算法与数据结构

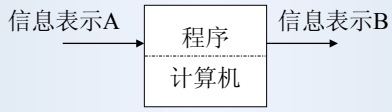
（2006.9～2007.1）

裘宗燕

北京大学数学学院

第一章　绪　　论

计算机程序实现信息表示形式之间的变换



如果：

- “信息表示A”表达了需要求解的某个问题
- “信息表示B”表达了相应的求解结果

就可以认为：这个程序完成该问题的求解

1.1 问题求解

使用计算机是为了解决问题。

为解决一个实际问题，就需要在计算机里建立起这个问题的求解模型：

1，把描述实际问题中的各种对象及其相互关系的信息映射为计算机存储器中的某种表示形式；

2，把对象领域中的问题求解过程映射到一个计算过程，用程序实现这个计算过程。

程序设计/软件开发就是要实现这些映射

问题：应该怎么做？

为解决一个实际问题而开发程序的工作通常可以分成以下四个阶段（常常需要重复）：

1，分析阶段：弄清需要求解的问题

2，设计阶段：设计出与问题对应的求解方案和有关实现细节的方案（信息的映射，求解过程等）
（与本课程关系最密切）

3，编码阶段：用某种计算机可以执行的形式，实现第2阶段的设计
（与本课程有关）

4，测试和维护：确认得到的程序能解决问题，为了某些目的而修改程序，扩充功能等等

问题求解示例

为一个多叉路口设计信号灯管理系统

不同行驶路线间可能出现冲突，存在安全问题

需要做出安排，对可能行驶路线分组，使得：

组内各方向行驶的车辆可同时安全行驶

(组尽可能大，提高效率)

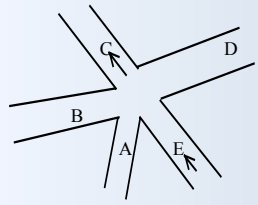


图1.1　一个交叉路口的模型

箭头表示单行线方向

1，问题分析

所有可能通行方向

A→B　A→C　A→D

B→A　B→C　B→D

D→A　D→B　D→C

E→A　E→B　E→C

E→D

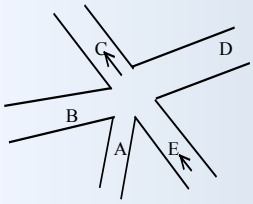


图1.1　一个交叉路口的模型

箭头表示单行线方向

用 AB 表示 A→B，余类推

有些方向不能同时行驶，在相应结点间画一条连线。

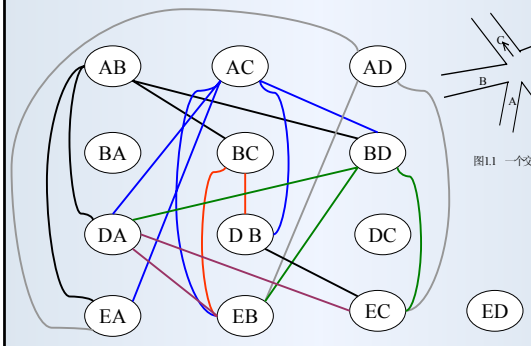


图1.1　一个交叉路口的模型

图1.2　交叉路口的图示模型（冲突图）

求解线索：把图1.2中结点分组，使得有边相连的结点不同组，同组的行驶方向互不冲突，可同时行驶

问题：哪些结点可同组，共分为多少个组？

一个显然解：每个方向独立作为一个组

进一步目标：分组最少（同时通行方向多，路通畅）

地图着色问题（一种模型）：

图中结点看作国家，结点间连线看作两国边界。上述问题就变成著名的“着色问题”：求可将图中所有国家着色，使相邻国的颜色不同的最少颜色数。

由于从具体问题得到的图（可能）不是平面图，因此需要的颜色数可能多于4。

2，算法设计

前面讨论构造了求解模型。算法设计研究求解方法

解决本问题有许多方法。

方法1．逐一检查所有可能组合。记录最小分组数和对应分组。一定能找到一个“最优”解（分组数最少的解）

缺点：可能的组合个数太多，逐个枚举需要指数时间

这一方法的效率太低。如果不同方向的集合比较大，求解时间可能长得无法忍受

方法2．“贪心法”

