算法设计与分析

主讲教师：王新宇

分支限界法的思想与回溯法相似，都是通过搜索问题的解空间树来寻找问题的解，同时使用剪枝函数提升搜索效率。

但分支限界法的求解目标不同于回溯法，使得分支限界法是采用广度优先的方法搜索解空间树。这表明，面向一个问题究竟应当选择回溯法还是分支限界法决定于具体的求解目标，即需要具体问题具体分析。

具体问题具体分析是辩证唯物主义的一条基本要求和重要原理，在选择一种算法设计方法求解问题时，需要秉持这一原则，特别是在类似回溯法和分支限界法等相似方法之间进行选择时，更要厘清它们的不同之处。

第13章 分支限界法

目　录

13.1 分支限界法概述

13.1.1 分支限界法的基本思想

分支限界法类似于回溯法，也是一种在问题的解空间树上搜索问题解的方法。

- 分支限界法与回溯法的区别

（1）求解目标不同

回溯法通常是在解空间树中寻找满足约束条件的所有解，在此基础上可以寻找任一个解或最优解。

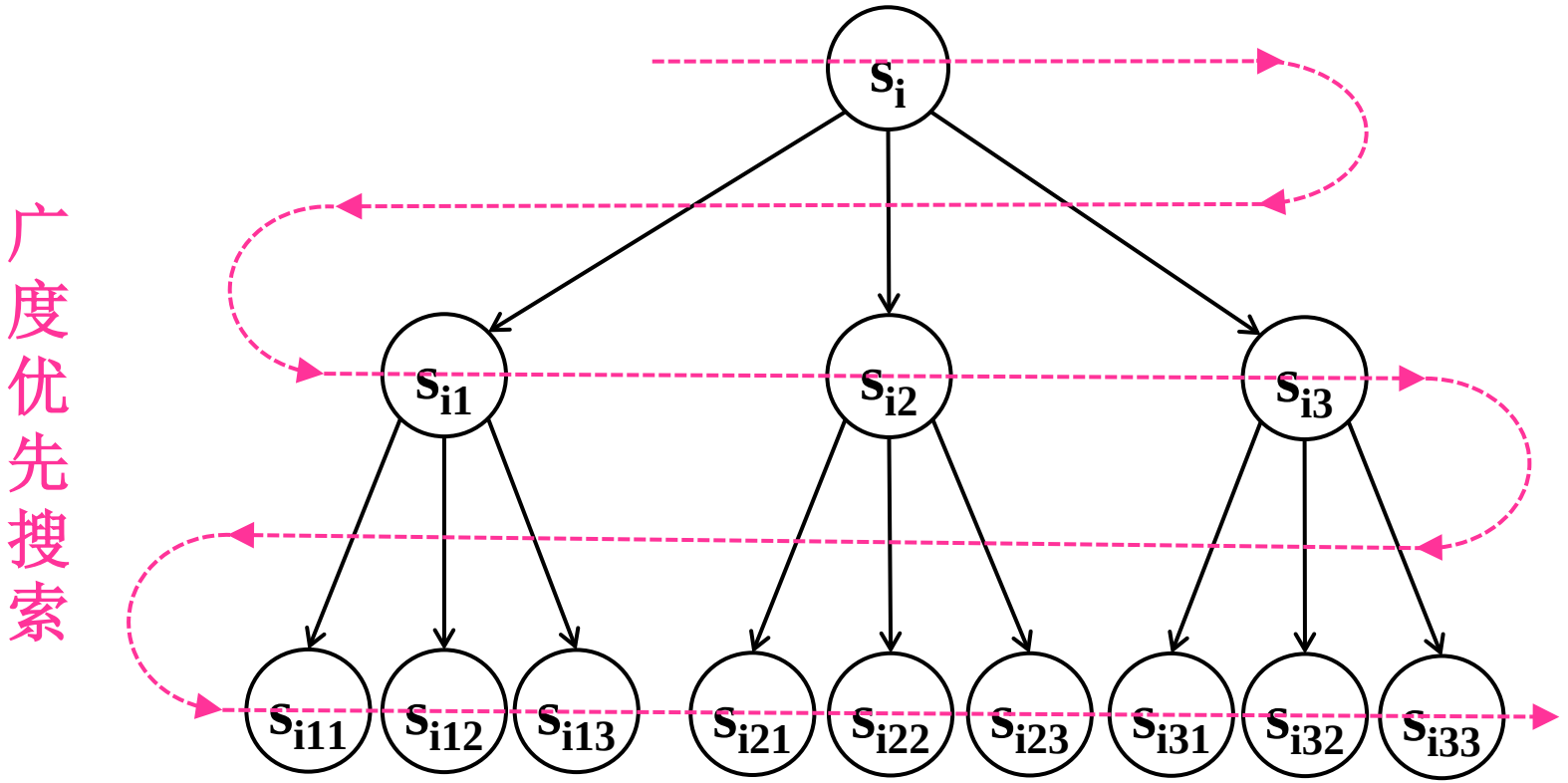
分支限界法是在解空间树中寻找满足约束条件的一个解或最优解，即通常不使用分支限界法寻找问题的所有可行解。

