“十二五”普通高等教育本科国家级规划教材

国家精品课程教材

计算机算法设计与分析习题解答

（第5版）

◎ 王晓东　编著

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

“十二五”普通高等教育本科国家级规划教材
国家精品课程教材

计算机算法设计与分析习题解答

（第5版）

王晓东 编著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书是与“十二五”普通高等教育本科国家级规划教材《计算机算法设计与分析（第5版）》配套的辅助教材和国家精品课程教材，分别对主教材中的算法分析题和算法实现题给出了解答或解题思路提示。为了提高学生灵活运用算法设计策略解决实际问题的能力，本书还将主教材中的许多习题改造成算法实现题，要求学生设计出求解算法并上机实现。本书教学资料包含各章算法实现题、测试数据和答案，可在华信教育资源网免费注册下载。

本书内容丰富，理论联系实际，可作为高等学校计算机科学与技术、软件工程、信息安全、信息与计算科学等专业本科生和研究生学习计算机算法设计的辅助教材，也是工程技术人员和自学者的参考书。

图书在版编目（CIP）数据

计算机算法设计与分析习题解答/王晓东编著. —5版. —北京：电子工业出版社，2018.10

ISBN 978-7-121-34438-1

Ⅰ. ① 计… Ⅱ. ① 王… Ⅲ. ① 电子计算机－算法设计－高等学校－题解 ② 电子计算机－算法分析－高等学校－题解 Ⅳ. ① TP301.6-44

中国版本图书馆CIP数据核字（2018）第120711号

策划编辑：章海涛
责任编辑：章海涛
印　　刷：三河市良远印务有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编　100036
开　　本：787×1092　1/16　　印张：22.75　　字数：580千字
版　　次：2005年8月第1版
　　2018年10月第5版
印　　次：2018年10月第1次印刷
定　　价：56.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：192910558（QQ群）。

前　言

一些著名的计算机科学家在有关计算机科学教育的论述中认为，计算机科学是一种创造性思维活动，其教育必须面向设计。“计算机算法设计与分析”正是一门面向设计，且处于计算机学科核心地位的教育课程。通过对计算机算法系统的学习与研究，理解掌握算法设计的主要方法，培养对算法的计算复杂性正确分析的能力，为独立设计算法和对算法进行复杂性分析奠定坚实的理论基础，对每一位从事计算机系统结构、系统软件和应用软件研究与开发的科技工作者都是非常重要和必不可少的。课程结合我国高等学校教育工作的现状，追踪国际计算机科学技术的发展水平，更新了教学内容和教学方法，以算法设计策略为知识单元，在内容选材、深度把握、系统性和可用性方面进行了精心设计，力图适合高校本科生教学对学时数和知识结构的要求。

本书是“十二五”普通高等教育本科国家级规划教材《计算机算法设计与分析（第5版）》（ISBN 978-7-121-34439-8）配套的辅助教材，对《计算机算法设计与分析（第5版）》一书中的全部习题做了详尽的解答，旨在让使用该书的教师更容易教，学生更容易学。为了便于对照阅读，本书的章序与《计算机算法设计与分析（第5版）》一书的章序保持一致，且一一对应。

本书的内容是对《计算机算法设计与分析（第5版）》的较深入的扩展，许多教材中无法讲述的较深入的主题通过习题的形式展现出来。为了加强学生灵活运用算法设计策略解决实际问题的能力，本书将主教材中的许多习题改造成算法实现题，要求学生不仅设计出解决具体问题的算法，而且能上机实现。作者的教学实践反映出这类算法实现题的教学效果非常好。作者还结合国家精品课程建设，建立了“算法设计与分析”教学网站。国家精品资源共享课地址：http://www.icourses.cn/sCourse/course_2535.html。欢迎广大读者访问作者的教学网站并提出宝贵意见。

在本书编写过程中，福州大学“211工程”计算机与信息工程重点学科实验室为本书的写作提供了优良的设备与工作环境。电子工业出版社负责本书编辑出版工作的全体同仁为本书的出版付出了大量辛勤劳动，他们认真细致、一丝不苟的工作精神保证了本书的出版质量。在此，谨向每位曾经关心和支持本书编写工作的各方面人士表示衷心的谢意！

作　者

目　录

第6章　分支限界法

算法分析题6

6-1　0-1背包问题的栈式分支限界法。

栈式分支限界法将活结点表以后进先出（LIFO）的方式存储于一个栈中。试设计一个解0-1背包问题的栈式分支限界法，并说明栈式分支限界法与回溯法的区别。

分析与解答：函数StackKnapsack()实施对子集树的栈式分支限界法，假定各物品依其单位重量价值从大到小排好序。相应的排序过程可在算法的预处理部分完成。

算法中enode是当前扩展结点，cw是该结点所相应的重量，cp是相应的价值，up是价值上界。算法的while循环不断扩展结点，首先检查当前扩展结点的左儿子结点的可行性。如果该左儿子结点是可行结点，则将它加入到子集树和活结点栈中。当前扩展结点的右儿子结点一定是可行结点，仅当右儿子结点满足上界约束时，才将它加入子集树和活结点栈。

具体算法描述如下。

```
template<class Typew,class Typep>
Typep Knap<Typew,Typep>::StackKnapsack() {                    // 栈式分支限界法，返回最大价值
  H=new STACK<HeapNode<Typep, Typew>>(1000);                  // 定义栈的容量为1000
  int  i = 1;                                                 // 初始化
  E = 0;
  cw = cp = 0;
  Typep bestp = 0;                                            // 当前最优值
  Typep up = Bound(1);                                        // 价值上界
  // 搜索子集空间树
  while(true) {                                               // 检查当前扩展结点的左儿子结点
    Typew  wt = cw+w[i];
    if(wt <= c) {                                             // 左儿子结点为可行结点
      if(cp+p[i] > bestp)
        bestp = cp+p[i];
      AddLiveNode(up, cp+p[i], cw+w[i], true, i+1);
    }
    up = Bound(i+1);
    // 检查当前扩展结点的右儿子结点
    if(up >= bestp)                                           // 右子树可能含最优解
      AddLiveNode(up, cp, cw, false, i+1);
    if(H->EMPTY())                                            // 取下一扩展结点
      return bestp;
    HeapNode<Typep, Typew> N;
    H->POP(N);
    E = N.ptr;
    cw = N.weight;
```

```
        cp = N.profit;
        up = N.uprofit;
        i = N.level;
    }
}
```

其中，AddLiveNode()函数将一个新的活结点插入到子集树和栈中。

```
template<class Typep,class Typew>
void Knap<Typep, Typew>::AddLiveNode(Typep up, Typep cp, Typew cw, bool ch, int lev) {
    bbnode  *b = new bbnode;
    b->parent = E;
    b->LChild = ch;
    HeapNode<Typep, Typew> N;
    N.uprofit = up;
    N.profit = cp;
    N.weight = cw;
    N.level = lev;
    N.ptr = b;
    if(lev <= n)
        H->PUSH(N);
}
```

算法 Knapsack 完成输入数据的预处理，将各物品依其单位重量价值从大到小排好序。然后调用 StackKnapsack()函数完成对子集树的栈式分支限界搜索。

```
template<class Typew, class Typep>
Typep Knapsack(Typep *p,Typew *w,Typew c,int n) {
    //初始化
    Typew  W = 0;                                   // 装包物品重量
    Typep  P = 0;                                   // 装包物品价值
    Object  *Q = new Object [n];                    // 定义依单位重量价值排序的物品数组
    for(int i=1; i <= n; i++) {                     // 单位重量价值数组
        Q[i-1].ID = i;
        Q[i-1].d = 1.0*p[i]/w[i];
        P += p[i];
        W += w[i];
    }
    if(W <= c)
        return P;                                   // 所有物品装包
    MergeSort(Q, n);                                // 依单位重量价值排序
    Knap<Typew, Typep> K;                           // 创建类 Knap 的数据成员
    K.p = new Typep [n+1];
    K.w = new Typew [n+1];
    for(i=1; i <= n; i++) {
        K.p[i] = p[Q[i-1].ID];
        K.w[i] = w[Q[i-1].ID];
    }
    K.cp = 0;
    K.cw = 0;
    K.c = c;
```

```
    K.n = n;
    Typep bestp = K.StackKnapsack();              // 调用 StackKnapsack 求问题的最优解
    delete [] Q;
    delete [] K.w;
    delete [] K.p;
    return bestp;
}
```

栈式分支限界法与回溯法的主要区别在于，对当前扩展结点所采用的扩展方式不同。在栈式分支限界法中，每个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点。在这些儿子结点中，导致不可行解或导致非最优解的儿子结点被舍弃，其余儿子结点被加入活结点表中。此后，从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所需的解或活结点表为空时为止。而回溯法中的结点有可能多次成为活结点。

6-2　释放结点空间的队列式分支限界法。

修改解装载问题的分支限界算法MaxLoading，使得算法在结束前释放所有已由EnQueue产生的结点。

分析与解答：用一个队列 Que 收集算法删除的扩展结点。算法结束后释放所有已由EnQueue产生的结点。修改后的算法如下。

```
template<class Type>
Type MaxLoading(Type w[], Type c, int n, int bestx[]) { // 队列式分支限界法，返回最优载重量，bestx 返回最优解
    // 初始化
    Queue<QNode<Type>*>  Q, Que;                  //活结点队列
    Q.Add(0);                                     //同层结点尾部标志
    int  i = 1;                                   //当前扩展结点所处的层
    Type  Ew = 0, bestw = 0, r = 0;
    for(int j=2; j <= n; j++)
      r += w[i];
    QNode<Type> *E=0, *bestE;
    // 搜索子集空间树
    while(true) {                                 // 检查左儿子结点
      Type wt = Ew+w[i];
      if(wt <= c) {                               // 可行结点
        if(wt>bestw)
          bestw = wt;
        EnQueue(Q, wt, i, n, bestw, E, bestE, bestx, true);
      }
      if(Ew+r>bestw)                              // 检查右儿子结点
        EnQueue(Q, Ew, i, n, bestw, E, bestE, bestx, false);
      Q.Delete(E);                                // 取下一扩展结点
      if(E)
        Que.Add(E);
      if(!E) {                                    // 同层结点尾部
        if(Q.IsEmpty())
          break;
        Q.Add(0);                                 // 同层结点尾部标志
        Q.Delete(E);                              // 取下一扩展结点
```

```
        if(E)
           Que.Add(E);
        i++;                                    // 进入下一层
        r-=w[i];
      }
      Ew = E->weight;                           // 新扩展结点所相应的载重量
   }
   for(j=n-1; j > 0; j--) {                     // 构造当前最优解
      bestx[j] = bestE->LChild;
      bestE = bestE->parent;
   }
   while(!Que.IsEmpty()){
      QNode<Type> *b;
      Que.Delete(b);
      delete b;
   }
   return bestw;
}
```

6-3 及时删除不用的结点。

解装载问题的分支限界算法中，由 EnQueue 产生的结点可以在算法结束前一次性删除，然而那些没有活儿子结点或没有叶结点的扩展结点可以立即被删除。试设计一个在算法中及时删除不用结点的方案，并讨论其时间与空间之间的折中。

分析与解答：修改后及时删除不用结点的算法如下。

```
// 队列式分支限界法，返回最优载重量，bestx 返回最优解
template<class Type>
Type MaxLoading(Type w[], Type c, int n, int bestx[]) {
   // 初始化
   Queue<QNode<Type>*>  Q, Que;                 // 活结点队列
   Q.Add(0);                                    // 同层结点尾部标志
   int  i = 1;                                  // 当前扩展结点所处的层
   Type  Ew = 0, bestw = 0, r = 0;
   for(int j=2; j <= n; j++)
      r += w[i];
   QNode<Type>  *E=0, *bestE;
   while(true) {                                // 搜索子集空间树
      Type  wt = Ew+w[i];                       // 检查左儿子结点
      bool  del = true;
      if(wt <= c) {                             // 可行结点
         if(wt > bestw)
            bestw = wt;
         EnQueue(Q, wt, i, n, bestw, E, bestE, bestx, true);
         del = false;
      }
      if(Ew+r > bestw) {                        // 检查右儿子结点
         EnQueue(Q, Ew, i, n, bestw, E, bestE, bestx, false);
         del = false;
      }
```

```
    if(E) {
      if(del)
        delete E;
      else
        Que.Add(E);
    }
    Q.Delete(E);                          // 取下一扩展结点
    if(!E) {                              // 同层结点尾部
      if(Q.IsEmpty())
        break;
      Q.Add(0);                           // 同层结点尾部标志
      Q.Delete(E);                        // 取下一扩展结点
      i++;                                // 进入下一层
      r -= w[i];
    }
    Ew = E->weight;                       // 新扩展结点所相应的载重量
  }
  for(j=n1; j > 0; j--) {                 // 构造当前最优解
    bestx[j] = bestE->LChild;
    bestE = bestE->parent;
  }
  while(!Que.IsEmpty()) {
    QNode<Type> *b;
    Que.Delete(b);
    delete b;
  }
  return bestw;
}
```

6-4 用最大堆存储活结点的优先队列式分支限界法。

试修改解装载问题和解 0-1 背包问题的优先队列式分支限界法，使其仅使用一个最大堆来存储活结点，而不必存储产生的解空间树。

分析与解答：对于解 0-1 背包问题的优先队列式分支限界法，在堆结点中增加记录当前解的数组 x。

```
template<class Tw, class Tp>
class HeapNode {
  friend Knap<Tw, Tp>;
public:
  operator Tp () const { return uprofit; }
private:
  Tp  uprofit, profit;
  Tw  weight;
  int  level, *x;
};
```

修改后的算法如下。

```
template<class Typew, class Typep>
class Knap{
```

```
  friend Typep Knapsack(Typep*, Typew*, Typew, int, int*);
public:
  Typep MaxKnapsack();
private:
  MaxHeap<HeapNode<Typep, Typew>>*H;
  Typep Bound(int i);
  void AddLiveNode(Typep up, Typep cp, Typew cw, bool ch, int level, int *x);
  Typew  c;                                    // 背包容量
  int  n;                                      // 物品总数
  Typew  *w;                                   // 物品重量数组
  Typep  *p;                                   // 物品价值数组
  Typew  cw;                                   // 当前装包重量
  Typep  cp;                                   // 当前装包价值
  int  *bestx;                                 // 最优解
};
```

上界函数 Bound()计算结点所相应价值的上界。

```
template<class Typew, class Typep>
Typep Knap<Typew,Typep>::Bound(int i) {        // 计算结点所相应价值的上界
  Typew  cleft = c-cw;                         // 剩余容量
  Typep  b = cp;                               // 价值上界
  while(i <= n && w[i] <= cleft) {             // 以物品单位重量价值递减序装填剩余容量
    cleft -= w[i];
    b += p[i];
    i++;
  }
  if(i <= n)                                   // 装填剩余容量装满背包
    b += p[i]/w[i]*cleft;
  return b;
}
```

修改后的函数 AddLiveNode()将一个新结点插入到优先队列中。

```
// 将一个新结点插入到最大堆 H 中
template<class Typep, class Typew>
void Knap<Typep, Typew>::AddLiveNode(Typep up, Typep cp, Typew cw, bool ch, int lev, int *x) {
  HeapNode<Typep, Typew> N;
  N.x = new int[n+1];
  for(int j=0; j <= n; j++)
    N.x[j] = x[j];
  N.uprofit = up;
  N.profit = cp;
  N.weight = cw;
  N.level = lev;
  N.x[lev-1] = ch;
  H->Insert(N);
}
```

