# 第一章 蛮力算法

大多数算法教材没有讨论蛮力算法的章节，我们讨论这部分内容有以下几个原因：(1) 通过蛮力算法可以把一些描述算法问题及其复杂性的基本概念以既严格又易懂的方式提前介绍给读者，这些概念包括搜索问题，优化问题，判定问题以及它们之间的相互联系，**PF**, **PC**, **P** 和 **NP** 这几个最重要的复杂度类以及它们之间的相互联系；(2) 我们以通用算法或称高阶算法的形式引入蛮力法，而不是当作一种算法策略，这为以后的通用搜索算法提供了准备；(3) 介绍了通用蛮力算法和基本数学对象的遍历算法后，大多数算法问题可以用很少的工作量和很小的难度快速实现第一个有效算法，尽管这一算法的复杂度较高，但它可以求解一定范围的问题实例，为其它更巧妙的算法提供正确性的实验参照。

## 1.1 组合构造问题和组合优化问题

算法课所讨论的问题基本上可以分为组合问题，代数和数论问题，几何问题这三大类， 其中组合问题数量最多，这类问题又可以分为组合构造问题和组合优化问题两个类别。组合构造问题要求对于给定的输入数据求出一个或多个满足特定条件的有限数学对象，如有限集合，有限函数（序列是自然数有限区间上的函数），有限图等。下面是两个在本课程中反复出现的组合构造问题的例子。

例1.1.1 子集和数问题：设$n$和$a$为正整数，$s$为从自然数集$N$的非空前段$D_n=\{0, ..., n-1\}$到正整数集合$N^+$的函数，求$D_n$的子集$A$, 使得$A$的全体元素的函数值和$\sum_{x\in A}s(x)$等于$a$。

例1.1.2 $n$皇后问题：8皇后问题要求在一个8×8格的国际象棋棋盘上放置8个皇后，使它们彼此不受到攻击。按照国际象棋的规则，一个皇后可以攻击与之处在同一行或同一列或同一斜线上的任何棋子。因此，8后问题等价于在国际象棋棋盘上放置8个皇后，使得任何两个皇后不能被放在同一行或同一列或同一斜线上。推广到$n\times n(n\geq 4)$的棋盘后称为$n$皇后问题。

可以通过算法求解的组合构造问题通常有一个共同的特点，就是对于给定的输入数据，所有满足要求的对象都在一个有限集合中，因此可以通过穷举的办法逐个检查求解，这种利用穷举和逐个检查的解题方法称为面向组合构造的蛮力法。

例1.1.1（续）因为$D_n$是有限集合，全体$D_n$的子集组成的集合，即$D_n$的幂集$P(D_n)$也是有限集合， 输入$<s, a>$的解都在这个有限集合$P(D_n)$中。

例1.1.2（续）将第i行第j列的位置用二元组$<i, j>$表示，对于$n\times n$的棋盘，全体位置的集合$A$是$n\times n$个元素的集合，在棋盘中选择若干位置放置皇后，相当于选择了集合$A$的一个子集，因此所有的解都在$A$的幂集$P(A)$，它是一个有限集。有$2^{n*n}$个元素。如果考虑到每行都要选择恰好一个位置，该位置可以用所在

列来刻画，因此可以把解看作是行的集合到列的集合的函数，全体这样的函数组成的集合B包含了所有解，它也是一个有限集，有$n^n$个元素。集合$B$比$P(A)$的元素少。如果考虑到每列只能有一个位置被选中，集合$B$的由全体一一对应函数组成的子集$C$也包含了所有解，$C$有$n!$个元素，数目比$B$又少了很多。

组合优化问题是一类特殊的组合构造问题，它们的特点是：对于给定的输入数据，可以引入一种优劣标准，通过该标准可以在所有满足要求的对象所组成的集合上定义一个全序（更确切地说，定义一个任意两个元素可比较的前序，全序条件中的反对称性质可以不满足），问题的要求是求最优解，确切地说是求出上述可比前序的最小（或最大）对象。通常这个可比前序是通过该集合上的一个整数值函数来定义的，最小对象就是该函数值最小的对象。下面是两个在本课程中反复出现的组合优化问题的例子。

例1.1.3　旅行售货员问题：某售货员要到若干城市去推销商品，已知各城市之间的路程（或旅费），要求选定一条从驻地出发，经过每个城市一次，最后回到驻地的路线，使总路程最短（或总费用最小）。用图论的术语，就是在一个完全网络中求长度最小的哈密尔顿回路。

例1.1.4　0-1背包问题：给定$n$种物品和一个背包，每个物品具有特定重量和价值，背包具有最大负荷重量，如何向背包中放入物品使得它们的总价值达到最大。

如果一个组合优化问题所对应的组合构造问题可以用面向组合构造的蛮力法求解，那么它本身可以用类似的蛮力方法求出最优解，该蛮力方法的特点是在原来的穷举和检查两个要素中增加比较操作作为第三个要素，称为面向组合优化的蛮力法。面向组合构造的蛮力法和面向组合优化的蛮力法统称蛮力法。

编程习题

习题1.1.1　求一个4位数$x$和一个一位数$y$，使得$x$与$y$的乘积$z$也是4位数，且$x, y, z$包含了所有1到9的数字。

习题1.1.2　求一个3位数$x$和一个2位数$y$，使得$x$与$y$的乘积$z$是4位数，且$x$，$y$，$z$包含了所有1到9的数字。

习题1.1.3　求3位数$x$，使得$2x$与$3x$也是3位数，且它们包含了所有1到9的数字。

习题1.1.4　求3位数$x$和2位数$y$，使得$x$与$y$的乘法算式只包含数字2，3，5，7。

## 1.2　具体算法和通用算法

蛮力法是一种相对简单的算法设计策略，但这并不能说明在使用这种策略设计和实现算法求解具体问题时不会遇到任何困难。对于一个特定问题，检查潜在解是否满足条件也可能是一个比较困难的任务。在更多情形下，设计一种穷举方法是一件技巧性很强的任务。然而在大多数情况下，蛮力法可以用一种非常规范的方式实现，常用的穷举方法也可以通过算法类库的形式一劳永逸地实现，并通过合适的接口与蛮力法的主流程模块进行方便的组合。这时的蛮力法已经不再是算法设计策略，而是一个通用算法了。本节讨论通用算法的一些基本概念。

通用算法并没有严格的定义，因为算法的通用性不存在精确的解释，通用算法与具体算法的关系是相对的。如果一个新算法基本上可以替代原有的一批算法，则这个新算法可以称为通用算法。更确切地说，设*a*是一个带参数的算法，*A*是一个算法集合，对*A*中的任一算法*b*，可以为*a*设计出一个实际参数*x*，使*a*代入参数*x*后在功能上相当于算法*b*，并且在效率上不次于*b*，参数*x*的实现难度明显小于算法*b*的实现难度，则算法*a*可以称为相对于算法集合*A*的一个通用算法。

通用算法的设计与实现既要建立在合适的数学模型之上，又离不开程序设计语言的支持。传统的过程式程序设计语言只适合用来对具有确定的数据结构的问题设计和实现算法。函数式程序设计语言和面向对象程序设计语言以不同的方式扩展了算法的概念，它们都为将其他算法作为参数的通用算法的构造与使用提供了足够的支持。面向对象程序设计语言通过对象、类、继承和接口等概念使算法可以通过标准组件的搭配来生成，与函数式程序设计语言不同，它是传统计算模型的改进而不是彻底的变革，因此对算法设计具有更重要的意义。为一批具体问题建立一个统一的数学模型和算法首先需要给出严格的数学定义，体现到算法设计与应用就是设计合适的接口，然后是两方面的工作，一是建立数学理论和相应的算法，二是将一个个具体问题描述成符合模型定义的形式。当这一批问题可以直接用数据表示时，后者可以不通过算法而直接实现，这样在第一步中建立的算法可以直接应用于这一批问题，绝大多数传统算法都属于这种情形。但是后一项工作也可能需要通过一个算法来完成，另一种可能是不同的问题需要不同的算法来将该问题描述成符合模型定义的形式。例如旅行售货商问题和0/1背包问题需要用不同的算法转换成整数线性规划问题。当在第一步中建立的面向数学模型的算法可以直接作用于具体的数据结构时，将两个或更多的算法组合起来解决一个具体问题用传统的过程式程序设计语言也能方便地实现，线性规划算法就是这样。当在第一步中建立的面向数学模型的算法本质上是一种将其他算法作为参数的高阶算法时，如果没有面向对象程序设计语言提供概念和工具的帮助，建立上述的数学模型和算法即使可能，也是非常困难的。

以蛮力法为例，使用作为算法策略的蛮力法为一个特定问题设计算法时，这个算法应具备穷举，检查，或者还包括比较等要素，但这些要素可能是紧密联系在一起的，穷举可以通过多重循环或递归进行并使用各种复杂数据结构，检查更是与特定问题高度相关，因此各种基于蛮力法的算法可以有多种多样的外在形式和变化。通用蛮力算法的思想则是把穷举的外在方式以遍历器的形式固定下来，蛮力算法由一个通用的主流程模块，一个遍历器模块和一个检查模块构成，主流程模块调用另外两个模块，检查模块以遍历器状态为参数，对于一个具体问题，只须单独实现检查模块，并从算法库中选择合适的遍历器模块。后面几节将给出详细的描述。

以状态空间搜索问题为例，回溯法的特征是将一个个具体问题转换为状态空间，然后对一个个特定的状态空间设计单独的搜索算法。通用搜索算法的思想则是首先为状态空间定义一个精确的计算模型，然后提出建立在该计算模型上的状态空间搜索算法*a*，设*b*是求解组合问题*P*的具体搜索算法，算法*a*对应于算法*b*的参数就是用状态空间表示组合问题*P*的算法。使用程序设计的术语，就是一个组

合问题的状态空间求解程序由通用搜索模块加上一个问题描述模块组成，不同问题具有不同的问题描述模块，通用搜索模块不随问题的改变而改变。

上述例子说明，通用算法作为一种高阶算法在很多情形下无法单独用来求解一个具体问题，需要同其它算法组合成一个问题求解方案。单独的通用算法也无法事先知道它所调用的算法是什么。因此通过接口调用其它算法的通用算法是无法单独分析时间和空间复杂度的，只有当它作为一个成分构成一个具体算法用来解决一个具体问题时，包含通用算法成分的具体算法才具有可以分析时空复杂度。但是通用算法是一个涵义广泛的概念，下面的例子给出了通用算法与具体算法相互关系的另一种情况。

例1.2.1 和差问题：小明和小红共吃了$m$个苹果，小明比小红多吃了$n$个，小明和小红各吃了几个苹果？($m$, $n$是正整数，$m>n$)

解答1 (蛮力法)：因为$m>n$, 说明小明没把$m$个苹果都吃了，小明最少吃了$m/2+1$(向下取整)个苹果，最多吃了$m-1$个苹果，在这些可能中检查小明比小红多吃了几个，若多吃了$n$个，这就是答案。
解答1的缺点：
1. 若$m$值很大，比如10000，该方法太麻烦。(算法效率低)
2. 有时无解，如$m$=10, $n$=3时, 6-4=2，7-3=4， 8-2=6，9-1=8哪个也不等于3。(算法失败)

解答2(调整法)：小明和小红先平分苹果，各得$m/2$个，然后小红给小明$n/2$个，结果小明吃了$(m+n)/2$个，小红吃了$(m-n)/2$个。

例1.2.2 鸡兔同笼问题：鸡兔同笼，共有头$m$只，腿$n$只，问鸡兔各几只？
($m$, $n$是正整数，n是偶数，$2m<n<4m$)

解答1 (形象思维法) ：训练鸡兔听从命令，然后让它们抬起两只腿，这样鸡两脚朝天，兔如猿人两腿直立，地上还有$n-2m$只腿, 因此有$n/2-m$只兔, $2m-n/2$只鸡.

