第 1 章
程序员与算法

本章的标题既然是“程序员与算法”，就必然要涉及一个基本问题，那就是“程序员是否必须会算法”。这是一个充满争议的问题，虽然并不像“生存还是毁灭”之类的选择那样艰难而沉重，但也绝不是一个轻松的话题。朋友们在我的“算法系列”博客专栏上发表的评论和回复，并不都是我所期待的赞美和鼓励，也常常会有一些冷言冷语。比如，“穷举也算是算法吗”或者“请你说明一下算法在 XX 系统中能起到什么作用”。

有一次，一个网友通过邮件问我：“你写的都是小儿科的东西，几十行代码就能搞定，能不能整一点高深的算法？”我反问他什么是他所理解的高深的算法，他答复说：“像遗传算法、蚁群算法之类的。”于是我给了他一个遗传算法求解 0-1 背包问题的例子（参见第 16 章），并告诉他，这也就是几十行代码的算法，怎么理解成是高深的算法？他刚开始不承认这是遗传算法，直到我给了他 Denis Cormier 公开在北卡罗来纳州立大学服务器上的遗传算法的源代码后，他才相信他一直认为深不可测的遗传算法的原理原来是这么简单。

还有一个网友直言我写的“用三个水桶等分 8 升水”之类的问题根本就称不上算法，他认为像“深蓝”那样的人工智能才算是算法。我告诉他计算机下棋的基本理论就是博弈树，或者再加一个专家系统。但是他认为博弈树也是很高深的算法，于是我给了他一个井字棋游戏（参见第 23 章），并告诉他，这就是博弈树搜索算法，非常智能，你绝对战胜不了它（因为井字棋游戏很简单，这个算法会把所有的状态都搜索完）。我相信他一定很震惊，因为这个算法也不超过 100 行代码。

对于上面提到的例子，我觉得主要原因在于大家对算法的理解有差异，很多人对算法的理解太片面，很多人觉得只有名字里包含“XX 算法”之类的东西才是算法。而我认为算法的本质是解决问题，只要是能解决问题的代码就是算法。在讨论程序员与算法这个问题之前，我们先探讨一个最基本的问题：什么是算法。

1.1　什么是算法

《算法导论》一书将算法（algorithm）描述为定义良好的计算过程，它取一个或一组值作为输入，并产生一个或一组值作为输出。Knuth 在《计算机程序设计艺术》一书中将算法描述为从一个步骤开始，按照既定的顺序执行完所有的步骤，最终结束（得到结果）的一个过程。Weiss 在《数据结构与算法分析》一书中将算法描述为一系列的计算步骤，将输入数据转换成输出的结果。

虽然没有被普遍接受的“算法”的正式定义，但是各种著作中对算法的基本要素或基本特征的定义都是明确的，Knuth 总结了算法的四大特征。

- 确定性。算法的每个步骤都是明确的，对结果的预期也是确定的。
- 有穷性。算法必须是由有限个步骤组成的过程，步骤的数量可能是几个，也可能是几百万个，但是必须有一个确定的结束条件。
- 可行性。一般来说，我们期望算法最后得出的是正确的结果，这意味着算法中的每一个步骤都是可行的。只要有一个步骤不可行，算法就是失败的，或者不能被称为某种算法。
- 输入和输出。算法总是要解决特定的问题，问题来源就是算法的输入，期望的结果就是算法的输出。没有输入的算法是没有意义的，没有输出的算法是没有用的。

算法需要一定的数学基础，但是没有任何文献资料将算法限定于只解决数学问题。有些人将贪婪法、分治法、动态规划法、线性规划法、搜索和枚举（包括穷尽枚举）等方法理解为算法，其实这些只是设计算法常用的设计模式（Knuth 称之为设计范式）。同样，计算机程序只是算法的一种存在形式，伪代码、流程图、各种符号和控制表格也是常见的算法展示形式。而顺序执行、并行执行（包括分布式计算）、递归方法和迭代方法则是常用的算法实现方法。

综合以上分析和引述，本人将算法定义为：算法是为解决一个特定的问题而精心设计的一套数学模型以及在这套数学模型上的一系列操作步骤，这些操作步骤将问题描述的输入数据逐步处理、转换，并最后得到一个确定的结果。使用“精心设计”一词，是因为我将算法的设计过程理解为人类头脑中知识、经验激烈碰撞的过程，将算法理解为最终“小宇宙爆发”一般得到的智力结果。

1.2　程序员必须要会算法吗

很多人可能是好莱坞大片看多了，以为计算机神通广大，但事实不是这样的。计算机其实是一种很傻的工具，傻到几乎没有智商（至少目前是这样）。它可以连续几年做同一件事情而毫无怨言，但是如果你不告诉它怎么做，它什么事情也不会做。最有创造性的活动其实是由一种被称为“程序员”的人做的，计算机做的只不过是人类不愿意做的体力活而已。比如图像识别技术，需要一个字节一个字节地处理数据，提取数据的特征值，然后在海量的数据中比较、匹配这些特征值，直到累得两眼昏花，人类才不会干这种傻事儿呢。计算机愿意做，但前提是你要告诉它怎

么做。算法可以理解为这样一种技术，它将告诉计算机怎么做。有人将编程理解为搭积木，直接用别人开发好的组件、库，甚至是类或 API 就行了，并且美其名曰“不用重复发明轮子”。我认为这其实就是所谓的系统集成，如果一个程序员每天的工作就是搭积木，那将是令人十分羡慕的事情，但是我知道，事实并不是这样的。这样搭积木式的编程计算机就可以做，没有必要让人来做，因为人工的成本高于计算机。我遇到的更多的是在论坛里发帖求助的人，比如“求代码，把一个固定格式的文本文件读入内存”，再比如“谁能帮我把这个结构数组排排序啊，书上的例子都是整数数组排序”。他们是如此地无助，如果不是论坛对回帖有积分奖励的话，恐怕不会有人理他们。

我要说的是，大多数程序员并不需要知道各种专业领域里的算法，但是你要会设计能够解决你面临问题的算法。一些领域内的经典问题，在前人的努力之下都有了高效的算法实现，本书的很多章节都介绍了这样的算法，比如稳定匹配问题，比如 A*算法，等等。但是更多情况下，你所面临的问题并没有现成的算法实现，需要程序员具有创新的精神。算法设计需要具备很好的数学基础，但数学并不是唯一需要的知识，计算机技术的一些基础学科（比如数据结构）也是必需的知识，有人说：程序 = 算法+数据结构，这个虽然不完全正确，但是提到了计算机程序最重要的两点，那就是算法和数据结构。算法和数据结构永远是紧密联系在一起的，算法可以理解为解决问题的思想，这是程序中最具有创造性的部分，也是一个程序有别于另一个程序的关键点，而数据结构就是这种思想的载体。

再次重申一遍，我和大多数人一样，并不是要求每个程序员都精通各种算法。大多数程序员可能在整个职业生涯中都不会遇到像 ACM（Association for Computing Machinery）组织的国际大学生程序设计竞赛中的问题，但是说用不到数据结构和算法则是不可想象的。说数据结构和算法没用的人是因为他们用不到，用不到的原因是他们想不到，而想不到的原因是他们不会。请息怒，我不是要打击任何人，很多情况下确实是因为不会，所以才用不到，下面就是一个典型的例子。

1.2.1 一个队列引发的惨案

我所在的团队负责一款光接入网产品的“EPON 业务管理模块”的开发和维护工作，这是电信级的网络设备，因此对各方面性能的要求都非常高。有一天，一个负责集成测试的小伙儿跑过来对我说，今天的每日构造版本出现异常，所有线卡（承载数据业务的板卡）的上线时间比昨天的版本慢了 4 分钟左右。我很惊讶，对于一个电信级网络设备来说，每次加电后的线卡上线时间就是业务恢复时间，业务恢复时间这么慢是不能接受的。于是我检查了一下前一天的代码入库记录，很快就找到了问题所在。原来当前版本的任务列表中有这样一项功能，那就是记录线卡的数据变更日志，需求的描述是在线卡上维护一个日志缓冲区，每当有用户操作造成数据变更时，就记录一条变更信息，线卡上线时的批量数据同步也属于操作数据变更，也要计入日志。因为是嵌入式设备，线卡上日志缓冲区的大小受限制，最多只能存储 1000 条记录，当记录的日志超过 1000 条时，新增的日志记录将覆盖旧的记录，也就是说，这个日志缓冲区只保留最近写入的 1000 条记录。一个新来的小伙儿接受了这个任务，并在前一天下班前将代码签入库中（程序员要记住啊，

一定不要在临下班前签入代码）。他的实现方案大致是这样的（注释是我加上的）：

```
#define SYNC_LOG_CNT            1000
#define SYNC_LOG_MEMOVER_CNT    50

typedef struct
{
    INT32U logCnt;
    EPON_SYNC_LOG_DATA syncLogs[SYNC_LOG_CNT];
}EPON_SYNC_LOG;

EPON_SYNC_LOG s_EponSyncLog;

void Epon_Sync_Log_Add(EPON_SYNC_LOG_DATA*pLogData)
{
    INT32U i = 0;
    INT32U syncLogCnt = 0;

    syncLogCnt = s_EponSyncLog.logCnt;
    if(syncLogCnt>=SYNC_LOG_CNT)
    {
        /*缓冲区已满，向前移动 950 条记录，为新纪录腾出 50 条记录的空间*/
        memmove(s_EponSyncLog.syncLogs,
                s_EponSyncLog.syncLogs + SYNC_LOG_MEMOVER_CNT,
                (SYNC_LOG_CNT-SYNC_LOG_MEMOVER_CNT) * sizeof(EPON_SYNC_LOG_DATA));
        /*清空新腾出来的空间*/
        memset(s_EponSyncLog.syncLogs + (SYNC_LOG_CNT - SYNC_LOG_MEMOVER_CNT),
               0, SYNC_LOG_MEMOVER_CNT * sizeof(EPON_SYNC_LOG_DATA));
        /*写入当前一条日志*/
        memmove(s_EponSyncLog.syncLogs + (SYNC_LOG_CNT - SYNC_LOG_MEMOVER_CNT),
                pLogData, sizeof(EPON_SYNC_LOG_DATA));
        s_EponSyncLog.logCnt = SYNC_LOG_CNT - SYNC_LOG_MEMOVER_CNT + 1;

        return;
    }
        /*如果缓冲区有空间，则直接写入当前一条记录*/
    memmove(s_EponSyncLog.syncLogs + syncLogCnt,
            pLogData, sizeof(EPON_SYNC_LOG_DATA));
    s_EponSyncLog.logCnt++;
}
```