函数 MaxKnapsack()实施对子集空间树的优先队列式分支限界搜索，假定各物品依其单位重量价值从大到小排好序。相应的排序过程可在算法的预处理部分完成。

```
// 优先队列式分支限界法，返回最大价值，bestx 返回最优解
template<class Typew, class Typep>
Typep Knap<Typew, Typep>::MaxKnapsack() {
  H = new MaxHeap<HeapNode<Typep, Typew> >(10000);          // 定义最大堆的容量为 10000
  HeapNode<Typep, Typew> N;
  N.x = new int[n+1];
  bestx = new int [n+1];                                    // 为 bestx 分配存储空间
  for(int k=0; k <= n; k++)
    bestx[k] = 0;
  int  i = 1;                                               // 初始化
  c w= cp = 0;
  Typep  bestp = 0;                                         // 当前最优值
  Typep  up = Bound(1);                                     // 价值上界
  // 搜索子集空间树
  while(i != n+1){                                          // 非叶结点
    Typew  wt = cw+w[i];                                    // 检查当前扩展结点的左儿子结点
    if(wt <= c) {                                           // 左儿子结点为可行结点
      if(cp+p[i]>bestp)
        bestp = cp+p[i];
      AddLiveNode(up, cp+p[i], cw+w[i], true, i+1, bestx);
    }
    up = Bound(i+1);
    // 检查当前扩展结点的右儿子结点
    if(up >= bestp)                                         // 右子树可能含最优解
      AddLiveNode(up, cp, cw, false, i+1, bestx);
    H->DeleteMax(N);                                        // 取下一扩展结点
    cw = N.weight;
    cp = N.profit;
    up = N.uprofit;
    i = N.level;
    for(int j=0; j <= n; j++)
      bestx[j] = N.x[j];
  }
  for(int j=0; j <= n; j++)                                 // 构造当前最优解
    bestx[j] = N.x[j];
  while(true) {                                             // 释放最小堆中所有结点
    try { H->DeleteMax(N); }
    catch(OutOfBounds) { break; }
  }
  return cp;
}
```

下面的算法 Knapsack 完成输入数据的预处理，将各物品依其单位重量价值从大到小排好序。然后调用 MaxKnapsack()函数，完成对子集空间树的优先队列式分支限界搜索。

```
template<class Typew, class Typep>
Typep Knapsack(Typep *p, Typew *w, Typew c, int n, int *bestx) {      // 返回最大价值，bestx 返回最优解
  // 初始化
  Typew  W = 0;                                             // 装包物品重量
  Typep  P = 0;                                             // 装包物品价值
```

```
    Object  *Q = new Object [n];                    // 定义依单位重量价值排序的物品数组
    for(int i=1; i <= n; i++) {                     // 单位重量价值数组
      Q[i-1].ID = i;
      Q[i-1].d = 1.0*p[i]/w[i];
      P += p[i];
      W += w[i];
    }
    if(W <= c)
      return P;                                     // 所有物品装包
    MergeSort(Q, n);                                // 依单位重量价值排序
    Knap<Typew, Typep> K;                           // 创建类 Knap 的数据成员
    K.p = new Typep [n+1];
    K.w = new Typew [n+1];
    for(i=1;i <= n; i++) {
      K.p[i] = p[Q[i-1].ID];
      K.w[i] = w[Q[i-1].ID];
    }
    K.cp = 0;
    K.cw = 0;
    K.c = c;
    K.n = n;
    Typep bestp = K.MaxKnapsack();                  // 调用 MaxKnapsack 求问题的最优解
    for(int j =1; j <= n; j++)
      bestx[Q[j1].ID] = K.bestx[j];
    delete [] Q;
    delete [] K.w;
    delete [] K.p;
    delete [] K.bestx;
    return bestp;
}
```

6-5　释放结点空间的优先队列式分支限界法。

试修改解装载问题和解 0-1 背包问题的优先队列式分支限界法，使得算法在运行结束时释放所有类型为 bbnode 和 HeapNode 的结点所占用的空间。

分析与解答：用一个队列 Que 收集算法删除的扩展结点。算法结束后释放所有类型为 bbnode 和 HeapNode 结点所占用的空间。修改后的解 0-1 背包问题的优先队列式分支限界法如下。

```
// 优先队列式分支限界法，返回最大价值，bestx 返回最优解
template<class Typew, class Typep>
Typep Knap<Typew, Typep>::MaxKnapsack() {
  H=new MaxHeap<HeapNode<Typep, Typew> >(10000);        // 定义最大堆的容量为 10000
  Queue<bbnode *> Que;
  bestx = new int [n+1];                                // 为 bestx 分配存储空间
  int  i = 1;                                           // 初始化
  E = 0;
  cw = cp = 0;
  Typep  bestp = 0;                                     // 当前最优值
  Typep  up = Bound(1);                                 // 价值上界
```

```
  // 搜索子集空间树
  while(i != n+1) {                                    // 非叶结点
    Typew wt=cw+w[i];                                  // 检查当前扩展结点的左儿子结点
    if(wt <= c) {                                      // 左儿子结点为可行结点
      if(cp+p[i] > bestp)
        bestp = cp+p[i];
      AddLiveNode(up, cp+p[i], cw+w[i], true, i+1);}
      up = Bound(i+1);
      // 检查当前扩展结点的右儿子结点
      if(up >= bestp)                                  // 右子树可能含最优解
        AddLiveNode(up, cp, cw, false, i+1);
      HeapNode<Typep, Typew> N;                        // 取下一扩展结点
      H->DeleteMax(N);
      E = N.ptr;    Que.Add(E);
      cw = N.weight;
      cp = N.profit;
      up = N.uprofit;
      i = N.level;
    }
    for(int j=n; j > 0; j--) {                         // 构造当前最优解
      bestx[j] = E->LChild;
      E = E->parent;
    }
    HeapNode<Typep, Typew> N;
    while(true) {                                      // 释放堆中所有结点
      try { H->DeleteMax(N); }
      catch(OutOfBounds) { break; }
      Que.Add(N.ptr);
    }
    while(!Que.IsEmpty()) {
      bbnode*b;
      Que.Delete(b);
      delete b;
    }
  }
  return cp;
}
```

修改后的解装载问题的优先队列式分支限界法如下。

```
template<class T>
T MaxLoading(T *w, T c, int n, int *bestx) {
  MaxHeap<HeapNode<T>> H(1000);
  Queue<bbnode *> Que;
  T  *r = new T[n+1];
  r[n] = 0;
  for(int j=n-1; j > 0; j--)
    r[j] = r[j+1]+w[j+1];
  int  i = 1;
  bbnode  *E = 0;
```

```
    int  Ew = 0;
    while(i != n+1) {
      if(Ew+w[i] <= c)
        AddLiveNode(H, E, Ew+w[i]+r[i], true, i+1);
      AddLiveNode(H, E, Ew+r[i], false, i+1);
      HeapNode<T> N;
      H.DeleteMax(N);
      i = N.level;
      E = N.ptr;
      Que.Add(E);
      Ew = N.uweight-r[i-1];
    }
    for(j=n; j > 0 ; j--) {
      bestx[j] = E->LChild;
      E = E->parent;
    }
    HeapNode<T> N;
    while(true) {
      try {  H.DeleteMax(N);  }
      catch(OutOfBounds) {  break;  }
      Que.Add(N.ptr);
    }
    while(!Que.IsEmpty()) {
      bbnode  *b;
      Que.Delete(b);
      delete b;
    }
    return Ew;
}
```

6-6　团顶点数的上界。

在解最大团问题的优先队列式分支限界法中，当前扩展结点满足 cn+$n-i\geq$bestn 的右儿子结点被插入到优先队列中。如果将这个条件修改为满足 cn+$n-i>$bestn 右儿子结点插入优先队列，仍能保证算法的正确性吗？为什么？

分析与解答：如果将条件 cn+$n-i\geq$bestn 修改为满足 cn+$n-i>$bestn 时右儿子结点插入优先队列，不能保证算法的正确性。因为在当前扩展结点处，团顶点数的上界为 cn+$n-i$+1。

6-7　团顶点数改进的上界。

考虑最大团问题的子集空间树中第 i 层的一个结点 x，设 MinDegree(x)是以结点 x 为根的子树中所有结点度数的最小值。

（1）设 $x.u$=min{x.cn+$n-i$+1, MinDegree(x)+1}，证明以结点 x 为根的子树中任意叶结点相应的团的大小不超过 $x.u$。

（2）依此 $x.u$ 的定义重写算法 BBMaxClique。

（3）比较新旧算法所需的计算时间和产生的排列树结点数。

分析与解答：

（1）在当前扩展结点 x 处，MinDegree(x)+1 显然是团顶点数的一个上界。主教材中在当前扩展结点 x 处团顶点数的上界为 x. cn+$n-i$+1。

由此可见，min{x. cn+n−i+1, MinDegree(x)+1}是当前扩展结点 x 处团顶点数的上界。

（2）在算法预处理时，先计算出每个顶点的 MinDegree(x)，然后在算法中修正团顶点数的上界。

（3）新算法所产生的排列树结点数较少。

6-8　修改解旅行售货员问题的分支限界法。

试修改解旅行售货员问题的分支限界法，使得 s=n−2 的结点不插入优先队列，而是将当前最优排列存储于 bestp 中。这样修改后，算法在下一个扩展结点满足条件 Lcost≥bestc 时结束。

分析与解答：在类 Traveling 中增加保存当前最优排列的数组 bestp。

```
template<class Type>
class Traveling{
   friend int main();
public:
   Type BBTSP(int v[]);
private:
   int  n, *bestp;
   Type  **a, NoEdge, c, bestc;
};
```

最小堆结点类型不变。

```
template<class Type>
class MinHeapNode {
   friend Traveling<Type>;
public:
   operator Type () const {  return lcost;  }
private:
   Type  lcost, cc, rcost;
   int  s, *x;
};
```

修改后的算法描述如下。

```
template<class Type>
Type Traveling<Type>::BBTSP(int v[]) {                    // 解旅行售货员问题的优先队列式分支限界法
   MinHeap<MinHeapNode<Type>> H(1000000);                 // 定义最小堆的容量为 1000000
   Type  *MinOut = new Type[n+1];
   // 计算 MinOut[i]=顶点 i 的最小出边费用
   Type  MinSum = 0;                                      // 最小出边费用和
   for(int i=1; i <= n; i++) {
      Type Min = NoEdge;
      for(int j=1; j <= n; j++)
         if(a[i][j] != NoEdge && (a[i][j] < Min || Min == NoEdge))
            Min = a[i][j];
      if(Min == NoEdge)
         return NoEdge;                                   // 无回路
      MinOut[i] = Min;
      MinSum += Min;
   }
```

```
// 初始化
MinHeapNode<Type> E;
E.x = new int [n];
for(i=0; i < n; i++)
   E.x[i] = i+1;
E.s = 0;
E.cc = 0;
E.rcost = MinSum;
Type  bestc = NoEdge;
// 搜索排列空间树
while (E.s < n-1 && E.lcost < bestc) {                    // 非叶结点
   // 当前扩展结点是叶结点的父结点再加 2 条边构成回路，所构成回路是否优于当前最优解
   if(E.s == n-2) {
      // 费用更小的回路
      if(a[E.x[n-2]][E.x[n-1] ]!= NoEdge && a[E.x[n-1]][1] != NoEdge
                  && (E.cc+ a[E.x[n-2]][E.x[n-1]]+a[E.x[n-1]][1] < bestc || bestc == NoEdge)) {
         bestc = E.cc+a[E.x[n-2]][E.x[n-1]]+a[E.x[n-1]][1];
         for(int j=0; j < n; j++)
            bestp[j] = E.x[j];
      }
      else
         delete [] E.x;                                   // 舍弃扩展结点
   }
   else {                                                 // 产生当前扩展结点的儿子结点
      for(int i=E.s+1; i < n; i++) {
         if(a[E.x[E.s]][E.x[i]] != NoEdge) {              // 可行儿子结点
            Type  cc = E.cc+a[E.x[E.s]][E.x[i]];
            Type  rcost = E.rcostMinOut[E.x[E.s]];
            Type  b = cc+rcost;                           // 下界
            if(b < bestc || bestc == NoEdge) {            // 子树可能含最优解，结点插入最小堆
               MinHeapNode<Type> N;
               N.x = new int [n];
               for(int j=0; j < n; j++)
                  N.x[j] = E.x[j];
               N.x[E.s+1] = E.x[i];
               N.x[i] = E.x[E.s+1];
               N.cc = cc;
               N.s = E.s+1;
               N.lcost = b;
               N.rcost = rcost;
               H.Insert(N);}
            }
            delete [] E.x;
         }                                                // 完成结点扩展
         try { H.DeleteMin(E); }                          // 取下一扩展结点
         catch (OutOfBounds){ break; }                    // 堆已空
      }
   }
}
```

```
  if(bestc == NoEdge)
    return NoEdge;                                  // 无回路
  for(i=0; i < n; i++)                              // 将最优解复制到 v[1:n]
    v[i+1] = bestp[i];
  while(true) {                                     // 释放最小堆中所有结点
    delete [] E.x;
    try { H.DeleteMin(E); }
    catch(OutOfBounds) { break; }
  }
  return bestc;
}
```

6-9 试修改解旅行售货员问题的分支限界法，使得算法保存已产生的排列树。

分析与解答： 排列树中结点类型定义为：

```
class bbnode {
public:
  bbnode *parent;
  int  s, *x;
};
```

保存已产生的排列树的旅行售货员问题的分支限界法如下。

```
template<class Type>
class Traveling {
  friend int main();
public:
  Type  BBTSP(int v[]);
private:
  int  n;
  Type  **a, NoEdge, cc, bestc;
};
template<class Type>
class MinHeapNode{
  friend Traveling<Type>;
public:
  operator Type () const { return lcost; }
private:
  Type  lcost, cc, rcost;
  bbnode *ptr;
};
template<class Type>
Type Traveling<Type>::BBTSP(int v[]) {                  // 解旅行售货员问题的优先队列式分支限界法
  MinHeap<MinHeapNode<Type>> H(100000);                 // 定义最小堆的容量为 100000
  Type *MinOut = new Type[n+1];
  Type  MinSum = 0;                                     // 计算 MinOut[i]=顶点 i 的最小出边费用
  for(int i=1; i <= n; i++) {
    Type  Min = NoEdge;
    for(int j=1; j <= n; j++)
      if(a[i][j] != NoEdge && (a[i][j]<Min || Min == NoEdge))
        Min = a[i][j];
```

```
    if(Min == NoEdge)
      return NoEdge;
    MinOut[i] = Min;
    MinSum += Min;
  }
  MinHeapNode<Type> E;                                    // 初始化
  bbnode  *bb = new bbnode;
  bb->parent = 0;
  bb->x = new int[n];
  bb->s = 0;
  for(int j=0; j < n; j++)
    bb->x[j] = j+1;
  E.ptr = bb;
  E.cc = 0;
  E.rcost = MinSum;
  Type  bestc = NoEdge;
  while (E.ptr->s < n-1) {                                // 搜索排列空间树
    if(E.ptr->s == n-2) {
      if(a[E.ptr->x[n-2]][E.ptr->x[n-1]] != NoEdge &&a[E.ptr->x[n-1]][1] != NoEdge
                  && (E.cc+a[E.ptr->x[n-2]][E.ptr->x[n-1]]+a[E.ptr->x[n-1]][1] < bestc
                  || bestc == NoEdge)) {
        bestc = E.cc+a[E.ptr->x[n-2]][E.ptr->x[n-1]]+a[E.ptr->x[n-1]][1];
        E.cc = bestc;
        E.lcost = bestc;
        E.ptr->s++;
        H.Insert(E);
      }
    }
    else {                                                // 产生当前扩展结点的儿子结点
      for(int i=E.ptr->s+1; i < n; i++) {
        if(a[E.ptr->x[E.ptr->s]][E.ptr->x[i]] != NoEdge)
          Type cc=E.cc+a[E.ptr->x[E.ptr->s]][E.ptr->x[i]];
        Type  rcost = E.rcost-MinOut[E.ptr->x[E.ptr->s]];
        Type  b = cc+rcost;
        if(b < bestc || bestc == NoEdge) {                // 子树可能含最优解，结点插入最小堆
          bbnode  *bb = new bbnode;
          bb->parent = E.ptr;
          bb->x = new int[n];
          bb->s = E.ptr->s+1;
          for(int j=0; j < n; j++)
            bb->x[j] = E.ptr->x[j];
          bb->x[E.ptr->s+1] = E.ptr->x[i];
          bb->x[i] = E.ptr->x[E.ptr->s+1];
          MinHeapNode<Type> N;
          N.cc = cc;
          N.lcost = b;
          N.rcost = rcost;
          N.ptr = bb;
          H.Insert(N);
```

```
        }
      }
    }                                                    // 完成结点扩展
    try { H.DeleteMin(E); }                              // 取下一扩展结点
    catch (OutOfBounds) { break; }
  }
  if(bestc == NoEdge)
    return NoEdge;
  bb = E.ptr;                                            // 将最优解复制到 v[1:n]
  v[n] = bb->x[n-1];
  for(j=n; j > 1; j--) {
    v[j-1] = bb->x[j-2];
    bb = bb->parent;
  }
  return bestc;
}
```