解答2 (调整法)：先设笼中全是鸡, 此时缺$n-2m$腿，然后用兔子换鸡，换一只多两条腿，共需换$n/2-m$只，因此有$n/2-m$只兔, $2m-n/2$只鸡.

例1.2.3 高斯消元法是用来解线性方程组的算法。相对于求解和差问题的算法和求解鸡兔同笼的算法，高斯消元法是一个通用算法。用高斯消元法求解例1.2.1, 只需输入矩阵(1, 1, $m$//1, -1, $n$)。用高斯消元法求解例1.2.2, 只需输入矩阵(1, 1, $m$//2, 4, $n$)。但高斯消元法又可以看作一个具体算法，因为它接受具体的输入数据，给出具体的答案。它不需要调用其它算法，具有明确的时空复杂度特征。它可以用来求解多种类别的问题，但它比起那些功能更为单一的算法具有更高的时空复杂度。象和差问题和鸡兔同笼问题如果使用高斯消元法求解，必须进行更多的计算。

在本课程中，我们将在特定的含义下使用通用算法这一术语，即把它当作高

阶算法的同义词，并且只在讨论蛮力法，回溯法，分枝限界法这三个传统的算法设计策略时引入通用算法概念，以表达上述三个策略更适合改为高阶算法这一结论。

## 1.3 通用蛮力算法的数学模型

通用算法通常需要建立在合适的数学模型之上作为其接口设计的依据。本节讨论与通用蛮力算法有关的一些数学概念。

定义1.3.1 一个搜索问题是一个三元组$P=<A, B, R>$, 其中$R \subseteq A\times B$, 称为$P$的基本关系，$A$称为$P$的输入集，$B$称为$P$的解答范围，对$<a, b>\in R$, $b$称为$a$的一个解答，对$a\in A$, $R[a]=\{b|<a, b>\in R\}$称为$a$的解集。若对每个$a\in A$,$R[a]$是有限集，则称为有限搜索问题。

通常我们遇到的组合构造问题都是有限搜索问题。如果一个搜索问题不是有限搜索问题，因为从实用的角度我们只需要求有限的解，通常是一个解，所以可以通过把每个非空解集限制到一个非空子集的方法将原问题改造为一个有限搜索问题。

例1.3.1 对于子集和数问题，令$N$-{0}为全体正整数的集合，$S$为全体正整数非空序列的集合，即全体从$D_n=\{0, ..., n-1\}(n\in N-\{0\})$到$N$-{0}的函数组成的集合，
$A=S\times(N-\{0\})$, $B$为全体自然数有限子集的集合，
$R=\{<<s, a>, b>| s\in S, a\in N-\{0\}, b\subseteq dom(s), \sum_{x\in b}s(x)=a\}$,
则三元组$<A, B, R>$构成表示子集和数问题的有限搜索问题。

例1.3.2 对于$n$皇后问题，令$A$为全体不小于4的正整数的集合，
$B$为全体自然数前段$D_n=\{0, ..., n-1\}(n\geq 4)$上的置换组成的集合，
对$D_n$上的置换$f$, $f(i)=j$表示在第$i$行第$j$列放置一个皇后，在棋盘上斜率为-1的斜线上，行号减列号的值相等，在棋盘上斜率为1的斜线上，行号加列号的值相等，因此若两个皇后放置的位置分别是$<i, f(i)>$和$<j, f(j)>$, 它们位于同一斜线上当且仅当$i-f(i)=j-f(j)$或$i+f(i)=j+f(j)$, 即$|i-j|=|f(i)-f(j)|$。
令$R=\{<n, f>| n\in A, f$为$D_n$上的置换, 对$i, j\in D_n$, 若$i\neq j$, 则$|i-j|\neq|f(i)-f(j)|\}$,
则三元组$<A, B, R>$构成表示$n$皇后问题的有限搜索问题。

用蛮力法求解有限搜索问题的第一步是为每个解集找出一个有限超集，即包含该解集的有限集。

定义1.3.2 有限搜索问题$P=<A, B, R>$的一个限制函数$C$是$A$上的函数，满足: (1) 对$a\in A$, $C(a)\subseteq B$是有限集; (2)对每个$a\in A$, $R[a]\subseteq C(a)$。

例1.3.1（续）对于表示子集和数问题的有限搜索问题$P=<A, B, R>$，令
$C:A\rightarrow P(B)$, 对$<s, a>\in A$, $C(s, a)=P(dom(s))$, 则$C$是$P$的一个限制函数。

例1.3.2 续）对于表示$n$皇后问题的有限搜索问题$P=<A, B, R>$，令$C:A\to P(B)$，
对$n\in A$, $C(n)$=全体$D_n$上的置换组成的集合，则$C$是$P$的一个限制函数。

用蛮力法求解有限搜索问题的第二步是对每个输入$a$设计一个遍历$C(a)$的方法。在讨论遍历器之前，需要复习一下离散数学中学到的前序，全序和字典序等概念。

定义1.3.3 设$A$为集合，$I_A=\{<a, a>\mid a\in A\}$, $R$为$A$上的关系，若$R$满足自反性（$I_A\subseteq R$）和传递性（$R\circ R\subseteq R$），则称$R$为$A$上的前序关系，满足反对称性（$R\cap R^{-1}\subseteq I_A$）的前序关系称为$A$上的偏序关系，满足可比性（$A\times A\subseteq R\cup R^{-1}$）的前序关系称为$A$上的可比前序关系，满足可比性的偏序关系称为$A$上的全序关系。

定义1.3.4 设≤是非空有限集$A$上的一个全序，$A^*$是全体$A$中元素的有限序列组成的集合，即全体从$D_n=\{0, ..., n-1\}(n\in N)$到$A$的函数组成的集合，$A^*$上的由≤所确定的字典序$\le_d$定义如下：
对$f, g\in A^*$, $f\le_d g$ 当且仅当下列四种情况有一种成立
(1) $f=g$; (2) $f$是$g$的真前缀；(3) $f$和$g$都非空，$f(0)<g(0)$;
(4) $f$和$g$长度都大于$n(n\in N^+)$, 对任意$i\in D_n=\{0, ..., n-1\}$, $f(i)=g(i)$, $f(n)<g(n)$.

容易证明字典序$\le_d$是$A^*$上的全序。绝大多数有限搜索问题的答案都可以表示成自然数的有限序列，即$N^*$中的元素，最常见的穷举方法是按照字典序进行遍历。因此在本节中我们从全序的角度定义遍历器。为此先复习离散数学中关于非空有限集上的全序关系的一个简单结论。

命题1.3.1 (1)一个集合$A$是非空有限集的充要条件是存在一个从集合$A$到一个自然数集非空前段$D_n=\{0, ..., n-1\}(n\in N-\{0\})$的双射。
(2)设$f$是非空有限集合$A$到$D_n(n\in N-\{0\})$的双射。
令$R=\{<a, b>\mid f(a)\le f(b)\}$, 则$R$是$A$上的一个全序关系。
(3)设$R$是非空有限集$A$上的一个全序关系，则存在唯一的一个从集合$A$到某个自然数集非空前段$D_n(n\in N-\{0\})$的保序双射。

命题1.3.1说明集合$A$到一个自然数集非空前段$D_n=\{0, ..., n-1\}$的一个双射与集合$A$上的一个全序关系是一对等价的概念，通过这两个概念的任何一个可以给出集合$A$上遍历器的定义。

定义1.3.5 设≤是非空有限集$A$上的一个全序，$\perp\notin A$，$g$是从集合$A$到某个自然数集非空前段$D_n=\{0, ..., n-1\}(n\in N-\{0\})$的保序双射，令$i=g^{-1}(0)$, $next$是$A$到$A\cup\{\perp\}$的函数，对$a\in A$,若$g(a)=n$, $next(a)=\perp$,否则$next(a)=g^{-1}(g(a)+1)$. 二元组$<i, next>$称为$A$关于全序≤或双射$g$的遍历器。

下面的命题给出了遍历器的一种等价描述。

命题1.3.2 (1)设≤是非空有限集$A$上的一个全序，$g$是从集合$A$到某个自然数集非空前段$D_n=\{0, ..., n-1\}(n\in N-\{0\})$的保序双射，$<i, next>$是$A$关于全序≤的遍历器，

$next'$ 是 $A\cup\{\perp\}$ 到 $A\cup\{\perp\}$ 的函数，满足 (a)$next'(\perp)=\perp$; (b) 对 $a\in A$, $next'(a)=next(a)$, $f$ 是自然数集 $N$ 到 $A\cup\{\perp\}$ 函数，满足 (a)$f(0)=i$, (b) 对 $k\in N$, $f(k+1)=next'(f(k))$。则 $f$ 在 $D_n$ 上的限制等于 $g^{-1}$。

(2)设 $A$ 是包含 $n$ 个元素的非空有限集合($n\in N-\{0\}$), $i\in A$, $\perp\notin A$，$next$ 是 $A\cup\{\perp\}$ 到 $A\cup\{\perp\}$ 的函数，$next(\perp)=\perp$, $f$ 是自然数集 $N$ 到 $A\cup\{\perp\}$ 函数，满足(a) $f(0)=i$; (b) 对 $k\in N$, $f(k+1)=next(f(k))$。若 $f(n)=\perp$, $f$ 在 $D_n$ 上的限制是单射，则 $f^{-1}$ 是集合 $A$ 到 $D_n$ 的双射，$<i, next>$ 是 $A$ 关于双射 $f^{-1}$ 的遍历器。

命题1.3.2说明，在使用蛮力法设计算法的过程中，关于穷举要素的实现可以先为每个 $C(a)$ 指定一个全序，再按照这个全序进行遍历，也可以直接设计每个 $C(a)$ 的起点和后继函数，起点和后继函数应满足命题1.3.2(2)中的条件，即在遍历结束前不会回到曾经到过的地方，也不会提前结束。今后我们把满足命题1.3.2(2)中条件的 $<i, next>$ 称为 $A$ 的一个遍历器。

下面给出有限搜索问题的蛮力模型定义，它综合了限制，穷举和检查等要素。

定义1.3.6 设 $P=<A, B, R>$ 为一个有限搜索问题，$P$ 的一个蛮力模型是满足以下条件的四元组 $<C, I, NEXT, T>$:

(1) $C$ 是 $P$ 的一个限制函数;

(2) $I$ 是 $A$ 到 $B$ 的函数，令 $V=\cup_{a\in A}C(a)$, $NEXT$ 是 $A\times V$ 到 $V$ 的函数，对 $a\in A$，令 $NEXT_a$: $C(a)\to C(a)$, 对 $x\in C(a)$, $NEXT_a(x)=NEXT(a, x)$, 则 $<I(a), NEXT_a>$ 是 $C(a)$ 的一个遍历器;

(3) $T$ 是 $\cup_{a\in A}(\{a\}\times C(a))$ 到 $\{0, 1\}$ 的可计算函数，对 $a\in A$, $x\in C(a)$, $T(a, x)=1$ 当且仅当 $<a, x>\in R$。

在本课程中讨论的有限搜索问题的例子中，判定函数 $T$ 很容易直接通过有限搜索问题的基本关系 $R$ 用算法实现，因此在讨论蛮力模型的例子时我们通常省略 $T$ 的定义。遍历器的设计将在1.5节中讨论。