这个方案使用一个长度为 1000 条记录的数组存储日志，用一个计数器记录当前写入的有效日志条数，数据结构的设计中规中矩，但是当缓冲区满，需要覆盖旧记录时遇到了麻烦，因为每次都要移动数组中的前 999 条记录，才能为新记录腾出空间，这将使 Epon_Sync_Log_Add()函数的性能急剧恶化。考虑到这一点，小伙儿为他的方案设计了一个阈值，就是 SYNC_LOG_MEMOVER_CNT 常量定义的 50。当缓冲区满的时候，就一次性向前移动 950 条记录，腾出 50 条记录的空间，避免了每新增一条记录就要移动全部数据的情况。可见这个小伙儿还是动了一番脑子的，在 Epon_Sync_Log_Add()函数调用不是很频繁的情况下，在功能和性能之间做了个折中，根据自测的情况，他觉得还可以，于是就在下班前匆匆签入代码，没有来得及安排代码走查和同行评审。但是他没有考虑到线卡上线时需要批量同步数据的情况，在这种情况下，Epon_Sync_Log_Add()函数

被调用的频度仍然超出了这个阈值所能容忍的程度。通过对任务的性能进行分析，我们发现大量的时间都花费在 Epon_Sync_Log_Add() 函数中移动记录的操作上，即便是设计了阈值 SYNC_LOG_MEMOVER_CNT，性能依然很差。

其实，类似这样的固定长度缓冲区的读写，环形队列通常是最好的选择。下面我们来看一下环形队列的示意图，如图 1-1 所示。

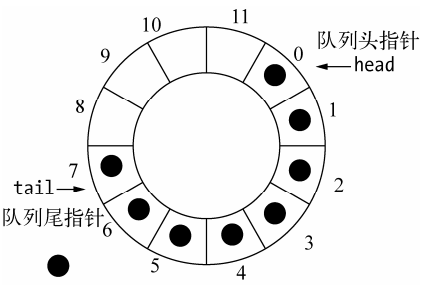

图 1-1 环形队列示意图

计算机内存中没有环形结构，因此环形队列都是用线性表来实现的，当数据指针到达线性表的尾部时，就将它转到 0 位置重新开始。实际编程的时候，也不需要每次都判断数据指针是否到达线性表的尾部，通常用取模运算对此做一致性处理。设模拟环形队列的线性表的长度是 N，队头指针为 head，队尾指针为 tail，则每增加一条记录，就可用以下方法计算新的队尾指针：

```
tail = (tail + 1) % N
```

对于本例的功能需求，当 tail + 1 等于 head 的时候，说明队列已满，此时只需将 head 指针向前移动一位，就可以在 tail 位置写入新的记录。使用环形队列，可以避免移动记录操作，本节开始时提到的性能问题就迎刃而解了。在这里，套用一句广告词：“没有做不到，只有想不到。”看看，我没说错吧？

1.2.2 我的第一个算法

我的第一份工作是为一个光栅图像矢量化软件编写一个图像预处理系统，这套光栅图像矢量化软件能够将从纸质工程图纸扫描得到的位图图纸识别成能被各种 CAD 软件处理的矢量化图形文件。在预处理系统中有一个功能是对已经二值化的光栅位图（黑白两色位图）进行污点消除。光栅位图上的污点可能是原始图纸上扫描前就存在的墨点，也可能是扫描仪引入的噪点，这些污点会对矢量化识别过程产生影响，会识别出错误的图形和符号，因此需要预先消除这些污点。

当时我不知道有小波算法，也不知道还有各种图像滤波算法，只是根据对问题的认识，给出了我的解决方案。首先我观察图纸文件，像直线、圆和弧线这样有意义的图形都是最少有 5 个点相互连在一起构成的，而污点一般都不会超过 5 个点连在一起（较大的污点都用其他的方法除掉了）。因此我给出了污点的定义：如果一个点周围与之相连的点的总数小于 5，则这几个相连在一起的点就是一个污点。根据这个定义，我给出了我的算法：从位图的第一个点开始搜索，如果这个点是 1（1 表示黑色，是图纸上的点；0 表示白色，是图纸背景颜色），就将相连点计数器加 1，然后继续向这个点相连的 8 个方向分别搜索，如果某个方向上的相邻点是 0 就停止这个方向的搜索。如果搜索到的相连点超过 4 个，说明这个点是某个图形上的点，就退出这个点的搜索。如果搜索完成后得到的相连的点小于或等于 4 个，就说明这个点是一个污点，需要将其颜色置为

0（清除污点）。

算法实现首先定义搜索过程中存储相连点信息的数据结构，这个数据结构定义如下：

```
typedef struct tagRESULT
{
    POINT pts[MAX_DIRTY_POINT];/*记录搜索过的前 5 个点的位置*/
    int count;
}RESULT;
```

这个数据结构有两个属性，count是搜索过程中发现的相连点的个数，pts是记录这些相连点位置的线性表。记录这些点的位置是为了在搜索结束后，如果判定这些点是污点，可以利用这些记录的位置信息直接清除这些点的颜色。

```
/*8 个方向*/
POINT dir[] = { {-1, 0}, {-1, -1}, {0, -1}, {1, -1}, {1, 0}, {1, 1}, {0, 1}, {-1, 1} };

void SearchDirty(char bmp[MAX_BMP_WIDTH][MAX_BMP_HEIGHT]
                 int x, int y, RESULT *result)
{
    for(int i = 0; i < sizeof(dir)/sizeof(dir[0]); i++)
    {
        int nx = x + dir[i].x;
        int ny = y + dir[i].y;
        if( (nx >= 0 && nx < MAX_BMP_WIDTH)
            && (ny >= 0 && nx < MAX_BMP_HEIGHT)
            && (bmp[nx][ny] == 1) )
        {
            if(result->count < MAX_DIRTY_POINT)
            {
                /*记录前 MAX_DIRTY_POINT 个点的位置*/
                result->pts[result->count].x = nx;
                result->pts[result->count].x = ny;
            }
            result->count++;
            if(result->count > MAX_DIRTY_POINT)
                break;

            SearchDirty(bmp, nx, ny, result);
        }
    }
}
```

向 8 个方向搜索使用了预置的矢量数组 dir，这是迷宫或棋盘类游戏搜索惯用的模式，本书介绍的算法会多次使用这种模式。SearchDirty()函数递归地调用自己，实现对 8 个方向的连通性搜索，最后的结果存在 result 中，如果 count 的个数大于 4，说明[x, y]位置的点是正常图形上的点，如果 count 的个数小于或等于 4，则说明[x, y]位置相邻的这个点是一个污点。污点相邻的点的位置都被记录在 pts 中，将这些位置的位图数据置 0 就消除了污点。算法没有做任何优化，不过好在图纸上大部分都是白色背景，需要搜索的点并不多。打开测试图纸一试，速度并不慢，效果也很好，几个故意点上去做测试用的污点都没有了，小的噪点也没有了，图纸一下就变白了。

不过这段代码最终并没有成为那个软件的一部分，学过机械制图的同学可能看出来了，这个算法会将一些细小的虚线和点划线一并干掉。

这是一个微不足道的问题，但却是我第一次为解决（当然，未遂）问题而设计了一个算法，并最终用程序将其实现。它让我领悟到了一个道理，软件被编写出来就是为了解决问题的，程序员的任务就是设计解决这些问题的算法。成功固然高兴，失败也没有什么代价，可以随时卷土重来。不要小看这些事情，不要以为只有各种专业领域的程序才会用到算法，每一个微小的设计都是算法创造性的体现，即使失败，也比放弃强。

1.3 算法的乐趣在哪里

算法有很多种存在形式，编写计算机程序只是其中一种，是程序员惯用的方式，本书要介绍的内容就是如何以计算机程序的方式研究算法。1.2 节介绍的两个例子都是我亲身经历过的事情，程序员在大部分时间里都是处理一些平凡而琐碎的程序，但有时候也需要做一些创造性的工作。记住，程序员就是计算机的“上帝”，计算机能解决问题是因为它的“上帝”告诉它怎么做。那么，当问题来临的时候，“上帝”是到各种论坛上发帖子求代码，还是自己解决问题？

如果要自己解决问题，应该如何解决问题？为什么要自己解决问题？先来回答第一个问题——如何设计算法解决问题？人类解决问题的方式是当遇到一个问题时，首先从大脑中搜索已有的知识和经验，寻找它们之间具有关联的地方，将一个未知问题做适当的转换，转化成一个或多个已知问题进行求解，最后综合起来得到原始问题的解决方案。编写计算机程序实现算法，让计算机帮我们解决问题的过程也不例外，也需要一定的知识和经验。为了让计算机帮我们解决问题，就要设计计算机能理解的算法程序。而设计算法程序的第一步就是要让计算机理解问题是什么。这就需要建立现实问题的数学模型。建模过程就是一个对现实问题的抽象过程，运用逻辑思维能力，抓住问题的主要因素，忽略次要因素。建立数学模型之后，第二个要考虑的问题就是输入输出问题，输入就是将自然语言或人类能够理解的其他表达方式描述的问题转换为数学模型中的数据，输出就是将数学模型中表达的运算结果转换成自然语言或人类能够理解的其他表达方式。最后就是算法的设计，其实就是设计一套对数学模型中的数据的操作和转换步骤，使其能演化出最终的结果。

数学模型、输入输出方法和算法步骤是编写计算机算法程序的三大关键因素。对于非常复杂的问题，建立数学模型是非常难的事情，比如天文物理学家研究的“宇宙大爆炸”模型，再比如热力学研究的复杂几何体冷却模型，等等。不过，这不是本书探讨的范围，程序员遇到的问题更多的不是这种复杂的理论问题，而是软件开发过程中常用和常见的问题，这些问题简单，但并不枯燥乏味。对于简单的计算机算法而言，建立数学模型实际上就是设计合适的数据结构的问题。这又引出了前面提到的话题，数据结构在算法设计过程中扮演着非常重要的角色。输入输出方式和算法步骤设计都是基于相应的数据结构设计的，相应的数据结构要能很方便地将原始问题转换成数据结构中的各个属性，也要能很方便地将数据结构中的结果以人们能够理解的方式输出，同