6-10 电路板排列问题的队列式分支限界法。

试设计解电路板排列问题的队列式分支限界法，并使算法在运行结束时输出最优解和最优值。

分析与解答：

```
class BoardNode{
  friend int FIFOArr(int **, int, int, int *);
public:
  operator int() const { return cd; }
private:
  int  *x, s, cd, *now;
};
```

解电路板排列问题的队列式分支限界法实现如下。

```
int FIFOArr(int **B, int n, int m, int * bestx) {          // 解电路板排列问题的队列式分支限界法
  Queue<BoardNode> H;                                      // 活结点队列
  BoardNode  E;                                            // 初始化
  E.x = new int[n+1];
  E.s = 0;
  E.cd = 0;
  E.now = new int[m+1];
  int  *total = new int[m+1];
  for(int i=1; i <= m; i++) {     // now[i]=x[1:s]所含连接块 i 中电路板数, total[i]=连接块 i 中电路板数
    total[i] = 0;
    E.now[i] = 0;
  }
  for(i=1; i <= n; i++) {
    E.x[i] = i;                                            // 初始排列为 1 2 3 … n
    for(int j=1; j <= m; j++)
      total[j] += B[i][j];                                 // 连接块 j 中电路板数
  }
  int  bestd = m+1;                                        // 当前最小密度
```

```
while(true) {                                          // 结点扩展
  if(E.s == n-1) {                                     // 仅一个儿子结点
    int  ld = 0;                                       // 最后一块电路板的密度
    for(int j=1; j <= m; j++)
      ld += B[E.x[n]][j];
    if(ld < bestd && E.cd < bestd) {                   // 密度更小的电路板排列
      bestd = max(ld, E.cd);
    for(int k=0; k <= n; k++)
      bestx[k] = E.x[k];
  }
  else
    delete[] E.x;
  delete[] E.now;
}
else {                                                 // 产生当前扩展结点的所有儿子结点
  for(int i=E.s+1; i <= n; i++) {
    BoardNode N;
    N.now = new int[m+1];
    for(int j=1; j <= m; j++)                          // 新插入的电路板
      N.now[j] = E.now[j]+B[E.x[i]][j];
    int  ld = 0;                                       // 新插入电路板的密度
    for(j=1; j <= m; j++)
      if(N.now[j] > 0 && total[j] != N.now[j])
        ld++;
    N.cd = max(ld, E.cd);
    if(Ncd<bestd){                                     // 可能产生更好的叶结点
      N.x = new int[n+1];
      N.s = E.s+1;
      for(int j=1; j <= n; j++)
        N.x[j] = E.x[j];
      N.x[N.s] = E.x[i];
      N.x[i] = E.x[N.s];
      H.Add(N);
    }
    else
      delete[]N.now;
  }
  delete[] E.x;
}                                                      // 完成当前结点扩展
if(H.IsEmpty())
  break;
else
  H.Delete(E);                                         // 取下一扩展结点
return bestd;
}
```

算法实现题 6

6-1　最小长度电路板排列问题。

问题描述：最小长度电路板排列问题是大规模电子系统设计中提出的实际问题。该问题的提法是，将 n 块电路板以最佳排列方案插入带有 n 个插槽的机箱中。n 块电路板的不同的排列方式对应不同的电路板插入方案。

设 $B=\{1, 2, \cdots, n\}$ 是 n 块电路板的集合。集合 $L=\{N_1, N_2, \cdots, N_m\}$ 是 n 块电路板的 m 个连接块。其中每个连接块 N_i 是 B 的一个子集，且 N_i 中的电路板用同一根导线连接在一起。在最小长度电路板排列问题中，连接块的长度是指该连接块中第 1 块电路板到最后 1 块电路板之间的距离。例如，设 n=8，m=5，给定 n 块电路板及其 m 个连接块如下：

$$B=\{1, 2, 3, 4, 5, 6, 7, 8\};\qquad L=\{N_1, N_2, N_3, N_4, N_5\}$$

$$N_1=\{4, 5, 6\};\quad N_2=\{2, 3\};\quad N_3=\{1, 3\};\quad N_4=\{3, 6\};\quad N_5=\{7, 8\}$$

这 8 块电路板的一个可能的排列如图 6-1 所示。

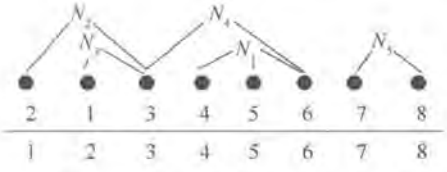


图 6-1　最小长度电路板排列

试设计一个队列式分支限界法找出所给 n 个电路板的最佳排列，使得 m 个连接块中最大长度达到最小。

算法设计：对于给定的电路板连接块，设计一个队列式分支限界法，找出所给 n 个电路板的最佳排列，使得 m 个连接块中最大长度达到最小。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m（$1 \leqslant m, n \leqslant 20$）。接下来的 n 行中，每行有 m 个数。第 k 行的第 j 个数为 0 表示电路板 k 不在连接块 j 中，为 1 表示电路板 k 在连接块 j 中。

结果输出：将计算的电路板排列最小长度及其最佳排列输出到文件 output.txt。文件的第 1 行是最小长度；接下来的 1 行是最佳排列。

输入文件示例	输出文件示例
input.txt	output.txt
8 5	4
1 1 1 1 1	5 4 3 1 6 2 8 7
0 1 0 1 0	
0 1 1 1 0	
1 0 1 1 0	
1 0 1 0 0	
1 1 0 1 0	
0 0 0 0 1	
0 1 0 0 1	

分析与解答：与最小密度电路板排列问题类似，结点元素类型是 BoardNode。

```
class BoardNode{
   friend int FIFOBoards(int **, int, int, int *&);
public:
   operator int() const{  return cd;  }
   int len();
private:
   int  *x, s, cd, *low, *high;
```

```
};
```

其中，len()函数计算当前排列的最小长度。

```
int BoardNode::len() {
   int  tmp = 0;
   for(int k=1; k <= m; k++)
      if(low[k] <= n && high[k] > 0 && tmp < high[k]-low[k])
         tmp = high[k]-low[k];
   return tmp;
}
```

解最小长度电路板排列问题的队列式分支限界法如下。

```
int FIFOBoards(int **B, int n, int m, int *&bestx) {
   Queue<BoardNode> Q;
   BoardNode E;
   E.x = new int[n+1];
   E.s = 0;
   E.cd = 0;
   E.low = new int[m+1];
   E.high = new int[m+1];
   for(int i=1; i <= m; i++) {
      E.high[i] = 0;
      E.low[i] = n+1;
   }
   for(i=1; i <= n; i++)
      E.x[i] = i;
   int  bestd = n+1;
   bestx = 0;
   do {
      if(E.s == n-1) {
         for(int j=1; j <= m; j++)
            if(B[E.x[n]][j] && n > E.high[j])
               E.high[j] = n;
         int  ld = E.len();
         if(ld < bestd) {
            delete[] bestx;
            bestx = E.x;
            bestd = ld;
         }
         else
            delete[] E.x;
         delete[] E.low;
         delete[] E.high;
      }
      else {
         int  curr = E.s+1;
         for(int i=E.s+1; i <= n; i++) {
            BoardNode N;
            N.low = new int[m+1];
```

```
            N.high = new int[m+1];
            for(int j=1; j <= m; j++) {
               N.low[j] = E.low[j];
               N.high[j] = E.high[j];
               if(B[E.x[i]][j]) {
                  if(curr < N.low[j])
                     N.low[j] = curr;
                  if(curr > N.high[j])
                     N.high[j] = curr;
               }
            }
            N.cd= N.len();
            if(N.cd < bestd) {
               N.x = new int[n+1];
               N.s= E.s+1;
               for(int j=1; j <= n; j++)
                  N.x[j] = E.x[j];
               N.x[N.s] = E.x[i];
               N.x[i] = E.x[N.s];
               Q.Add(N);
            }
            else {
               delete[] N.low;
               delete[] N.high;
            }
         }
         delete[] E.x;
      }
      try { Q.Delete(E); }
      catch(OutOfBounds) { return bestd; }
   } while(!Q.IsEmpty());
   return bestd;
}
```

6-2 最小权顶点覆盖问题。

问题描述：给定一个赋权无向图 $G=(V, E)$，每个顶点 $v\in V$ 都有权值 $w(v)$。如果 $U\subseteq V$，且对任意$(u, v)\in E$ 有 $u\in U$ 或 $v\in U$，就称 U 为图 G 的一个顶点覆盖。G 的最小权顶点覆盖是指 G 中所含顶点权之和最小的顶点覆盖。

算法设计：对于给定的无向图 G，设计一个优先队列式分支限界法，计算 G 的最小权顶点覆盖。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m，表示给定的图 G 有 n 个顶点和 m 条边，顶点编号为 1, 2, …, n。第 2 行有 n 个正整数表示 n 个顶点的权。接下来的 m 行中，每行有 2 个正整数 u 和 v，表示图 G 的一条边(u, v)。

结果输出：将计算的最小权顶点覆盖的顶点权之和以及最优解输出到文件 output.txt。文件的第 1 行是最小权顶点覆盖顶点权之和；第 2 行是最优解 x_i（$1\leqslant i\leqslant n$），$x_i=0$ 表示顶点 i 不在最小权顶点覆盖中，$x_i=1$ 表示顶点 i 在最小权顶点覆盖中。

输入文件示例	输出文件示例
input.txt	output.txt
7 7	13
1 100 1 1 1 100 10	1 0 1 1 0 0 1
1 6	
2 4	
2 5	
3 6	
4 5	
4 6	
6 7	

分析与解答：最小堆结点元素类型是 HeapNode。

```
class HeapNode{
   friend class VC;
public:
   operator int () const {  return cn;  }
private:
   int  i, cn, *x, *c;
};
```

解最小权顶点覆盖问题的优先队列式分支限界法如下。

```
class VC {
   friend MinCover(int **, int [], int);
private:
   void BBVC();
   bool cover(HeapNode E);
   void AddLiveNode(MinHeap<HeapNode> &H, HeapNode E, int cn, int i, bool ch);
   int  **a, n, *w, *bestx, bestn;
};
void VC::BBVC() {
   MinHeap<HeapNode> H(100000);
   HeapNode  E;
   E.x = new int[n+1];
   E.c = new int[n+1];
   for(int j=1; j <= n; j++)
      E.x[j] = E.c[j] = 0;
   int  i = 1, cn = 0;
   while(true) {
      if(i > n) {
         if(cover(E)) {
            for(int j=1; j <= n; j++)
               bestx[j] = E.x[j];
            bestn = cn;
            break;
         }
      }
      else {
         if(!cover(E))
```

```
            AddLiveNode(H, E, cn, i, 1);
         AddLiveNode(H, E, cn, i, 0);}
         if(H.Size() == 0)
            break;
         H.DeleteMin(E);
         cn = E.cn;
         i = E.i+1;
      }
}
```

cover()函数判定图是否已完全覆盖。

```
bool VC::cover(HeapNode E) {
   for(int j=1; j <= n; j++)
      if(E.x[j] == 0 && E.c[j] == 0)
         return false;
   return true;
}
```

AddLiveNode()函数将活结点加入堆中。

```
void VC::AddLiveNode(MinHeap<HeapNode> &H, HeapNode E, int cn, int i, bool ch) {
   HeapNode N;
   N.x = new int[n+1];
   N.c = new int[n+1];
   for(int j=1;j <= n; j++) {
      N.x[j] = E.x[j];
      N.c[j] = E.c[j];
   }
   N.x[i] = ch;
   if(ch) {
      N.cn = cn+w[i];
      for(int j=1; j <= n; j++)
         if(a[i][j])
            N.c[j]++;
   }
   else
      N.cn = cn;
   N.i = i;
   H.Insert(N);
}
```

MinCover()函数完成最小覆盖计算。

```
int MinCover(int **a, int v[], int n) {
   VC  Y;
   Y.w = new int [n+1];
   for(int j=1; j <= n; j++)
      Y.w[j]=v[j];
   Y.a=  a;
   Y.n = n;
   Y.bestx = v;
```

```
    Y.BBVC();
    return Y.bestn;
}
```

算法的主函数如下。

```
int main() {
    int  u, v;
    cin>>n>>e;
    Make2DArray(a, n+1, n+1);
    for(int i=0; i <= n; i++)
       for(int j=0; j <= n;j++)
          a[i][j] = 0;
       p = new int[n+1];
       for(i=1; i <= n; i++)
          cin >> p[i];
       for(i=1; i <= e; i++){
         cin >> u >> v;
       a[u][v] = 1;
       a[v][u] = 1;
    }
    cout << MinCover(a,p  , n)< < endl;
   for(i=1; i<= n; i++)
       cout << p[i] << " ";
    cout << endl;
     return 0;
}
```

6-3　无向图的最大割问题。

问题描述：给定一个无向图 $G=(V, E)$，设 $U \subseteq V$ 是 G 的顶点集。对任意$(u, v) \in E$，若 $u \in U$ 且 $v \in V-U$，就称(u, v)为关于顶点集 U 的一条割边。顶点集 U 的所有割边构成图 G 的一个割。G 的最大割是指 G 中所含边数最多的割。

算法设计：对于给定的无向图 G，设计一个优先队列式分支限界法，计算 G 的最大割。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m，表示给定的图 G 有 n 个顶点和 m 条边，顶点编号为 $1, 2, \cdots, n$。接下来的 m 行中，每行有 2 个正整数 u 和 v，表示图 G 的一条边(u, v)。

结果输出：将计算的最大割的边数和顶点集 U 输出到文件 output.txt。文件的第 1 行是最大割的边数；第 2 行是表示顶点集 U 的向量 x_i（$1 \leqslant i \leqslant n$），$x_i=0$ 表示顶点 i 不在顶点集 U 中，$x_i=1$ 表示顶点 i 在顶点集 U 中。

输入文件示例	输出文件示例
input.txt	output.txt
7 18	12
1 4	1 1 1 0 1 0 0
1 5	
1 6	
1 7	
2 3	
2 4	

2 5
2 6
2 7
3 4
3 5
3 6
3 7
4 5
4 6
5 6
5 7
6 7

分析与解答：最大堆结点元素类型是 HeapNode。

```
class HeapNode{
   friend class  MCut;
public:
   operator int () const {  return cut+e;  }
private:
   int  i, cut, e, *x;
};
```

解无向图最大割问题的优先队列式分支限界法如下。

```
class MCut{
   friend MaxCut(int **, int [], int, int);
private:
   void BBCut();
   void AddLiveNode(MaxHeap<HeapNode> &H, HeapNode E, bool ch);
   int  **a, n, e, *bestx, bestn;
};
void MCut::BBCut() {
   MaxHeap<HeapNode> H(HeapSize);
   HeapNode E;
   E.x = new int[n+1];
   E.cut = 0;
   E.e = e;
   E.i = 1;
   for(int j=1; j <= n; j++)
      E.x[j] = 0;
   while(true) {
      if(E.i > n) {
         if(E.cut > bestn){
            for(int j=1; j <= n; j++)
               bestx[j] = E.x[j];
            bestn = E.cut;
         }
      }
      else {
         AddLiveNode(H, E, 1);
```

```
        if(E.cut+E.e > bestn)
          AddLiveNode(H, E, 0);
      }
      if(H.Size() == 0)
        break;
      H.DeleteMax(E);
    }
}
void MCut::AddLiveNode(MaxHeap<HeapNode> &H, HeapNode E, bool ch) {
   HeapNode N;
   int  i=  E.i;
   N.x = new int[n+1];
   for(int j=1; j <= n; j++)
     N.x[j] = E.x[j];
   N.x[i] = ch;
   N.cut = E.cut;
   N.e = E.e;
   if(ch) {
     for(int j=1; j <= n; j++) {
       if(a[i][j]) {
         if(N.x[j] == 0) {
           N.cut++;
           N.e--;
         }
         else
           N.cut--;
       }
     }
   }
   N.i = i+1;
   H.Insert(N);
}
```

