ，l 表示二进制串的长度；种群规模：N ；选择策略：$s: I^N \to I^N$ ；遗传算子 g：通常包括繁殖算子 $O_r: I \to I$、杂交算子 $O_c: I \times I \to I \times I$ 和变异算子 $O_m: I \to I$；遗传算子的操作概率 p：包括繁殖概率 p_r、杂交概率 p_c 和变异概率 p_m；； $f: I \to R^+$是适应函数； $t: I^N \to \{0, 1\}$是终止准则。

遗传算法是一个具有理论基础的算法。下面介绍一些关于遗传算法的理论。

模式是指种群个体基因串中的相似样板，它用来描述基因串中某些特征位相同的结构。在二进制编码中，模式是基于三个字符集(0,1,*)的字符串，符号*代表任意字符，即 0 或者 1，如 101**1。模式 H 中确定位置的个数称为模式 H 的阶，记作 O(H)。例如 O(10**1)=3 。模式 H 中第一个确定位置和最后一个确定位置之间的距离称为模式 H 的定义距，记作 δ(H)。例如(10**1)=4 。

模式阶用来反映不同模式间确定性的差异，模式阶数越高，模式的确定性就越高，所匹配的样本数就越少。在遗传操作中，即使阶数相同的模式，也会有不同的性质，而模式的定义距就反映了这种性质的差异。

模式定理（Schema Theorem）（ Holland 1975，《自然系统和人工系统的自适应性》 ）：具有低阶、短定义距以及平均适应度高于种群平均适应度的模式在子代中呈指数增长。

模式定理保证了较优的模式（遗传算法的较优解）的数目呈指数增长，确认了结构重组遗传操作对于获得隐并行性的重要性，从理论上保证了遗传算法是一个可以寻求最优可行解的优化过程，为解释遗传算法机理提供了数学基础，被认为是遗传算法的基本定理。

积木块假设：遗传算法通过短定义距、低阶以及高平均适应度的模式（积木块），在遗传操作下相互结合，最终接近全局最优解。

模式定理保证了较优模式的样本数呈指数增长，从而使遗传算法找到全局最优解的可能性存在；而积木块假设则指出了在遗传算子的作用下，能生成全局最优解。

下面看一下遗传算法的收敛性问题。种群太小则不能提供足够的采样点，以致算法性能很差；种群太大，尽管可以增加优化信息，阻止早熟收敛的发生，但无疑会增加计算量，造成收敛时间太长，表现为收敛速度缓慢。选择操作使高适应度个体能够以更大的概率生存，从而提高了遗传算法的全局收敛性。如果在算法中采用最优保存策略，即将父代群体中最佳个体保留下来，不参加交叉和变异操作，使之直接进入下一代，最终可使遗传算法以概率 1 收敛于全局最优解。交叉操作用于个体对，产生新的个体，实质上是在解空间中进行有效搜索。交叉概率太大时，种群中个体更新很快，会造成高适应度值的个体很快被破坏掉；概率太小时，交叉操作很少进行，从而会使搜索停滞不前，造成算法的不收敛。 变异操作是对种群模式的扰动，有利于增加种群的多样性 。但是，变异概率太小则很难产生新模式，变异概率太大则会使遗传算法成为随机搜索算法。

遗传算法本质上是对染色体模式所进行的一系列运算，即通过选择算子将当前种群中的优良模式遗传到下一代种群中，利用交叉算子进行模式重组，利用变异算子进行模式突变。通过这些遗传操作，模式逐步向较好的方向进化，最终得到问题的最优解。

遗传算法有以下特点：

群体搜索，易于并行化处理；

不是盲目穷举，而是启发式搜索；

适应度函数不受连续、可微等条件的约束，适用范围很广。

在遗传算法中，会产生遗传欺骗问题：在遗传算法进化过程中，有时会产生一些超常的个体，这些个体因竞争力太突出而控制了选择运算过程，从而影响算法的全局优化性能，导致算法获得某个局部最优解。对此可以在以下方面进行改进：

（1）对编码方式的改进

（2）对遗传算子 的改进

（3）对控制参数的改进

（4）对执行策略的改进

在设计遗传算法时，编码表示与遗传算子尤其是杂交算子是同步考虑的，如果编码处理不当，在遗传空间中因杂交而生成的个体映射到问题空间时，有可能成为无用解，我们称形成无用解的染色体为致死因子。同时，即使逆映射到问题空间的是有用解，但也可能是和双亲完全无缘的解。编码要避免在问题空间中形成这些不适当的个体。评估编码策略常采用以下 3 个规范：