下面讨论组合优化问题的建模。

定义1.3.7 (1)一个有限优化搜索问题是一个四元组 $P=<A, B, R, \le>$, 其中 $<A, B, R>$ 为一个有限搜索问题，称为 $P$ 的有限搜索原型, $<A, B, R>$ 的输入集，解答范围和基本关系也称为 $P$ 的输入集，解答范围和基本关系。$\le$ 是 $R$ 上的前序关系，满足: (a) 对 $a_1, a_2\in A$，$b_1, b_2\in B$, 若 $<a_1, b_1>\le<a_2, b_2>$，则 $a_1=a_2$; (b) 对 $a\in A$，$b_1, b_2\in B$, 若 $<a, b_1>\in R$, $<a, b_2>\in R$, 则 $<a, b_1>\le<a, b_2>$ 或 $<a, b_2>\le<a, b_1>$。 $\le$ 称为 $P$ 的评价标准。

(2) $P$ 的有限搜索原型 $P'$ 的限制函数也称为 $P$ 的限制函数，$P'$ 的蛮力模型也称为 $P$ 的蛮力模型。

(3)设 $P=<A, B, R, \le>$ 是一个有限优化搜索问题，令

$R' = \{<a, b>|<a, b>\in R,$ 对任意 $b'\in B,$ 若 $<a, b'>\in R,$ 则 $<a, b>\le<a, b'>\}$,

则 $P'=<A, B, R'>$ 是一个有限搜索问题，称为 $P$ 导出的有限搜索问题，对 $a\in A$，$a$ 关于 $P'$ 的一个解答称为 $a$ 关于 $P$ 的一个解答。

尽管根据定义一个有限优化搜索问题可以导出一个有限搜索问题，但由于有限优化搜索问题$P$所导出的有限搜索问题$P'$通常并不存在简单的蛮力模型，所以$P$的求解主要依据的是$P$的有限搜索原型的蛮力模型，以及前序关系≤的判定算法。

绝大多数有限优化搜索问题$P$的评价标准是通过在$P$的基本关系$R$上定义一个整数值函数所导出的。容易证明如下命题：

命题1.3.3　设$<A, B, R>$为一个有限搜索问题，$v$是从$R$到整数集$\boldsymbol{Z}$的函数，
令$\leq_v$是$R$上的关系，对$<a_1, b_1>, <a_2, b_2> \in R$,
$<a_1, b_1> \leq_v <a_2, b_2>$ 当且仅当 $a_1=a_2$ 且 $v(a_1, b_1) \leq v(a_2, b_2)$.
则$<A, B, R, \leq_v>$组成一个有限优化搜索问题。

如果一个有限优化搜索问题$<A, B, R, \leq>$的评价标准≤是通过$R$到整数集$\boldsymbol{Z}$的函数$v$按命题1.3.3中的方式定义的，我们今后直接称四元组$P=<A, B, R, v>$为一个有限优化搜索问题，$v$称为$P$的评价函数。由于评价函数并不能对输入$a$导出$C(a)$上的一个全序，$a$的最优解通常不是唯一的，但是对于有限优化搜索问题，现实中一般只要求找出一个最优解。而对于有限搜索问题，有时要求找出多个解。

例1.3.3　对于旅行售货员问题，令$N$-{0}为全体正整数的集合，
$D_n=\{0, ..., n-1\}(n\geq 3)$，$I_{Dn}=\{<i, i>| i\in D_n\}$,
$A$为全体结点数不小于3的完全网络的集合，
即全体从$(D_n\times D_n)-I_{Dn}(n\geq 3)$到$N$-{0}的函数组成的集合，
$B$为全体自然数前段$D_n=\{0, ..., n-1\}(n\geq 3)$上的置换组成的集合，
$R=\{<g, p>| g\in A, f$为$D_n$上的置换, $g$的结点数$=n\}$,
$v:R\rightarrow \boldsymbol{Z}$, 对$<g, p>\in R$, 设$g$的结点数$=n$，$v(g, p)=\sum_{i\in Dn} g(i,(i+1) \bmod n)$
则四元组$P=<A, B, R, v>$构成表示旅行售货员问题的有限优化搜索问题。
令$C:A\rightarrow P(B)$, 对$n\in A$, $C(n)$=全体$D_n$上的置换组成的集合, 则$C$是$P$的一个限制函数。

例1.3.4　对于0-1背包问题，令$N$-{0}为全体正整数的集合，$S$为全体正整数非空序列的集合，即全体从$D_n=\{0, ..., n-1\}(n\in N-\{0\})$到$N$-{0}的函数组成的集合，
$A=\{<w, u, a>| w\in S, u\in S, a\in N-\{0\}, dom(w)=dom(u)\}$,
$B$为全体自然数有限子集的集合，
$R=\{<<w, u, a>, b>| <w, u>\in A, b\subseteq dom(s), \sum_{x\in b} w(x)\leq a\}$,
$v:R\rightarrow \boldsymbol{Z}$, 对$<<w, u, a>, b>\in R$, $v(<w, u, a>, b)=-(\sum_{x\in b} u(x))$.
则四元组$P=<A, B, R, v>$构成表示0-1背包问题的有限优化搜索问题。
令$C:A\rightarrow P(B)$, 对$<w, u, a>\in A$, $C(w, u, a)=P(dom(w))$, 则$C$是$P$的一个限制函数。

带有评价函数的有限优化搜索问题可以通过引入一个上界的方式转化为有限搜索问题，虽然输入多了一个分量，但它容易实现，用途广泛。

定义1.3.8 设$P=<A, B, R, v>$为一个有限优化搜索问题，令$A'=A\times \boldsymbol{Z}$,
$R'=\{<<a, k>, b>| k\in \boldsymbol{Z}, <a, b>\in R, v(a, b)\leq k\}$, 则$P'=<A', B, R'>$构成一个有限搜索问题，称为$P$量化导出的有限搜索问题。

## 1.4 基于蛮力模型的通用蛮力算法

本节我们给出有限搜索问题和有限优化搜索问题的通用蛮力算法，它们需要调用有关有限搜索问题的蛮力模型的实现算法。

算法1.4.1 对有限搜索问题*P*=<*A*, *B*, *R*>求*k*(*k*>0)个解的通用蛮力算法
输入：*a*, 接口：*P*的蛮力模型<*C, I, NEXT, T*>

```
1  num←0; b←I(a);
2  while (num<k and b!=⊥)
3  {      if  (T(a, b)=1)
4         {  ++num;  输出b; }
5         b←NEXT(a, b);  }
6  if  (num=0)
7      输出"无解！"
```

注：(1) 对于输入*a*, 若*a*的解数目不大于*k*，算法A1.4.1输出*a*的全部解后结束。因此算法A1.4.1可以用来求有限搜索问题的全部解，只须将*k*设置得足够大；
(2) 算法A1.4.1也可以只用来对输入*a*求解答数目，只须将*k*设置得足够大，删去循环体中输出答案的语句，在算法最后增加输出*num*的语句。

采用算法1.4.1作为主流程的具体蛮力算法其时间复杂度依赖于它所求解的有限搜索问题*P*的输入集和解答范围中元素的表示方法，以及蛮力模型中*NEXT*和*T*的实现算法的时间复杂度。其最坏时间复杂度主要取决于限制函数*C*的值所包含的元素个数相对于输入规模的增长速度。其平均时间复杂度主要取决于*P*关于输入*a*的解集*R*[*a*]相对于*C*(*a*)的密度，以及*R*[*a*]相对于*NEXT*所确定的遍历次序的分布情况。事实上有限搜索问题的蛮力模型是一个非常灵活的框架，它并不一定意味着低效率。如果限制函数能达到接近解集的程度，或者遍历器的次序能够以检查的成功率来确定，采用算法1.4.1作为主流程的具体蛮力算法也可以成为效率很高的快速算法。然而针对特定问题设计的具有高度技巧性的蛮力模型只是外表具备蛮力法的特征，其实质已经是专门算法了。普通的蛮力模型通常采用幂集，全体置换，全体序列等作为限制函数，用字典序来遍历，因此跟算法1.4.1组合后只能产生复杂度很高的算法，只适合求解规模不大的例子，或作为原型速成的第一个算法版本。

算法1.4.2 对有限优化搜索问题*P*=<*A*, *B*, *R*, ≤>求 一个最小解的通用蛮力算法
输入：*a*, 接口：*P*的蛮力模型<*C, I, NEXT, T*>

```
1  b←I(a);   q←true;   cur←null;
2  while (q and b!=⊥)
3  {    if  (T(a, b)=1)
4       {  cur←b;   q←false; }
5       b←NEXT(a, b);  }
6  if  (q)
```

```
7       输出"无解！"
8   else
9   {   while (b!=⊥)
10      {   if (T(a, b)=1 and <a, b> < <a, cur>)
11              cur←b;
12          b←NEXT(a, b);   }
13      输出cur;   }
```

注：(1) 为了降低空间复杂度，*NEXT*的实现通常采用原地变化的方式，第4行和第11行中的 *cur*←*b*; 是将*b*的一个复制赋值给*cur*；

(2) 第10行中的<*a, b*> < <*a, cur*>指的是<*a, b*>≤<*a, cur*> and !(<*a, cur*>≤<*a, b*>), 其中≤的含义是调用*P*的评价标准的实现算法，若≤是通过评价函数*v*所导出的，可以将<*a, b*> < <*a,* cur>改为*v*(*a, b*)<*v*(*a,* cur), 这里的<指的是普通的整数比较符，*v*(*a, b*)的含义是调用评价函数*v*的实现算法。

跟算法A1.4.1不同，采用算法A1.4.2作为主流程的具体蛮力算法其平均时间复杂度与最坏时间复杂度是一致的。其时间复杂度主要取决于*P*关于输入*a*的解集*R*[*a*]相对于*C*(*a*)的密度，遍历次序的选择不再影响主循环体的执行次数，但可能影响*NEXT*的实现算法的时间复杂度。

编程习题

习题1.4.1 实现通用蛮力算法A1.2.1，并用来求解子集和数问题。

习题1.4.2 实现通用蛮力算法A1.2.2，并用来求解0-1背包问题。

## 1.5 常见数学对象的遍历算法

有限搜索问题和有限优化搜索问题的解答范围通常是全体自然数有限序列的集合*N**的子集。其限制函数值的常见形式有以下四种：(1) 由全体从$D_n$到$D_m$(*m*>1, *n*>1)的函数组成的集合，通俗地说就是全体*m*个元素的长度为*n*的允许重复的排列，这里的*m*通常由问题本身确定，*n*通常是输入的规模, 例如$D_n$到$D_2$的函数可以表示*n*元集合的子集；(2) 由全体长度不大于*n*(*n*>1)的$D_n$中元素的递增序列组成的集合，用来表示*n*元集合的幂集；(3) 全体从$D_n$到$D_m$(1<*n*≤*m*)的严格递增函数组成的集合， 通俗地说就是全体从*m*个元素中取*n*个元素的组合，这里的*n*通常由问题本身确定，*m*通常是输入的规模, 它可以表示*m*元集合的全体*n*个元素的子集；(4) 全体从$D_n$到$D_m$(1<*n*≤*m*)的一一函数组成的集合, 通俗地说就是全体从*m*个元素中取*n*个元素的不允许重复的排列，这里的*n*通常由问题本身确定，*m*通常是输入的规模, 大多数情况下*n*=*m*，称为*n*个元素的全排列或者置换，*n*皇后问题和旅行售货员问题都属于这种情况。本节对这四种情况分别给出按字典序的遍历算法。