MaxCut()函数完成最大割计算。

```
int MaxCut(int **a, int v[], int n, int e) {
   MCut Y;
   Y.a =a;
   Y.n = n;
   Y.e = e;
   Y.bestn = 0;
   Y.bestx = v;
   Y.BBCut();
   return Y.bestn;
}
```

算法的主函数如下。

```
int main() {
   int  e, u, v;
   cin> > n >> e;
```

```
    Make2DArray(a, n+1, n+1);
    for(int i=0; i <= n; i++)
      for(int j=0; j <= n; j++)
        a[i][j] = 0;
    p=  new int[n+1];
    for(i=1; i<= e; i++) {
      cin >> u>> v;
      a[u][v] = 1;
      a[v][u] = 1;
    }
    cout << MaxCut(a, p, n, e) << endl;
    for(i=1; i <= n; i++)
      cout << p[i] << " ";
    cout << endl;
    return 0;
}
```

6-4　最小重量机器设计问题。

问题描述：设某一机器由 n 个部件组成，每种部件都可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量，c_{ij} 是相应的价格。设计一个优先队列式分支限界法，给出总价格不超过 d 的最小重量机器设计。

算法设计：对于给定的机器部件重量和机器部件价格，设计一个优先队列式分支限界法，计算总价格不超过 d 的最小重量机器设计。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n、m 和 d。接下来的 $2n$ 行，每行 n 个数。前 n 行是 c，后 n 行是 w。

结果输出：将计算的最小重量，以及每个部件的供应商输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3 3 4	4
1 2 3	1 3 1
3 2 1	
2 2 2	
1 2 3	
3 2 1	
2 2 2	

分析与解答：状态空间树中结点类型为 bbnode。

```
class bbnode{
   friend Mach<int, int>;
   friend int Machine(int **, int **, int, int, int, int *);
private:
   bbnode  *parent;
   int  mj;
};
```

堆结点元素类型是 HeapNode。

```
template<class Typew, class Typep>
class HeapNode {
```

```
  friend Mach<Typew, Typep>;
public:
  operator Typew () const {  return weight;  }
private:
  Typep  profit;
  Typew  weight;
  int  level;
  bbnode  *ptr;
};
```

解最小重量机器设计问题的优先队列式分支限界法如下。

```
template<class Typew, class Typep>
class Mach {
  friend Typep Machine(Typep **, Typew **, Typew, int, int, int *);
public:
  Typep MinWeightMachine();
private:
  MinHeap<HeapNode<Typep, Typew>> *H;
  void AddLiveNode(Typep cp, Typew cw, int i, int j);
  bbnode  *E;
  int  n, m, *bestx;
  Typew  cc, cw, **w;
  Typep  cp, **c;
};
template<class Typew, class Typep>
Typep Mach<Typew, Typep>::MinWeightMachine(){
  H = new MinHeap<HeapNode<Typep, Typew>>(HeapSize);
  bestx = new int[n+1];
  int  i= 1;
  E = 0;
  cw = cp = 0;
  Typep  besTypep = 0;
  while(i != n+1) {
    for(int j=1; j <= m; j++) {
      Typew wt = cw+w[i][j];
      Typep ct =cp+c[i][j];
      if(ct <= cc)
        AddLiveNode(ct, wt, i+1, j);
    }
    HeapNode<Typep, Typew> N;
    try {  H->DeleteMin(N);  }
    catch(…) {  break;  }
    E = N.ptr;
    cw = N.weight;
    cp = N.profit;
    i = N.level;
  }
  if(i <= n)
    return 0;
```

```
  for(int j=n; j > 0; j--) {
    bestx[j] = E->mj;
    E = E->parent;
  }
  return cw;
}
void Mach<int, int>::AddLiveNode(int cp, int cw, int i, int j) {
  bbnode  *b = new bbnode;
  b->parent = E;
  b->mj = j;
  HeapNode<Typep, Typew> N;
  N.profit=  cp;
  N.weight = cw;
  N.level = i;
  N.ptr = b;
  H->Insert(N);
}
```

Machine()函数完成最小重量机器设计。

```
int Machine(int **c, int **w, int cc, int n, int m, int bestx[]) {
  Mach<int, int> K;
  K.c = c;
  K.w = w;
  K.cp=  0;
  K.cw=  0;
  K.cc= cc;
  K.n = n;
  K.m = m;
  K.bestx = bestx;
  int  besTypep = K.MinWeightMachine();
  for(int j=1; j<= n; j++)
    bestx[j] = K.bestx[j];
  return besTypep;
}
```

算法的主函数如下。

```
int main() {
  int  cc, n, m, **c, **w, *bestx;
  cin>>n>>m>>cc;
  Make2DArray(c, n+1, m+1);
  Make2DArray(w, n+1, m+1);
  bestx = new int[n+1];
  for(int i=1; i <= n ; i++)
    for(int j=1; j <= m; j++)
      cin >> c[i][j];
  for(i=1; i <= n; i++)
    for(int j=1; j <= m; j++)
      cin >> w[i][j];
  int  answer = Machine(c, w, cc, n, m, bestx);
```

```
    if(answer > 0) {
       cout << answer << endl;
       for(int i=1; i <= n; i++)
          cout << bestx[i] << " ";
        cout << endl;
    }
    else
       cout << "No Solution!" << endl;
    return 0;
}
```

6-5 运动员最佳配对问题。

问题描述：羽毛球队有男女运动员各 n 人。给定 2 个 $n \times n$ 矩阵 $\boldsymbol{P}$ 和 $\boldsymbol{Q}$。$P[i][j]$是男运动员 i 和女运动员 j 配对组成混合双打的男运动员竞赛优势；$Q[i][j]$是女运动员 i 和男运动员 j 配合的女运动员竞赛优势。由于技术配合和心理状态等因素影响，$P[i][j]$不一定等于 $Q[j][i]$。男运动员 i 和女运动员 j 配对组成混合双打的男女双方竞赛优势为 $P[i][j] \times Q[j][i]$。设计一个算法，计算男女运动员最佳配对法，使各组男女双方竞赛优势的总和达到最大。

算法设计：设计一个优先队列式分支限界法，对于给定的男女运动员竞赛优势，计算男女运动员最佳配对法，使各组男女双方竞赛优势的总和达到最大。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n（$1 \leqslant n \leqslant 20$）。接下来的 $2n$ 行，每行 n 个数。前 n 行是 p，后 n 行是 q。

结果输出：将计算的男女双方竞赛优势的总和的最大值输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	52
10 2 3	
2 3 4	
3 4 5	
2 2 2	
3 5 3	
4 5 1	

分析与解答：堆结点元素类型是 pref。

```
class pref{
public:
   operator int() const {  return val;  }
   int getbest();
private:
   void init();
   void Compute(int ii);
   int  s, val, *r;
};
```

解运动员最佳配对问题的优先队列式分支限界法如下。

```
void pref::init() {
   cin >> n;
   s = 0;
```

```
  r = new int[n+1];
  bestr = new int[n+1];
  Make2DArray(p, n+1, n+1);
  Make2DArray(q, n+1, n+1);
  for(int i=1; i <= n; i++)
    r[i] = i;
  for(i=1; i <= n; i++)
    for(int j=1; j <= n; j++)
      cin >> p[i][j];
  for(i=1; i <= n; i++)
    for(int j=1; j <= n; j++)
      cin >> q[i][j];
}
void pref::Compute(int ii) {
  for(int i=1,temp=0; i <= ii; i++)
    temp += p[i][r[i]]*q[r[i]][i];
  val=temp;
}
int pref::getbest() {
  MaxHeap<pref> H(HeapSize);
  pref E;
  E.init();
  while(true) {
    if(E.s == n-1) {
      E.Compute(n);
      if(E.val > best) {
        delete [] bestr;
        bestr = E.r;
        best = E.val;
      }
      else
        delete [] E.r;
    }
    else {
      for(int i=E.s+1; i <= n; i++) {
        pref N;
        N.r = new int[n+1];
        N.s = E.s+1;
        N.val = E.val;
        for(int j=1; j <= n; j++)
          N.r[j] = E.r[j];
        N.r[N.s] = E.r[i];
        N.r[i] = E.r[N.s];
        N.Compute(N.s);
        H.Insert(N);
      }
      delete [] E.r;
    }
    try { H.DeleteMax(E); }
```

```
    catch(OutOfBounds) {  return best;  }
  }
}
```

6-6　n 后问题。

问题描述：在 $n \times n$ 格的棋盘上放置彼此不受攻击的 n 个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。n 皇后问题等价于在 $n \times n$ 格的棋盘上放置 n 个皇后，任何两个皇后不放在同一行或同一列或同一斜线上。

算法设计：设计一个解 n 后问题的队列式分支限界法，计算在 $n \times n$ 个方格上放置彼此不受攻击的 n 个皇后的一个放置方案。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n。

结果输出：将计算的彼此不受攻击的 n 个皇后的一个放置方案输出到文件 output.txt。文件的第 1 行是 n 个皇后的放置方案。

输入文件示例	输出文件示例
input.txt	output.txt
5	1 3 5 2 4

分析与解答：

排列空间树中结点类型为 QNode。

```
template<class Type>
class QNode{
public:
  operator int() const {  return cd;  }
  int  *x, i;
};
```

用队列式分支限界法搜索排列树的一般算法 fifobb 如下。

```
template<class Type>
void Queen<Type>::fifobb() {
  Queue<QNode<Type>*> Q;                          // 活结点队列
  QNode<Type> *E, *N;                             // 当前扩展结点
  Init(E);
  while(true) {                                   // 搜索排列空间树
    if(Answer(E))
      Save(E);
    else
      for(int i=E->i+1; i <= n; i++) {
        NewNode(N, E, i);
        if(Constrain(N))
          Q.Add(N);
      }
    if(!Getnext(Q, E))                            // 取下一扩展结点
      break;
  }
  Output();
}
```

结点可行性判定函数 Constraint()等函数通过类 Queen 的私有函数传递。

```
template<class Type>
class Queen{
   friend int main();
private:
   void Init(QNode<Type> *&E);
   bool Answer(QNode<Type> *E);
   void Save(QNode<Type> *&E);
   void NewNode(QNode<Type> *&N, QNode<Type> *E, int i);
   bool Constrain(QNode<Type> *E);
   bool Getnext(Queue<QNode<Type>*> &Q, QNode<Type> *&E);
   void Output();
   void fifobb();
   int  n, *bestx;
   bool  found;
};
template<class Type>
void Queen<Type>::Init(QNode<Type> *&E) {
   E = new QNode<Type>;
   E->x = new int[n+1];
   for(int j=1; j <= n; j++)
      E->x[j] = j;
   E->i = 0;
   bestx = new int[n+1];
   found = false;
}
template<class Type>
bool Queen<Type>::Answer(QNode<Type> *E) {
   return E->i == n;
}
template<class Type>
void Queen<Type>::Save(QNode<Type> *&E) {
   for(int k=1; k <= n; k++)
      bestx[k] = E->x[k];
   found = true;
}
template<class Type>
void Queen<Type>::NewNode(QNode<Type> *&N, QNode<Type> *E,int i) {
   N = new QNode<Type>;
   N->x = new int[n+1];
   N->i = E->i+1;
   for(int j=1; j <= n; j++)
      N->x[j] = E->x[j];
   N->x[N->i] = E->x[i];
   N->x[i] = E->x[N->i];
}
template<class Type>
bool Queen<Type>::Constrain(QNode<Type> *E) {
   for(int j=1; j < E->i; j++)
      if((abs(E->i-j) == abs(E->x[j]-E->x[E->i])) || (E->x[j]==E->x[E->i]))
```

```
        return false;
    return true;
}
template<class Type>
bool Queen<Type>::Getnext(Queue<QNode<Type>*> &Q,QNode<Type> *&E) {
    if(found || Q.IsEmpty())
        return false;
    Q.Delete(E);
    return true;
}
template<class Type>
void Queen<Type>::Output() {
    for(int i=1; i <= n; i++)
        cout << bestx[i] << " ";
    cout << endl;
}
```

6-7 布线问题。

问题描述：假设要将一组元件安装在一块线路板上，为此需要设计一个线路板布线方案。各元件的连线数由连线矩阵 conn 给出。元件 i 和元件 j 之间的连线数为 conn(i,j)。如果将元件 i 安装在线路板上位置 r 处，而将元件 j 安装在线路板上位置 s 处，则元件 i 和元件 j 之间的距离为 dist(r,s)。确定了所给的 n 个元件的安装位置，就确定了一个布线方案。与此布线方案相应的布线成本为 $\text{dist}(r,s)\times\sum_{1\leqslant i<j\leqslant n}\text{conn}(i,j)$。试设计一个优先队列式分支限界法，找出所给 n 个元件的布线成本最小的布线方案。

算法设计：对于给定的 n 个元件，设计一个优先队列式分支限界法，计算最佳布线方案，使布线费用达到最小。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n（$1\leqslant n\leqslant 20$）。接下来的 $n-1$ 行，每行 $n-i$ 个数，表示元件 i 和元件 j 之间连线数（$1\leqslant i<j\leqslant 20$）。

结果输出：将计算的最小布线费用以及相应的最佳布线方案输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	10
2 3	1 3 2
3	

分析与解答：堆结点元素类型是 BoardNode。

```
class BoardNode {
    friend int BBArrangeBoards(int **, int, int *&);
public:
    operator int() const {  return cd;  }
    int len(int **, int ii);
private:
    int  *x, s, cd;
};
```

解布线问题的优先队列式分支限界法如下。

```
int BoardNode::len(int **conn,int ii) {
   for(int i=1,sum=0; i <= ii; i++) {
      for(int j=i+1; j <= ii; j++) {
         int  dist = x[i]>x[j] ? x[i]-x[j] : x[j]-x[i];
         sum += conn[i][j]*dist;
      }
   }
   return sum;
}
int BBArrangeBoards(int **conn, int n, int *&bestx) {
   MinHeap<BoardNode> H(HeapSize);
   BoardNode E;
   E.x = new int[n+1];
   E.s = 0;
   E.cd = 0;
   for(int i=1; i <= n; i++)
      E.x[i] = i;
   int  bestd = INT_MAX;
   bestx = 0;
   while(E.cd < bestd) {
      if(E.s == n-1) {
         int  ld = E.len(conn, n);
         if(ld < bestd) {
            delete[] bestx;
            bestx = E.x;
            bestd = ld;
         }
         else
            delete[] E.x;
         }
         else {
            for(int i=E.s+1; i <= n; i++) {
               BoardNode N;
               N.x=new int[n+1];
               N.s=E.s+1;
               for(int j=1; j <= n; j++)
                  N.x[j] = E.x[j];
               N.x[N.s] = E.x[i];
               N.x[i] = E.x[N.s];
               N.cd = N.len(conn, N.s);
               if(N.cd < bestd)
                  H.Insert(N);
               else
                  delete[] N.x;
            }
         }
         delete[] E.x;
      }
      try { H.DeleteMin(E); }
```

```
      catch(OutOfBounds) { return bestd; }
   }
   while(true) {
      delete[] E.x;
      try { H.DeleteMin(E); }
      catch(…) { break; }
   }
   return bestd;
}
```

算法的主函数如下。

```
int main() {
   cin >> n;
   p = new int[n+1];
   int **B;
   Make2DArray(B, n+1, n+1);
   for(int i =1; i <= n-1; i++)
      for(int j=i+1; j <= n; j++)
         cin >> B[i][j];
   cout << BBArrangeBoards(B, n, p) << endl;
   for(i=1; i <= n; i++)
      cout << p[i] << " ";
   cout << endl;
   return 0;
}
```

6-8 最佳调度问题。

问题描述：假设有 n 个任务由 k 个可并行工作的机器完成。完成任务 i 需要的时间为 t_i。试设计一个算法找出完成这 n 个任务的最佳调度，使得完成全部任务的时间最早。

算法设计：对任意给定的整数 n 和 k，以及完成任务 i 需要的时间为 t_i（i=1, 2, …, n）。设计一个优先队列式分支限界法，计算完成这 n 个任务的最佳调度。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k。第 2 行的 n 个正整数是完成 n 个任务需要的时间。

结果输出：将计算的完成全部任务的最早时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
7 3	17
2 14 4 16 6 5 3	

分析与解答：堆结点元素类型是 Machine。

```
class Machine{
   friend int BBMachine();
public:
   operator int() const { return len[i]; }
private:
   int i, dep, *len;
};
```