时，也要为算法转换过程中各个步骤的演化提供最便利的支持。使用线性表还是关联结构，使用树还是图，都是在设计输入输出和算法步骤时就要考虑的问题。

为什么要自己解决问题？爱因斯坦说过："兴趣是最好的老师。"这就是说，只要一个人对某事物产生兴趣，就会主动去学习、去研究，并在学习和研究的过程中产生愉快的情绪。我把从算法中体会到的乐趣分成三个层次：初级层次是找到特定的算法解决特定的实际问题，这种乐趣是解决问题后的成就感；中级层次是有些算法本身就是充满乐趣的，搞明白这种算法的原理并写出算法的程序代码，能为自己今后的工作带来便利；高级层次是自己设计算法解决问题，让其他人可以利用你的算法享受到初级层次的乐趣。有时候问题可能是别人没有遇到过的，没有已知的解法，这种情况下只能自己解决问题。这是本书一直强调算法的乐趣的原因。只有体会到乐趣，才有动力去学习和研究，而这种学习和研究的结果是为自己带来正向的激励，为今后的工作带来便利。回想一下 1.2.1 节的例子，环形队列相关的算法是固定长度缓冲区读写的常用模式，如果知道这一点，就不会有这种问题了。

1.4　算法与代码

本书讲到的算法都是以计算机程序作为载体展示的，其基本形式就是程序代码。作为一个软件开发人员，你希望看到什么样的代码？是这样的代码：

```
double kg = gScale * 102.1 + 55.3;
NotifyModule1(kk);
double kl1 = kg / l_mask;
NotifyModule2(kl1);
double kl2 = kg * 1.25 / l_mask;
NotifyModule2(kl2);
```

还是这样的代码：

```
double globalKerp = GetGlobalKerp();
NotifyGlobalModule(globalKerp);
double localKrep = globalKerp / localMask;
NotifyLocalModule(localKrep);
double localKrepBoost = globalKerp * 1.25 / localMask;
NotifyLocalModule(localKrepBoost);
```

程序员都有一种直觉，那就是能看懂的代码就是好代码。但是"能看懂"是一个非常主观的感觉，同样的代码给不同的人看，能否看懂有着天壤之别。《重构》一书的作者为不好的代码总结了 21 条"坏味道"规律，希望能够对号入座地判断一下代码中的"坏代码"。但是这 21 条规律仍然太主观，于是人们又给代码制定了很多量化指标，比如代码注释率（这个指标因为没有意义，已经被很多组织抛弃了）、平均源代码文件长度、平均函数长度、平均代码依赖度、代码嵌套深度、测试用例覆盖度，等等。做这些工作的目的在于人们希望看到漂亮的代码，这不仅仅是主观审美的需要，更是客观上对软件质量的不懈追求。漂亮的代码有助于改善软件的质量，这已经是公认的事实，因为程序员在把他们的代码变得漂亮的过程中，能够通过一些细小却又重要的方式改善代码的质量，这些细小却又重要的方式包括但不限于更好的设计、可测试性和可维护性

等方面的方法。

在保证软件行为正确性的基础上，人们都用什么词来形容好的代码呢？好看、漂亮、整洁、优雅、艺术品、像诗一样？我看过很多软件的代码，有开源软件的代码，也有商业软件的代码，好的代码给我的感觉就是以上这些形容词，当然也见过不好的代码，给我的感觉就是“一堆代码”而已。我在写“算法系列”博客专栏的时候，就特别注意这一点，即便别人已经发布过类似的算法实现，我也希望我的算法呈现出来的是完全不一样的代码。设计算法也和设计软件一样，应该是漂亮的代码，如果几百行代码堆在一起，不分主次，关系凌乱，只是最后堆出了一个正确的结果，这不是我所希望的代码，即虐人又虐己。大部分人来看你的博客，应该还是为了看懂吧。在我准备这本书的时候，我把很多算法又重新写了一遍，不仅算法有趣，研究代码也是一种乐趣。如果算法本身很有趣，但是最后的代码实现却是毫无美感的“一堆代码”，那真是太扫兴了。

1.5 总结

本章借用了多部知名著作中对算法的定义，只是想让大家对算法有一个“宽容”一点的理解。通过我亲身经历的两个例子，说明了程序员与算法之间“剪不断，理还乱”的关系。除此之外，还简单探讨了算法乐趣的来源、算法和代码的关系，以及研究代码本身的乐趣等内容。

如果你认同我的观点，就可以继续阅读本书了。本书的每一章都是独立的，没有前后关系，你可以根据自己的喜好直接阅读相关的章节。希望本书能使你有所收获，并体会到算法的乐趣。

1.6 参考资料

[1] Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[2] Knuth D E. *The Art of Computer Programming* (*Third Edition*), Vol 1. Addison-Wesley, 1997

[3] Weiss M A. *Data Structures and Algorithm Analysis* (*Second Edition*). Addison-Wesley, 2001

[4] Oram A, Wilson G. *Beautiful Code*. O’Reilly Media, Inc, 2007

[5] Fowler M, et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999

第 7 章
稳定匹配与舞伴问题

每年凤凰花开、蝉鸣绿叶的季节，都是毕业的季节，也是同学们找工作的季节。很显然，学生和雇主之间从来都是双向选择的关系，然而学霸们往往先人一步，早早地就抓了一把 offer。无奈，即便是学霸也分身无术，最终只能选择一个 offer。毫无疑问，学霸们会根据自己的偏好对 offer 排队，选其中最好的一个。有时候我会想，其他也给了学霸 offer 的公司岂不是少了一个名额？显然我是多虑了，其实这些雇主公司也有一个偏好列表作为备用，如果空出了名额，他们会从这个备用的偏好队列中再选一个。但这总归不是一个最高效的资源配置方式，大量的撤销和重新选择会浪费很多社会资源。有没有一种方法，在双向选择公开透明的基础上，按照资源配置的最优原则给学生和雇主配对，直接得到一个学生和雇主之间的完备匹配或稳定匹配？

幸运的是，确实有人在研究稳定匹配问题（stable matching problem）。戴维·盖尔（David Gale）和劳埃德·沙普利（Lloyd Shapley）就是两位这样的专家，他们从 20 世纪 60 年代就开始研究这个问题。他们最早研究的问题是稳定婚姻问题（stable marriage problem），其实这适用于所有带偏爱或优先选择的双向选择问题。本章我们就以稳定婚姻问题为例，介绍一下盖尔和沙普利研究的稳定匹配算法（Gale-Shapley 算法）的原理，并给出一个解决舞伴匹配问题的 Gale-Shapley 算法实现。

7.1 稳定匹配问题

1962 年，盖尔和沙普利发表了一篇名为“大学招生与婚姻的稳定性”[1]的论文，首次提出了稳定婚姻问题，该问题后来成为研究稳定匹配的典型例子。在介绍稳定匹配问题之前，我们先来了解几个概念。

7.1.1 什么是稳定匹配

假设 n 个未婚男人的集合 $M=\{m_1,m_2,\cdots,m_n\}$ 和 n 个未婚女人的集合 $W=\{w_1,w_2,\cdots,w_n\}$，令 $M\times W$ 为所有可能的形如(m_i,w_i)的有序对的集合，其中 $m_i\in M$，$w_i\in W$。根据上述定义，我们给出匹配

的概念，匹配 S 是来自 $M \times W$ 的有序对的集合，并且具有以下性质：每个 M 的成员和每个 W 的成员至多出现在 S 的一个有序对中。接下来是完美匹配的概念，完美匹配 S' 是一个具有以下性质的匹配：M 的每个成员和 W 的每个成员恰好出现在 S' 的一个对里。S 和 S' 这两个定义的差别就是“至多”和“恰好”两个词，对很多人来说，区分这两个概念就像区分落基山大角羊和沙漠大角羊一样困难。我来解释一下，可以将 S 理解为 M 和 W 的成员配对结婚，但是 M 和 W 中不一定所有成员都能配对成功，还有剩余的男人和女人是单身。而完美匹配 S' 则是 S 的一种特殊情况，即 S' 是所有人都配对成功，不存在落单的男人和女人。

很显然，盖尔和沙普利研究的稳定婚姻问题是在一夫一妻制度下男人和女人的配对关系，每个男人最终都要和一个女人结婚。现在在完美匹配的背景下引入优先或偏好的概念，每个男人都按照个人喜好对所有女人排名，如果某个男人 m 给女人 w 的排名高于给 w' 的排名，就可以理解为 m 喜欢 w 胜过 w'。反过来也一样，每个女人也按照自己的喜好对所有的男人排名。以上排名必须区分先后顺序，不能有排名并列的情况出现。那么什么是稳定匹配呢？稳定匹配就是在引入优先排名的情况下，一个完美匹配 S 如果不存在不稳定因素，则称这个完美匹配是稳定匹配。什么是不稳定因素呢？假设在完美匹配 S 中存在两个配对（m, w）和（m', w'），但是从优先排名上看，m 更喜欢 w' 而不喜欢 w，同时 w' 也更喜欢 m 而不喜欢 m'，如图 7-1 所示。在这种情况下，我们称这个完美匹配 S 是不稳定的，像（m, w'）这样有“私奔”倾向的不稳定对（unstable pair）就是 S 的一个不稳定因素。

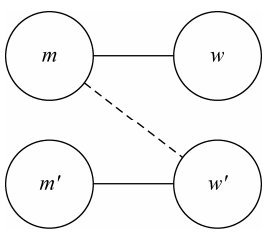

图 7-1 不稳定因素示意图

稳定匹配满足两个条件：首先，它是一个完美匹配；其次，它不含有任何不稳定因素。在给定的众多复杂关系中，如何求得一个稳定匹配？盖尔和沙普利在 1962 年提出的 Gale-Shapley 算法就是一种著名的稳定匹配算法，接下来我们就来简单介绍一下 Gale-Shapley 算法的原理。

7.1.2 Gale-Shapley算法原理

盖尔和沙普利的策略是一种寻找稳定婚姻的策略，不管男女之间有何种偏好，这种策略总可以得到一个稳定的婚姻匹配。先来看一下 Gale-Shapley 算法实现的伪代码：

```
初始化所有的 m∈ M，w∈ W，所有的 m 和 w 都是自由状态;
while (存在男人是自由的，并且他还没有对每个女人都求过婚)
{
    选择一个这样的男人 m;
    w = m 的优先选择表中还没有求过婚的排名最高的女人;
    if (w 是自由状态)
    {
        将(m, w)的状态设置为约会状态;
    }
    else /*w 已经和其他男人约会了*/
    {
        m' = w 当前约会的男人;
        if (w 更喜爱 m' 而不是 m)
```

```
            {
                m 保持单身状态（w 不更换约会对象）;
            }
            else /*w 更喜爱 m 而不是 m'*/
            {
                将(m, w)的状态设置为约会状态;
                将 m' 设置为自由状态;
            }
        }
    }
    输出已经匹配的集合 S;
```