允许重复的排列是问题答案的常见形式，也是表示子集的最简洁的方法。它的遍历算法也最简单，只需要将排列看作*m*进制数做加一运算就可以了。

算法1.5.1对由全体从$D_n$到$D_m$($m$>1, $n$>1)函数组成的集合按字典序求后继的算法
输入：$m,n$($m$>1, $n$>1)，长度为$n$的整数数组$a$，满足0≤$a[i]$<$m$ (0≤$i$<$n$)

```
1   i←n-1;
2   while (i≥0 and a[i]=m-1)
3   {   a[i]←0;   --i;   }
4   if (i<0)
5         输出⊥;
6   else
7   {     ++a[i];
8         输出a;   }
```

注：本节的遍历算法都采用原地变化的方式以降低时空复杂度，它们与高级语言类库中针对数据结构的遍历器是不同的。

算法1.5.1的最坏时间复杂度是O($n$)。但后继算法是在一个规模更大的算法如通用蛮力算法中反复执行的一个成分，只有计算从初始元素起连续执行$k$次的总运行时间才有意义。在求解有限优化搜索问题的通用蛮力算法中，$k=m^n$，在求解有限搜索问题的通用蛮力算法中，$k\le m^n$. 通过简单的平摊分析，执行$k$次算法1.5.1的时间复杂度是O($k$)。

严格递增序列可以表示子集，比子集的2进制数表示时空效率更高。

例1.5.1 {a1, a2, a3, a4}的全体子集的严格递增序列表示以字典序排列如下：
∅，0，01，012，0123，013，02，023，03，1，12，123，13，2，23，3。

设由全体长度不大于$n$($n$>1)的$D_n$中元素的递增序列组成的集合为$C_n$，即
$C_n$={$f$|存在$i\in D_{n+1}$, $f$:$D_i\to D_n$, $f$为严格递增函数}。

算法1.5.2 对$C_n$按字典序求后继的算法
输入：$n$($n$>1)，长度为$n$的整数数组$a$，$a$的真实长度$m$(0≤$m$≤$n$)，满足
0 ≤ $a[i]$ <$n$(0≤$i$<$m$)，$a[i-1]$<$a[i]$(0<$i$<$m$)，

```
1   if   (m=0)
2   {     a[0]←0;   m←1;   }
3   else if (a[m-1]!=n-1)
4   {     a[m]←a[m-1]+1;   ++m;   }
5   else   if (m=1)
6   {     输出⊥; 算法停止;   }
7   else
8   {     --m;   ++a[m-1];   }
9   输出<a, m>;
```

算法1.5.2不包含任何循环成分，最坏时间复杂度为O(1)。用它遍历子集比算法A1.5.1效率更高。

有些有限搜索问题和有限优化搜索问题的解答范围是有限集合的大小固定的子集，这时前面给出的两个算法都无法使用，需要引入新的遍历算法。

例1.5.2　{a1, a2, a3, a4, a5}的全体3-子集递增序列表示以字典序排列如下：

012，013，014，023，024，034，123，124，134，234.

算法A1.5.3　对由全体从$D_n$到$D_m$(1<$n$≤$m$)的严格递增函数组成的集合按字典序求后继的算法

输入：$m$,$n$(1<$n$≤$m$)，长度为$n$的整数数组$a$，满足0≤$a$[$i$]<$m$ (0≤$i$<$n$)，$a$[$i$-1]<$a$[$i$](0<$i$<$n$)，

```
1   i←n-1;
2   while (i≥0 and a[i]=m-n+i)
3         --i;
4   if (i<0)
5         输出⊥;
6   else
7   {     ++a[i];   ++i;
8         while (i<n)
9         {     a[i]←a[i-1]+1;   ++i;   }
10        输出a;   }
```

对于不允许重复的排列，本节只讨论全排列的遍历，这对于大多数情况已经足够了。对于一般的排列，可以将算法1.5.3跟全排列遍历算法结合使用，但这时遍历次序已经不是字典序了。

例1.5.3　4个元素的全排列(或置换)以字典序排列如下：

0123，0132，0213，0231，0312，0321，1023，1032，1203，1230，
1302，1320，2013，2031，2103，2130，2301，2310，3012，3021
3102，3120，3201，3210.

算法1.5.4 对全体$D_n$上的置换组成的集合按字典序求后继的算法

输入：$n$($n$>1)，长度为$n$的整数数组$a$，满足0≤$a$[$i$]<$n$(0≤$i$<$n$), $a$[$i$]≠$a$[$j$](0≤$i$, $j$<$n$, $i$≠$j$)

```
1   i←n-1;
2   while (i>0 and a[i-1]>a[i]m-n+i)
3         --i;
4   if (i=0)
5         输出⊥;
6   else
7   {     --i;   j←n-1;
8         while (a[j]<a[i])
9               --j;
10        a[i]↔a[j];   ++i;   j←n-1;
11        while (i<j)
12        {     a[i]↔a[j];   ++i;   --j;     }
```

13　　输出*a*;　}

注：第10行和第12行中的↔表示将两个变量的内容互换。

本节讨论的四种情况还有很多其它的有趣算法，例如子集遍历的格雷算法，全排列遍历的Trotter-Johnson算法等。有限搜索问题和有限优化搜索问题的解答范围还可能是更复杂的数学对象，如有限集合的全体分划，自然数的全体分划，有限树的全体子树等，这些对象的遍历也都提出了引人入胜的问题。限于篇幅不再介绍，有兴趣的同学可参考Kreher和Stinson合写的《组合算法》，以及Knuth的《计算机编程艺术》第四卷。

编程习题

习题1.5.1　如何将1，2，3，4，5，6，7，12分别代入a-h, 使得下面的算式 成立？(减法和除法的顺序是从上倒下，从左到右)

a　*　b　=　c
=　　/
d　　e
-　　=
f　=　g　+　h

习题1.5.2: 如何将+,-, *, /分别代入A-D,将1，2，3，4，5，6，7，12分别代入a-h, 使得下面的算式 成立？ (减法和除法的顺序是从上倒下，从左到右)

a　A　b　=　c
=　　B
d　　e
C　　=
f　=　g　D　h

习题1.5.3 用通用蛮力算法A1.2.2和遍历算法A1.5.4求解旅行售货员问题。

习题1.5.4 用通用蛮力算法A1.2.1和遍历算法A1.5.2求解子集和数问题。

习题1.5.5: 用通用蛮力算法A1.2.2和遍历算法A1.5.2求解0-1背包问题。

## 1.6　具体搜索问题及其复杂性

有限搜索问题的定义为组合搜索问题提供了一个简单，灵活且实用的模型。绝大多数组合优化问题也可以看作特殊的有限搜索问题。然而它没有涉及问题的输入输出的表示形式，因而无法有效地反映问题复杂度特征。为此本节引入问题表示和具体搜索问题的概念以便讨论搜索问题的复杂度。

定义1.6.1 设$\Sigma$是一个有限字符集。

(1) 一个有限搜索问题$P=<A, B, R>$若满足$A=B=\Sigma^*$, 则称为$\Sigma$上的具体搜索问题，这时可以用$R$代替$P$;

(2) 有限搜索问题$P=<A, B, R>$的一个$\Sigma$表示是二元组$<f, g>$, 其中$f$, $g$分别是$A$, $B$到$\Sigma$的一一函数, 令$R'=\{<f(a), g(b)>|a\in A, b\in B\}$, 则$P'=<ran(f), ran(g), R'>$构成一个具体搜索问题，称为$P$的由$\Sigma$表示$<f, g>$导出的具体搜索问题。

例1.6.1 设$\Sigma$={0,1, #}, $P$=<$A$, $B$, $R$>是例1.3.1中给出的表示子集和数问题的有限搜索问题, 对$n\in N$-{0}，设$\theta(n)$是$n$的二进制表示，对从$D_n$={0, ..., $n$-1}($n\in N$-{0})到$N$-{0}的函数$s$, $a\in N$-{0}, 令$f(s, a)=\theta(s(0))\#\theta(s(1))\#...\#\theta(s(n-1))\#\theta(n)$, 对$N$的有限集合$b$={$i_1$, $i_2$, ..., $i_{n-1}$}，($i_1< i_2< ...< i_{n-1}$), 令$g(b)=\theta(i_1 s(0))\#\theta(i_2)\#...\#\theta(i_{n-1})$, 则<$f$, $g$>是$P$的一个$\Sigma$表示。

例1.6.2 设$\Sigma$={0,1, #}, $P$=<$A$, $B$, $R$>是例1.3.1中给出的表示$n$皇后问题的有限搜索问题, 对$n\in N$，设$\theta(n)$是$n$的二进制表示，对不小于4的正整数$a$, 令$f(a)=\theta(a)$; 对$D_n$上的置换$b$, 令$g(b)=\theta(b(0))\#\theta(b(1))\#...\#\theta(b(n-1))$, 则<$f$, $g$>是$P$的一个$\Sigma$表示。

定义1.6.2 对$\Sigma$上的一个具体搜索问题$R$，若存在具有多项式复杂度的算法$\boldsymbol{A}$, 使得对任意$x\in\Sigma^*$, 若$x$有解，$\boldsymbol{A}$输出$x$的一个解，否则$\boldsymbol{A}$输出$x$无解的信息，则称问题$R$可有效求解，全体可有效求解的具体搜索问题组成的类称为**PF**。

例1.6.3 目前尚未找到求解子集和数问题的具有多项式复杂度的算法，绝大多数学者相信子集和数问题不在**PF**中。

例1.6.4 $n$皇后问题显然不在**PF**中。因为在多项式时间内连输出都无法完成，输出长度是输入长度的指数级函数。

绝大多数无法找到具有多项式复杂度算法的具体搜索问题具有至多是指数级增长的限制函数，以及对基本关系的具有多项式复杂度的判定算法。下面的定义给出准确的表达。

定义1.6.3 一个可有效检查的具体搜索问题是满足以下两个条件的具体搜索问题$R$，(1) $R$是多项式平衡的，即存在$N$到$N$的多项式函数$p$, 使得每个$R$中的二元组<$x$, $y$>有$|y|\le p(|x|)$($|a|$表示字符串$a$的长度)。(2) $R$是多项式可判别的，即存在具有多项式复杂度的算法$\boldsymbol{A}$判别二元组<$x$, $y$>是否在$R$中。全体可有效检查的具体搜索问题组成的类称为**PC**。

例1.6.3（续）容易验证子集和数问题的基本关系是多项式平衡的和多项式可判别的，它是**PC**中的问题。

例1.6.4（续）$n$皇后问题的基本关系不是多项式平衡的，因此它不在**PC**中。

## 1.7 判定问题及其复杂性

每个搜索问题都对应一个更加简单的判定问题，即对于一个输入，该搜索问题是否存在解。优化问题所量化导出的搜索问题其对应的判定问题也称为原优化问题所对应的判定问题。判定问题因为形式简单，在计算复杂性理论中是更加重要的研究对象，尤其是在证明一个问题的复杂度较高时。如果一个搜索问题或优化问题所对应的判定问题都具有较高复杂度的话，原来的问题至少具有同样较

高的复杂度。

定义1.7.1 (1) 一个$\Sigma$上的判定问题是$\Sigma^*$到二元集合{0,1}的一个函数，也可以等价地看作$\Sigma^*$的一个子集，也称为$\Sigma$上的一个语言。
(2) $\Sigma$上的具体搜索问题$R$的定义域作为一个语言，称为$R$对应的判定问题。

定义1.7.2 (1) 对$\Sigma$上的判定问题$D$，若存在具有多项式复杂度的算法$\boldsymbol{A}$正确回答该问题，则称$D$可以有效判定，全体可有效判定的判定问题组成的类称为**P**。
(2) 对$\Sigma$上的判定问题$D$，若存在**PC**中的具体搜索问题$P$，使得$P$所对应的判定问题为$D$，则称$D$可以有效检查，全体可有效检查的判定问题组成的类称为**NP**。