是一类典型算法，其宗旨是根据当时掌握的信息，尽可能地向得到解的方向推进

初始时：

集合 V_1 包含图G中的所有结点

集合 T = { }，准备存放分组

重复做：

while (存在未着色结点，即 V_1 不空) {

　选择一种新颜色;

　在未着色结点中给尽量多的无边互连的点着色;

}

采用贪心法，按结点排列顺序试探：

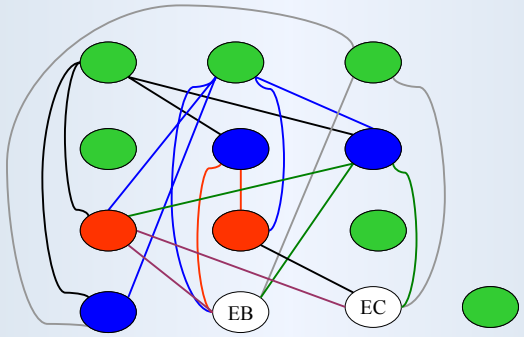


图1.2　交叉路口的图示模型

贪心法应用于图1.2，得到的分组：

绿色：AB，AC，AD，BA，DC，ED

蓝色：BC，BD，EA

红色：DA，DB

白色：EB，EC

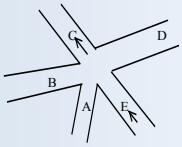
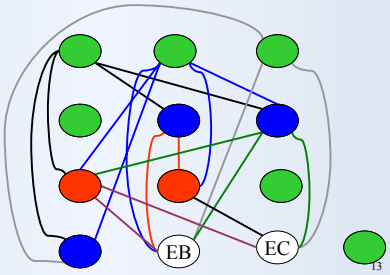


图1.1 一个交叉路口的模型



用一种新颜色着色的算法

设需要着色的图是G，集合V_1包括图中所有未被着色的结点，着色开始时V_1是G所有结点集合。NEW表示已确定可以用新颜色着色的结点集合。

找出V_1中可用新颜色着色的结点集的程序框架：

```
NEW={};
for (每个v ∈ V1) do
      if (v与NEW中所有结点间都没有边) {
            从V1中去掉v;
            把 v 加入NEW;
      }
```

结束时NEW中是可以用一种新颜色着色的结点。

上述算法的实现要基于集合和图的操作。

所需的集合和图操作：

- 判断一个集合是否为空：isSetEmpty(V1);
- 置一个集合为空：emptySet(NEW);
- 从集合中去掉一个元素：removeFromSet(V1,v);
- 向集合里增加一个元素：addToSet(NEW,v);
- 检查结点v与结点集NEW中各结点间在图G中是否有边连接：notAdjacentWithSet(NEW,v,G);

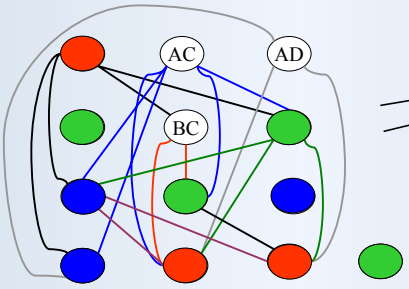
有了图、集合和其上的操作，程序实现就不难了

贪心着色算法（程序）：

```
int coloring( Graph G) {
  int color=0;   V1=G.V;
  while(!isSetEmpty(V1)) {
    emptySet(NEW);
    while(∃v∈V1&&notAdjacentWithSet(NEW,v,G)) {
      addToSet(NEW,v);
      removeFromSet(V1,v);
    }
    ++color;  save(NEW);
  }
  return(color);
}
```

收集起各个NEW集合，就可以得到一种分组。

分组并不唯一，另一种可能分组：



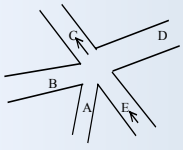
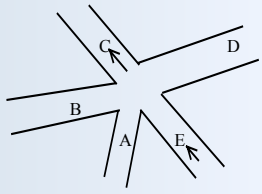