解最佳调度问题的优先队列式分支限界法如下。

```
int BBMachine() {
   MinHeap<Machine> H(HeapSize);
   Machine  E;
   E.len = new int[k];
   E.i = 0;
   E.dep = 0;
   for(int i=0; i < k; i++)
      E.len[i] = 0;
   int  dep = 0;
   while(true) {
      if(dep == n) {
         int  tmp = comp(E.len);
         if(tmp < best)
            best = tmp;
         delete[] E.len;
      }
      else {
         for(int i=0; i < k; i++) {
            Machine  N;
            N.len = new int[k];
            N.i = i;
            N.dep = dep;
            for(int j=0; j < k; j++)
               N.len[j] = E.len[j];
            N.len[i] += t[dep];
            if(N.len[i] < best)
               H.Insert(N);
            else
               delete[] N.len;
         }
         delete[] E.len;
      }
      try {  H.DeleteMin(E);  }
      catch(OutOfBounds) {  return best;  }
      dep = E.dep+1;
   }
   while(true) {
     delete[] E.len;
     try {  H.DeleteMin(E);  }
     catch(…) {  break;  }
   }
   return best;
}
```

算法的主函数如下。

```
int main() {
   readin();
```

```
    cout << BBMachine() << endl;
    return 0;
}
```

readin()函数读入初始数据并进行初始化计算。

```
void readin() {
    cin >> n >> k;
    len = new int[k];
    t.resize(n);
    for(int i=0; i < n; i++)
        cin >> t[i];
    for(i=0; i < k; i++)
        len[i] = 0;
    best = bound();
}
```

bound()函数计算上界值。

```
int bound() {
    sort(t.begin(), t.end(), greater<int>());
    for(int i=0; i < n; i++)
        len[ind(len)] += t[i];
    return comp(len);
}
int ind(int *len) {
    int  tmp = 0;
    for(int i=1; i < k; i++)
        if(len[i] < len[tmp])
            tmp = i;
    return tmp;
}
int comp(int *len) {
    int  tmp = 0;
    for(int i=0; i < k; i++)
        if(len[i] > tmp)
            tmp = len[i];
    return tmp;
}
```

6-9　无优先级运算问题。

问题描述：给定 n 个正整数和 4 个运算符+、−、*、/，且运算符无优先级，如 2+3*5=25。对于任意给定的整数 m，试设计一个算法，用以上给出的 n 个数和 4 个运算符，产生整数 m，且用的运算次数最少。给出的 n 个数中每个数最多只能用一次，但每种运算符可以任意使用。

算法设计：对于给定的 n 个正整数，设计一个优先队列式分支限界法，用最少的无优先级运算次数产生整数 m。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m。第 2 行是给定的用于运算的 n 个正整数。

结果输出：将计算的产生整数 m 的最少无优先级运算次数以及最优无优先级运算表达

式输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 25	2
5 2 3 6 7	2+3*5

分析与解答：堆结点元素类型是 Arit。

```
class Arit {
public:
   Arit(int n);
   Arit() { };
   operator int() const {  return dep;  }
   int  dep, *num, *oper, *flag;
};
Arit::Arit(int n) {
   num = new int[n];
   oper = new int[n];
   flag = new int[n];
   for(int i=0; i < n; i++)
      num[i] = oper[i] = flag[i] = 0;
   dep =0;
}
```

解无优先级运算问题的优先队列式分支限界法如下。

```
int BBArit() {
   MinHeap<Arit> H(HeapSize);
   Arit E(n);
   while(true) {
      if(found(E)) {
         out(E);
         return 1;
      }
      else {
         for(int i=0; i < n; i++) {
            if(!E.flag[i]) {
               for(int j=0; j < 4; j++) {
                  Arit N(n);
               newnode(N, E, i, j, dep);
               H.Insert(N);
               }
            }
      }
      try {  H.DeleteMin(E);  }
      catch(OutOfBounds) {  return 0;  }
      dep = E.dep+1;
   }
}
```

其中，found()函数判定是否找到解，out()函数输出解，newnode()函数生成新结点。

```
bool found(Arit E) {
   int x=E.num[0];
   for(int i=0; i < E.dep; i++) {
      switch(E.oper[i]) {
         case 0:    x += E.num[i+1];  break;
         case 1:    x = E.num[i+1];  break;
         case 2:    x *= E.num[i+1];  break;
         case 3:    x /= E.num[i+1];  break;
      }
   }
   return (x == m);
}
void out(Arit E) {
   cout << E.dep << endl;
   for(int i=0; i < E.dep; i++) {
      cout<<E.num[i];
      switch (E.oper[i]) {
         case 0:    cout<<"+";  break;
         case 1:    cout<<"_";  break;
         case 2:    cout<<"*";  break;
         case 3:    cout<<"/";   break;
      }
   }
   cout << E.num[E.dep] << endl;
}
void newnode(Arit &N, Arit E, int i, int j, int dep) {
   for(int k=0; k < n; k++) {
      N.num[k] = E.num[k];
      N.oper[k]=  E.oper[k];
      N.flag[k] = E.flag[k];
   }
   N.dep = dep;
   N.oper[dep] = j;
   N.num[dep] = a[i];
   N.flag[i] = 1;
}
```

算法的主函数如下。

```
int main() {
   readin();
   if(!BBArit())
      cout << "No Solution!" << endl;
   return 0;
}
void readin() {
   cin >> n >> m;
   a = new int[n];
   for(int i=0; i < n; i++)
      cin>>a[i];
```

}

6-10 世界名画陈列馆问题。

问题描述：世界名画陈列馆由 $m \times n$ 个排列成矩形阵列的陈列室组成。为了防止名画被盗，需要在陈列室中设置警卫机器人哨位。除了监视所在的陈列室，每个警卫机器人还可以监视与它所在的陈列室相邻的上、下、左、右 4 个陈列室。试设计一个安排警卫机器人哨位的算法，使名画陈列馆中每个陈列室都在警卫机器人的监视下，且所用的警卫机器人数最少。

算法设计：设计一个优先队列式分支限界法，计算警卫机器人的最佳哨位安排，使名画陈列馆中每个陈列室都在警卫机器人的监视下，且所用的警卫机器人数最少。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n（$1 \leqslant m, n \leqslant 20$）。

结果输出：将计算的警卫机器人数及其最佳哨位安排输出到文件 output.txt。文件的第 1 行是警卫机器人数；接下来的 m 行中每行 n 个数，0 表示无哨位，1 表示有哨位。

输入文件示例	输出文件示例
input.txt	output.txt
4 4	4
	0 0 1 0
	1 0 0 0
	0 0 0 1
	0 1 0 0

分析与解答：堆结点元素类型是 HeapNode。

```
class HeapNode{
public:
   HeapNode(int n, int m);
   HeapNode() { };
   operator int() const {  return k;  }
   void out(int n, int m);
   int  i, j, k, t;
   int  **x, **y;
};
HeapNode::HeapNode(int n, int m) {
   Make2DArray(x, n+2, m+2);
   Make2DArray(y, n+2, m+2);
   for(int a=0; a <= n+1; a++) {
      for(int b=0; b <= m+1; b++) {
         x[a][b] = 0;
         y[a][b] = 0;
      }
   }
   for(a=0; a <= m+1; a++) {
      y[0][a] = 1;
      y[n+1][a] = 1;
   }
   for(a=0; a <= n+1; a++) {
      y[a][0] = 1;
      y[a][m+1] = 1;
   }
```

```
    i = 1;
    j = 1;
    k = t = 0;
}
```

解世界名画陈列馆问题的优先队列式分支限界法如下。

```
class Robot{
    friend int main();
private:
  void copy(int **x, int **y);
  void change(MinHeap<HeapNode> &H, HeapNode E, int i, int j);
  void init();
  void output();
  void pqbb();
  void compute();
  int  **bestx;
  int  n, m, best;
  bool  p;
};
void Robot::pqbb() {
    MinHeap<HeapNode> H(HeapSize);
    HeapNode E(n, m);
    while(true) {
       int  i = E.i, j = E.j, k = E.k, t = E.t;
       if(t == n*m) {
          best=k;
          copy(bestx, E.x);
          return;
       }
       else {
          if(i < n)
             change(H, E, i+1, j);
          if((j < m) && ((E.y[i][j+1] == 0) || (E.y[i][j+2] == 0)))
             change(H, E, i, j+1);
          if(((E.y[i+1][j]= = 0) && (E.y[i][j+1] == 0)))
             change(H, E, i, j);
       }
       try {  H.DeleteMin(E);  }
       catch(OutOfBounds) {  break;  }
    }
}
void Robot::change(MinHeap<HeapNode> &H, HeapNode E, int i, int j) {
    HeapNode N(n, m);
    N.i = E.i;
    N.j = E.j;
    N.k = E.k+1;
    N.t = E.t;
    for(int a=0; a <= n+1; a++) {
       for(int b=0; b <= m+1; b++) {
```

```
            N.x[a][b] = E.x[a][b];
            N.y[a][b] = E.y[a][b];
        }
    }
    N.x[i][j] = 1;
    for(int s=1; s <=5 ; s++) {
        int  p = i+d[s][1], q = j+d[s][2];
        N.y[p][q]++;
        if(N.y[p][q] == 1)
            N.t++;
    }
    while(!(N.y[N.i][N.j] == 0 || N.i>n)) {
        N.j++;
        if(N.j > m) {
            N.i++;
            N.j=1;
        }
    }
    H.Insert(N);
}
void Robot::copy(int **x,int **y) {
    for(int i=0; i <= n; i++)
        for(int j=0; j <= m; j++)
            x[i][j] = y[i][j];
}
```

用 compute()函数完成计算。

```
void Robot::compute() {
    Make2DArray(bestx, n+2, m+2);
    if(n == 1 && m == 1) {
        cout << 1 << endl << 1 << endl;
        return;
    }
    pqbb();
    output();
}
```

算法的主函数如下。

```
int main() {
    Robot X;
    X.init();
    X.compute();
    return 0;
}
void Robot::init() {
    cin >> n >> m;
    p = false;
    if(n < m) {
        Swap(n, m);
```

```
        p = true;
    }
}
```

6-11　子集空间树问题。

问题描述：试设计一个用队列式分支限界法搜索子集空间树的函数，其参数包括结点可行性判定函数和上界函数等必要的函数，并将此函数用于解装载问题。

装载问题描述如下：有一批共 n 个集装箱要装上一艘载重量为 c 的轮船，其中集装箱 i 的重量为 w_i。找出一种最优装载方案，将轮船尽可能装满，即在装载体积不受限制的情况下，将尽可能重的集装箱装上轮船。

算法设计：对于给定的 n 个集装箱和轮船的载重量，计算最优装载方案。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 c。n 是集装箱数，c 是轮船的载重量。接下来的 1 行中有 n 个正整数，表示集装箱的重量。

结果输出：将计算的最大装载重量输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 10	10
7 2 6 5 4	

分析与解答：用队列式分支限界法搜索子集空间树的一般算法 fifobb 如下。

```
template<class Type>
void Loading<Type>::fifobb() {
   Queue<QNode<Type>*> Q;                    // 活结点队列
   int  i = 1;                               // 当前扩展结点所处的层
   QNode<Type>  *E = 0,                      // 当前扩展结点
                *bestE;                      // 当前最优扩展结点
   while(true) {                             // 搜索子集空间树
      if(Constraint(E, bestE, i))            // 检查左儿子结点
         EnQueue(Q, E, i, 1);
      if(Bound(E, i))                        // 检查右儿子结点
         EnQueue(Q, E, i, 0);
      if(!Getnext(Q, E, i))                  // 取下一扩展结点
         break;
   }
   Solution(bestE);                          // 构造最优解
   Output();
}
```

其中，子集树的结点类型为 QNode：

```
template<class Type>
class QNode {
public:
   QNode  *parent;
   int  i, LChild;
   Type  weight;
};
```

结点可行性判定函数 Constraint()和上界函数 Bound()等必要的函数通过类 Loading 的私

有函数传递。

```
template<class Type>
class Loading {
   friend MaxLoading(Type *, Type, int, int *);
private:
   void maxLoading(int i);
   bool Constraint(QNode<Type> *E, QNode<Type> *&bestE, int i);
   bool Bound(QNode<Type> *E, int i);
   void EnQueue(Queue<QNode<Type>*> &Q, QNode<Type> *E, int i, int ch);
   bool Getnext(Queue<QNode<Type>*> &Q, QNode<Type> *&E, int &i);
   void Solution(QNode<Type> *bestE);
   void Output();
   void fifobb();
   int  n, *bestx;
   Type  *w, c, r, Ew, wt, bestw;
};
template<class Type>
bool Loading<Type>::Constraint(QNode<Type> *E, QNode<Type>*&bestE, int i) {
   wt = Ew+w[i];
   if(wt <= c) {
      if(wt > bestw) {
         bestw = wt;
         bestE = E;
      }
      if(i < n)
         return true;
   }
   return false;
}
template<class Type>
bool Loading<Type>::Bound(QNode<Type> *E, int i) {
   return (Ew+r > bestw && i < n);
}
template<class Type>
void Loading<Type>::EnQueue(Queue<QNode<Type>*> &Q, QNode<Type> *E, int i, int ch) {
   QNode<Type>*b;                                    // 将活结点加入到活结点队列Q中
   b = new QNode<Type>;
   if(ch)
      b->weight = wt;
   else
      b->weight = Ew;
   b->parent = E;
   b->i = i;
   b->LChild = ch;
   Q.Add(b);
}
template<class Type>
bool Loading<Type>::Getnext(Queue<QNode<Type>*> &Q, QNode<Type> *&E, int &i) {
```

```
    if(Q.IsEmpty())
        return false;
    Q.Delete(E);                                    // 取下一扩展结点
    Ew = E->weight;                                 // 新扩展结点所相应的载重量
    if(i < E->i)
        r -= w[i];                                  // 剩余集装箱重量
    i = E->i+1;
    return true;
}
template<class Type>
void Loading<Type>::Solution(QNode<Type> *bestE) {          // 构造最优解
    int  besti = bestE->i+1;
    for(int j=n; j > besti; j--)
        bestx[j] = 0;
    bestx[besti] = 1;
    for(j=besti-1; j > 0; j--) {
        bestx[j] = bestE->LChild;
        bestE = bestE->parent;
    }
}
template<class Type>
void Loading<Type>::Output() {
    cout << bestw << endl;
    for(int i=1; i <= n; i++)
        cout << bestx[i] << " ";
    cout << endl;
}
```

MaxLoading()函数实现装载问题的队列式分支限界法。

```
template<class Type>
Type MaxLoading(Type w[], Type c, int n, int bestx[]) {
    Loading<Type> X;
    X.c = c;
    X.n = n;
    X.Ew = 0;
    X.bestw = 0;
    X.r = 0;
    X.w = w;
    X.bestx = bestx;
    for(int j=2; j <= n; j++)
        X.r += w[j];
    X.fifobb();
    return X.bestw;
}
```

算法的主函数如下。

```
int main() {
    int  n, c;
    cin >> n >> c;
```

```
    int  *w = new int[n+1];
    int  *x = new int[n+1];
    for(int i=1; i <= n; i++)
       cin >> w[i];
    cout << "Max loading is" << MaxLoading(w, c, n, x) << endl;
    cout << "Loading vector is" << endl;
    for(i=1; i <= n; i++)
       cout << x[i] << " ";
    cout << endl;
    return 0;
}
```

6-12　排列空间树问题。

问题描述：试设计一个用队列式分支限界法搜索排列空间树的函数，其参数包括结点可行性判定函数和上界函数等必要的函数，并将此函数用于解电路板排列问题。

电路板排列问题是大规模电子系统设计中提出的实际问题。该问题的提法是，将 n 块电路板以最佳排列方案插入带有 n 个插槽的机箱中。n 块电路板的不同的排列方式对应不同的电路板插入方案。设 $B=\{1, 2, \cdots, n\}$ 是 n 块电路板的集合，集合 $L=\{N_1, N_2, \cdots, N_m\}$ 是 n 块电路板的 m 个连接块，每个连接块 N_i 是 B 的一个子集，且 N_i 中的电路板用同一根导线连接在一起。在设计机箱时，插槽一侧的布线间隙由电路板排列的密度确定。因此，电路板排列问题要求对于给定电路板连接条件（连接块），确定电路板的最佳排列，使其具有最小密度。