定理1.7.1 **P**$\subseteq$**NP**，但**PF**$\subseteq$**PC**不成立。

定理1.7.2 **PC**$\subseteq$**PF** 当且仅当 **P=NP**。

例1.7.1 子集和数问题所对应的判定问题目前尚未找到具有多项式复杂度的实现算法，绝大多数学者相信该问题不在**P**中，由例1.6.3的结论，它在**NP**中。

例1.7.2 通过数学推导可证明，$n$皇后问题所对应的判定问题是取值1的常函数，根据例1.6.2引入的表示，具体的$n$皇后问题所对应的判定问题只须检查输入是否是一个不小于4的正整数，它在**P**中，根据定理1.7.1，它也在**NP**中。需要注意的是，一个判定问题D可以是多个不同的搜索问题所对应的判定问题，这些搜索问题只要有一个在**PC**中，D就在**NP**中。

# 第二章 组合搜索算法

本章讨论组合构造问题的搜索求解方法。这些内容通常在名为“回溯法”的章节里讲授。我们不使用回溯法这一名称有以下几个原因：(1) 回溯法指的是一种算法策略，本章介绍的是通用算法；(2) 回溯法的搜索对象是状态空间树，本章也讨论一般状态空间的搜索；(3) 回溯法是深度优先的搜索方法，本章对状态空间树也讨论宽度优先搜索；(4) 回溯法也包括对优化问题的搜索，本章只讨论组合构造问题的搜索。我们认为比起搜索策略的差别，问题种类的不同是更加重要的算法分类标准。

## 2.1 隐式图和显式图

大部分组合构造问题可以把输入数据组织成一个有向图，图中的结点集包括一个初始结点和一些目标结点，关于输入的解答可以表示成初始结点到目标结点的有向路径。给定有向图中的一个结点，它的后继结点可以通过问题所规定的规则算出，但是输入数据无法直接表示成一个有向图。本节我们讨论这类问题的数学模型。

定义2.1.1 一个隐式空间是五元组$S = <V, C, F, q_0, T>$，其中$V$是非空结点集，$C$, $F$是$V$上的函数，满足:

(1)对$v \in V$, $C(v)$是有限集;

(2) 对$v \in V$, $F(v):C(v) \to V_\perp$，其中$V_\perp = V \cup \{\perp\}$，$\perp \notin V$;

(3) 对$v \in V$, $x, y \in C(v)$, $F(v)(y) \neq v$，

若$x \neq y$, $F(v)(x) \neq \perp$, $F(v)(y) \neq \perp$，则$F(v)(x) \neq F(v)(y)$;

(4) $q_0 \in V$, $T \subseteq V$。$q_0$称为$S$的初始结点，$T$称为$S$的目标结点集。

若对每个$v \in V$，有自然数$n$，使$C(v) = D_n = \{0, \ldots, n-1\}$，则称$S$为简单隐式空间，此时可以用$n$代替$D_n$作为$C(v)$的值。

(5) 目标结点集为空集的隐式空间称为隐式图，将隐式空间$S$的目标结点集替换为空集后所得的隐式图称为$S$的基础图。

对$v \in V$, $C(v)$中的元素称为$v$的后继结点指针。一个结点可以通过后继结点指针来计算后继结点。后继结点指针集只保证不遗漏全体后继结点，不同的后继结点指针不会生成相同的后继结点，但不保证每个后继结点指针都确实生成一个合法的后继结点。对$v \in V$, $x \in C(v)$, $F(v)(x) = \perp$表示结点$v$无法通过后继结点指针$x$求出合法的后继子结点。

例 2.1.1 布线问题：印刷电路板将布线区域划分成 $n \times m$ 个方格阵列，要求确定连接方格 $q_0$ 的中点到方格 $q_1$ 的中点的一个或全体布线方案。在布线时，电路只能沿直线或直角布线，为了避免线路相交，已布线了的方格作了封锁标记，其它线路不允许穿过被封锁的方格。

下面给出布线问题的隐式空间描述。描述布线问题的隐式空间其结点是所有的空白方格，因此我们建立一函数 $f: D_m \times D_n \to \{0, 1\}$，对于 $i \in D_m, j \in D_n$，若

方格<$i, j$>是空白方格，令 $f(i, j) = 0$；若方格<$i, j$>是已经布了线的方格，
令$f(i, j) =1$，设$V = f^{-1}(0)$，$S = \{-1, 0, 1\}$，对$v \in V$, $C(v)=4$，
令$g: D_4 \to (S \times S)$，其中$g(0) = <1, 0>$, $g(1) = <0, 1>$, $g(2) = <-1, 0>$,
$g(3) = <0, -1>$。令$F: V \to (D_4 \to V_\perp)$，对$v \in V$, $i \in D_4$，设$a = v+g(i)$,
其中+为向量加法，若$a \in V$，令$F(v)(i) = a$，否则令$F(v)(i) = \perp$。
则$S = <V, C, F, q_0, \{q_1\}>$是描述从$q_0$ 到$q_1$的布线问题的一个隐式空间。

例2.1.2 平分牛奶问题，一个$2n(n \in N, n>1)$升的容器中装满了牛奶，另外有一个$a$升和一个$b$升的空容器$(a, b \in N)$，如何用它们来平分牛奶？

下面给出平分牛奶问题的隐式空间描述。

设$V = \{<i, j, k> | i \in D_a, j \in D_b, k \in D_{2n}, i+j+k=2n\}$, $V$的元素<$i, j, k$>表示$a$升容器中有$i$升牛奶，$b$升容器中有$j$升牛奶，$2n$升容器中有$k$升牛奶。操作方法有6种，每个容器可以向另外两个容器中倒奶，直到倒空或倒满为止。对$v \in V$, $C(v)=6$，令 $F: V \to (D_6 \to V_\perp)$，对 $v=<i, j, k> \in V$, $F(v)(0)$是 $a$ 升容器向 $b$ 升容器中倒奶的结果，若 $i=0$ 或 $j=b$, $F(v)(0)=\perp$，否则若 $i<b-j$，应该倒空， $F(v)(0)=<0, i+j, k>$，否则倒满，$F(v)(0)=<i+j-b, b, k>$，对其它的$m \in D_6$, $F(v)(m)$的定义与$F(v)(0)$类似，在此省略。$q_0 =<0, 0, 2n>$, $T=\{<i, j, k> | <i, j, k> \in V, i=n$ 或 $j=n$ 或 $k=n\}$。

则 $S= <V, C, F, q_0, T>$是描述平分牛奶问题的一个隐式空间。

每个隐式空间对应着一个显式空间，隐式空间的定义描述状态空间的构造方法，显式空间的定义描述状态空间的构造结果。

定义2.1.2 (1) 一个显式空间是四元组$S = <V, E, q_0, T>$，其中$V$是非空的结点集，$E$是$V$上的二元关系，它的元素用来表示有向边，<$V, E$>构成简单有向图，即<$V, E$>没有自环$(E \cap I_V = \varnothing)$，$q_0 \in V$, $T \subseteq V$, $q_0$称为$S$的初始结点，$T$称为$S$的目标结点集，满足:

(a) 对$V$的每个结点$v$，在<$V, E$>中$q_0$到$v$有一条有向路;

(b) 对$V$的每个结点$v$，在<$V, E$>中$v$只有有限条伸出边。

在<$V, E$>中从$q_0$到$T$中结点的一条有向路称为$S$的一个解。

(2) 设$S = <V, C, F, q_0, T>$是一个隐式空间，

令$E = \{<u, v> | u, v \in V$，存在$x \in C(u), F(u)(x) = v\}$，

<$V_1, E_1$>为$q_0$在<$V, E$>中导出的单向连通分支，即$V_1$是由在<$V, E$>中从$q_0$出发通过有向路能到达的全体$V$中结点所组成的集合，$E_1 = E \cap (V_1 \times V_1)$，则

$S_1 = <V_1, E_1, q_0, T \cap V_1>$是一个显式空间，称为$S$的导出空间，$S_1$的一个解也称为$S$的一个解。

(3) 目标结点集为空集的显式空间称为有源图，将显式空间$S$的目标结点集替换为空集后所得的有源图称为$S$的基础图。

隐式图与显式图的区别在于：结点之间的联系在显式图中是通过一个二元关系给出的，而在隐式图中则是通过结点集上的后继结点计算函数给出的。显式空间与隐式空间统称为状态空间。

例2.1.1（续）设$S = <V, C, F, q_0, \{q_1\}>$是如前所述的描述布线问题的隐式空间，

$E = \{<<a, b>, <c, d>> | <a, b>, <c, d> \in V, (a = c \wedge |b-d| = 1) \vee (b = d \wedge |a-c| = 1)\}$，

$<V_1, E_1>$为 $q_0$ 在$<V, E>$中导出的单向连通分支，则 $S_1 = <V_1, E_1, q_0, \{q_1\} \cap V_1>$是 $S$ 的导出空间。

## 2.2 简单隐式空间的通用搜索算法

对于具有隐式空间模型的组合构造问题$S$，可以利用图搜索的思想设计算法，但更合适的方法是设计隐式空间的通用搜索算法$A$和$S$的隐式空间的实现算法，二者的结合就是$S$的求解算法，不同的组合构造问题需要给出不同的隐式空间实现算法，通用搜索算法$A$则可以重用。与搜索算法$A$相比，隐式空间的实现算法通常是非常简单的任务，

隐式空间的通用搜索算法跟显式图的搜索算法类似，最常见的搜索策略是深度优先和广度优先。不同之处是需要设置一个集合对象存放已经生成的结点，基本的图操作不再是表示图的数据对象的访问，而是隐式空间的实现算法的调用。本节将给出隐式空间的几个通用搜索算法，包括深度优先搜索和广度优先搜索，深度优先搜索又分为狭义和广义两种类型。狭义深度优先搜索算法每次只生成当前结点的一个后继结点，然后就把新生成的结点当作当前结点。广义深度优先搜索算法一次生成当前结点的所有后继结点，再按深度优先的策略选择新的当前结点。因为描述实际问题隐式空间的绝大多数是简单隐式空间，本节只讨论搜索简单隐式空间的搜索。

算法2.2.1 对简单隐式空间求$k$($k$>0)个解的狭义深度优先搜索算法
输入：初始结点$q_0$ 接口：简单隐式空间$S = <V, C, F, q_0, T>$

```
1   堆栈path和集合nodeset初始化为空；num←0;
2   Push (<q0, 0, C(q0)>, path);   Add (q0, nodeset);
3   while   (path ≠∅   and   num<k)
4   {         <q, i, w>←Pop (path);
5         if   (i==0 and q∈T )
6         {   输出path的相关内容；++num;   }
7         found←false;
8         while   (!found   and   i<w)
9         {   sub←F(q)(i);   found←sub ≠⊥ and ! (sub∈nodeset);   ++i;   }
10        if   (found)
11        {      Push (<q, i, w>, path);   Push (<sub, 0, C(sub)>, path);
12              Add (sub, nodeset);   }   }
```

注：(1) 算法2.2.1中的集合操作都是针对标准接口，不需要实现确定具体实现形式，在使用算法2.2.1跟其它算法组合成求解具体组合构造问题的算法时再选择合适的集合对象，可以使用高级语言类库中提供的类，也可以针对具体问题自己实现效率更高的集合类；
(2) 算法2.2.1也可以设计成递归函数的形式，其时空复杂度都保持不变。