13.1.1 分支限界法的基本思想

（2）搜索解空间树的方法不同

回溯法采用深度优先的方法搜索解空间树的节点。

分支限界法采用广度优先的方法搜索解空间树的节点。



13.1.1 分支限界法的基本思想

- 分支限界法搜索解空间树的过程

从根节点出发，以广度优先的方法搜索整棵解空间树。

在搜索过程中，每一个活节点只有一次机会成为扩展节点。活节点一旦成为扩展节点，将一次性产生所有的孩子节点，舍弃其中无法产生可行解或最优解的孩子节点，其余孩子节点被加入活节点表。

在处理完当前扩展节点之后，从活节点表取下一节点成为当前扩展节点，重复上述的节点扩展过程，直到找到满足要求的解或活节点表为空时为止。

13.1.2 分支限界法的三个关键问题

- 如何舍弃无法产生可行解或最优解的孩子节点？

剪枝函数：约束函数和限界函数。

（1）约束函数

判断节点的扩展是否满足约束条件，用于舍弃无法产生可行解的分支。

（2）限界函数

判断节点的扩展是否可能产生更优解，用于舍弃无法产生最优解的分支。

若问题的求解目标是寻找最大值，则计算节点的上界与已有的最大值比较，舍弃节点上界小于已有最大值的分支。

若问题的求解目标是寻找最小值，则计算节点的下界与已有的最小值比较，舍弃节点下界大于已有最小值的分支。

13.1.2 分支限界法的三个关键问题

- 如何组织活节点表？{ 队列 / 优先队列 }

（1）队列式分支限界法

采用队列存储活节点，按照“先进先出”的原则依次选择活节点作为扩展节点。搜索过程如下：

① 将根节点加入活节点队列。

② 从活节点队列取出队头节点作为当前扩展节点。

③ 对于当前扩展节点，根据扩展规则依次产生它的所有孩子节点，用剪枝函数检查每一个分支，将满足剪枝函数的孩子节点加入活节点队列。

④ 重复②和③，直到找到一个解或活节点队列为空为止。

搜索解空间树的方法：完全的广度优先搜索

13.1.2 分支限界法的三个关键问题

（2）优先队列式分支限界法

这种方法采用优先队列存储活节点，每次选择优先级最高的活节点作为扩展节点。搜索过程如下：

① 计算起始节点（根节点）的优先级并加入优先队列。

② 从优先队列取出优先级最高的节点作为当前扩展节点。

③ 对于当前扩展节点，根据扩展规则依次产生它的所有孩子节点，用剪枝函数检查由此产生的每一个分支，将满足剪枝函数的孩子节点计算优先级加入优先队列。

④ 重复②和③，直到找到一个解或优先队列为空为止。

搜索解空间树的方法：最小耗费或最大效益优先的广度优先搜索

13.1.2 分支限界法的三个关键问题

◆优先队列的实现方法

① 使用普通队列，当节点入队时根据优先级排序，优先级高的节点排在前面，优先级低的节点排在后面。

② 使用堆作为优先队列，大顶堆用于高优先级先出队的优先队列，小顶堆用于低优先级先出队的优先队列。

13.1.2 分支限界法的三个关键问题

- 如何记录搜索过程中的解？

① 每个节点都带有一个解向量，记录从根节点到该节点经过路径上的决策。当搜索到一个叶子节点时，该节点中的解向量就是问题的一个可行解。

② 在搜索过程中建立解空间树的结构。每个节点都带有一个指向双亲的指针，当搜索到一个叶子节点时，从该节点的双亲指针开始，不断向上回溯直至找到根节点，这一路径上的决策构成问题的一个可行解。

13.1.3 分支限界法的设计步骤

（1）确定问题的解空间，包括确定解向量的形式、显约束和隐约束。

（2）确定节点的扩展规则。

（3）确定剪枝函数。结合节点扩展规则、约束条件和目标函数的优化目标确定剪枝函数。不同分支的剪枝函数可能不同。

（4）确定活节点表的组织方式和解向量的记录方式。

（5）以广度优先的方式搜索解空间树，在搜索过程中采用剪枝函数避免无效搜索。

13.1.4 分支限界法的时间性能

设问题的解向量为$X=(x_1, x_2, \ldots, x_n)$，$x_i$（$1 \le i \le n$）的取值范围为某个有限集合$S_i=(s_{i1}, s_{i2}, \ldots, s_{ir})$，使用$|S_i|$表示集合$S_i$的元素数目，则：

- 第1层的根节点有$|S_1|$棵子树
- 第2层有$|S_1|$个节点，每个节点有$|S_2|$棵子树
- 第3层有$|S_1| \times |S_2|$个节点，每个节点有$|S_3|$棵子树