算法设计：对于给定电路板连接条件，计算电路板的最佳排列，使其具有最小密度。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是 2 个正整数 n 和 m，表示有 n 块电路板和 m 个连接块。接下来的 n 行，每行有 m 个数，第 i 行的第 j 个数 a[i][j]=1 表示第 i 块电路板在第 j 个连接块中，否则第 i 块电路板不在第 j 个连接块中。

结果输出：将计算的最小密度和电路板的最佳排列输出到文件 output.txt。文件的第 1 行是最小密度，第 2 行是电路板的最佳排列。

输入文件示例	输出文件示例
input.txt	output.txt
8 5	4
1 1 1 1 1	2 3 4 5 1 6 7 8
0 1 0 1 0	
0 1 1 1 0	
1 0 1 1 0	
1 0 1 0 0	
1 1 0 1 0	
0 0 0 0 1	
0 1 0 0 1	

分析与解答：用队列式分支限界法搜索排列空间树的一般算法 fifobb 如下。

```
template<class Type>
void Board<Type>::fifobb() {
   Queue<QNode<Type>*> Q;                    // 活结点队列
   QNode<Type>  *E, *N;                      // 当前扩展结点
   Init(E);
   while(true) {                             // 搜索排列空间树
```

```
    if(Answer(E))
      Save(E);
    else {
      for(int i=E->i+1; i <= n; i++) {
        NewNode(N, E, i);
        if(Constrain(N) && Bound(N))
          EnQueue(Q, N, E, i);
        else
          DelNode(N);
      }
    }
    if(!Getnext(Q, E))                                    // 取下一扩展结点
      break;
  }
  Output();
}
```

其中，排列树的结点类型为 QNode。

```
template<class Type>
class QNode{
public:
  operator int() const {  return cd;  }
  int  *x, i;
  Type  cd, *now;
};
```

结点可行性判定函数 Constraint()和上界函数 Bound()等必要的函数通过类 Board 的私有函数传递。

```
template<class Type>
class Board {
  friend int main();
private:
  void Init(QNode<Type> *&E);
  bool Answer(QNode<Type> *E);
  void Save(QNode<Type> *&E);
  void NewNode(QNode<Type> *&N, QNode<Type> *E, int i);
  bool Constrain(QNode<Type> *E);
  bool Bound(QNode<Type> *E);
  void EnQueue(Queue<QNode<Type>*> &Q, QNode<Type> *N, QNode<Type> *E, int i);
  bool Getnext(Queue<QNode<Type>*> &Q, QNode<Type> *&E);
  void DelNode(QNode<Type> *&E);
  void Output();
  void fifobb();
  int  n, m, *bestx;
  Type  **B, bestd, *total;
};
template<class Type>
void Board<Type>::Init(QNode<Type> *&E) {                 // 结点初始化
  E = new QNode<Type>;
```

```
  E->x = new int[n+1];
  E->i = 0;
  E->cd = 0;
  E->now = new Type[m+1];
  total = new Type[m+1];
  for(int i=1; i <= m; i++) {
     total[i] = 0;
     E->now[i] = 0;
  }
  for(i=1; i <= n; i++) {
     E->x[i] = i;
     for(int j=1; j <= m; j++)
        total[j] += B[i][j];
  }
  bestd = m+1;
}
template<class Type>
bool Board<Type>::Answer(QNode<Type> *E) {                          // 叶结点判定
  return E->i == n1;
}
template<class Type>
void Board<Type>::Save(QNode<Type> *&E) {                           // 保存最优解
  int  ld = 0;                                                      // 最后一块电路板的密度
  for(int j=1; j <= m; j++)
     ld += B[E->x[n]][j];
  if(ld < bestd && E->cd < bestd) {
     bestd = max(ld, E->cd);
     for(int k=0; k <= n; k++)
        bestx[k] = E->x[k];
  }
  else
     delete[] E->x;
  delete[] E->now;
}
template<class Type>
void Board<Type>::NewNode(QNode<Type> *&N, QNode<Type> *E, int i) {  // 产生新结点
  N = new QNode<Type>;
  N->now = new Type[m+1];
  for(int j=1; j <= m; j++)
     N->now[j] = E->now[j]+B[E->x[i]][j];
  int  ld = 0;                                                      // 新插入电路板的密度
  for(j=1; j <= m; j++)
     if(N->now[j] > 0 && total[j] != N->now[j])
        ld++;
  N->cd = max(ld, E->cd);
}
template<class Type>
bool Board<Type>::Constrain(QNode<Type> *E) {                       // 可行性约束
  return true;
```

```
}
template<class Type>
bool Board<Type>::Bound(QNode<Type> *E) {                          // 边界约束
    return E->cd<bestd;
}
template<class Type>
void Board<Type>::EnQueue(Queue<QNode<Type>*> &Q, QNode<Type> *N, QNode<Type> *E, int i) {
  N->x = new int[n+1];                                             // 结点入队列
  N->i = E->i+1;
  for(int j=1; j <= n; j++)
     N->x[j] = E->x[j];
  N->x[N>i] = E->x[i];
  N->x[i] = E->x[N>i];
  Q.Add(N);
}
template<class Type>
void Board<Type>::DelNode(QNode<Type> *&E) {                       // 删除结点
  delete[] E->now;
}
template<class Type>
bool Board<Type>::Getnext(Queue<QNode<Type>*> &Q, QNode<Type> *&E) {  // 取队列中下一结点
  if(Q.IsEmpty())
     return false;
  Q.Delete(E);
  return true;
}
template<class Type>
void Board<Type>::Output() {                                       // 输出最优解
  cout << bestd << endl;
  for(int i=1; i <= n; i++)
     cout << bestx[i] << " ";
  cout << endl;
}
```

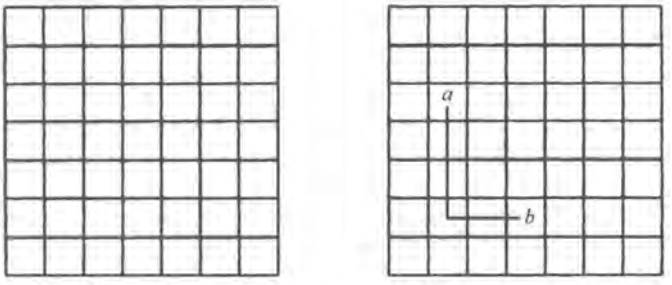
6-13　一般解空间的队列式分支限界法。

问题描述：试设计一个用队列式分支限界法搜索一般解空间的函数，其参数包括结点可行性判定函数和上界函数等必要的函数，并将此函数用于解布线问题。

印制电路板将布线区域划分成 $n \times m$ 个方格阵列（见图 6-3(a)）。精确的电路布线问题要求确定连接方格 a 的中点到方格 b 的中点的最短布线方案。在布线时，电路只能沿直线或直角布线（见图 6-3(b)）。为了避免线路相交，已布线了的方格做了封锁标记，其他线路不允许穿过被封锁的方格。

算法设计：对于给定的布线区域，计算最短布线方案。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n、m、k，分别表示布线区域方格阵列的行数、列数和封闭的方格数。接下来的 k 行中，每行 2 个正整数，表示被封闭的方格所在的行号和列号。最后的 2 行，每行也有 2 个正整数，分别表示开始布线的方格(p, q)和结束布线的方格(r, s)。



(a) 布线区域　　(b) 沿直线或直角布线

图 6-3　印制电路板问题

结果输出：将计算的最短布线长度和最短布线方案输出到文件 output.txt。文件的第 1 行是最短布线长度。从第 2 行起，每行 2 个正整数，表示布线经过的方格坐标。如果无法布线，则输出“No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
8 8 3	11
3 3	2 1
4 5	3 1
6 6	4 1
2 1	5 1
7 7	6 1
	7 1
	7 2
	7 3
	7 4
	7 5
	7 6
	7 7

分析与解答：用队列式分支限界法搜索一般解空间的算法 fifobb 如下。

```
template<class Type>
void Maze<Type>::fifobb() {
  Queue<QNode<Type>*> Q;                    // 活结点队列
   QNode<Type> *E,*N;                       // 当前扩展结点
   Init(E);
   while(true) {                            // 搜索一般解空间树
      if(Answer(E))
         Save(E);
      else {
         for(int i=f(n,E); i <= g(n,E); i++) {
            NewNode(N, E, i);
            if(Constrain(N) && Bound(N))
               EnQueue(Q, N, E, i);
            else
               DelNode(N);
         }
      }
```

```
    if(!Getnext(Q,E))                    // 取下一扩展结点
      break;
  }
  Output();
}
```

其中，解空间树的结点类型为 QNode。

```
template<class Type>
class QNode {
public:
  operator int() const {  return row;  }
  Type row, col;
};
```

结点可行性判定函数 Constraint()和上界函数 Bound()等必要的函数通过类 Maze 的私有函数传递。

```
template<class Type>
class Maze {
  friend int main();
private:
  void Init(QNode<Type>*&E);
  bool Answer(QNode<Type>*E);
  void Save(QNode<Type>*&E);
  int f(int n, QNode<Type>*E);
  int g(int n, QNode<Type>*E);
  void NewNode(QNode<Type>*&N, QNode<Type>*E, int i);
  bool Constrain(QNode<Type>*E);
  bool Bound(QNode<Type>*E);
  void EnQueue(Queue<QNode<Type>*> &Q, QNode<Type>*N, QNode<Type>*E, int i);
  bool Getnext(Queue<QNode<Type>*> &Q, QNode<Type>*&E);
  void DelNode(QNode<Type>*&E);
  void Output();
  void fifobb();
  int  n, m;
  bool  found;
  Type  **grid, PathLen;
  QNode<Type> start, finish, offset[4], *path;
};
template<class Type>
void Maze<Type>::Init(QNode<Type> *&E) {                    //设置方格阵列"围墙"
  for(int i=0; i <= m+1; i++)
    grid[0][i] = grid[n+1][i] = 1;                          // 顶部和底部
  for(i=0; i <= n+1; i++)
     grid[i][0] = grid[i][m+1] = 1;                         // 左翼和右翼
  // 初始化相对位移
  offset[0].row = 0;   offset[0].col = 1;                   // 右
  offset[1].row = 1;   offset[1].col = 0;                   // 下
  offset[2].row = 0;   offset[2].col = 1;                   // 左
  offset[3].row = 1;   offset[3].col = 0;                   // 上
```

```
  E = new QNode<Type>;
  E->row = start.row;
  E->col = start.col;
  grid[start.row][start.col] = 2;
}
template<class Type>
bool Maze<Type>::Answer(QNode<Type>*E) {  return false;  }
template<class Type>
void Maze<Type>::Save(QNode<Type>*&E) {  }
template<class Type>
int Maze<Type>::f(int n,QNode<Type>*E) {  return 1;  }
template<class Type>
int Maze<Type>::g(int n,QNode<Type>*E) {  return 4;  }
template<class Type>
void Maze<Type>::NewNode(QNode<Type>*&N, QNode<Type>*E, int i) {
  N = new QNode<Type>;
  N->row = E->row+offset[i-1].row;
  N->col = E->col+offset[i-1].col;
}
template<class Type>
bool Maze<Type>::Constrain(QNode<Type>*E) {
  return grid[E->row][E>col] == 0;
}
template<class Type>
bool Maze<Type>::Bound(QNode<Type>*E) {
  if(!found)
    found = E->row == finish.row && E->col == finish.col;
  return true;
}
template<class Type>
void Maze<Type>::EnQueue(Queue<QNode<Type>*> &Q, QNode<Type>*N, QNode<Type>*E, int i) {
  grid[N->row][N>col] = grid[E->row][E->col]+1;
  if(!found)
    Q.Add(N);
}
template<class Type>
void Maze<Type>::DelNode(QNode<Type>*&E) {  }
template<class Type>
bool Maze<Type>::Getnext(Queue<QNode<Type>*> &Q, QNode<Type>*&E) {
  if(found || Q.IsEmpty())
    return false;
  Q.Delete(E);
  return true;
}
template<class Type>
void Maze<Type>::Output() {
  if(!found) {
    cout << "No path!" << endl;
    return;
```

```
    }
    PathLen = grid[finish.row][finish.col]-2;          // 构造最优解
    path = new QNode<Type> [PathLen];
    QNode<Type> N, E= finish;                          // 从目标位置finish开始向起始位置回溯
    for(int j=PathLen-1; j >= 0; j--) {
      path[j] = E;
      for(int i=0; i < 4; i++) {                       // 找前驱位置
        N.row = E.row+offset[i].row;
        N.col = E.col+offset[i].col;
        if(grid[N.row][N.col] == j+2)
          break;
      }
      E = N;                                                    // 向前移动
    }
    cout << PathLen << endl;
    cout << start.row << " " << start.col << endl;
    for(j=0; j < PathLen; j++)
      cout << path[j].row << " " << path[j].col << endl;
}
```

实现算法的主函数如下。

```
int main() {
    int  n, m, a, b, x;
    Maze<int> X;
    cin >> n >> m >> x;
    X.n = n;
    X.m = m;
    X.found = false;
    Make2DArray(X.grid, n+2, m+2);
    for(a=0; a < n+2; a++)
      for(b=0; b < m+2; b++)
        X.grid[a][b] = 0;
    for(x=x; x >= 1; x--) {
      cin >> a >> b;
      X.grid[a][b] = 1;
    }
     cin >> X.start.row> > X.start.col >> X.finish.row >> X.finish.col;
    X.fifobb();
    return 0;
}
```

6-14　子集空间树问题。

问题描述：试设计一个用优先队列式分支限界法搜索子集空间树的函数。该函数的参数包括结点可行性判定函数和上界函数等必要的函数，并将此函数用于解 0-1 背包问题。

0-1 背包问题描述如下：给定 n 种物品和一背包。物品 i 的重量是 w_i，其价值为 v_i，背包的容量为 C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大，在选择装入背包的物品时，对每种物品 i 只有两种选择，即装入背包或不装入背包。不能将物品 i 装入背包多次，也不能只装入部分的物品 i。

0-1 背包问题形式化描述如下：给定 $C>0$，$w_i>0$，$v_i>0$（$1\leqslant i\leqslant n$），要求 n 元 0-1 向量$(x_1, x_2, \cdots, x_n)$，$x_i\in\{0, 1\}$（$1\leqslant i\leqslant n$），使得 $\sum_{i=1}^{n} w_i x_i \leqslant C$，而且 $\sum_{i=1}^{n} v_i x_i$ 达到最大。因此，0-1 背包问题是一个特殊的整数规划问题。

$$\begin{cases} \max \sum_{i=1}^{n} v_i x_i \\ \max \sum_{i=1}^{n} w_i x_i \leqslant C \end{cases} \qquad x_i \in \{0,1\}, 1\leqslant i\leqslant n$$

算法设计：对于给定的 n 种物品的重量和价值，以及背包的容量，计算可装入背包的最大价值。

数据输入：由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 C，分别表示有 n 种物品，背包的容量为 C。接下来的 2 行中，每行有 n 个数，分别表示各物品的价值和重量。

结果输出：将最佳装包方案及其最大价值输出到文件 output.txt。文件的第 1 行是最大价值，第 2 行是最佳装包方案。

输入文件示例	输出文件示例
input.txt	output.txt
5 10	15
6 3 5 4 6	1 1 0 0 1
2 2 6 5 4	

分析与解答：用优先队列式分支限界法搜索子集树的一般算法 pqbb 如下。

```
template<class Typew, class Typep>
void Knap<Typew, Typep>::pqbb() {
   NewHeap(H);
   NewHeapNode(N);
   int  i = 1;
   Init();
   while(true) {                                    // 搜索子集空间树
      if(Constraint(N, i))                          // 检查左儿子结点
         AddLiveNode(N, i, 1);
      if(Bound(N, i))                               // 检查右儿子结点
         AddLiveNode(N, i, 0);
      if(!Getnext(N, i))
         break;                                     // 取下一扩展结点
   }
   Solution(N);                                     // 构造最优解
   Output();
}
```

其中，堆结点类型为 HeapNode。

```
template<class Typew, class Typep>
class HeapNode {
   friend Knap<Typew, Typep>;
public:
```

```
  operator Tp() const {  return uprofit;  }
private:
  Typep  uprofit, profit;
  Typew  weight;
  int  level, *x;
};
```