图1.1 一个交叉路口的模型

找出最小分组数不是简单工作。实际上，目前发现的算法都是通过枚举组合的方式



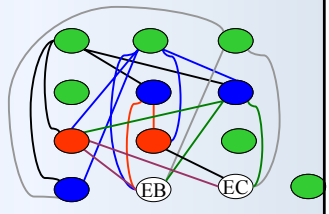


图1.1 一个交叉路口的模型

解决了原问题吗？

得到分组：

1：AB，AC，AD，BA，DC，ED

2：BC，BD，EA

3：DA，DB

4：EB，EC

如果希望提供最大行驶可能，问题就不是安全划分，而是基于安全划分的分组最大化。可从划分扩充得到

第一种分组的扩充：

1: AB, AC, AD, BA, DC, ED
2: BC, BD, EA, BA, DC, ED
3: DA, DB, BA, DC, ED, AD
4: EB, EC, BA, DC, ED, EA

后两组扩充时，都可在 AD 和 EA 中选择

其他问题：

- 分组如何更替？
- 持续时间（公平、实际需要）？
- ……

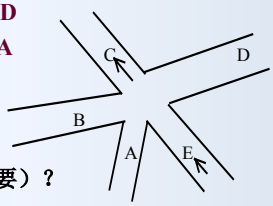


图1.1 一个交叉路口的模型

设希望做出一个程序，给它任一交叉路口的信息，它就能给出一种可行分组。

上面分析已给出了一种解决问题的方案（算法）。

解决问题的下一步是做出程序。怎么做？

如果语言本身提供了集合、图，以及所需的操作，前面的算法就可以直接翻译为程序了。

大多数语言没有这类高级结构，只有基本数据类型、数组、结构和指针等。

这时就需要自己实现集合、图及其相应操作。

如何有效实现这类高级结构就是数据结构研究的问题。

1.2 数据结构

- 数据(Data)：所有能输入计算机，由计算机程序处理的符号的总和。
- 数据元素(Data Element)：数据的基本单位，在程序中通常作为一个整体进行考虑和处理。
- 数据结构（Data Structures)：一组数据元素（结点）按照一定方式构成的复合数据形式，以及作用于这些元素或者结构上的一些函数或运算。

本课程讨论的主要数据结构：

线性表/字符串/堆栈/队列/树/二叉树/字典/集合/图

一种数据结构包含如下三个方面：

1. 逻辑结构：表示数据元素之间的逻辑关系。（元素之间的抽象关系，与具体实现无关）

   B = (E, R)

   元素取自集合E，元素间有关系R

2. 物理结构：数据的逻辑结构在计算机存储器中的映射（或表示），又称存储结构或存储表示
3. 结构的行为特征。作用于数据结构上的各种运算。例如：检索元素，插入元素，删除元素等

还有具体实现问题：所采用的语言，在该语言里采用什么具体方式和技术实现数据结构

1.3 算法（Algorithm)

算法是解题过程的精确描述，由有限条可完全机械执行的、含义明确的指令/语句构成。有如下性质：

1. 将它作用于所求解问题的给定输入集，或作用于给定问题的某种描述，将产生唯一确定的动作序列，而且此序列是有穷的
2. 此序列或终止于给定问题的解，或者终止并且指出本问题对于当前的输入无解