  ……

- 第$n+1$层有$|S_1| \times |S_2| \times \ldots \times |S_n|$个节点，它们都是叶子节点，代表问题的所有可能解。

因此，问题的解空间由笛卡儿积$S_1 \times S_2 \times \ldots \times S_n$构成。

在最坏情况下，搜索过程没有发生剪枝，则算法的时间复杂度是指数阶。

13.1.5 分支限界法的适用条件

（1）问题的求解过程可以划分为若干个依次进行的决策阶段，每个阶段对解的一个分量做出选择，整个问题的解可以表示为一个向量。

（2）问题要满足多米诺性质。

13.2 分支限界法的应用实例

13.2.1 0-1背包问题

设给定n种物品和一个背包，物品i的质量是w_i，其价值为v_i，背包的容量为c。

对每种物品i只有两种选择：装入背包或不装入背包，既不能将物品i装入背包多次，也不能只装入部分的物品i。

应当如何选择装入背包的物品，在物品质量不超过背包容量的情况下，使得装入物品后的背包价值最大？

13.2.1 0-1背包问题

【问题分析】

（1）确定问题的解空间

设$X=(x_1, x_2, \ldots, x_n)$表示问题的解向量，其中，$x_i$（$i=1, 2, \ldots, n$）表示对第$i$种物品的决策。

显约束：$x_i \in \{0, 1\}$（$i=1, 2, \ldots, n$），$x_i=0$表示第i种物品不装入背包，$x_i=1$表示第i种物品装入背包。