采用算法2.2.1作为主流程的具体搜索算法其时间复杂度依赖于它所求解的

简单隐式空间实现算法的时间复杂度。其最坏时间复杂度主要取决于隐式空间的边数相对于输入规模的增长速度。其平均时间复杂度还取决于后继结点的排列次序，对于一个结点的后继结点集，如果能将离目标结点集距离小的元素排列在前面，就能降低具体搜索算法的平均时间复杂度。一个问题可以有许多不同的隐式空间表示，其导出空间呈现出不同的规模，对搜索算法的最坏时间复杂度起着重大影响。即使两个隐式空间具有相同的导出空间，它们仍然可以导致搜索算法具有不同的平均时间复杂度。集合对象的实现方式对具体搜索算法的时间复杂度也有影响，但集合操作时间复杂度的降低通常以空间复杂度的提高为代价。

算法2.2.2 对简单隐式空间求$k$($k$>0)个解的广义深度优先搜索算法
输入：初始结点$q_0$ 接口：简单隐式空间$S$ = <$V$, $C$, $F$, $q_0$, $T$>

```
1   堆栈path和集合nodeset初始化为空；num←0;
2   Push (<q0, null>, path);   Add (q0, nodeset);
3   while (path ≠∅ and num<k)
4   {       <q, p>←Pop (path);   m←C(q);   i←0;
5           while (i<m)
6           {    sub=F(q)(i);
7                if (sub ≠⊥ and ! (sub∈nodeset))
8                {    Push (<sub, <q, p>>, path);   Add (sub, nodeset);
9                     if (sub∈T )
10                    {  沿返回指针链输出sub到q0的路径；++num;  }  }
11               ++i;  }  }
```

简单隐式空间的广义深度优先搜索算法在一定程度上是狭义深度优先搜索算法与广度优先搜索算法的折中。它的时间复杂度特征与算法2.2.1类似。

简单隐式空间的广度优先搜索算法与广义深度优先搜索算法的结构很相似，只是把算法2.2.2中的堆栈改成了队列。

算法2.2.3 对简单隐式空间求$k$($k$>0)个解的广度优先搜索算法
输入：初始结点$q_0$ 接口：简单隐式空间$S$ = <$V$, $C$, $F$, $q_0$, $T$>

```
1   队列path和集合nodeset初始化为空；num←0;
2   Push (<q0, null>, path);   Add (q0, nodeset);
3   while (path ≠∅ and num<k)
4   {       <q, p>←Pop (path);   m←C(q);   i←0;
5           while (i<m)
6           {    sub=F(q)(i);
7                if (sub ≠⊥ and ! (sub∈nodeset))
8                {    Push (<sub, <q, p>>, path);   Add (sub, nodeset);
9                     if (sub∈T )
10                    {  沿返回指针链输出sub到q0的路径；++num;  }  }
11               ++i;  }  }
```

简单隐式空间的广度优先搜索算法求出的第一个解是路径长度最短的解答，

如果解的路径长度可以作为解的评价函数，算法2.2.3可以用来求最优解。它的时间复杂度不再依赖于后继结点的排列方式，隐式空间的建模方式对搜索算法的影响只体现在隐式空间的实现复杂度和它的导出空间的规模。

编程习题

用算法2.1.1,算法2.1.2和算法2.1.3分别求解下列习题：

习题2.1 1 一个16升的容器中装满了牛奶，另外有一个6升和一个11升的空容器，如何用它们来平分牛奶？

习题2.1 2 小明拿着一个7升和一个11升的空容器来到水龙头前，他如何才能带回6升水呢？

习题2.1 3 小明拿着一个7升,一个13升和一个19升的空容器来到水龙头前，他如何才能在两个容器中各盛10升水呢？

习题2.1 4: 小明有两个100毫升的容器中装满了牛奶，另外有一个40毫升和一个70毫升的空碗，如何用它们在两个碗中各盛30毫升的牛奶，并且一点儿牛奶也不泼掉呢？

习题2.1 5 布线问题(例2.1.1).

## 2.3 展开树，标号空间和层次图

描述组合构造问题的隐式空间绝大多数其导出空间是一棵外向树。这是因为在蛮力算法一章中我们提到绝大多数有限搜索问题的解答范围是是全体自然数有限序列的集合$N^*$的子集，自然数的有限序列，不管是允许重复的，不允许重复的，还是严格递增的，都可以组织成一棵外向树。另外，每个显式空间都有一个展开树跟原空间具有同样的解集。下面给出严格定义。

定义2.3.1 (1) 一个有源图$G = <V, E, q_0, \varnothing>$如果满足：

(a) 对$V$中每个结点$v$，在$<V, E>$中$q_0$到$v$恰好有一条路;

(b) 对$V$中每个结点$v$，不存在从$v$到$q_0$的路；则称$G$为一个外向树。

在外向树中若$<v_1, v_2> \in E$, 则$v_1$称为$v_2$的父结点，$v_2$称为$v_1$的子结点。

一个显式空间$S$ 如果其基础图是一个外向树，则称$S$为显式空间树。

一个隐式空间$S$，若$S$的导出空间是显式空间树，则称$S$为隐式空间树。

(2) 有向图中的有向途径指的是允许重复的结点有限序列，其中相邻结点的前一个到后一个组成的有序对在$E$中。

设$S = <V, E, q_0, T>$是一个显式空间，令$V_1$为在$<V, E>$中起点为$q_0$的全体有向途径的集合，$V_2$为在$<V, E>$中起点为$q_0$的全体有向路的集合，

对$i$ =1, 2, $E_i = \{<\alpha,\beta> | \alpha,\beta \in V_i, \beta$包含$\alpha$并且比$\alpha$多一个结点$\}$,

对$\alpha \in V_i$，设$Last(\alpha)$为$\alpha$的末结点，

对$i$=1, 2, $T_i=\{\alpha | \alpha \in V_i$，$Last(\alpha) \in T\}$, $S_i = <V_i, E_i, q_0, T_i>$, 则$S_i$是显式空间树,

$S_1$和$S_2$分别称为$S$的广义展开树和狭义展开树。

广义展开树和狭义展开树统称展开树。

注：路是图论中的基本概念，这里不再给出定义，它跟途径的区别是在途径中允许出现重复结点，而路不允许出现重复结点。

有源图的展开树是外向树。有限显式空间的狭义展开树是有限显式空间树。无圈显式空间的广义展开树等同于它的狭义展开树。有圈显式空间的广义展开树是无穷显式空间树，对它的搜索通常限制在一定的深度。一个显式空间的解集同它的狭义展开树的解集是一致的，因此对每个显式空间的搜索都可以转换成对它的狭义展开树的搜索。然而这一转换不仅没有实现任何简化，反而大大增加了搜索的时间复杂度。回溯法的最常见的误用就是用树搜索来求解更适合采用图搜索的组合问题。

例2.3.1　$n(n>1)$个结点的有向完全图的广义展开树和狭义展开树。

设$V = \{-1\} \cup D_n$, $E = V \times D_n$, $q_0 = -1$，则$G = <V, E, q_0, \varnothing>$是一个有源图。

设$G_1$和$G_2$分别为$G$的广义展开树和狭义展开树，

则$G_1$的深度为$k$的结点表示从$D_n$中取$k$个结点的允许重复的排列，

即从$D_k$到$D_n$的映射。

当$n = 2$时，$G_1$的全体$k$层结点可以表示$D_k$的全部子集。

$G_2$的深度为$k$的结点表示从$D_n$中取$k$个结点的部分排列，即从$D_k$到$D_n$的单射。

$G_2$的最大深度为$n$，这一层全是叶结点，表示全体$D_n$元素的全排列，即$D_n$上的全体置换，$G_2$共有$n!$个叶结点。

例2.3.2　$n(n>1)$个结点的全序关系图的展开树。

设$V = \{-1\} \cup D_n$, $E = \{<x, y> \mid x, y \in V, x < y\}$, $q_0 = -1$,

则$G = <V, E, q_0, \varnothing>$是一个无圈有源图。

它的广义展开树$G_1$等同于它的狭义展开树。

$G_1$的深度为$k(k<=n)$的结点表示从$D_n$中取$k$个元素的组合，

即从$D_k$到$D_n$的严格递增映射。

它也表示集合$D_n$的由$k$个元素组成的一个子集，因此$G_1$可以表示$N$的幂集。

如果使用上一个例子所描述的有向完全图的展开树的子树$T$来表示子集，则$T$的结点比$G_1$的结点多一倍。因此$G_1$是表示子集的更理想的树。

对于隐式空间也可以定义对应的展开树概念。

定义2.3.2(1)　设$G = <V, C, F, q_0, T>$是一个隐式空间，令

$V_1 = \{\alpha \mid \alpha$是$V$中结点的有限序列且$\alpha$的头元素为$q_0\}$，

$V_2 = \{\alpha \mid \alpha$是$V$中结点的不允许重复的部分排列且$\alpha$的头元素为$q_0\}$;

对$i = 1, 2$，令$C_i$是$V_i$上的函数，对$\alpha \in V_i$，

设$Last(\alpha)$为$\alpha$的末结点，$C_i(\alpha) = C(Last(\alpha))$;

$F_i$是$V_i$上的函数,。对$\alpha \in V_i$，$F_i(\alpha): C_i(\alpha) \to (V_i)_\perp$，

对$x \in C_i(\alpha)$，设$y = F(Last(\alpha))(x)$，

若 $y \neq \perp$，令$F_1(\alpha)(x) = \alpha \bullet y$, 其中$\bullet$表示并置运算，否则令$F_1(\alpha)(x) = \perp$;

若$y \neq \perp$, $\alpha \bullet y \in V_2$，令$F_2(\alpha)(x) = \alpha \bullet y$，否则令$F_2(\alpha)(x) = \perp$;

对$i = 1, 2$, $T_i = \{\alpha \mid \alpha \in V_i$，$Last(\alpha) \in T\}$,

则$G_i = <V_i, C_i, F_i, q_0, T_i>$是隐式空间树，

$G_1$和$G_2$分别称为$G$的广义展开树和狭义展开树。

显式空间与隐式空间的展开树存在如下的联系。

定理2.3.1 隐式空间$G$的广义或狭义展开树其导出空间等于$G$的导出空间的广义或狭义展开树。

例2.3.3 骑士巡游问题：在$2n\times 2n(n>2)$的国际象棋棋盘上，求一个骑士按移动规则遍历整个棋盘的一个方案。更准确地说，如果骑士从某个位置出发，遍历所有位置后回到原处的巡游称为闭巡游，如果骑士从某个位置出发，遍历所有位置，但不要求回到原处的巡游称为开巡游。相关问题分别称为骑士闭巡游问题和骑士开巡游问题。

下面给出骑士巡游问题的隐式空间描述。设$V=D_{2n}\times D_{2n}$，$S=\{-2,-1,0,1,2\}$，对$v\in V$, $C(v)=8$，令$g:D_8\to(S\times S)$，其中$g(0)$到$g(7)$的值分别是

<2, 1>, <1, 2>, <-1, 2>, <-2, 1>, <-2, -1>, <-1, -2>, <1, -2>, <2, -1>;

令$F: V\to(D_8\to V_\perp)$，对$v\in V$, $i\in D_8$，设$a=v+g(i)$，

其中+为向量加法，若$a\in V$，令$F(v)(i)=a$，否则令$F(v)(i)=\perp$；

任取$q_0\in V$，则$S=<V, C, F, q_0, \varnothing>$是描述骑士移动规则的隐式图。$S$本身还无法描述骑士巡游问题。设$D$是$S$的导出空间，$S'$是$S$的狭义展开树，