看起来总是男人主动选择，女人被动接受，事实上这个算法并没有做这个假设。基于男女平等的原则，也可以是女人主动选择，男人被动接受，这就是这个算法常被提到的两个策略，即“男士优先”还是“女士优先”。

从 Gale-Shapley 算法的策略来看，男人们一轮一轮地选择自己中意的女人，女人则可以选择接受追求者，或拒绝追求者。只要女人是单身的自由状态，男人的追求就不会被拒绝，但这并不表示男人总是能选到自己最中意的女人，因为女人是可以毁约的。男人被拒绝的情况有两种，一种情况是男人追求的女人已经有约会对象，并且女人喜欢自己的约会对象胜过当前追求她的男人；另一种情况是女人面对另一个男人的追求时，如果她喜欢这个追求她的男人胜过自己当前的约会对象，女人会利用毁约的权利拒绝当前约会对象。男人每被拒绝一次，就只能从自己的优先选择表中选择下一个女人。男人不能重复尝试约会那些已经拒绝过他的女人，因此这种选择总是无奈地向越来越不中意的方向发展。每一轮选择之后都会有一些男人或女人脱离单身的自由状态，当某一轮过后没有任何一个男人或女人是单身状态时，这个算法就结束了。在 Gale-Shapley 算法中，n 个男人共需要进行 n 轮选择，每一个男人需要向 n 个中意对象求婚，因此，算法最多需要 $n*n$ 轮循环就可以结束。

这个算法的流程非常简单，但是是否有效呢？也就是说 Gale-Shapley 算法结束后得到的一个匹配一定是稳定匹配吗？还记得上一节介绍的稳定匹配的两个条件吗？稳定匹配首先是完美匹配，其次是要求没有不稳定因素。下面我们就从这两方面分别证明一下这个算法的结果是否是稳定匹配。

首先，我们要证明 Gale-Shapley 算法结束得到的是一个完美匹配。直接证明这个问题比较困难，所以我们采用反证法。假设算法结束后有一个男人 m 还是单身，因为规则是一个男人只能和一个女人约会，这就意味着必定有一个女人 w 也是单身。根据算法规则，女人只要是单身，一定会接受男人的求婚，现在 w 是单身，说明 w 没有收到任何求婚请求。这时就出现矛盾了，因为根据算法流程，m 肯定是向包括 w 在内的所有女人都求过婚的，所以假设应该是不成立的，也就是说，能够证明 Gale-Shapley 算法得到的是一个完美匹配。

接下来证明 Gale-Shapley 算法的结果没有任何不稳定因素，仍然采用反证法。假设匹配结果中存在不稳定因素，也就是说，存在 m 和 w，他们各自都已经有了伴侣，但是 m 喜欢 w 胜过喜欢自己现在的伴侣，同样，w 也喜欢 m 胜过喜欢自己现在的伴侣。但是根据算法规则，m 肯

定是向 w 求过婚的，如果 w 更喜欢 m，w 应该选择 m 而不是当前的伴侣，因此这个假设也是不成立的。

由以上证明可知，Gale-Shapley 算法的结果是一个稳定匹配，也就证明了 Gale-Shapley 算法的正确性。

7.2 Gale-Shapley 算法的应用实例

本节利用舞伴问题介绍一个 Gale-Shapley 算法的应用实例。舞伴问题是这样的：有 n 个男孩与 n 个女孩参加舞会，每个男孩和女孩均交给主持一个名单，写上他（她）中意的舞伴名字。无论男孩还是女孩，提交给主持人的名单都是按照偏爱程度排序的，排在前面的都是他们最中意的舞伴。试问主持人在收到名单后，是否可以将他们分成 n 对，使每个人都能和他们中意的舞伴结对跳舞？为了避免舞会上出现不和谐的情况，要求这些舞伴的关系是稳定的。

假如有两对分好的舞伴：（男孩 A，女孩 B）和（男孩 B，女孩 A），但是男孩 A 更偏爱女孩 A，女孩 A 也更偏爱男孩 A，同样，女孩 B 更偏爱男孩 B，而男孩 B 也更偏爱女爱 B。在这种情况下，这两对舞伴就倾向于分开，然后重新组合，这就是不稳定因素。很显然，这个问题需要的是一个稳定匹配的结果，适合使用 Gale-Shapley 算法。

7.2.1 算法实现

首先定义舞伴的数据结构，根据题意，一个舞伴至少要包含两个属性，就是每个人的偏爱舞伴列表和他（她）们当前选择的舞伴。根据 Gale-Shapley 算法的规则，还需要有一个属性表示下一次要向哪个偏爱舞伴提出跳舞要求。当然，这个属性并不是男生和女生同时需要的，当使用“男士优先”策略时，男生需要这个属性，当使用“女士优先”策略时，女生需要这个属性。为了使程序输出更有趣味，需要为每个角色提供一个名字。综上所述，舞伴的数据结构定义如下：

```
typedef struct tagPartner
{
    char *name;   //名字
    int next;     //下一个邀请对象
    int current;  //当前舞伴，-1 表示还没有舞伴
    int pCount;   //偏爱列表中舞伴个数
    int perfect[UNIT_COUNT]; //偏爱列表
}PARTNER;
```

UNIT_COUNT 是男孩或女孩的数量（稳定匹配问题总是假设男孩和女孩的数量相等），pCount 是偏爱列表中的舞伴个数。根据标准的“稳定婚姻问题”的要求，pCount 的值应该是和 UNIT_COUNT 一致的，但是某些情况下（比如一些算法比赛题目的特殊要求）也会要求伙伴们提供的偏爱列表可长可短，因此我们增加了这个属性。但是有一点需要注意，如果允许舞伴的 pCount 小于 UNIT_COUNT，则 7.1.2 节的证明就不适用了，需要设置相应的条件并使用更复杂的证明方法。关键是，最后不一定能得到稳定匹配的结果。这里给出的实现算法使用数组来存储参加舞会的男孩和

女孩列表，因此这个数据结构中的 next、current 和 perfect 列表中存放的都是数组索引，了解这一点有助于理解算法的实现代码。

Gale-Shapley 算法的实现非常简单，将 7.1.2 节给出算法伪代码翻译成编程语言即可。完整的算法代码如下：

```
bool Gale_Shapley(PARTNER *boys, PARTNER *girls, int count)
{
    int bid = FindFreePartner(boys, count);
    while(bid >= 0)
    {
        int gid = boys[bid].perfect[boys[bid].next];
        if(girls[gid].current == -1)
        {
            boys[bid].current = gid;
            girls[gid].current = bid;
        }
        else
        {
            int bpid = girls[gid].current;
            //女孩喜欢 bid 胜过其当前舞伴 bpid
            if(GetPerfectPosition(&girls[gid], bpid) > GetPerfectPosition(&girls[gid], bid))
            {
                boys[bpid].current = -1; //当前舞伴恢复自由身
                boys[bid].current = gid; //结交新舞伴
                girls[gid].current = bid;
            }
        }
        boys[bid].next++; //无论是否配对成功，对同一个女孩只邀请一次
        bid = FindFreePartner(boys, count);
    }

    return IsAllPartnerMatch(boys, count);
}
```

FindFreePartner()函数负责从男孩列表中找一个还没有舞伴、并且偏好列表中还有没有邀请过的女孩的男孩，返回男孩在列表（数组）中的索引。如果返回值等于–1，表示没有符合条件的男孩了，于是主循环停止，算法就结束了。GetPerfectPosition()函数用于判断女孩喜欢一个舞伴的程度，通过返回舞伴在自己的偏爱列表中的位置来判断，位置越靠前，也就是 GetPerfectPosition()函数的返回值越小，说明女孩越喜欢这个舞伴。GetPerfectPosition()函数的实现代码如下：

```
int GetPerfectPosition(PARTNER *partner, int id)
{
    for(int i = 0; i < partner->pCount; i++)
    {
        if(partner->perfect[i] == id)
        {
            return i;
        }
    }
```

```
    //返回一个非常大的值，意味着根本排不上对
    return 0x7FFFFFFF;
}
```

按照“稳定婚姻问题”的要求，这个函数应该总是能够得到 ID 指定的异性舞伴在 partner 的偏爱列表中的位置，因为每个 partner 的偏爱列表包含所有异性舞伴。但是当题目有特殊需求时，partner 的偏爱列表可能只有部分异性舞伴。比如 partner 非常恨一个人，他们绝对不能成为舞伴，那么 partner 的偏爱列表肯定不会包含这个人。考虑到算法的通用性，GetPerfectPosition() 函数默认返回一个非常大的数，返回这个数这意味着 ID 指定的异性舞伴在 partner 的偏爱列表中根本没有位置（非常恨），根据算法的规则，partner 最不喜欢的异性舞伴的位置都比 id 指定的异性舞伴位置靠前。这也是算法一致性处理的一个技巧，GetPerfectPosition()函数当然可以设计成返回–1 表示 ID 指定的异性舞伴不在 partner 的偏爱列表中，但是大家想一想，算法中是不是要对这个返回值做特殊处理？原来代码中判断位置关系的一行代码处理：

```
if(GetPerfectPosition(&girls[gid], bpid) > GetPerfectPosition(&girls[gid], bid))
```

就会变得非常繁琐，让我们看看会是什么情况：

```
if((GetPerfectPosition(&girls[gid], bpid) == -1)
    && (GetPerfectPosition(&girls[gid], bid) == -1))
{
    //当前舞伴 bpid 和 bid 都不在女孩的喜欢列表中，太糟糕了
    ...
}
else if(GetPerfectPosition(&girls[gid], bpid) == -1)
{
    //当前舞伴 bpid 不在女孩的喜欢列表中，bid 有机会
    ...
}
else if(GetPerfectPosition(&girls[gid], bid) == -1)
{
    //bid 不在女孩的喜欢列表中，当前舞伴 bpid 维持原状
    ...
}
else if(GetPerfectPosition(&girls[gid], bpid) > GetPerfectPosition(&girls[gid], bid))
{
    //女孩喜欢 bid 胜过其当前舞伴 bpid
    ...
}
else
{
    //女孩喜欢当前舞伴 bpid 胜过 bid
    ...
}
```

这是我最不喜欢的代码逻辑，真的，太糟糕了。可见，这个小小的技巧为代码的逻辑处理带来了极大的好处。类似的技巧被广泛应用，在排序算法中经常使用“哨兵”位，避免每次都要判断是否比较完全部元素。面向对象技术中常用的“Dummy Object”技术，也是类似的思想。