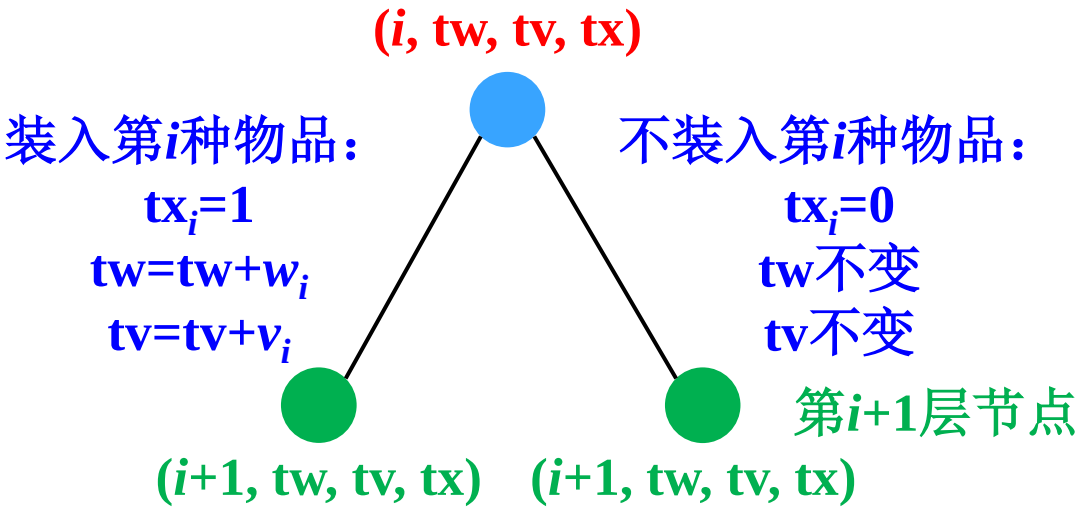
隐约束：$\sum_{i=1}^{n} w_i x_i \leq c$

13.2.1 0-1背包问题

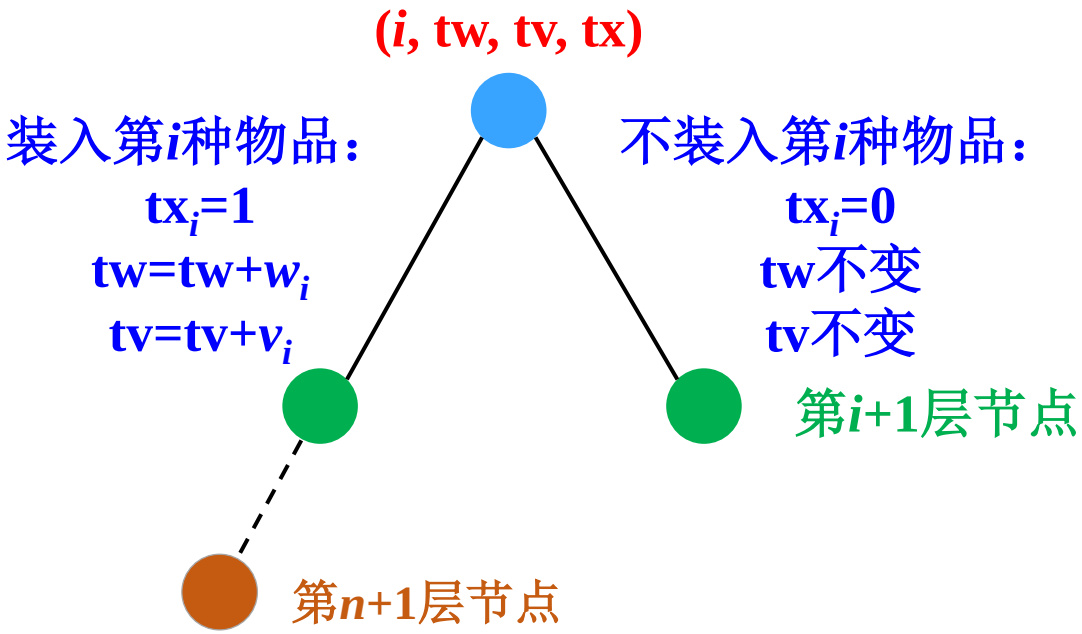
（2）确定节点的扩展规则

为解空间树的每个节点设置状态：**(*i*, tw, tv, tx)**，其中，*i*表示节点的层次，tw表示当前的背包质量，tv表示当前的背包价值，tx记录当前的解向量。

第*i*层节点的扩展规则：装入或不装入第*i*种物品。



13.2.1 0-1背包问题



- 叶子结点表示已经对n种物品做出了决策。

13.2.1 0-1背包问题

（3）确定剪枝函数

① 左剪枝

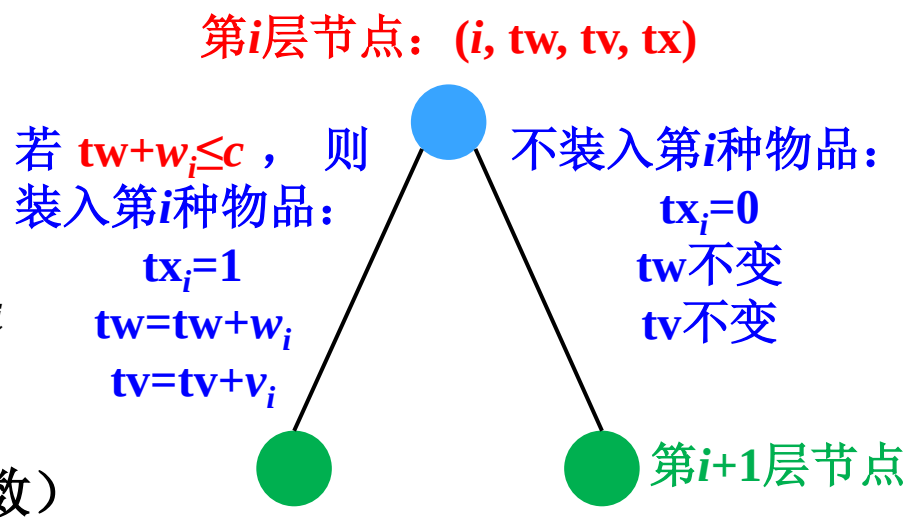
左分支的扩展规则：装入物品。

约束条件：$\sum_{i=1}^{n} w_i x_i \leq c$

目标函数的优化目标：$\max \sum_{i=1}^{n} v_i x_i$

剪枝函数：$tw+w_i \leq c$（约束函数）

$bound(i)>maxv$（限界函数）



其中，$bound(i)=tv+v_i+rv$表示装入第i种物品能够产生的背包价值的上界；rv表示第i+1~n种物品能够产生的装入物品价值；maxv表示当前最优的物品选择方案产生的背包价值。