令$T_1=\{x|x$是$S'$的结点，$x$的长度为$n^2\}$,

$T_2=\{x|x$是$S'$的结点，$x$的长度为$n^2$, $x$的末结点到$q_0$在$D$中存在有向边$\}$，

用$T_1$替换$\varnothing$作为$S'$的目标结点集所得的新的隐式空间树$S_1$就是描述骑士开巡游问题的隐式空间树。

用$T_2$替换$\varnothing$作为$S'$的目标结点集所得的新的隐式空间树$S_2$就是描述骑士闭巡游问题的隐式空间树。

一类常见的组合问题是求满足特定条件的一组对象的排列，用来表示这些排列问题的有源图通常是例2.3.1中的有向完全图的狭义展开树的子树。狭义展开树的实现算法相对复杂，通过给状态空间中的结点标号，并将状态空间解的要求增强到不仅组成有向路的结点必须互不相同，而且这些结点的标号值也必须互不相同，可以避免使用狭义展开树这一概念，把问题的难度从问题的描述转移到搜索算法的设计，而后者可以通过通用搜索算法一劳永逸地解决。状态空间的展开树是使用标号函数的最常见的例子，但标号状态空间的用途并不仅限于展开树的搜索。

定义2.3.2(1) 一个标号空间是三元组$S'=<S, A, L>$，其中$S$为一个状态空间，

$A$是一个集合，称为$S'$的标号集，

$L$是$S$的结点集到$A$的函数，称为$S'$的标号函数。

对$S$的一个解$s$，若$s$的任意两个不同结点的标号函数值都不相同，

则称$s$为$S'$的一个解。

(2) 设$S$是一个状态空间，$S_1$和$S_2$分别为$S$的广义展开树和狭义展开树，

$V$为$S$的结点集，对$i=1, 2$, $V_i$为$S_i$的结点集，

$L_i:V_i\to V$，对$\alpha\in V_i$，$L_i(\alpha)=\alpha$的末结点，

$S'_i=<S_i, V, L_i>$，则$S'_i$是标号状态空间，

$S'_1$和$S'_2$分别称为$S$的广义标号展开树和狭义标号展开树。

根据上面的定义，显式空间$G$的解集等同于其狭义展开树的解集，也等同于$G$

的广义标号展开树的解集。后者比前者更易于实现。

除了标号展开树以外，标号空间的另一种常见例子是有源图的层次图，下面给出定义。

定义2.3.3 (1) 设$S = <V, E, q_0, T>$是一个显式空间，$n \in N-\{0\}$，令$V'=V\times D_{n+1}$，
$E'=\{<<a, k>, <b, k+1>> | <a, b>\in E, k\in D_n\}$, $T'=T\times\{n\}$,
设$<V'', E''>$是$q_0$在$<V, E>$中导出的单向连通分支，$T''=T'\cap V''$,
则$S'' = <V'', E'', <q_0,, 0>, T''>$是一个显式空间，称为$S$的$n$次层次图。
(2) 设$G = <V, C, F, q_0, T>$是一个隐式空间，$n\in N-\{0\}$，令$V'=V\times D_{n+1}$, $T'=T\times\{n\}$,
$C'$是$V'$上的函数，对$<a, k>\in V'$，$C'(a, k)=C(a)$,
$F'$是$V'$上的函数，对$\alpha\in V'$，$F'(\alpha):C'(\alpha)\rightarrow(V')_{\perp}$，
对$a\in V, k\in D_n$，$x\in C'(a, k)$, $F'(a, k)(x)=F(a)(x)$,
对$a\in V, x\in C'(a, n)$, $F'(a, n)(x)=\perp$;
则$S'= <V', C', F', <q_0,, 0>, T'>$是一个隐式空间，称为$G$的$n$次层次图。
(3) 设$S$是一个状态空间，$V$为$S$的结点集，$n\in N-\{0\}$, $S'$为$S$的$n$次层次图,
$V'$为$S'$的结点集，$L:V'\rightarrow V$，对$a\in V, k\in D_{n+1}$，$L(a, k) = a$，
则$<S', V, L>$是标号状态空间，称为$S$的$n$次标号层次图。

显式空间与隐式空间的层次图存在如下的联系。

定理2.3.2 隐式空间$G$的$n$次层次图其导出空间等于$G$的导出空间的$n$次层次图。

例2.3.1（续）$n(n>1)$个结点的有向完全图的层次图。
设$V = \{-1\}\cup D_n$, $E = V\times D_n$, $q_0 = -1$，则$G = <V, E, q_0, V>$是一个显式空间。
设$k\in N-\{0\}$, $G$的$k$次层次图的解集对应于全体从$D_n$中取$k$个结点的允许重复的排列，即从$D_k$到$D_n$的映射。
当$n = 2$时，$G$的$k$次层次图的解集对应于$D_k$的全体子集。
当$k\in D_{n+1}-\{0\}$时，$G$的$k$次标号层次图的解集对应于全体从$D_n$中取$k$个结点的部分排列，即从$D_k$到$D_n$的单射。
$G$的$n$次标号层次图的解集对应于全体$D_n$元素的全排列，即$D_n$上的全体置换。

例2.3.2（续）$n(n>1)$个结点的全序关系图的层次图。
设$V = \{-1\}\cup D_n$, $E = \{<x, y> \mid x, y\in V, x < y\}$, $q_0 = -1$,
则$G = <V, E, q_0, V>$是一个显式空间。
对$k\in D_{n+1}-\{0\}$, $G$的$k$次层次图的解集对应于全体从$D_n$中取$k$个元素的组合，
即从$D_k$到$D_n$的严格递增映射。
该解集也对应于集合$D_n$的全体由$k$个元素组成的子集所组成的集合。

例2.3.3（续）设$S = <V, C, F, q_0, \varnothing>$是如前所述描述骑士移动规则的隐式图，
设$S' = <V, C, F, q_0, V>$, $D$是$S$的导出空间，
设$S_1=<V', C', F', <q_0,, 0>, T>$为$S'$的$n^2$次标号层次图，
$T'=\{<a, n^2> | a\in V, <a, q_0>$是$D$中的有向边$\}$，$S_2=<V', C', F', <q_0,, 0>, T'>$,
则$S_1$就是描述骑士开巡游问题的标号隐式空间，

$S_2$就是描述骑士闭巡游问题的标号隐式空间。

## 2.4 隐式空间树的通用搜索算法

隐式空间树的搜索算法是简化形式的隐式空间搜索算法，涉及集合操作的语句不再出现。

算法2.4.1 对简单隐式空间树求$k$($k$>0)个解的狭义深度优先搜索算法
输入：初始结点$q_0$ 接口：简单隐式空间树$S = <V, C, F, q_0, T>$

```
1  堆栈path初始化为空；num←0;
2  Push (<q0, 0, C(q0)>, path);
3  while  (path ≠∅  and  num<k)
4  {        <q, i, w>←Pop (path);
5        if  (i==0 and q∈T )
6        {  输出path的相关内容；++num;  }
7        found←false;
8        while  (!found  and  i<w)
9        {  sub←F(q)(i);  found←sub ≠⊥;  ++i;  }
10       if (found)
11       {  Push (<q, i, w>, path);  Push (<sub, 0, C(sub)>, path);  }  }
```

算法2.4.2 对简单隐式空间树求$k$($k$>0)个解的宽度优先搜索算法
输入：初始结点$q_0$ 接口：简单隐式空间$S = <V, C, F, q_0, T>$

```
1  队列path初始化为空；num←0;
2  Push (<q0, null>, path);
3  while (path ≠∅ and num<k)
4  {     <q, p>←Pop (path);  m←C(q);  i←0;
5        while (i<m)
6        {  sub=F(q)(i);
7           if (sub ≠⊥)
8           {  Push (<sub, <q, p>>, path);
9              if (sub∈T )
10             {  沿返回指针链输出sub到q0的路径；++num;  }  }
11          ++i;  }  }
```

算法2.4.3 对简单标号隐式空间树求$k$($k$>0)个解的狭义深度优先搜索算法
输入：初始结点$q_0$ 接口：简单标号隐式空间树 $S = <V, C, F, q_0, T, A, L>$

```
1  堆栈path和集合nodeset初始化为空；num←0;
2  Push (<q0, 0, C(q0)>, path);  Add (L(q0), nodeset);
3  while  (path ≠∅ and num<k)
4  {  <q, i, w>←Pop (path);
```

```
5        if   (i=0 and q∈T )
6        {    输出path的相关内容；++num;   }
7        found←false;
8        while (!found and i<w)
9        {    sub←F(q)(i);   ++i;
10            found←sub ≠⊥ and ! (L(sub)∈nodeset);   }
11       if   (found)
12       {    Push (<q, i, w>, path);   Push (<sub, 0, C(sub)>, path);
13            Add (L(sub), nodeset);   }
14       else
15            Remove(L(q), nodeset);   }
```

编程习题

习题2.4 1 用完全2叉树表示子集，用算法2.4.1和算法2.4.2求解子集和数问题。

习题2.4 2 用递增排列树表示子集，用算法2.4.1求解子集和数问题，用算法2.4.2寻找包含元素最少的解。

习题2.4.3 用算法2.4.3求解骑士开巡游问题，从一个角开始。

习题2.4.4 用算法2.4.3求解骑士闭巡游问题。

习题2.4.5 用算法2.4.3求解布线问题，跟习题2.1 5的程序比较运行时间。

## 2.5 回溯空间及其通用搜索算法

一个回溯空间是六元组$B=<V, C, F, q_0, T, P>$,其中$S=<V, C, F, q_0, T>$是树形隐式空间，$P: (V-\{q_0\})\rightarrow V$，满足:对$v\in V$，$x\in C(v)$，设 $F(v)(x)=s$，若$s\neq\perp$，则 $P(s)=v$. 回溯空间中的函数$P$称$B$为$B$的返回函数。若$S$是简单树形隐式空间，则称$B$为简单回溯空间。

隐式空间的(广义或狭义)展开空间是回溯空间的最常见的例子，其中作为展开空间结点的原隐式空间的结点序列其返回函数值是删去尾结点后所得的新结点序列。

因为狭义展开空间的解空间与原隐式空间的解空间存在一一对应，所以每个隐式空间都对应一个回溯空间，每个有限隐式空间都对应一个有限回溯空间。树形隐式空间搜索转换为回溯空间搜索可以不增加计算复杂度。

回溯空间是特殊的树形隐式空间，它的每个子结点包含了父结点的全部信息，因此在做狭义深度优先搜索时，只须保存一个当前结点。

算法2.5.1 对简单回溯空间求$k$($k$>0)个解的狭义深度优先搜索算法

输入：初始结点$q_0$ 接口：简单回溯空间$B=<V, C, F, q_0, T, P>$

```
1   堆栈path初始化为空；q←q0;   num←0;
```

```
2   Push (<0, C(q0)>, path);
3   while (path ≠∅ and num<bound){
4         <i, w>←Pop (path);
5         if   (i=0 and q∈T ){
6                输出path的相关内容；++num; }
7         found←false;
8         while   (!found   and   i<w){
9               sub←F(q)(i); found =   sub ≠⊥; ++i; }
10       if   (found){
11             Push(<i, w>, path);   Push(<0, C(sub)>, path);   q←sub;
12       }else q←P(q);
13 }
```

例6　0-1矩阵问题：求一个0-1矩阵，使每行的和与每列的和都等于给定的数值。

更准确地说，令$M = \{0, \ldots, m\text{-}1\}$，$N = \{0, \ldots, n\text{-}1\}$，

$row: M \to \omega$, $col: N \to \omega$, 求函数$f: M \times N \to \{0, 1\}$, 使得对每个$i \in M$,