Gale-Shapley 算法原来如此简单，你是不是为沙普利能获得诺贝尔奖愤愤不平？其实不然，算法原理的简单并不等于其解决的问题也简单，本书介绍的很多算法都是如此，小算法解决大问题。

7.2.2　改进优化：空间换时间

Gale_Shapley()函数给出的算法还有点问题，主要是 GetPerfectPosition()函数的策略，这个函数每次都要遍历 partner 的偏爱舞伴列表才能确定 bid 的位置，很可能导致理论上时间复杂度为 $O(n^2)$的算法在实际实现时变成 $O(n^3)$的时间复杂度。为了避免算法在多轮选择过程中频繁遍历每个 partner 的偏爱舞伴列表，需要对 partner 到底更偏爱哪个舞伴的判断策略进行改进。

改进的原则就是“以空间换时间”。简单来讲，以空间换时间的方法就是用一张事先初始化好的表存储这些位置关系，在使用个过程中，以 $O(1)$时间复杂度的方式直接查表确定偏爱舞伴的关系。这样的表可以是线性表，也可以是哈希表这样的映射表。对于这个问题，我们选择使用二维表来存储这些位置关系。假设存在二维表 priority[n][n]，我们用 priority[w][m]表示 m 在 w 的偏爱列表中的位置，这个值越小，表示 m 在 w 的偏爱列表中的位置越靠前。在算法开始之前，首先初始化这个关系表：

```
for(int w = 0; w < UNIT_COUNT; w++)
{
    //初始化成最大值，原理同上
    for(int j = 0; j < UNIT_COUNT; j++)
    {
        priority[w][j] = 0x7FFFFFFF;
    }
    //给偏爱舞伴指定位置关系
    int pos = 0;
    for(int m = 0; m < girls[w].pCount; m++)
    {
        priority[w][girls[w].perfect[m]] = pos++;
    }
}
```

最后，将对 GetPerfectPosition()函数的调用替换成查表：

```
if(priority[gid][bpid] > priority[gid][bid])
```

对于一些在算法执行过程中不会发生变化的静态数据，如果算法执行过程中需要反复读取这些数据，并且读取操作存在一定时间开销的场合，比较适合使用这种“以空间换时间”的策略。用合理的方式组织这些数据，使得数据能够在 $O(1)$时间复杂度内实现是这种策略的关键。对本问题应用“以空间换时间”的策略，需要在算法开始的准备阶段初始化好 priority 二维表，这需要一些额外的开销，但是相对于 n^2次查询节省的时间来说，这点开销是能够容忍的。

“以空间换时间”也是算法设计常用的技巧，在很多算法中都有应用。比如本书第 15 章介绍的快速傅里叶变换算法，经过蝶形变换后每个点的数据位置都发生了变化，需要将这些点的位置

还原。可以利用一个二重循环将这些错位的数据还原，也可以利用蝶形变换的位置变换关系表，采用查表的方式将两个错位的数据交换位置，后者采用的就是“以空间换时间”的策略。

7.3 有多少稳定匹配

当参加舞会的男孩和女孩按照一定的顺序排好队，位置固定之后，使用 Gale-Shapley 算法能够得到一个确定的稳定匹配结果。但是对这群男孩和女孩来说，稳定匹配的结果肯定不是唯一的，其实只要将计算策略从“男士优先”转换成“女士优先”，就可以得到另外一个完全不同的稳定匹配结果。同样，调整一下男孩们的位置顺序，比如让最后一个男孩排在第一的位置，让他第一个邀请女孩，则 Gale-Shapley 算法也可以得到一个完全不同的稳定匹配结果。

很显然，对于任意情况下的 n 个男孩和 n 个女孩来说，肯定有多个稳定匹配，那么，到底有多少个稳定匹配？稳定匹配首先必须是完美匹配，而且稳定匹配的个数小于或等于完美匹配，所以，我们可以先从理论上计算一下完美匹配的数量，估算一下问题的规模，然后再决定是否能用算法找出全部的稳定匹配。从理论上分析，只要每个人的偏爱列表都包含全部异性舞伴，那么完美匹配的个数就可以通过公式计算出来。首先，假设男孩们已经排好了队，准备按照顺序邀请女孩跳舞，在不考虑稳定匹配的情况下，每个男孩选择一个女孩之后，还没有舞伴的女孩的总数就减 1，剩下的男生的可选范围就变小了。第一个男孩选择的可能情况是 C_n^1，第二个男孩可能的选择就只有 C_{n-1}^1 种。以此类推，可以计算出完美匹配的可能情况是 $M = C_n^1 C_{n-1}^1 C_{n-2}^1 \dots C_1^1 = n!$种。如果仅仅从排列组合问题的角度考虑舞伴问题，随着男孩们的顺序变化，这个数字会成倍增加。那么男孩们有多少种顺序变化呢？n 个男孩全排列，结果也是 $n!$(P_n^n)种变化，因此最终的结果应该是$(n!)^2$。但是舞伴问题并不是单纯的排列组合问题，因为这些男孩和女孩之间通过各自的偏爱列表建立了某种联系，这使得一些组合结果实际上是没有意义的重复。举个例子说明一下，假如 m 在第一轮选择，他选择 w 作为舞伴，m'在第二轮选择，他选择 w'作为舞伴。现在转换一下选择顺序，改为 m'在第一轮选择，但是 m'的偏爱列表中，w'排在前面，于是 m'仍然选择 w'作为舞伴，m 也只能选择 w 作为舞伴，虽然选择的顺序变了，但是结果和前一次一样。

由此看来，虽然对男孩的选择顺序进行全排列有 $n!$种可能，但是这 $n!$种选择顺序最终得到的匹配结果都只是 $n!$种结果的重复出现，实际的完美匹配只有 $n!$种。接下来我们要给出的穷举算法也验证了这一点，对于 3 个男孩和 3 个女孩的情况，穷举算法得到了 36（3!×3!=36）个完美匹配结果，排除掉重复结果后得到 6（3×2×1=6）个结果。对于 4 个男孩和 4 个女孩的情况，穷举算法得到了 576（4!×4!=576）个完美匹配结果，排除掉重复结果后得到 24（4×3×2×1=24）个结果。

7.3.1 穷举所有的完美匹配

如果想知道到底有多少个稳定匹配，首先要知道有多少个完美匹配。具体的方法就是使用穷举的方法找到全部的完美匹配，然后根据条件将包含不稳定因素的完美匹配过滤掉，剩下的就是稳定匹配。遵循这个原则，我们先来研究一下穷举所有完美匹配的算法。

穷举算法的数据结构定义仍然沿用 7.2.1 节算法实现中使用的 PARTNER 定义，只是 next 属性用不上。穷举的方法就是每次为一个男孩选择一个舞伴，选择的方法就是从男孩的偏爱列表中找一个还没有舞伴的女孩，确定为这个男孩的舞伴，同时将男孩和女孩对应的 PARTNER 定义中的 current 属性指向对方。判断一个女孩是否已经有舞伴的方法就是判断她的 current 属性是否是 –1，如果不是–1 就说明这个女孩已经有舞伴了。按照男孩的顺序逐个为他们选择舞伴，当最后一个男孩也确定了舞伴之后，就得到了一个完美匹配，可以打印这个结果，用于检查是否正确。

SearchStableMatch()函数是搜索算法的核心，采用递归方式实现，每次为一个男孩选择舞伴。index 参数是男孩按照顺序的编号，从 0 开始编号，刚好对应 boys 数组的下标，简化了代码实现。当 index 等于 UNIT_COUNT（男孩的个数）时，表示已经为所有男孩找到了舞伴，如果算法没有错误，这应该就是一个完美匹配。算法的主体就是遍历 index 对应的男孩的偏爱列表，从列表中找到一个还没有舞伴并且也喜欢自己的女孩作为舞伴，互相设置 current 属性。需要注意的是，算法主体包含一个回溯处理，当某一级搜索结束后，要重置相关男孩和女孩的舞伴关系，以便后序的递归搜索能够正常进行。具体代码可看 SearchStableMatch()函数的 for 循环主体部分。

```
void SearchStableMatch(int index, PARTNER *boys, PARTNER *girls)
{
    if(index == UNIT_COUNT)
    {
        if(IsStableMatch(boys, girls))
        {
            PrintResult(boys, girls, UNIT_COUNT);
        }
        return;
    }

    for(int i = 0; i < boys[index].pCount; i++)
    {
        int gid = boys[index].perfect[i];

        if(!IsPartnerAssigned(&girls[gid]) && IsFavoritePartner(&girls[gid], index))
        {
            boys[index].current = gid;
            girls[gid].current = index;
            SearchStableMatch(index + 1, boys, girls);
            boys[index].current = -1;
            girls[gid].current = -1;
        }
    }
}
```

7.3.2　不稳定因素的判断算法

7.1.1 节给出了完美匹配中不稳定因素的定义，当一个男孩和一个女孩同时有比他们当前舞伴更“强烈的”愿望结为舞伴的时候，他们就倾向于与各自的舞伴分开，然后结合在一起成为舞伴。不稳定因素的判断算法就是在一个完美匹配中找出图 7-1 所示的情况，这种情况有两个特征：

首先，男孩的当前舞伴不是他的偏爱列表中排在第一位的女孩，也就是说，男孩更偏爱其他女孩胜过自己当前的舞伴；其次，男孩更偏爱的那个（或那几个女孩中的一个）女孩刚好也喜欢这个男孩胜过自己当前的舞伴。

于是，不稳定因素的判断算法就呼之欲出了，重点就是上述两个特征的识别。判断一个完美匹配是否是稳定匹配的算法流程如下。

(1) 找出这个男孩的当前舞伴在男孩的偏爱列表中的位置，如果当前舞伴排在偏爱列表的第一位，则表示这个男孩不存在不稳定因素的可能，转步骤(4)。如果当前舞伴不是男孩偏爱列表的第一位，则转到步骤(2)。

(2) 男孩的偏爱列表中如果还有排在当前舞伴之前但还没有进行判断处理的女孩，则转步骤(3)，否则转步骤(4)。

(3) 找到女孩的当前舞伴在女孩的偏爱列表中的位置和当前处理的男孩在女孩的偏爱列表中的位置，如果女孩当前舞伴的位置比当前处理的男孩的位置靠前，则表示对该女孩不存在不稳定因素，转步骤(2)。如果当前处理的男孩的位置比女孩当前舞伴的位置靠前，则表示存在不稳定因素，直接转步骤(6)。