由于$bound(i)=bound(i-1)>maxv$，所以可以省略限界函数，仅需使用约束函数，扩展满足$tw+w_i \leq c$的左孩子节点。

13.2.1 0-1背包问题

② 右剪枝

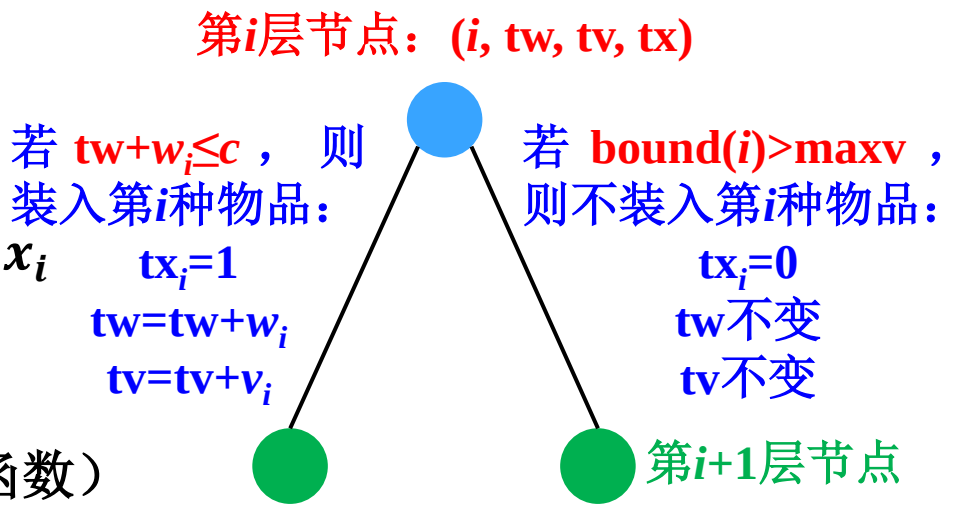
右分支的扩展规则：不装入物品。

约束条件：$\sum_{i=1}^{n} w_i x_i \leq c$

目标函数的优化目标：$\max \sum_{i=1}^{n} v_i x_i$

剪枝函数：无需约束函数

bound(*i*)>maxv（限界函数）



其中，bound(*i*)=tv+rv表示放弃第*i*种物品能够产生的背包价值的上界。

右分支仅需使用限界函数，扩展tv+rv>maxv的右孩子节点。

显然，rv越小，剪枝越多，为了构造尽可能小的rv，可以预先将所有物品按照单位质量价值递减排列。

13.2.1 0-1背包问题

【算法描述】

设n表示物品种类；c表示背包容量。

一维结构体数组A记录物品信息，其中，A[i]记录第i种物品的信息，A[i].no表示物品编号，A[i].w表示物品的质量，A[i].v表示物品的价值，A[i].p表示物品单位质量的价值（1≤i≤n）。

结构体变量e表示解空间树的当前扩展节点，其中，e.i表示节点层次，e.tw表示节点对应的背包质量，e.tv表示节点对应的背包价值，e.tx表示节点对应的解向量；结构体变量e1和e2分别表示扩展节点e的左孩子和右孩子。

maxv记录最优的物品选择方案产生的背包价值；一维数组x记录最优解向量；bound记录当前扩展节点的背包价值上界。

13.2.1 0-1背包问题

应用分支限界法求解0-1背包问题的算法描述如下。

Step 1：将所有物品按照单位质量的价值递减排序，并存储于结构体数组A。

Step 2：初始化最大背包价值maxv为0。

Step 3：初始化解空间树的根节点为当前扩展节点e，并入队。

Step 4：当队列不为空时，反复进行如下处理，否则结束算法。

Step 4.1：根据出队规则出队队头节点e。

Step 4.2：判断是否可以扩展左分支，即是否满足e.tw+A[e.i].w≤c，满足则选择装入第i种物品，构造左孩子节点e1，继续向下执行，否则转至Step 4.4。

13.2.1 0-1背包问题

Step 4.3：判断左孩子节点**e1**是否解空间树的叶子节点，即是否满足**e1.i>n**，分为如下两种情况。

① **e1.i>n**，表明找到问题的一个可行解，进一步判断是否是更优解，即是否满足**e1.tv>maxv**，若满足，则表明找到问题的一个更优解，更新**maxv**和**x**。

② **e1.i≤n**，将**e1**入队。

Step 4.4：判断是否可以扩展右分支，即是否满足**bound>maxv**，若满足，则选择不装入第**i**种物品，构造右孩子节点**e2**，继续向下执行，否则跳过**Step 4.5**。

13.2.1 0-1背包问题

Step 4.5：判断右孩子节点e2是否解空间树的叶子节点，即是否满足e2.i>n，分为如下两种情况。

① e2.i>n，表明找到问题的一个可行解，进一步判断是否是更优解，即是否满足e2.tv>maxv，若满足，则表明找到问题的一个更优解，更新maxv和x。

② e2.i≤n，将e2入队。

13.2.1 0-1背包问题

【算法实现】

（1）队列式分支限界算法

```
#include "LinkQueue.h"          //使用第3章的链队列
struct Item  {                  //物品类型
    int no;                     //物品编号
    double w, v, p;       //物品的质量、价值和物品单位质量价值
    //重载小于运算符用于按照单位质量价值递减排序
    bool operator<(const Item &s) const
    {    return p > s.p;    }
};
```

13.2.1 0-1背包问题

```
struct STNode  {        //解空间树的节点类型
    int i;              //当前节点在解空间树中的层次
    double tw, tv;      //当前节点的背包质量和背包价值
    int tx[MAX];        //当前节点包含的解向量
};
```

13.2.1 0-1背包问题

```
//计算扩展节点的背包价值上界
double Bound(int i, Item * A, STNode & e, int n, int c)  {
    double tw = e.tw, tv = e.tv;          //初始化背包当前质量和价值
    i++;                                   //从第i+1种物品开始计算
    while(i<=n && tw + A[i].w <= c)  {//依次判断剩余物品
        tw += A[i].w;  tv += A[i].v;    //可以整个装入第i种物品
        i++;
    }
    if(i <= n)  return tv += (c - tw) * A[i].p;  //第i种物品只能装入部分
    return tv;
}
```

13.2.1 0-1背包问题

```
//队列式分支限界法求解0-1背包问题
void Knapsack01(int n, int c, Item * A, double &maxv, int * x)  {
    int j;
    STNode e, e1, e2;                    //定义3个解空间树节点
    LinkQueue<STNode> queue;             //定义一个队列
    for(int i = 1; i <= n; i++)          //求物品单位质量的价值
        A[i].p = A[i].v / A[i].w;
    sort(A + 1, A + n + 1);              //将物品按单位质量价值递减排序
    //初始化根节点
    e.i = 1;  e.tw = 0;  e.tv = 0;
    for(j=1; j<=n; j++)  e.tx[j] = 0;
```