结点可行性判定函数 Constraint()和上界函数 Bound()等通过类 Knap 的私有函数传递。

```
template<class Typew, class Typep>
class Knap {
  friend Typep Knapsack(Typep *, Typew *, Typew, int, int *);
public:
  void pqbb();
private:
  MaxHeap<HeapNode<Typep, Typew>> *H;
  HeapNode<Typep, Typew> N;
  void NewHeap(MaxHeap<HeapNode<Typep, Typew>> *&H);
  void NewHeapNode(HeapNode<Typep, Typew> &N);
  void Init();
  Typep Bd(int i);
  bool Constraint(HeapNode<Typep, Typew> N, int i);
  bool Bound(HeapNode<Typep, Typew> N, int i);
  void AddLiveNode(HeapNode<Typep, Typew> &N, int i, bool ch);
  bool Getnext(HeapNode<Typep, Typew> &N, int &i);
  void Solution(HeapNode<Typep, Typew> N);
  void Output();
  Typew  c, cw, wt, *w;
  Typep  cp, up, bestp, *p;
  int  n, *bestx;
};
template<class Typew, class Typep>
void Knap<Typew, Typep>::NewHeap(MaxHeap<HeapNode<Typep, Typew>> *&H) {
  H = new MaxHeap<HeapNode<Typep,Typew> >(10000);
}
template<class Typew, class Typep>
void Knap<Typew, Typep>::NewHeapNode(HeapNode<Typep, Typew> &N) {
  N.x = new int[n+1];
}
template<class Typew, class Typep>
void Knap<Typew, Typep>::Init() {                          // 初始化
  bestx = new int [n+1];
  for(int k=0; k <= n; k++)
    bestx[k] = 0;
  cw = cp = 0;
  bestp = 0;
  up = Bd(1);
}
template<class Typew, class Typep>
Typep Knap<Typew, Typep>::Bd(int i) {                      // 计算结点所相应价值的上界
```

```
  Typew cleft = c-cw;                                    // 剩余容量
  Typep b = cp;                                          // 价值上界
  while(i <= n && w[i] <= cleft) {                       // 以物品单位重量价值递减序装填剩余容量
    cleft = w[i];
    b += p[i];
    i++;
  }
  if(i <= n)                                             // 装填剩余容量装满背包
    b += p[i]*cleft/w[i];
  return b;
}
template<class Typew, class Typep>
bool Knap<Typew, Typep>::Constraint(HeapNode<Typep, Typew> N, int i) {
  wt = cw + w[i];
  if(wt <= c) {
    if(cp+p[i] > bestp)
      bestp = cp+p[i];
    return true;
  }
  return false;
}
template<class Typew, class Typep>
bool Knap<Typew, Typep>::Bound(HeapNode<Typep, Typew> N, int i) {
  up = Bd(i+1);
  return up >= bestp;
}
template<class Typep, class Typew>
void Knap<Typep, Typew>::AddLiveNode(HeapNode<Typep, Typew> &N, int i, bool ch) {
  N.x = new int[n+1];                                    // 将一个新结点插入到最大堆H中
  for(int j=0; j <= n; j++)
    N.x[j] = bestx[j];
  N.uprofit = up;
  N.profit = cp;
  N.weight = cw;
  if(ch) {
    N.weight = cw+w[i];
    N.profit = cp+p[i];
  }
  N.level = i+1;
  N.x[i] = ch;
  H->Insert(N);
}
template<class Typew, class Typep>
bool Knap<Typew, Typep>::Getnext(HeapNode<Typep, Typew> &N, int &i) {
  H->DeleteMax(N);
  cw = N.weight;
  cp = N.profit;
  up = N.uprofit;
  i = N.level;
```

```
   for(int j=0; j <= n; j++)
      bestx[j] = N.x[j];
   if(i > n)
      return false;
   else
      return true;
}
template<class Typep, class Typew>
void Knap<Typep, Typew>::Solution(HeapNode<Typep, Typew> N) {          // 构造当前最优解
   for(int j=0; j <= n; j++)
      bestx[j] = N.x[j];
   while(true) {                                                       // 释放堆中所有结点
      try { H->DeleteMax(N); }
      catch(OutOfBounds) { break; }
   }
}
template<class Typep, class Typew>
void Knap<Typep, Typew>::Output() {
   cout << cp << endl;
   for(int j=1; j <= n; j++)
      cout << bestx[j] << " ";
   cout << endl;
}
```

Knapsack()函数实现 0-1 背包问题的优先队列式分支限界法。

```
template<class Typew, class Typep>
Typep Knapsack(Typep *p, Typew *w, Typew c, int n, int *bestx) { // 返回最大价值，bestx 返回最优解
   // 初始化
   Typew  W = 0;                                                   // 装包物品重量
   Typep  P= 0;                                                    // 装包物品价值
   Object  *Q = new Object[n];                                     // 定义依单位重量价值排序的物品数组
   for(int i=1; i <= n; i++) {                                     // 单位重量价值数组
      Q[i-1].ID = i;
      Q[i-1].d = 1.0*p[i]/w[i];
      P += p[i];
      W += w[i];
   }
   if(W <= c)
      return P;                                                    // 所有物品装包
   MergeSort(Q, n);                                                // 依单位重量价值排序
   Knap<Typew, Typep> K;                                           // 创建类 Knap 的数据成员
   K.p = new Typep [n+1];
   K.w = new Typew [n+1];
   for(i=1; i <= n; i++) {
      K.p[i] = p[Q[i-1].ID];
      K.w[i] = w[Q[i-1].ID];
   }
   K.cp = 0;
   K.cw = 0;
```

```
    K.c = c;
    K.n = n;
    K.pqbb();                                          // 调用 pqbb()求问题的最优解
    bestp = K.bestp;
    for(int j =1; j <= n; j++)
      bestx[Q[j-1].ID] = K.bestx[j];
    delete [] Q;
    delete [] K.w;
    delete [] K.p;
    delete [] K.bestx;
    return bestp;
}
```

算法的主函数如下。

```
int main() {
    cin >> n >> c;
    p = new int[n+1];
    w = new int[n+1];
    bestx = new int[n+1];
    for(int i=1; i <= n; i++)
      cin >> p[i];
    for(i=1; i <= n; i++)
      cin >> w[i];
    for(i=1; i <=n;  i++)
      bestx[i] = 1;
    cout << Knapsack(p, w, c, n, bestx) << endl;
     for(i=1; i <= n; i++)
       cout << bestx[i] << " ";
    cout << endl;
    return 0;
}
```

6-15 排列空间树问题。

问题描述：试设计一个用优先队列式分支限界法搜索排列空间树的函数，其参数包括结点可行性判定函数和上界函数等必要的函数，并将此函数用于解批处理作业调度问题。给定 n 个作业的集合 $J=\{J_1, J_2, \cdots, J_n\}$。每个作业 J_i 都有 2 项任务分别在 2 台机器上完成。每个作业必须先由机器 1 处理，再由机器 2 处理。作业 J_i 需要机器 j 的处理时间为 t_{ji}（i=1, 2, …, n；j=1, 2）。对于一个确定的作业调度，设 F_{ji} 是作业 i 在机器 j 上完成处理的时间。所有作业在机器 2 上完成处理的时间和 $f=\sum_{i=1}^{n} F_{2i}$ 称为该作业调度的完成时间和。

批处理作业调度问题要求对于给定的 n 个作业，制定最佳作业调度方案，使其完成时间和达到最小。

算法设计：对于给定的 n 个作业，计算最佳作业调度方案。

数据输入：由文件 input.txt 提供输入数据。文件第 1 行有 1 个正整数 n，表示作业数。接下来的 n 行中，每行有 2 个正整数 i 和 j，分别表示在机器 1 和机器 2 上完成该作业所需的处理时间。

结果输出：将最佳作业调度方案及其完成时间和输出到文件 output.txt。文件的第 1 行是完成时间和，第 2 行是最佳作业调度方案。

输入文件示例	输出文件示例
input.txt	output.txt
3	18
2 1	1 3 2
3 1	
2 3	

分析与解答：用优先队列式分支限界法搜索排列空间树的一般算法 pqbb 如下。

```
template<class Type>
void Flowshop<Type>::pqbb() {
  MinHeap<HeapNode<Type>> H(HeapSize);
  HeapNode<Type> E;
  Init(E);
  while(true){                                          // 搜索排列空间树
    if(Answer(E))
      Save(E);
    else {
      for(int i=E.s; i < n; i++) {
        Swap(E.x[E.s], E.x[i]);
        if(Constrain(E) && Bound(E))
          AddLiveNode(H, E);
        Swap(E.x[E.s], E.x[i]);
      }
    }
    if(!Getnext(H, E))                                  // 取下一扩展结点
      break;
  }
  Output();
}
```

其中，堆结点类型为 HeapNode。

```
template<class Type>
class HeapNode {
  friend Flowshop<Type>;
public:
  operator Type()const {  return bb;  }
private:
  void Init(int), NewNode(HeapNode<Type>, Type, Type, Type, int);
  int  s,                                               // 已安排作业数
       *x;                                              // 当前作业调度
  Type  f1,                                             // 机器 1 上最后完成时间
        f2,                                             // 机器 2 上最后完成时间
        sf2,                                            // 当前机器 2 上的完成时间和
        bb;                                             // 当前完成时间和下界
};
template<class Type>
void HeapNode<Type>::Init(int n) {                      // 堆结点初始化
```

```
  x=new int[n];
  for(int i=0; i<n; i++)
    x[i] = i;
  s = 0;
  f1 = 0;
  f2 = 0;
  sf2 = 0;
  bb = 0;
}
template<class Type>
void HeapNode<Type>::NewNode(HeapNode<Type> E, Type Ef1, Type Ef2, Type Ebb, int n) {     // 新结点
  x = new int[n];
  for(int i=0; i < n; i++)
    x[i] = E.x[i];
  f1 = Ef1;
  f2 = Ef2;
  sf2 = E.sf2+f2;
  bb = Ebb;
  s = E.s+1;
}
```

结点可行性判定函数Constraint()和上界函数Bound()等必要的函数通过类Flowshop的私有函数传递。

```
template<class Type>
class Flowshop{
  friend Type Flow(Type **M, int n, int bestx[]);
public:
  void pqbb();
private:
  Type Bd(HeapNode<Type>);
  void Sort();
  void Init(HeapNode<Type> &E);
  bool Answer(HeapNode<Type> E);
  void Save(HeapNode<Type> &E);
  bool Constrain(HeapNode<Type> E);
  bool Bound(HeapNode<Type> E);
  void AddLiveNode(MinHeap<HeapNode<Type> > &H, HeapNode<Type> E);
  bool Getnext(MinHeap<HeapNode<Type> > &H, HeapNode<Type> &E);
  void Output();
  int  n,                                          // 作业数
       *bestx;                                     // 最优解
  Type **M,                                        // 各作业所需的处理时间数组
       **b,                                        // 各作业所需的处理时间排序数组
       **a,                                        // 数组M和b的对应关系数组
       bestc,                                      // 最小完成时间和
       f1, f2, bb;
  bool **y;                                        // 工作数组
};
template<class Type>
```

```
void Flowshop<Type>::Sort() {                                    // 对各作业在机器1和2上所需时间排序
   int  *c = new int[n];
   for(int j=0; j < 2; j++) {
      for(int i=0; i< n; i++) {
         b[i][j] = M[i][j];
         c[i] = i;
      }
      for(i=0; i < n1; i++) {
         for(int k=n-1; k > i; k--) {
            if(b[k][j]<b[k-1][j]) {
               Swap(b[k][j], b[k-1][j]);
            Swap(c[k], c[k-1]);
         }
         for(i=0; i < n; i++)
            a[c[i]][j] = i;
      }
   }
   delete[] c;
}
template<class Type>
Type Flowshop<Type>::Bd(HeapNode<Type> E) {                      // 计算完成时间和下界
   for(int k=0; k < n; k++)
      for(int j=0; j < 2; j++)
         y[k][j] = false;
   for(k=0; k <= E.s; k++)
      for(int j=0; j < 2; j++)
         y[a[E.x[k]][j]][j]=  true;
   f1 = E.f1+M[E.x[E.s]][0];
   f2 = ((f1>E.f2) ? f1 : E.f2) + M[E.x[E.s]][1];
   Type  sf2 = E.sf2+f2;
   Type  s1 = 0, s2 = 0, k1 = n-E.s, k2 = n-E.s, f3 = f2;
   for(int j=0; j< n; j++)                                       // 计算s1的值
      if(!y[j][0]) {
         k1--;
         if(k1 == n-E.s-1)
            f3 = (f2>f1+b[j][0]) ? f2 : f1+b[j][0];
         s1 += f1+k1*b[j][0];
      }
   for(j=0; j < n; j++) {                                        // 计算s2的值
      if(!y[j][1]) {
         k2--;
      s1 += b[j][1];
      s2 += f3+k2*b[j][1];
   }
   return sf2 + ((s1>s2) ? s1 : s2);                             // 返回完成时间和下界
}
template<class Type>
void Flowshop<Type>::Init(HeapNode<Type> &E) {
   Sort();                                                       // 对各作业在机器1和2上所需时间排序
```

```
   E.Init(n);
}
template<class Type>
bool Flowshop<Type>::Answer(HeapNode<Type> E) {
   return E.s == n;
}
template<class Type>
void Flowshop<Type>::Save(HeapNode<Type> &E) {
   if(E.sf2 < bestc) {
      bestc = E.sf2;
      for(int i=0; i < n; i++)
         bestx[i] = E.x[i];
   }
   delete[] E.x;
}
template<class Type>
bool Flowshop<Type>::Constrain(HeapNode<Type> E) {
   return true;
}
template<class Type>
bool Flowshop<Type>::Bound(HeapNode<Type> E) {
   bb = Bd(E);
   return bb<bestc;
}
template<class Type>
void Flowshop<Type>::AddLiveNode(MinHeap<HeapNode<Type> > &H,HeapNode<Type> E) {  // 将新结点插入堆中
   HeapNode<Type> N;
   N.NewNode(E, f1, f2, bb, n);
   H.Insert(N);
}
template<class Type>
bool Flowshop<Type>::Getnext(MinHeap<HeapNode<Type> > &H, HeapNode<Type> &E) {
   try {  H.DeleteMin(E);  }                                                    // 取下一扩展结点
   catch(OutOfBounds) {  return false;  }
   return true;
}
template<class Type>
void Flowshop<Type>::Output(){
   cout << bestc << endl;
   for(int j=0; j < n; j++)
      cout << bestx[j]+1 << " ";
   cout << endl;
}
```

Flow()函数实现批处理作业调度问题的优先队列式分支限界法。

```
template<class Type>
Type Flow(Type **M,int n, int bestx[]) {
   Flowshop<Type> X;
   X.M = M;
```

```
    X.n = n;
    X.bestx = bestx;
    X.bestc = INT_MAX;
    Make2DArray(X.a, n, 2);
    Make2DArray(X.b, n, 2);
    Make2DArray(X.y, n, 2);
    X.pqbb();
    return X.bestc;
}
```

算法的主函数如下。

```
int main() {
    cin >> n;
    Make2DArray(M, n, 2);
    bestx = new int[n+1];
    for(int i=0; i < n; i++)
       for(int j=0; j < 2; j++)
          cin >> M[i][j];
    cout << Flow(M, n, bestx) << endl;
    for(i=0; i < n; i++)
       cout << bestx[i]+1 << " ";
    cout << endl;
    return 0;
}
```

6-16 一般解空间的优先队列式分支限界法。

问题描述：试设计一个用优先队列式分支限界法搜索一般解空间的函数，其参数包括结点可行性判定函数和上界函数等必要的函数，并将此函数用于解布线问题。

印刷电路板将布线区域划分成 $n \times m$ 个方格阵列（见图 6-3(a)）。精确的电路布线问题要求确定连接方格 a 的中点到方格 b 的中点的最短布线方案。在布线时，电路只能沿直线或直角布线（见图 6-3(b)）。为了避免线路相交，已布线了的方格做了封锁标记，其他线路不允许穿过被封锁的方格。