(4) 如果对全部男孩判断完毕，转步骤(5)。否则，继续对下一个男孩进行不稳定因素判断，转步骤(1)。

(5) 结束，没有找到不稳定因素。

(6) 结束，找到不稳定因素，此完美匹配不是稳定匹配。

根据以上算法流程，我们给出判断稳定匹配的算法实现，如 IsStableMatch()函数所示，非常简单，相关的注释和以上算法流程的表述都能对上，此处就不再过多解释。


```
bool IsStableMatch(PARTNER *boys, PARTNER *girls)
{
    for(int i = 0; i < UNIT_COUNT; i++)
    {
        //找到男孩当前舞伴在自己的偏好列表中的位置
        int gpos = GetPerfectPosition(&boys[i], boys[i].current);
        //在 position 位置之前的舞伴，男孩喜欢她们胜过 current
        for(int k = 0; k < gpos; k++)
        {
            int gid = boys[i].perfect[k];
            //找到男孩在这个女孩的偏好列表中的位置
            int bpos = GetPerfectPosition(&girls[gid], i);
            //找到女孩的当前舞伴在这个女孩的偏好列表中的位置
            int cpos = GetPerfectPosition(&girls[gid], girls[gid].current);
            if(bpos < cpos)
            {
                //女孩也是喜欢这个男孩胜过喜欢自己当前的舞伴，这是不稳定因素
                return false;
            }
        }
    }
```

```
        return true;
    }
```

7.3.3　穷举的结果

至此，我们有了穷举法搜索全部稳定匹配结果的算法，来看看结果吧。假设有以下男孩和女孩的数据，冒号后是对应男孩和女孩的偏爱列表。

男孩

```
Albert: Laura, Nancy, Marcy
Brad: Marcy, Nancy, Laura
Chuck: Laura, Marcy, Nancy
```

女孩

```
Laura: Chuck, Albert, Brad
Marcy: Albert, Chuck, Brad
Nancy: Brad, Albert, Chuck
```

应用算法搜索后得到以下结果：

```
Albert[1] <---> Nancy[1]
Brad[0] <---> Marcy[2]
Chuck[0] <---> Laura[0]

Albert[2] <---> Marcy[0]
Brad[1] <---> Nancy[0]
Chuck[0] <---> Laura[0]

Total Matchs : 6, Stable Matchs : 2
```

看来，有两个稳定匹配的结果，用 7.2.1 节给出的 Gale-Shapley 算法得到的只是前一个稳定匹配的结果。参考资料[6]给出了一个有意思的结论，就是稳定匹配的个数总是 2 的整数幂，有兴趣的读者可阅读一下该资料，看看这个结论的来龙去脉。另外，这个资料还给出了只有一种稳定匹配结果的情况，即所有的女孩的偏爱列表都完全一样的时候，无论男孩们的偏爱列表如何选择，最终都只有一种稳定匹配结果，有兴趣的读者也可以自己研究研究。

7.4　二部图与二分匹配

之前讨论稳定匹配问题的时候，我们把完美匹配定义为每个男人和女人都属于匹配中的某个对，并不是很直观，现在我们准备用图的术语更一般地表达完美匹配的概念。首先介绍一下二部图，二部图 $G=(V,E)$是这样的一个图，它的顶点集合 V 可以划分为 X 和 Y 两个集合，它的边集合 E 中的每条边都有一个端点在 X 集合，另一个端点在 Y 集合。图 7-2 就是一个二部图。

现在给出针对二部图的匹配的定义，给定一个二部图 $G=(V,E)$的子图 M，如果 M 的边集中任意两条边都不依附于同一个顶点，则称 M 是一个匹配。简单地说，图 7-2 中 x_2、x_3、x_4 等点都有

多条边与之连接，也就是说有多个边依附于这些点，因此图 7-2 所示的二部图不是一个匹配。现在考虑删除一些边，最终得到如图 7-3 所示的一个 G 的子图。该子图中没有任何边同时连接 X 或 Y 中的同一个顶点，因此这是一个匹配。

如果 G 的一系列子图 $M_0,M_1,\cdots,M_n$ 都是匹配，那么包含边数最多的那个匹配就是图 G 的最大匹配。如果一个最大匹配中所有的点都有边与之相连，没有未覆盖点，则这个最大匹配就是完美匹配。未覆盖点的定义是：图 G 的一个顶点 V_i，如果 V_i 不与任何一条属于匹配 M 的边相连，则成 V_i 是一个未覆盖点。图 7-3 就是一个完美匹配。

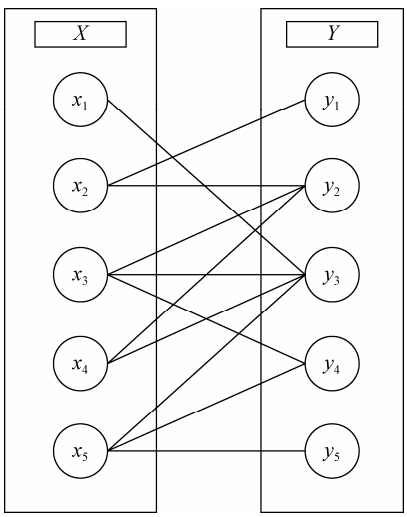

图 7-2 简单的二部图

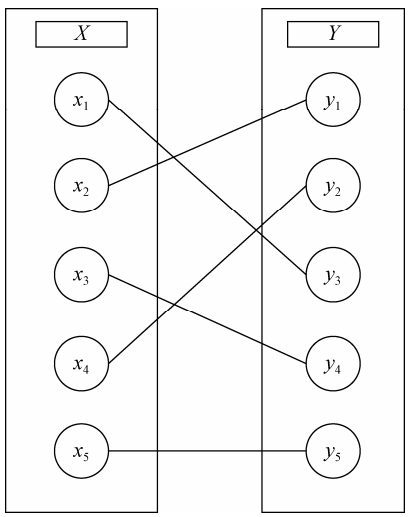

图 7-3 一个完美匹配的二部图

根据以上定义，如果 G 的一个匹配 M 是最大匹配，并且没有未覆盖点，则这个匹配就是完美匹配。可见，图 G 的匹配和完美匹配正好就是之前介绍的“稳定婚姻问题”中的匹配和完美匹配。用图论的方法寻找完美匹配，需要首先找到最大匹配，当二部图中两个顶点集合中的顶点个数相等时，这个最大匹配同时也是完美匹配。求二部图的最大匹配可以使用最大流（maximal flow）或匈牙利算法（Hungarian algorithm），接下来我们就来介绍匈牙利算法。

7.4.1 最大匹配与匈牙利算法

寻找二部图最大匹配的匈牙利数学家埃德蒙德斯（Edmonds）在 1965 年提出的一个简化的最大流算法。该算法根据二部图匹配这个问题的特点将最大流算法进行了简化，提高了效率。普通的最大流算法一般都是基于带权网络模型的，二部图匹配问题不需要区分图中的源点和汇点，也不关心边的方向，因此不需要复杂的网络图模型，这就是匈牙利算法简化的原因。正是因为这个原因，匈牙利算法成为一种很简单的二分匹配算法，其基本流程是：

```
将图 G 最大匹配初始化为空
while(从 Xi 点开始在图 G 中找到新的增广路径)
{
    将增广路径假如到最大匹配中;
}
输出图 G 的最大匹配;
```

根据匈牙利算法的流程看，寻找图 G 中的增广路径（Augment Path）是匈牙利算法的关键。先来看看什么是增广路径，二部图中的增广路径具有以下性质：

- 路径中边的条数是奇数；
- 路径的起点在二部图的左半边，终点在二部图的右半边；
- 路径上的点一个在左半边，一个在右半边，交替出现，整条路径上没有重复的点；
- 只有路径的起点和终点都是未覆盖的点，路径上其他的点都已经配对；
- 对路径上的边按照顺序编号，所有奇数编号的边都不在已知的匹配中，所有偶数编号的边都在已知的匹配中；
- 对增广路径进行“取反”操作，新的匹配数就比已知匹配数增加一个，也就是说，可以得到一个更大的匹配。

所谓的增广路径取反操作，就是把增广路径上奇数编号的边加入到已知匹配中，并把增广路径上偶数编号的边从已知匹配中删除。每做一次“取反”操作，得到的匹配就比原匹配多一个。匈牙利算法的思路就是不停地寻找增广路径，增加匹配的个数，当不能再找到增广路径时，算法就结束了，得到的匹配就是最大匹配。

增广路径的起点总是在二部图的左边，因此寻找增广路径的算法总是从一侧的顶点开始，逐个顶点搜索。从 X_i 顶点开始搜索增广路径的流程如下：

```
while(从 Xi 的邻接表中找到下一个关联顶点 Yj)
{
    if(顶点 Yj 不在增广路径上)
    {
        将 Yj 加入增广路径;
        if(Yj 是未覆盖点或者从与 Yj 相关连的顶点（Xk）能找到增广路径)
        {
            将 Yj 的关联顶点修改为 Xi;
            从顶点 Xi 开始有增广路径，返回 true;
        }
    }
    从顶点 Xi 开始没有增广路径，返回 false;
}
```

在这个算法流程中，“从与 Y_j 相关连的顶点（X_k）能找到增广路径”这一步体现的是一个递归过程。因为如果之前的搜索已经将 Y_j 加入到增广路径中，说明 Y_j 在 X 集合中一定有一个关联点，我们假设 Y_j 在 X 集合中的这个关联点是 X_k，所以要从 X_k 开始继续寻找增广路径。当从 X_k 开始的递归搜索完成后，通过“将 Y_j 的关联顶点修改为 X_i”这一步操作，将其与 X_i 连在一起，形成一条更长的增广路径。

到现在为止，匈牙利算法的流程已经很清楚了，现在我们来给出实现代码。首先定义求最大匹配的数据结构，这个数据结构要能表示二部图的边的关系，还要能体现最终的增广路径结果，我们给出如下定义：

```
typedef struct tagMaxMatch
{
    int edge[UNIT_COUNT][UNIT_COUNT];
    bool on_path[UNIT_COUNT];
    int path[UNIT_COUNT];
    int max_match;
}GRAPH_MATCH;
```

edge 是顶点与边的关系表，用来表示二部图，on_path 用来表示顶点 Y_j 是否已经在当前搜索过程中形成的增广路径上了，path 是当前找到的增广路径，max_match 是当前增广路径中边的条数，当算法结束时，如果 max_match 不等于顶点个数，说明有顶点不在最大增广路径上，也就是说，找不到能覆盖所有点的增广路径，此二部图没有最大匹配。从 X_i 寻找增广路径的算法实现如下：

```
bool FindAugmentPath(GRAPH_MATCH *match, int xi)
{
    for(int yj = 0; yj < UNIT_COUNT; yj++)
    {
        if((match->edge[xi][yj] == 1) && !match->on_path[yj])
        {
            match->on_path[yj] = true;
            if( (match->path[yj] == -1)
                || FindAugmentPath(match, match->path[yj]) )
            {
                match->path[yj] = xi;
                return true;
            }
        }
    }

    return false;
}
```