可行性；有穷性；确定性；输入；输出

程序：用某种计算装置能处理的语言描述的算法。

实际应用中的算法，表现形式千变万化。但是与数据结构的情况类似，许多算法的设计思想具有相似之处，可以对它们分类，进行学习和研究（第9章）

常见算法模式：

- 贪心法
- 分治法
- 回溯法
- 动态规划法
- 分支限界法

1.4 算法分析

需要考察算法的性质，比较不同算法的优劣，理解一个算法的适应范围。算法分析：度量算法性质的过程

最常用的算法度量特性是：

空间代价(空间复杂性)：被解决问题的规模(以某种单位计)为n时，某求解算法所需存储空间按某种单位为$S(n)$，称该算法的空间代价（空间复杂性）为$S(n)$

时间代价(时间复杂性)：当问题规模（以某种单位）为n时，算法所耗时间以某种单位计算为$T(n)$，则称该算法的时间代价（时间复杂性）为$T(n)$

三个重要概念：

* 问题规模
* 空间单位
* 时间单位

需要根据实际问题的情况确定。

例如，对n × n矩阵的常规的矩阵乘法算法

主要运算：乘法（或者加法）

空间单位：一个元素所占的存储空间s

时间单位：一次数值乘法所用时间t

时间复杂性：$f(n) = n \times n \times n = n^3$（单位为t）

大O记法：

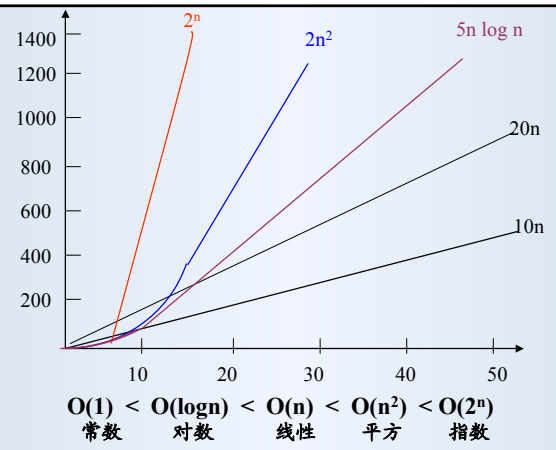
人们通常关心算法复杂性的量级，而不是具体函数

定义：若存在正常数c和n_0，当问题规模$n \geqslant n_0$后，某算法的时间(或空间)代价$T(n) \leqslant c \cdot f(n)$，则说该算法的时间代价(或者空间代价)为 $O(f(n))$

这时也说该算法的时间(或空间)代价的增长率为f(n)。也就是说，当n充分大时，该算法所需时间（空间）不大于 f(n) 的某常数倍

- 大O记法表示算法复杂性的上界（的量级）
- 还可以考虑下界，上确界，下确界问题

类似于无穷大的阶（级）



求解某问题的一个具体算法，对问题的不同实例，也可能需要耗费不同的时间和空间，因此人们常考虑：

- 最坏情况时间复杂性
- 平均时间复杂性
- 最好情况时间复杂性（不太有用）

对空间复杂性也可以有类似考虑

算法时间复杂性的计算规则

1. 加法规则（顺序复合）

算法分为两部分时，复杂性是两部分的复杂性之和

$$T(n) = T_1(n) + T_2(n) = O(f_1(n)) + O(f_2(n))$$
$$= O(\max(f_1(n), f_2(n)))$$

2. 乘法规则（循环）

循环 $T_1(n)$ 次，每次$T_2(n)$ 时间，则

$$T(n) = T_1(n) \times T_2(n)$$
$$= O(f_1(n)) \times O(f_2(n)) = O(f_1(n) \times f_2(n))$$