13.2.1 0-1背包问题

```
queue.EnQueue(e);                    //根节点入队
while(!queue.IsEmpty()) {            //当队列不为空时循环
    queue.DelQueue(e);               //队头节点出队成为当前扩展节点
    if(e.tw + A[e.i].w <= c) {  //左分支剪枝
        //构造左孩子节点
        e1.i = e.i + 1;  e1.tw = e.tw + A[e.i].w;
        e1.tv = e.tv + A[e.i].v;
        for(j = 1; j <= n; j++)  e1.tx[j] = e.tx[j];  //复制解向量
        e1.tx[e.i] = 1;    //记录本次选择，即装入第i种物品
```

13.2.1 0-1背包问题

```
        if(e1.i > n)  {                          //找到一个叶子节点
            if(e1.tv > maxv)  {                  //找到一个更优解
                maxv = e1.tv;                    //更新最优值
                for (j = 1; j <= n; j++) //更新最优解
                    x[j] = e1.tx[j];
            }
        }
        else  queue.EnQueue(e1);                 //左孩子入队
    }
```

13.2.1 0-1背包问题

```
        if(Bound(e.i, A, e, n, c) > maxv) {          //右分支剪枝
            //构造右孩子节点
            e2.i = e.i + 1;  e2.tw = e.tw;  e2.tv = e.tv;
            for(j = 1; j <= n; j++)  e2.tx[j] = e.tx[j];  //复制解向量
            e2.tx[e.i] = 0;          //记录本次选择，即不装入第i种物品
            if(e2.i > n) {                           //找到一个叶子节点
                if(e2.tv > maxv) {                   //找到一个更优解
                    maxv = e2.tv;                    //更新最优值
                    for(j=1; j<=n; j++)              //更新最优解
                        x[j] = e2.tx[j];
                }
            }
            else  queue.EnQueue(e2);                 //右孩子入队
        }
    }//End of while
}
```

13.2.1 0-1背包问题

（2）优先队列式分支限界算法

取节点的背包价值上界作为节点的优先级，背包价值上界越大的节点越先出队，因此，使用大顶堆作为优先队列。

```
#include "MaxHeap.h"          //使用第5章的大顶堆作为优先队列
struct Item  {                //物品类型
    int no;                   //物品编号
    double w, v, p;       //物品质量，物品价值，物品单位质量价值
    //重载小于运算符用于按照单位质量价值递减排序
    bool operator<(const Item &s) const
    {   return p > s.p;   }
};
```

13.2.1 0-1背包问题

```
struct STNode  {          //解空间树的节点类型
    int i;                //当前节点在解空间树中的层次
    double tw, tv;        //当前节点的背包质量和背包价值
    double ub;            //当前节点的背包价值上界
    int tx[MAX];          //当前节点包含的解向量
    //重载小于运算符用于大顶堆的元素比较
    bool operator<(const STNode &s) const
    {    return ub < s.ub;    }
    //重载<=运算符用于大顶堆的元素比较
    bool operator<=(const STNode &s) const
    {    return ub <= s.ub;    }
```

13.2.1 0-1背包问题

```
    //重载>=运算符用于大顶堆的元素比较
    bool operator>=(const STNode &s) const
    {   return ub >= s.ub;   }
    //重载赋值运算符用于大顶堆的元素赋值
    STNode& operator=(const STNode &s)  {
        if(this != &s)  {
            i = s.i;  tw = s.tw;  tv = s.tv;  ub = s.ub;
            for(int j = 0; j < MAX; j++)   tx[j] = s.tx[j];
        }
        return *this;
    }
};
```

13.2.1 0-1背包问题

```
//计算扩展节点的背包价值上界
double Bound(int i, Item * A, STNode & e, int n, int c)  {
    double tw = e.tw, tv = e.tv;          //初始化背包当前质量和价值
    i++;                                  //从第i+1种物品开始计算
    while(i<=n && tw + A[i].w <= c)  {  //依次判断剩余物品
        tw += A[i].w;  tv += A[i].v;     //可以整个装入第i种物品
        i++;
    }
    if(i <= n)  return tv += (c - tw) * A[i].p;  //第i种物品只能装入部分
    return tv;
}
```

13.2.1 0-1背包问题

```
//优先队列式分支限界法求解0-1背包问题
void Knapsack01(int n, int c, Item * A, double &maxv, int * x) {
    int j;
    STNode e, e1, e2;                    //定义3个解空间树节点
    MaxHeap<STNode> queue(MAX);  //定义大顶堆作为优先队列
    for(int i = 1; i <= n; i++)          //求物品单位质量的价值
        A[i].p = A[i].v / A[i].w;
    sort(A + 1, A + n + 1);      //将物品按单位质量价值递减排序
    //初始化根节点
    e.i = 1;  e.tw = 0;  e.tv = 0;  e.ub = Bound(0, A, e, n, c);
    for(j = 1; j <= n; j++)  e.tx[j] = 0;
```