1. 完备性(completeness)：问题空间中的所有点(候选解)都能作为演化空间中的点(染色体)表现；

2. 健全性(soundness)：演化空间中的所有染色体能对应问题空间中的候选解；

3.非冗余性(nonredundancy)：染色体和候选解一一对应。

二进制编码是将原问题的解空间映射到位串空间$\{0,1\}^l$上，然后在位串空间上进行演化操作。结果再通过解码过程还原成其表现型以进行适应值的评估。很多数值与非数值问题都可以用二进制编码以应用演化算法。 二进制编码有如下优点：

二进制编码类似于生物染色体的组成，算法易于用生物遗传理论来解释并使得演化操作如杂交、变异很容易实现；

采用二进制编码时，算法处理的模式数最多；

这种编码方式便于模式定理进行分析，因为模式定理就是以二值编码为基础的。

二进制编码也并未都是有点，它在在求解连续数值优化问题时有一些缺点：

相邻整数的二进制编码可能具有较大的 Hamming 距离，这种缺陷将降低演化算子的搜索效率，二进制编码的这一缺点有时称为 Hamming 悬崖（Hamming Cliffs）；

二进制编码时，所需解的精度确定后，串长也就确定了，当需要改变精度时，很难在算法执行过程中进行调整，从而使算法缺乏微调的功能；

当应用于高维、高精度数值问题时，产生的二进制位串很长，搜索空间非常大，从而使算法的搜索效率很低。

为了克服二进制编码的缺点，对于问题的变量是实向量的情形，可以直接采用十进制进行编码。这样，便可直接在解的表现型上进行遗传操作。从而便于引入与问题领域相关的启发式信息以增加遗传算法的搜索能力。

“在人工遗传和演化搜索方案中，实型编码或浮点基因的使用已经有很长的历史了，尽管是有争议的，但它们在以后的使用趋势仍在上升。这种上升的使用趋势多少有些使熟知基本遗传算法(GA)理论的研究者感到惊讶，因为简单的分析似乎建议通过使用低基数性的字母表可以增强模式的处理，而来自于实算的结论似乎直接反驳了这种观点，实型编码在许多实际问题里工作得较好。”（ Goldberg ）

对实数编码的情形，从理论上讲，二进制编码下的各种遗传操作同样可以使用，因为在机器中实数也是采用二进制表示的。但实际应用时却很少使用这些基于内部点操作的算子，

而是针对实型编码的特性，引入其它一些遗传算子。

实验证明，对于大部分数值优化问题，通过一些专门设计的遗传算子的引入，采用实数编码比采用二进制编码时算法的平均效率要高。与二进制编码比较，由于浮点编码遗传算法有精度高和便于大空间搜索的优点。近年来，遗传算法在求解高维或复杂优化问题时也大多使用实数编码，并产生了很好的效果，实数编码的使用越来越广泛。

结构式编码：对很多问题其更自然的表示是树或图的形式，这时采用其它形式的编码可能是很困难的。将问题的解表示成树或图的形式的编码称为结构式编码。它随机产生一个与个体编码长度相同的二进制屏蔽字 $P = W_1W_2\ldots W_n$ ；按下列规则从 A、B 两个父代个体中产生两个新个体 X、Y：若 $W_i = 0$，则 X 的第 i 个基因继承 A 的对应基因，Y 的第 i 个基因继承 B 的对应基因；若 $W_i = 1$，则 A、B 的第 i 个基因相互交换，从而生成 X、Y 的第 i 个基因。

在对控制参数的改进上，Srinvivas 等人提出自适应遗传算法，即 P_C 和 P_m 能够随适应度自动改变，当种群的各个个体适应度趋于一致或趋于局部最优时，使二者增加，而当种群适应度比较分散时，使二者减小，同时对适应值高于群体平均适应值的个体，采用较低的 P_C 和 P_m，使性能优良的个体进入下一代，而低于平均适应值的个体，采用较高的 P_C 和 P_m，使性能较差的个体被淘汰 。

总之，遗传算法的特点是：以优化变量的遗传编码为运算及搜索对象；非单个操作，使用群体搜索策略，本质是并行的；使用概率搜索机制，无需其他信息；具有全局搜索能力，最善于搜索复杂问题和非线性问题，如多参数、多变量、多目标、多区域的 NP-hard 问题。