$\Sigma\{f(i, k) \mid k \in N\} = row(i)$, 对每个$j \in N$, $\Sigma\{f(k, j) \mid k \in M\} = col(j)$。

一个枚举回溯空间是七元组$B=<V, C, F, q_0, T, P, I>$, 其中$<V, C, F, q_0, T, P>$$V, C, F, q_0>$是一个回溯空间, 对$v \in V$, $I(v)$是$C(v)$关于某个全序$R$的遍历器。

算法2.5.2 对枚举回溯空间求$k$($k$>0)个解的狭义深度优先搜索算法

输入：初始结点$q_0$　接口：枚举回溯空间$B= <V, C, F, q_0, T, P, I>$

```
1   堆栈path初始化为空；q←q0;   num←0;
2   Push (I(q0), path);
3   while (path ≠∅ and num<k)
4   {     itr←Pop (path);     found←false;
5         while (!found and itr!=⊥)
6         {     sub←F(q)(itr); found ← sub ≠⊥;    itr←next(itr);    }
7         if (found)
8         {     Push(itr, path);    Push(I(sub), path);    p←sub;
9               if (q∈T )
10              {     输出path的相关内容；++num;    }   }
11        else
12              p=P(p);    }
```

编程习题

习题2.5 1　用算法2.5.1求解子集和数问题。

习题2.5 2　用算法2.5.1求解骑士闭巡游问题。

习题2.5 3　用算法2.5.1求解$n$皇后问题。

习题2.5 4　用算法2.5.1求解0-1矩阵问题。

习题2.5 5 用算法2.5.2求解0-1矩阵问题。
习题2.5 6 用算法2.5.2求解循环赛日程安排问题。

## 2.6 排列树及其通用搜索算法

如果被搜索的树形状态空间是排列树或其子树，可以对算法2.4.1和算法2.5.1做一些优化，从而降低时间复杂度。

算法2.6.1 对作为排列树子树的回溯空间求$k$($k$>0)个解的狭义深度优先搜索算法
输入：初始结点$q_0$ 接口：作为$n$阶排列树子树的回溯空间$B$= <$V$, $C$, $F$, $q_0$, $T$, $P$>

```
1   堆栈path初始化为空；定义一个n个元素的整数数组perm;
2   q←q0;   num←0;   lev←0;   for (int i=0; i<n; ++i) perm[i]←i;
2   Push (lev, path);
3   while (path ≠∅ and num<k)
4   {       i←Pop (path);
5           if (i=lev and q∈T )
6           {       输出path的相关内容；++num;   }
7           found←false;
8           while (!found    and i<n)
9           {       sub←F(q)(perm[i]);   found← sub ≠⊥; ++i;   }
10          if (found)
11          {       perm[lev]↔perm[i];   ++lev;
11                  Push(i, path);   Push(lev, path);   q←sub;   }
12          else
13          {    q←P(q);   --lev;   perm[lev]↔perm[i];   }   }
```

编程习题
习题2.6.1 用算法2.6.1求解$N$皇后问题。

## 2.7 基于结点重排的通用搜索算法

经验表明，如果在树形状态空间的深度优先搜索时，如果先遍历根结点分叉少的子树，就会更快地搜索到第一个解。编写和实现算法验证这一结论。选择$N$皇后问题和跳马问题作为搜索对象。

# 第三章 优化搜索算法

本章讨论组合优化问题的搜索求解方法。这些内容通常在名为“分枝限界法”的章节里讲授。我们不使用分枝限界法这一名称有以下几个原因：(1) 回溯法指的是一种算法策略，本章介绍的是通用算法；(2) 分枝限界法的搜索对象是状态空间树，本章也讨论一般加权状态空间的优化搜索；(3) 分枝限界法是广度优先和最小代价优先的搜索方法，本章对状态空间树也讨论深度优先搜索。

## 3.1 赋值空间和单调赋值空间

设S为一个树形的隐式(或显式, 标号, 回溯)空间, V是S的结点集，S的一个赋值函数是V到整数集Z的函数，若赋值函数f在每个结点上的函数值都不小于其父节点的函数值，则称f是单调的赋值函数。带有(单调)赋值函数的隐式(或显式, 标号, 回溯)空间称为空间(单调)赋值隐式(或显式, 标号, 回溯)空间。

一个赋值隐式(或显式, 标号, 回溯)空间的最优解指的是终结点赋值函数值最小的从根到目标节点的有向路。

## 3.2 一般赋值(回溯)空间的搜索

算法3.2.1 一般赋值空间$\langle S, f\rangle$的深度优先搜索算法

```
1  堆栈path，线性表ans初始化为空, least=MAXINT;
2   Push (<q0, 0, C(q0)>, path);
3   while (path ≠∅){
4         <q, i, w>←Pop (path);
5         if   (i=0   and   q∈T   and f(q)<least)
6         {        将path复制到ans；least←f(q);   }
7         found←false;
8         while   (!found and i<w)
9         {       sub=F(q)(i); found←sub ≠⊥; ++i; }
10        if (found)
11        {      Push (<q, i, w>, path);   Push (<sub, 0, C(sub)>, path);   }   }
```

算法3.2.2 一般赋值回溯空间$\langle S, f\rangle$的深度优先搜索算法

```
1   堆栈path初始化为空, q←q0; ans=空指针;  least←MAXINT;
2   Push (<0, C(q0)>, path);
3   while (path≠∅)
4   {       <i, w>←Pop (path);
5         if   (i=0   and   q∈T   and f(q)<least)
```

```
6        {    将q复制到ans；least←f(q);  }
7        found←false;
8        while (!found and  i<w)
9        {     sub←F(q)(i); found ←sub  ≠⊥; ++i; }
10       if (found)
11       {    Push (<i, w>, path);  Push (<0, C(sub)>, path);  q←sub; }
12       else
13            q←P(q);  }
```

例3.2.1 连续邮资问题：假设国家发行了n种不同面值的邮票，并且规定每张信封上最多只允许贴m张邮票。连续邮资问题要求对于给定的n和m的值，给出邮票面值的最佳设计，在一张信封上可贴出从邮资1开始，增量为1的最大连续邮资区间。例如，n=5,m=4时，面值为(1, 3, 11, 15, 32)的5种邮票可以贴出邮资的最大连续区间是1-70.

思考题3.1：实现算法3.1并用来求解连续邮资问题。
思考题3.2：用算法3.1来求解0-1背包问题。
思考题3.3：实现算法3.2并用来求解0-1背包问题。

## 3.3 单调赋值空间的搜索

算法3.3.1 单调赋值空间<*S*, *f*>的深度优先搜索算法

```
1   堆栈path，线性表ans初始化为空, least←MAXINT;
2   Push (<q0, 0, C(q0)>, path);
3   while (path≠∅)
4   {    <q, i, w>←Pop (path);
5        if  (i=0  and  q∈T)
6        {     将path复制到ans；least←f(q); }
7        found←false;
8        while  (!found  and  i<w)
9        {    sub←F(q)(i); found ←sub  ≠⊥and f(sub)<least; ++i; }
10       if (found)
11       {    Push (<q, i, w>, path);  Push (<sub, 0, C(sub)>, path);  }  }
```

算法3.3.2 单调赋值空间<*S*, *f*>的最小代价优先搜索算法

```
1 优先队列queue初始化为空；
```

```
2   Add (<q0, null>, queue);
3.   found←false;
4   while (path ≠∅   and Not found)
5   {        <q, p>←PopMin (queue);   m←C(q);
6        for   (int i=0;   i<m; ++i)
7         {   sub←F(q)(i);
8                 if (sub ≠⊥)
9                 {         Add (<sub, <q, p>>, queue);
10                          if (sub ∈T )
11                         {          沿返回指针链输出sub到q0的路径;
12                                    found=true;   }    }    }   }
```

算法3.3.3 单调赋值回溯空间<*S*, *f*>的深度优先搜索算法A3.5：

```
1   堆栈path初始化为空, q←q0; ans←空指针;  least←MAXINT;
2   Push (<0, C(q0)>, path);
3   while   (path  ≠∅)
4   {      <i, w>←Pop (path);
5          if (i=0   and   q∈T )
6          {      将q复制到ans;  least←f(q);   }
7          found←false;
8          while (!found   and   i<w)
9          {      sub←F(q)(i); found ← sub  ≠⊥and f(sub)<least;   ++i; }
10         if (found)
11         {    Push (<i, w>, path);   Push (<0, C(sub)>, path);   q←sub; }
12         else
13             q←P(q);   }
```

思考题3.4：实现算法3.3并用来求解0-1背包问题。
思考题3.5：实现算法3.4并用来求解0-1背包问题。
思考题3.6：实现算法3.5并用来求解0-1背包问题。
思考题3.7：用动态规划来求解0-1背包问题。
思考题3.8：设计并实现单调赋值排列树的深度优先搜索算法，用它来求解旅行商问题。
思考题3.9：设计并实现单调赋值回溯排列树的深度优先搜索算法，用它来求解旅行商问题。
思考题3.10：用动态规划来求解旅行商问题。

## 3.4 限界函数

设*V*是单调赋值空间<*S*, *f*>的结点集*h*是*V*到*Z*的函数，满足：

(1) $h \geq f$; (2) 若$v$是$u$的后代结点且$v$是目标结点，则$h(u) \leq f(v)$；则称$h$是$<S, f>$的一个限界函数。

定理：若$h$是单调赋值空间$<S, f>$的一个限界函数，则在算法3.3， 3.4， 3.5中，用$h$函数代替$f$函数后，算法仍然保持正确。

思考题3.11：为旅行商问题设计限界函数，用来求解旅行商问题，通过比较检查限界函数的效果。

## 3.4 若干优化问题

例3.2 作业调度问题： n个作业要在由两个机器M1和M2组成的流水线上完成加工，每个作业的顺序都是先在M1上加工，再在M2上加工。M1和M2加工作业i所需的时间分别是$a_i$和$b_i$，给定一个作业加工顺序后，设作业i的完工时间是$t_i$，要求确定这n个作业的最优加工顺序，使得最后完工时间$Max(t_i)$或平均完工时间$\sum t_i$达到最小。

思考题3.12：为作业调度问题设计限界函数，用来求解作业调度问题，通过比较检查限界函数的效果。

例3.3 稳定婚姻问题：n个男性和n个女性按照自己的喜欢程度给所有异性打分排序。每个男性都凭自己好恶给每个女性打分：我最爱a，其次爱b，再次爱c...同样，每个女性也同样给每个男性打分。现在需要给他们搭配出n对新郎新娘，并且要保证所得到是稳定婚姻的搭配。那么，什么是不稳定的婚姻呢？所谓不稳婚姻是说，比如说有两对夫妇(M1、F1)和(M2、F2)，M1的配偶是F1，但他更爱F2；而F2的配偶虽说是M2，但她更爱M1——这样的婚姻就是不稳婚姻，M1和F2理应结合，他们现在各自的婚姻都是错误。设在一个稳定婚姻搭配中，男性i的配偶是第$a_i$满意的对象，女性j的配偶是第$b_j$满意的对象，要求寻找稳定婚姻搭配使得$\sum a_i$(总体上男性最满意)或$\sum b_j$达到最小(总体上女性最满意)。

思考题3.13：为稳定婚姻问题设计限界函数，用来求解稳定婚姻问题，通过比较检查限界函数的效果。

例3.4 最大团问题：求无向图的最大完全子图。

例3.5 最大独立集问题：无向图的独立集是指其中任意两点没有边相连的结点子集，求无向图的最大独立集。

例3.6 最小支配集问题：无向图的支配集是指一个结点子集，它包含每条边的至少一个结点。

## 3.5 隐式空间上的 Dijstra 算法