13.2.1 0-1背包问题

```
queue.InsertElem(e);                    //根节点入队
while(!queue.IsEmpty())  {              //当队列不为空时循环
    queue.DeleteTop(e); //队头节点出队成为当前扩展节点
    if(e.tw + A[e.i].w <= c) {          //左分支剪枝
        //构造左孩子节点
        e1.i = e.i + 1;  e1.tw = e.tw + A[e.i].w;
        e1.tv = e.tv + A[e.i].v;  e1.ub = e.ub;
        for(j = 1; j <= n; j++)  e1.tx[j] = e.tx[j];//复制解向量
        e1.tx[e.i] = 1;      //记录本次选择，即装入第i种物品
```

13.2.1 0-1背包问题

```
            if(e1.i > n) {                          //找到一个叶子节点
                if(e1.tv > maxv) {                  //找到一个更优解
                    maxv = e1.tv;                   //更新最优值
                    for (j = 1; j <= n; j++) //更新最优解
                        x[j] = e1.tx[j];
                }
            }
            else  queue.InsertElem(e1);             //左孩子入队
        }
```

13.2.1 0-1背包问题

```
        if((e2.ub = Bound(e.i, A, e, n, c)) > maxv) {//右分支剪枝
            //构造右孩子节点
            e2.i = e.i + 1;  e2.tw = e.tw;  e2.tv = e.tv;
            for(j = 1; j <= n; j++)  e2.tx[j] = e.tx[j];   //复制解向量
            e2.tx[e.i] = 0;          //记录本次选择，即不装入第i种物品
            if(e2.i > n) {                                  //找到一个叶子节点
                if(e2.tv > maxv) {                          //找到一个更优解
                    maxv = e2.tv;                           //更新最优值
                    for (j = 1; j <= n; j++)                //更新最优解
                        x[j] = e2.tx[j];
                }
            }
            else  queue.InsertElem(e2);                     //右孩子入队
        }
    }//End of while
}
```