算法设计：对于给定的布线区域，计算最短布线方案。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n、m、k，分别表示布线区域方格阵列的行数、列数和封闭的方格数。接下来的 k 行中，每行 2 个正整数，表示被封闭的方格所在的行号和列号。最后的 2 行，每行也有 2 个正整数，分别表示开始布线的方格(p, q)和结束布线的方格(r, s)。

结果输出：将计算的最短布线长度和最短布线方案输出到文件 output.txt。文件的第 1 行是最短布线长度。从第 2 行起，每行 2 个正整数，表示布线经过的方格坐标。如果无法布线，则输出“No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
8 8 3	11
3 3	2 1
4 5	3 1
6 6	4 1

```
2 1                    5 1
7 7                    6 1
                       7 1
                       7 2
                       7 3
                       7 4
                       7 5
                       7 6
                       7 7
```

分析与解答：用优先队列式分支限界法搜索一般解空间的算法 pqbb 如下。

```
template<class Type>
void Maze<Type>::pqbb() {
   NewHeap(H);
   HeapNode<Type>  E, N;
   Init(E);
   while(true) {                                    // 搜索一般解空间树
      if(Answer(E))
         Save(E);
      else {
         for(int i=f(n, E); i <= g(n, E); i++) {
            NewNode(N, E, i);
            if(Constrain(N) && Bound(N))
               AddLiveNode(N, E, i);
            else
               DelNode(N);
         }
      }
      if(!Getnext(E))                               // 取下一扩展结点
         break;
   }
   Output();
}
```

其中，堆结点类型为 HeapNode。

```
template<class Type>
class HeapNode {
public:
   operator int() const {  return len;  }
   Type  row, col, len;
};
```

结点可行性判定函数 Constraint()和上界函数 Bound()等通过类 Maze 的私有函数传递。

```
template<class Type>
class Maze {
   friend int main();
private:
   MinHeap<HeapNode<Type>> *H;
   void NewHeap(MinHeap<HeapNode<Type>> *&H);
```

```
    void Init(HeapNode<Type> &E);
    bool Answer(HeapNode<Type> E);
    void Save(HeapNode<Type> &E);
    int f(int n, HeapNode<Type> E);
    int g(int n, HeapNode<Type> E);
    void NewNode(HeapNode<Type> &N, HeapNode<Type> E, int i);
    bool Constrain(HeapNode<Type> E);
    bool Bound(HeapNode<Type> E);
    void AddLiveNode(HeapNode<Type> N, HeapNode<Type> E, int i);
    bool Getnext(HeapNode<Type> &E);
    void DelNode(HeapNode<Type> &E);
    void Output();
    void pqbb();
    int  n, m;
    bool  found;
    Type  **grid, PathLen;
    HeapNode<Type> start, finish, offset[4], *path;
};
template<class Type>
void Maze<Type>::NewHeap(MinHeap<HeapNode<Type> > *&H) {
    H = new MinHeap<HeapNode<Type>>(HeapSize);
}
template<class Type>
void Maze<Type>::Init(HeapNode<Type> &E) {                    // 设置方格阵列"围墙"
    for(int i=0; i <= m+1; i++)
        grid[0][i] = grid[n+1][i] = 1;                        // 顶部和底部
    for(i=0; i <= n+1; i++)
        grid[i][0] = grid[i][m+1] = 1;                        // 左翼和右翼
    // 初始化相对位移
    offset[0].row = 0;          offset[0].col = 1;            // 右
    offset[1].row= 1;           offset[1].col = 0;            // 下
    offset[2].row =0;           offset[2].col = 1;            // 左
    offset[3].row= -1;          offset[3].col = 0;            // 上
    E.row = start.row;
    E.col = start.col;
    E.len = 0;
    grid[start.row][start.col] = 2;
}
template<class Type>
bool Maze<Type>::Answer(HeapNode<Type> E) {  return false;  }
template<class Type>
void Maze<Type>::Save(HeapNode<Type> &E) {  }
template<class Type>
int Maze<Type>::f(int n, HeapNode<Type> E) {  return 1;  }
template<class Type>
int Maze<Type>::g(int n, HeapNode<Type> E) {  return 4;  }
template<class Type>
void Maze<Type>::NewNode(HeapNode<Type> &N, HeapNode<Type> E, int i) {
    N.row = E.row+offset[i-1].row;
```

```
   N.col =E.col+offset[i-1].col;
}
template<class Type>
bool Maze<Type>::Constrain(HeapNode<Type> E) {  return grid[E.row][E.col] == 0;  }
template<class Type>
bool Maze<Type>::Bound(HeapNode<Type> E) {
   if(!found)
      found = E.row == finish.row && E.col == finish.col;
    return true;
}
template<class Type>
void Maze<Type>::AddLiveNode(HeapNode<Type> N, HeapNode<Type> E, int i) {
   grid[N.row][N.col] = grid[E.row][E.col]+1;
   N.len = grid[N.row][N.col];
   if(!found)
      H->Insert(N);
}
template<class Type>
void Maze<Type>::DelNode(HeapNode<Type> &E) {  }
template<class Type>
bool Maze<Type>::Getnext(HeapNode<Type> &E) {
   if(found)
      return false;
   H->DeleteMin(E);
   return true;
}
template<class Type>
void Maze<Type>::Output() {
   if(!found) {
      cout << "No path!" << endl;
      return;
   }
   PathLen = grid[finish.row][finish.col]-2;          // 构造最优解
   path = new HeapNode<Type> [PathLen];
   HeapNode<Type>  N, E = finish;                     // 从目标位置 finish 开始向起始位置回溯
   for(int j=PathLen-1; j >= 0; j--) {
      path[j] = E;
      for(int i=0; i < 4; i++) {                      // 找前驱位置
         N.row = E.row+offset[i].row;
         N.col = E.col+offset[i].col;
         if(grid[N.row][N.col] == j+2)
            break;
      }
      E = N;                                          // 向前移动
   }
   cout << PathLen << endl;
   cout << start.row << " " << start.col << endl;
   for(j=0; j < PathLen; j++)
      cout << path[j].row << " " << path[j].col << endl;
```

```
}
```

实现算法的主函数如下。

```
int main() {
   int  n, m, a, b, x;
   Maze<int> X;
   cin >> n >> m >> x;
   X.n = n;
   X.m = m;
   X.found = false;
   Make2DArray(X.grid, n+2, m+2);
   for(a=0; a < n+2; a++)
      for(b=0; b < m+2; b++)
         X.grid[a][b] = 0;
   for(x=x; x >= 1; x--) {
      cin >> a >> b;
      X.grid[a][b] = 1;
   }
   cin >> X.start.row >> X.start.col >> X.finish.row >> X.finish.col;
   X.pqbb();
   return 0;
}
```

6-17　推箱子问题。

问题描述：码头仓库是划分为 $n \times m$ 个格子的矩形阵列。有公共边的格子是相邻格子。当前仓库中有的格子是空闲的，有的格子则已经堆放了沉重的货物。由于堆放的货物很重，单凭仓库管理员的力量是无法移动的。仓库管理员有一项任务：要将一个小箱子推到指定的格子上去。管理员可以在仓库中移动，但不能跨过已经堆放了货物的格子。管理员站在与箱子相对的空闲格子上时，可以做一次推动，把箱子推到另一相邻的空闲格子。推箱时只能向管理员的对面方向推。由于要推动的箱子很重，仓库管理员想尽量减少推箱子的次数。

算法设计：对于给定的仓库布局，以及仓库管理员在仓库中的位置和箱子的开始位置和目标位置，设计一个解推箱子问题的分支限界法，计算出仓库管理员将箱子从开始位置推到目标位置所需的最少推动次数。

数据输入：由文件 input.txt 提供输入数据。输入文件第 1 行有 2 个正整数 n 和 m（$1 \leqslant n, m \leqslant 100$），表示仓库是 $n \times m$ 个格子的矩形阵列。接下来有 n 行，每行有 m 个字符，表示格子的状态。

S——格子上放了不可移动的沉重货物；　　P——箱子的初始位置；

w——格子空闲；　　K——箱子的目标位置。

M——仓库管理员的初始位置；

结果输出：将计算的最少推动次数输出到文件 output.txt。如果仓库管理员无法将箱子从开始位置推到目标位置则输出“No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
10 12	7
SSSSSSSSSSSS	

SwwwwwwwSSSS
SwSSSSwwSSSS
SwSSSSwwSKSS
SwSSSSwwSwSS
SwwwwwPwwwww
SSSSSSSwSwSw
SSSSSSMwSwww
SSSSSSSSSSSS
SSSSSSSSSSSS

分析与解答：与布线问题类似，结点元素类型是 Position。

```
class Position {
public:
   operator int() const {  return row;  }
   int  row, col, dir;
};
```

它的私有成员 row 和 col 分别表示方格所在的行和列，dir 表示推的方向。

算法用到如下全局变量。

```
int  op[4] = {1, 0, 3, 2};
int  m, n, totm, markr;
int  **grid, **reach, **mark, **low, **totr, ***comp;
long  ***ans;
Position  start, finish, man;
Position  offset[4];
```

Init()函数实现数据输入及预处理。

```
void Init() {
   char  c;
   cin >> n >> m;
   Make2DArray(grid, n+2, m+2);
   Make2DArray(reach, n+1, m+1);
   Make2DArray(mark, n+1, m+1);
   Make2DArray(low, n+1, m+1);
   Make2DArray(totr, n+1, m+1);
   Make3DArray(ans, n+1, m+1, 4);
   Make3DArray(comp, n+1, m+1, 10);
   for(int i=0; i < n+2; i++)
      for(int j=0; j < m+2; j++)
         grid[i][j] = 0;
   for(i=1; i <= n; i++) {
      for(int j=1; j <= m; j++) {
         cin >> c;
         while(!isupper(c) && !islower(c))
            cin >> c;
         if(c == 'M')
            man.row = i, man.col = j;
         if(c == 'P')
            start.row = i, start.col = j;
```

```
        if(c == 'K')
          finish.row = i, finish.col = j;
        if(c == 'S')
          grid[i][j] = 1;
      }
      for(i=0; i <= m+1; i++)                          // 设置方格阵列"围墙"
        grid[0][i] = grid[n+1][i] = 1;                 // 顶部和底部
      for(i=0; i <= n+1; i++)
        grid[i][0] = grid[i][m+1] = 1;                 // 左翼和右翼
      // 初始化相对位移
      offset[0].row = 0;      offset[0].col = -1;      // 左
      offset[1].row = 0;      offset[1].col =1;        // 右
      offset[2].row = -1;     offset[2].col =  0;      // 上
      offset[3].row = 1;      offset[3].col = 0;       // 下
      Prepro();
      for(i=0; i <= n; i++) {
        for(int j=0; j <= m; j++){
          reach[i][j] = 0;
          for(int k=0; k < 4; k++)
            ans[i][j][k] = LONG_MAX;
        }
      }
    }
    dfs(man.row, man.col);
}
```

其中，Prepro()函数对方格连通性进行预处理计算。

```
void Prepro() {
    totm = 0;
    markr = 0;
    for(int i=1; i <= n; i++) {
      for(int j=1; j <= m; j++) {
        mark[i][j] = 1;
        low[i][j] = INT_MAX;
        totr[i][j] = 0;
        reach[i][j] = -1;
      }
    }
    for(i=1; i <= n; i++)
      for(int j=1; j <= m; j++)
        if(grid[i][j] == 0 && mark[i][j] == -1)
          fill(i, j);
}
void put(int x, int y, int a, int b) {
    if(x==a && y==b)
      return;
    if(reach[x][y] == 2)
      return;
    comp[x][y][totr[x][y]] = markr;
```

```
    totr[x][y]++;
    reach[x][y] = 2;
    for(int i=0; i < 4; i++) {
      int  x1 = x+offset[i].row, y1 = y+offset[i].col;
      if(grid[x1][y1] == 0)
        put(x1, y1, a, b);
    }
}
void fill(int x, int y) {
    for(int i=0; i < 4; i++) {
      int  x1 = x+offset[i].row, y1 = y+offset[i].col;
      if(grid[x1][y1] == 0) {
        if(mark[x1][y1] == -1) {
          mark[x1][y1] = totm;
          totm++;
        }
        fill(x1, y1);
        low[x][y] = min(low[x][y], low[x1][y1]);
        if(low[x1][y1] >= mark[x][y]) {
          markr++;
          put(x1, y1, x, y);
          comp[x][y][totr[x][y]] = markr;
          totr[x][y]++;
          reach[x][y] = 1;
        }
      }
      else
        low[x][y] = min(low[x][y], mark[x1][y1]);
    }
}
```

预处理后，由 connect()函数计算方格连通性。

```
int connect(int x1, int y1, int x2, int y2) {
    for(int i=0; i < totr[x1][y1]; i++)
      for(int j=0; j < totr[x2][y2]; j++)
        if(comp[x1][y1][i] == comp[x2][y2][j])
          return 1;
    return 0;
}
```

dfs()函数计算初始可达性。

```
void dfs(int x, int y) {
    if(reach[x][y] == 1)
      return;
    reach[x][y] = 1;
    for(int i=0; i< 4; i++) {
      int  x1 = x+offset[i].row, y1 = y+offset[i].col;
      if(grid[x1][y1] == 0 &&(x1 != start.row || y1 != start.col))
        dfs(x1, y1);
```

```
    }
}
```

解推箱子问题的队列式分支限界法 FIFOBB 如下。

```
void FIFOBB() {
   Queue<Position>  Q;
   Position  here, nbr;
   for(int i=0; i < 4; i++) {
      nbr.row = start.row;
      nbr.col = start.col;
      nbr.dir = i;
      if(ok(start.row+offset[i].row, start.col+offset[i].col)) {
         ans[start.row][start.col][i] = 0;
         Q.Add(nbr);
      }
   }
   while(!Q.IsEmpty()) {
      Q.Delete(here);
      int  d = here.dir, x = here.row+offset[op[d]].row, y = here.col+offset[op[d]].col;
      if(grid[x][y] == 0 && ans[x][y][d] > ans[here.row][here.col][d]+1) {
         ans[x][y][d] = ans[here.row][here.col][d]+1;
         nbr.row = x;
         nbr.col = y;
         nbr.dir = d;
         Q.Add(nbr);
         for(i=0; i<  4; i++)
            if(i != d) {
               int  x1 = x+offset[i].row, y1=  y+offset[i].col;
               if(grid[x1][y1] == 0 && connect(x1, y1, here.row, here.col) && ans[x][y][i]>ans[x][y][d]) {
                  ans[x][y][i] = ans[x][y][d];
               nbr.row = x;
               nbr.col = y;
               nbr.dir = i;
               Q.Add(nbr);
            }
         }
      }
   }
}
bool ok(int a, int b) {  return grid[a][b] == 0 && reach[a][b] == 1;  }
```

算法的主函数如下。

```
int main() {
   Init();
   FIFOBB();
   Out();
   return 0;
}
```

Out()函数输出计算结果。

```
void Out() {
   for(long min=ans[finish.row][finish.col][0], i=1; i < 4; i++)
      if(ans[finish.row][finish.col][i] < min)
         min = ans[finish.row][finish.col][i];
   if(min == LONG_MAX)
      cout << "No Solution!" << endl;
   else
      cout << min << endl;
}
```

计算机算法设计与分析习题解答（第5版）

本书是与“十二五”普通高等教育本科国家级规划教材《计算机算法设计与分析（第5版）》配套的辅助教材和国家精品课程教材，分别对主教材中的算法分析题和算法实现题给出了解答或解题思路提示。为了提高学生灵活运用算法设计策略解决实际问题的能力，本书还将主教材中的许多习题改造成算法实现题，要求学生设计出求解算法并上机实现。本书教学资料包含各章算法实现题、测试数据和答案，可在华信教育资源网免费注册下载。

本书内容丰富，理论联系实际，可作为高等学校计算机科学与技术、软件工程、信息安全、信息与计算科学等专业本科生和研究生学习计算机算法设计的辅助教材，也是工程技术人员和自学者的参考书。

提升学生“知识—能力—素质”	体现“基础—技术—应用”内容
把握教学“难度—深度—强度”	提供“教材—教辅—课件”支持

相关图书：《计算机算法设计与分析（第5版）》 ISBN 978-7-121-34439-8

策划编辑：章海涛
责任编辑：章海涛
封面设计：张 昱

ISBN 978-7-121-34438-1

定价：56.00 元