算法实现基本上是按照之前的算法流程实现的，不需要做特别说明，唯一需要注意的是 path 中存放增广路径的方式。读者可能已经注意到了，存放的方式是以 Y 集合中的顶点为索引存放，其值是对应的关联顶点在 X 集合中的索引。搜索是按照 X 集合中的顶点索引进行的，增广路径以 Y 集合中的顶点为索引存储，关系是反的。输出结果的时候，需要结合 Y 集合中的顶点索引输出，如果需要以 X 集合的顺序输出结果，需要反向转换，转换的方法非常简单：

```
int path[UNIT_COUNT] = { 0 };

for(int i = 0; i < match->max_match; i++)
{
    path[match->path[i]] = i;
}
```

转换后 path 中就是以 X 集合的顺序存放的结果。

结合之前给出的匈牙利算法基本流程，最后给出匈牙利算法的入口函数实现：

```
bool Hungary_Match(GRAPH_MATCH *match)
{
    for(int xi = 0; xi < UNIT_COUNT; xi++)
    {
        if(FindAugmentPath(match, xi))
        {
            match->max_match++;
        }

        ClearOnPathSign(match);
    }
    return (match->max_match == UNIT_COUNT);
}
```

每完成一个顶点的搜索，需要重置 Y 集合中相关顶点的 on_path 标志，ClearOnPathSign()函数就负责干这个事情。

我们用图 7-2 中的二部图数据初始化 GRAPH_MATCH 中的顶点关系表 edge，然后调用 Hungary_Match()函数得到一组匹配：

```
X1<--->Y3
X2<--->Y1
X3<--->Y4
X4<--->Y2
X5<--->Y5
```

结果与图 7-3 一致，因为这个最大匹配没有未覆盖点，所以是完美匹配。

匈牙利算法的实现以顶点集合 V 为基础，每次 X 集合中选一个顶点 X_i 做增广路径的起点搜索增广路径。搜索增广路径需要遍历边集 E 内的所有边，遍历方法可以采用深度优先遍历（DFS），也可以采用广度优先遍历（BFS），无论什么方法，其时间复杂度都是 $O(E)$。匈牙利算法每个顶点 V_i 只能选择一次，因此算法的整体时间复杂度是 $O(V*E)$，总的来说，是一个相当高效的算法。除了匈牙利算法，求二部图的最大匹配还可以使用 Hopcroft-Karp 算法。Hopcroft-Karp 算法是由 Hopcroft 和 Karp 在 1972 年提出的一种算法，也是最大流算法的一种改进算法。算法的基本思想就是在每次搜索增广路径的时候不是只找一条增广路径，而是同时找几条互不相交的增广路径，形成最大增广路径集合，然后沿着集合中的几条增广路径同时扩大增广路径长度。通过进一步的分析，Hopcroft-Karp 算法的时间复杂度可以达到 $O(\text{sqrt}(V)*E)$，也非常高效。

7.4.2　带权匹配与Kuhn-Munkres算法

上一节我们介绍了二部图的最大匹配算法，用匈牙利算法寻找最大匹配，不要求每个个体给出的偏爱列表包含全部异性成员。比如在舞伴问题中，如果 Albert 非常讨厌 Marcy，那么 Albert 的偏爱列表无论如何也不会包含 Marcy。在这一节，我们让舞伴问题再复杂一点，于是为舞伴

问题引入带权优先表的概念，为每一个配对指定一个权重，表明我们更希望哪一对成为舞伴。通过控制每一对舞伴关系的权重，使得最后的完美匹配结果中有尽量多的舞伴是我们所期望的配对关系。

这个问题变得有点像最优解问题了，一提到与图有关的最优解问题，你会想到穷举法。穷举所有的完美匹配，然后计算每个完美匹配中各边的权重之和，取权重之和最大的一个作为最后的结果。这是一种解决方案，但是穷举法虽然是万能方法，但是不到万不得已最好不要用穷举法。仔细思考一下，其实这个问题已经演化成了求解二部图的带权匹配问题，所谓二部图的*带权匹配*其实就是求出一个匹配集合，使得集合中各边的权值之和最大或最小。对于本问题，给每一个配对（图中的边）指定一个权重之后问题就变成了求二部图的带权最大匹配问题。

通过之前对 Gale-Shapley 算法和匈牙利算法的介绍，我们已经了解了完美匹配、稳定匹配和最大匹配这些概念，那么带权匹配和之前的这些概念是什么关系呢？答案是没有半毛钱关系，至少和完美匹配与最大匹配之间不存在包含或等于关系。二部图的最大权或最小权匹配，只是要求得到的一个匹配中各边的权值之和最大或最小，并不要求这个匹配是完美匹配或最大匹配。如果这个权值最大（或最小）的匹配同时又是完美匹配，则这样的结果就被称为*最佳匹配*。本节我们要介绍的 Kuhn-Munkres 算法是求最大权或最小权匹配的算法，如果期望 Kuhn-Munkres 算法得到的结果同时是一个完美匹配（最佳匹配），那么要求算法运行的数据必须存在完美匹配（比如两个顶点集合的顶点个数必须相等之类的条件）。很多同学会忽略这一点，以为 Kuhn-Munkres 算法可以在任何情况下得到带权的最大匹配，这个理解是错误的。

Kuhn-Munkres 算法，也称 KM 算法，是 Kuhn 和 Munkres 二人在 1955 ~ 1957 年各自独立提出的一种算法，是一种求解最大最小权匹配问题的经典算法。最初的 Kuhn-Munkres 算法以矩阵为基础结构，但是 Edmonds 在 1965 年发布了匈牙利算法之后，Kuhn-Munkres 算法也基于匈牙利算法进行了改进。当给定的二部图存在完美匹配的情况下，Kuhn-Munkres 算法通过给每个顶点设置一个标号（叫作顶标）的方式把求最大权匹配的问题转化为求完美匹配的问题的，最终得到一个最大权完美匹配。那么这个转换是如何实现的呢？这就需要分析一下 Kuhn-Munkres 算法的原理了。

我们假设二部图中 X 顶点集合中每个顶点 X_i 的顶标是 $A[i]$，Y 顶点集合中每个顶点 Y_i 的顶标是 $B[i]$，顶点 X_i 和 Y_i 之间的边的权重是 weight[i][j]，则 Kuhn-Munkres 算法的原理就是基于以下定理：

若由二部图中所有满足 $A[i]+B[j]$ = weight[i,j]的边(X_i,Y_j)构成的子图（称作相等子图）有完美匹配，那么这个完美匹配就是二分图的最大权匹配。

现在明白转换原理了吧，就是先找出问题对应的相等子图，然后求相等子图的完美匹配即可。现在的问题是，这个定义成立吗？答案是只要在算法过程中始终满足“$A[i]+B[j] \geqslant$ weight[i,j]”这个条件，这个定理就成立。因为对于二分图的任意一个匹配，如果这个匹配是相等子图的匹配，那么它的边权重之和等于所有顶点的顶标之和（显然这是最大的）；如果这个匹配不是相等子图

的匹配（它的某些边不属于相等子图），那么它的边权重之和小于所有顶点的顶标和。所以只要始终满足“$A[i]+B[j] \geqslant \text{weight}[i,j]$”条件的相等子图的完备匹配一定是二分图的最大权匹配。

根据以上分析可知，Kuhn-Munkres 算法的实现流程大致如下所示。

(1) 初始化各个顶点的顶标值；

(2) 找出符合“$A[i]+B[j] = \text{weight}[i,j]$”条件的边构成相等子图，使用匈牙利算法寻找相等子图的完美匹配；

(3) 如果找到相等子图的完美匹配，则算法结束，否则调整相关顶点的顶标值；

(4) 重复步骤(2)(3)，直到找到完美匹配为止。

第(1)步初始化顶点顶标值可采用式(7-1)计算：

$$\begin{cases} A[x_i] = \max\{\text{weight}[x_i][y_0], \text{weight}[x_i][y_1], \cdots, \text{weight}[x_i][y_n]\}, x_i \in X \\ B[y_i] = 0, y_i \in Y \end{cases} \tag{7-1}$$

因为 $A[i]$总是取与之相邻的边中最大的权值作为初始值，因此初始阶段能保证满足“$A[i]+B[j] \geqslant \text{weight}[i,j]$”条件。如果在第(2)步的相等子图中没有找到完美匹配，说明相等子图中某个顶点出发的增广路径不能覆盖所有顶点。此时需要调整各个顶点的顶标值，然后重新在相等子图中寻找完美匹配。调整顶标的目的是为了扩大相等子图，使得更多的边进入相等子图，并最终能够找到一个完美匹配。设当前增广路径上所有属于 X 集合的顶点构成一个子集 S，所有属于 Y 集合的顶点构成一个子集 T，$\mathrm{d}x$ 为顶标调整的变化量，则 $\mathrm{d}x$ 可采用式(7-2)给出的方法计算：

$$\mathrm{d}x = \min\{A[x_i] + B[y_j] - \text{weight}[x_i][y_j], x_i \in S, y_j \notin T\} \tag{7-2}$$

由 $\mathrm{d}x$ 的计算公式可知，如果把 S 集合中所有顶点的顶标都减少 $\mathrm{d}x$，一定会有一条一端在 S 中，另一端不在 T 中的边因满足“$A[i]+B[j] = \text{weight}[i,j]$”的条件而进入相等子图，这就扩大了相等子图。对 S 集合中所有顶点的顶标都减少 $\mathrm{d}x$ 之后，为了使原来已经在相等子图中的边继续留在相等子图中，需要将 T 集合中所有顶点的顶标值增加 $\mathrm{d}x$，使 $A[i]+B[j]$之和不变。

现在总结一下顶标调整的方法，首先采用式(7-2)计算出调整变化量 $\mathrm{d}x$，然后将 S 集合中所有顶点的顶标值减少 $\mathrm{d}x$，同时将 T 集合中所有顶点的顶标值增加 $\mathrm{d}x$，这样的调整，对整个图上的所有顶点会产生如下四种结果。