例：矩阵乘法

$C_{n\times n} = A_{n\times n} \times B_{n\times n}$

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++) {
        c[i][j]=0;
        for(k=0; k<n; k++)
            c[i][j] = c[i][j]+a[i][k]*b[k][j];
    }
```

$T(n) = O(f_1(n)\times f_2(n)\times(O(1)+O(n)))$

$= O(f_1(n))\times O(f_2(n))\times O(n)$

$= O(n)\times O(n)\times O(n) = O(n^3)$

1.5 抽象数据类型

假设要在C程序里处理一些复数。可能方式：

1. 直接用一对double变量表示一个复数。

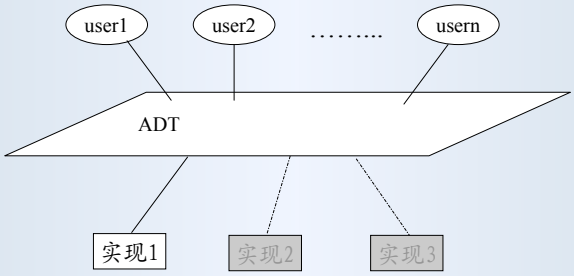
   不好！很难用（尤其是程序里有多个复数时）

2. 定义一种结构（类型）表示复数，在程序里使用复数时直接去操作这种结构的变量。

   不好！对复数结构的操作散布在程序各处，每个地方都要保证正确，修改复数的表示很困难……

3. 定义一种结构类型表示复数，并定义一组基本操作。程序里只通过这些操作使用复数，不直接使用复数的内部表示。（抽象数据类型的方式）

抽象数据类型（ Abstract Data Type，ADT ）的意义



ADT建立基于数据类型的抽象屏蔽，将程序的实现与使用部分隔开，使之相互独立。

抽象数据类型(ADT)是一种思想，也是一种程序技术

大系统必须模块化（复杂程序分为相对独立的片段）

函数、过程是最基本的控制抽象，也是模块化。但仅有控制抽象还不够（方式单一，不够灵活）

ADT为模块划分提供了另一重要形式：复杂程序的基本部分可看作（定义为）一些不同层次的ADT。

实践证明，基于数据类型的分层技术非常有效，这已经成为构造复杂软件的最重要的技术基础

一个ADT可看作定义了一组操作的一种抽象数据模型。它提供一种可创建的抽象的数据“值”，所定义的操作可作用于这种值。各操作之间有特定语义关系

ADT通过操作刻画所定义数据对象的性质，与对象的表示无关。隐藏了运算实现的细节和内部数据结构。

ADT需为用户提供使用该数据类型所需的完整信息，通常包括创建对象的手段和各种操作使用形式的说明（函数原型）——这些构成数据类型的使用界面

例：可把集合定义为ADT，提供创建集合的操作，判别集合是否空集，元素包含检查，加入删除元素的操作，以及集合的并、交、差集运算等。

一个ADT的操作集由设计者根据实际需要确定。

以集合为例，如果需要，也可以增加一些其他操作，例如判断两个集合是否相等，等等。

例：抽象数据类型Circle

```
ADT Circle is
data
        real  r;  real x, y;
operations
        constructor( )  构造一个圆
        real  area ( ) { return(3.14*r*r);}
        real  circumference( ) { return(2*3.14*r); }
        ... ...
end ADT Circle;
```

在多数语言里可以用某种方式“实现”ADT

不同语言对ADT的支持程度不同

C语言里的ADT技术

C没有为ADT提供专门支持（C语言设计时还没认识到ADT的重要性），但可通过程序技术模拟

在C里实现一个ADT通常用两个文件，.h 文件定义数据表示（定义类型）和操作原型，.c 文件实现操作

以Circle为例，circle.h 文件：

```
typedef struct { double x, y, r; } Circle;
Circle createCircle (double x, double y, double r);
double area (Circle c);
double circumference (Circle c);
... ... /* 其他操作的声明（原型） */
```


circle.c 文件：

```
#include "circle.h"
const double pi = 3.14159265;
circle createCircle (double x, double y, double r) {
   Circle c;
   c.x = x; c.y = y; c.r = r;
   return c;
}
double area (Circle c) {   return c.r *pi *pi; }
double circumference (Circle c) {   return 2 * c.r * pi;  }
... ... /* 其他操作的实现 */
```

使用Circle类型的文件应 #include "circle.h"，做可执行程序时应把文件 circle.c 作为其中一部分

数据结构的基本实现技术

实现数据元素和数据元素之间的关系：

- 简单元素直接映射到语言的基本类型，复杂元素的实现可以借助数据结构实现技术
- 数据元素间的关系映射到计算机里的基本方式：
  - 顺序表示：借助于数据元素的位置关系表示它们之间的逻辑关系。简单，但只能表示元素之间的顺序关系
  - 链接表示：以某种方式（显式）存储元素间关系（如用指针）。这样表示占存储，要维护和处理（要花时间），但更灵活，通过设计可以表示各种复杂关系
  - 在一个数据结构的实现里，完全可能采用不同方式表示元素间的多种关系

小　　结

在用计算机解题的过程中，算法和数据结构是相辅相成两个方面。数据结构是算法处理的对象，也是设计算法的基础。

一个具体问题的数据往往可采用多种不同的数据结构表示；一个计算过程的实现常有多种可用算法。因此算法和数据结构的选择就成为实现程序的过程中最重要的一个问题。

抽象数据类型技术提高了程序设计的抽象层次，也为数据结构和算法的研究提供了一种分层模型。

重点掌握数据结构和算法的基本知识，理解它们在问题求解过程中的地位和作用。

需要掌握和理解的概念：数据的逻辑结构、存储结构和操作，抽象数据类型，算法分析，空间代价和时间代价等。