- 对于两端点都在当前相等子图的增广路径上的边（x_i, y_j），其顶标值 $A[i]+B[j]$的和没有变化。也就是说，原来属于相等子图的边，调整后仍然属于相等子图。
- 对于两端点都不在当前相等子图的增广路径上的边（x_i, y_j），其顶标值 $A[i]$和 $B[j]$的值没有变化。也就是说，此边与相等子图的隶属关系没有变化，原来属于相等子图的边现在仍然属于相等子图，原来不属于相等子图的边现在仍然不属于相等子图。
- 对于 x_i 在当前相等子图的增广路径上，y_j 不在当前相等子图的增广路径上的边（x_i, y_j），其顶标值 $A[i]+B[j]$的和略有减小，原来不属于相等子图，现在有可能属于相等子图，使得相等子图有机会得到扩大。

- 如果 x_i 不在当前相等子图的增广路径上，y_j 在当前相等子图的增广路径上的边（x_i, y_j），其顶标值 $A[i]+B[j]$的和略有增加，原来不属于相等子图，现在仍不属于相等子图。

由此可见，每次调整顶标，都能在图的基本状态保持不变的情况下扩大相等子图，使得相等子图有机会找到一个完美匹配，这就是顶标值调整在算法中的意义所在。

现在，我们就结合一个带权最大匹配问题的实例，给出这个算法在实际应用中的一个实现。问题是这样的，某公司有 5 名技术工人，他们都可以完成公司流程中的 5 种工作，但是每个工人的技术侧重点不同，熟练程度也不同，因此他们完成同样的工作所产生的经济效益也不相同。如果用 0 ~ 5 范围内的值对每个工人完成每种工作所产生的经济效益进行评价，可得到如表 7-1 所示的经济效益矩阵。假如你是这家公司的负责人，你需要找到一种工人和工作之间的匹配关系，使得这种匹配关系能产生的经济效益最大。根据之前对 Kuhn-Munkres 算法的分析，我们针对这个问题设计了 KM_MATCH 匹配数据结构，如下所示：

```
typedef struct tagKmMatch
{
    int edge[UNIT_COUNT][UNIT_COUNT]; //Xi 与 Yj 对应的边的权重
    bool sub_map[UNIT_COUNT][UNIT_COUNT];// 二分图的相等子图，sub_map[i][j] = 1 代表 Xi 与 Yj 有边
    bool x_on_path[UNIT_COUNT]; // 标记在一次寻找增广路径的过程中，Xi 是否在增广路径上
    bool y_on_path[UNIT_COUNT]; // 标记在一次寻找增广路径的过程中，Yi 是否在增广路径上
    int path[UNIT_COUNT]; // 匹配信息，其中 i 为 Y 中的顶点标号，path[i]为 X 中顶点标号
}KM_MATCH;
```

相对于匈牙利算法中的 GRAPH_MATCH 定义，KM_MATCH 的主要变化是增加了 sub_map 作为相等子图定义和标识 y_i 是否在增广路径上的 y_on_path 标识。相对于我们前面对 Kuhn-Munkres 算法的分析，edge 对应于边的权重表 weight，sub_map 对应于算法执行过程中的相等子图，x_on_path 和 y_on_path 分别用于标识 X 集合和 Y 集合中的顶点是否属于增广路径上的 S 集合和 T 集合，path 就是最后匹配的结果。

表7-1 不同工人完成不同工作的经济效益

	工作1	工作2	工作3	工作4	工作5
工人1	3	5	5	4	1
工人2	2	2	0	2	2
工人3	2	4	4	1	0
工人4	0	1	1	0	0
工人5	1	2	1	3	3

下面给出 Kuhn-Munkres 算法的具体实现代码，Kuhn_Munkres_Match()函数虽然很长，但是并不难理解，因为这个代码是严格按照之前给出的 Kuhn-Munkres 算法的流程实现的。包括顶标的初始化、使用匈牙利算法求完美匹配和顶标调整在内的三个主要算法步骤在 Kuhn_Munkres_Match()函数中都得到体现，并且界定非常清晰。其中寻找增广路径的 FindAugmentPath()函数与之前介绍匈牙利算法时给出的 FindAugmentPath()函数实现非常类似，区别就是使用 sub_map 而不是直接使用 edge，并且额外记录了 x_on_path 标识。ResetMatchPath()函数负责每次开始寻找相等子图之前

清除上一次搜寻产生的临时增广路径，ClearOnPathSign()函数负责在每次搜寻增广路径之前清除顶点是否属于 S 集合或 T 集合的标识，大家可以从本书的配套代码中找到此函数的代码。

```
bool Kuhn_Munkres_Match(KM_MATCH *km)
{
    int i, j;
    int A[UNIT_COUNT], B[UNIT_COUNT];
    // 初始化 Xi 与 Yi 的顶标
    for(i = 0; i < UNIT_COUNT; i++)
    {
        B[i] = 0;
        A[i] = -INFINITE;
        for(j = 0; j < UNIT_COUNT; j++)
        {
            A[i] = std::max(A[i], km->edge[i][j]);
        }
    }
    while(true)
    {
        // 初始化带权二分图的相等子图
        for(i = 0; i < UNIT_COUNT; i++)
        {
            for(j = 0; j < UNIT_COUNT; j++)
            {
                km->sub_map[i][j] = ((A[i]+B[j]) == km->edge[i][j]);
            }
        }
        //使用匈牙利算法寻找相等子图的完备匹配
        int match = 0;
        ResetMatchPath(km);
        for(int xi = 0; xi < UNIT_COUNT; xi++)
        {
            ClearOnPathSign(km);
            if(FindAugmentPath(km, xi))
                match++;
            else
            {
                km->x_on_path[xi] = true;
                break;
            }
        }
        //如果找到完备匹配就返回结果
        if(match == UNIT_COUNT)
        {
            return true;
        }
        //调整顶标，继续算法
        int dx = INFINITE;
        for(i = 0; i < UNIT_COUNT; i++)
        {
            if(km->x_on_path[i])
            {
                for(j = 0; j < UNIT_COUNT; j++)
```

```
                {
                    if(!km->y_on_path[j])
                        dx = std::min(dx, A[i] + B[j] - km->edge[i][j]);
                }
            }
        }
        for(i = 0; i < UNIT_COUNT; i++)
        {
            if(km->x_on_path[i])
                A[i] -= dx;
            if(km->y_on_path[i])
                B[i] += dx;
        }
    }

    return false;
}
```

根据表 7-1 提供的数据初始化 KM_MATCH 数据结构，然后调用 Kuhn_Munkres_Match()函数，得到一个最大权匹配的结果，因为原数据存在完美匹配，因此这个结果就是最佳匹配结果：

工人 1 分配 工作 3（经济效益评价是 5）
工人 2 分配 工作 1（经济效益评价是 2）
工人 3 分配 工作 2（经济效益评价是 4）
工人 4 分配 工作 5（经济效益评价是 0）
工人 5 分配 工作 4（经济效益评价是 3）

最后获得最大经济效益评价是 14。需要说明的是，对于同一个问题，其最大权匹配的结果可能不唯一，也就是说，存在多个匹配的权重之和同为最大值的情况。Kuhn-Munkres 算法可以找出其中的一个，但是无法找到全部匹配结果。

7.5 总结

各种匹配问题可不是仅仅用来娱乐的算法竞赛题目，它们在现实生活中都有着广泛的应用。比如稳定匹配原理作为一种资源的分配方法，就在美国的医疗体系中得到了广泛应用。19 世纪 40 年代，在先进医疗技术的引领下，美国的医疗体系得到了巨大的发展，但是稀缺的医学院毕业生成了这个体系的心病。为了争抢稀缺资源，医院被迫在学生毕业前好几年就向他们提供实习机会。学生们则在还没有被证明有资格从事医疗工作的情况下就已经完成了工作配对，同时，如果医院提供的实习机会没有被学生接受，那么再向别的候选人提供机会就太晚了。很显然，这个市场没有稳定匹配，于是在 19 世纪 50 年代，美国启动了一个名为“国家住院医生匹配项目”（NRMP）的计划，旨在解决这个问题。从 1984 年开始，阿尔文·罗思（Alvin Roth）在论文中研究了这个项目使用的算法并发现了它与 Gale-Shapley 算法原理类似。他随之假设出 NRMP 项目成功的根本原因就是它使用了稳定匹配算法。后来，随着女医生越来越多，情侣们在一个地区寻找实习机会的现象越来越普遍，他们可不喜欢 NRMP 项目的这套匹配机制，这使得情侣们被

很容易安排在两个地方，这又引入了不稳定因素，那就是会导致情侣分居两地。于是在 1995 年，罗思为这个项目设计了一个新算法，这个新算法在 1997 年被 NRMP 所采纳，从那个时候开始到现在，该算法每年为超过 2 万个医生找到了合适的工作岗位。

2012 年诺贝尔经济学奖授予了两位美国学者：阿尔文・罗思和劳埃德・沙普利，以表彰他们在“如何让不同人为了互惠互利而联系在一起”这个课题上的出色研究。没错，这就是本章介绍的 Gale-Shapley 算法中提到的罗思和沙普利，他们被称为“数理经济学家”。

Gale-Shapley 算法又称为“求婚–拒绝算法”（propose-and-reject algorithm），以舞伴问题的整个求解过程来看，女孩从接受第一个邀请开始就有了舞伴，并且舞伴会越来越好，因为女孩可以根据自己的排序表确定是否选择更好的舞伴。与此同时，男孩如果被拒绝，他的选择对象会越来越差（因为男孩是根据自己的排序表从好到差开始选择的）。然而实际情况却并不是这样的，Gale-Shapley 算法中“求婚”的一方总是以最佳可能的稳定岗匹配结束，被“求婚”的一方总是以最差可能的稳定匹配结束，因为选择的主动权掌握在“求婚”者手中。现实生活中的道理也是如此，婚姻中男人如果不主动争取，条件好的女孩就会投入别人的怀抱，留给自己的机会就越来越差。学校里那些勇气可嘉、敢于主动示爱的男生，都是学过 Gale-Shapley 算法的，不信你问问他们。

7.6　参考资料

[1] Gale D, Shapley L S. *College Admissions and the Stability of Marriage*. American Mathematical Monthly, 1962,69:9-15

[2] Levitin A. 算法设计与分析基础. 潘彦译. 北京：清华大学出版社，2007

[3] Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[4] Knuth D E. *The Art of Computer Programming*(*Third Edition*), Vol 2. Addison-Wesley, 1997

[5] Kleigberg J, Tardos E. *Algorithm Design*. Addison-Wesley, 2005

[6] 稳定配对问题：http://www.docin.com/p-590496944.html