13.2.1 0-1背包问题

【一个简单实例的求解过程】

已知一个0-1背包问题，背包的容量c=16，物品种类n=3，物品信息如图（a）所示，按照单位质量价值递减排序后的物品结果如图（b）所示。

物品编号 no	质量 w	价值 v
1	8	14
2	9	18
3	8	12

（a）物品信息

序号 i	物品编号 no	质量 w	价值 v	单位质量价值 p
1	2	9	18	2
2	1	8	14	1.75
3	3	8	12	1.5

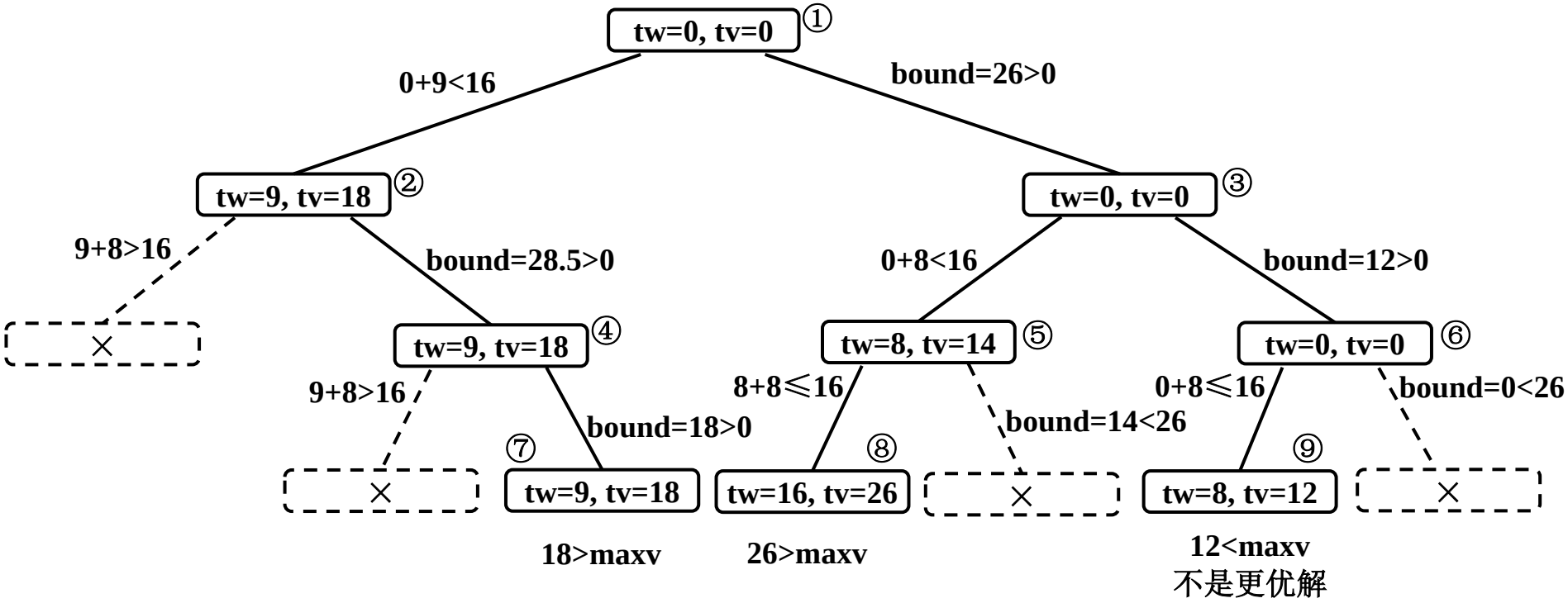
（b）按照单位质量价值递减排序后的结果

13.2.1 0-1背包问题

i	no	*w*	*v*	*p*
1	2	9	18	2
2	1	8	14	1.75
3	3	8	12	1.5

c=16

（1）队列式分支限界法的求解过程



maxv=26

x=(0, 1, 1)

队列

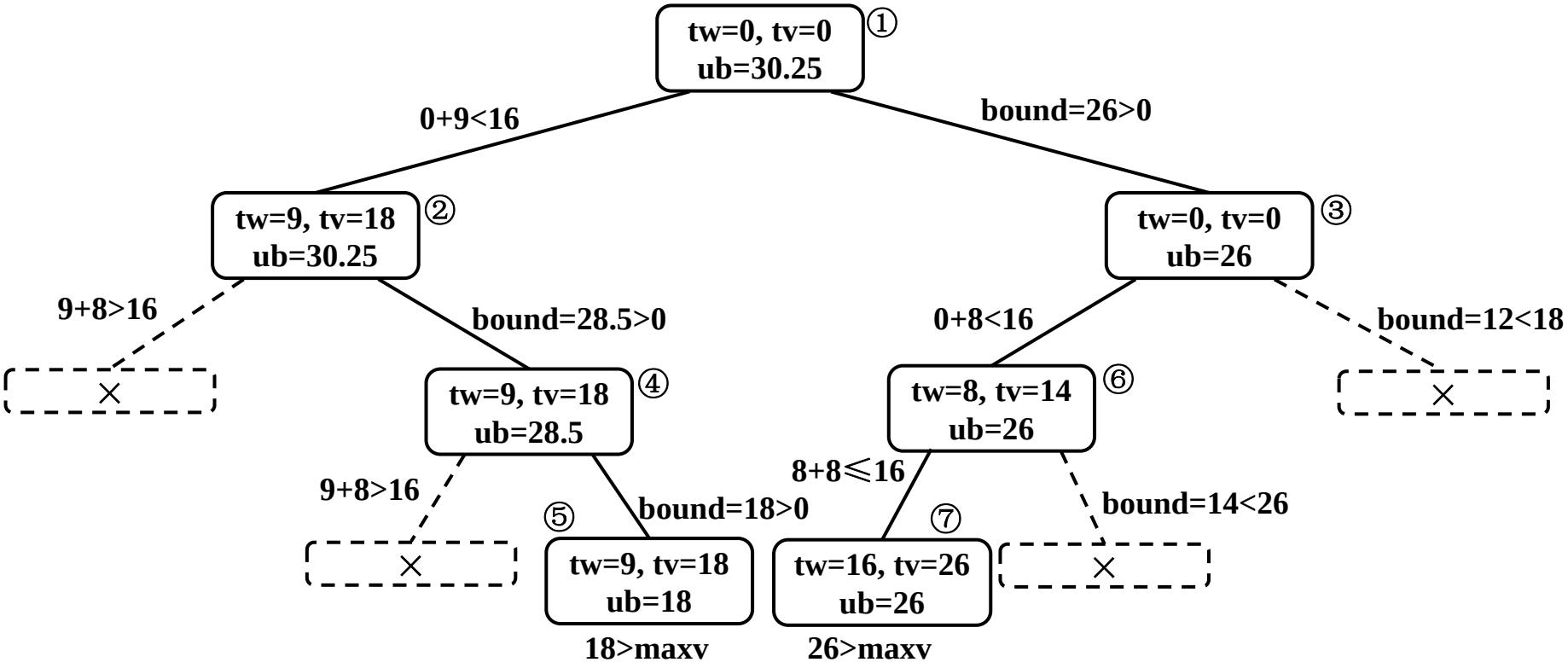
出队 ⑥

13.2.1 0-1背包问题

i	no	*w*	*v*	*p*
1	2	9	18	2
2	1	8	14	1.75
3	3	8	12	1.5

c=16

（2）优先队列式分支限界法的求解过程



maxv=26

x=(0, 1, 1)

优先队列

出队 ⑥

13.2.1 0-1背包问题

从上述实例的求解过程可知：

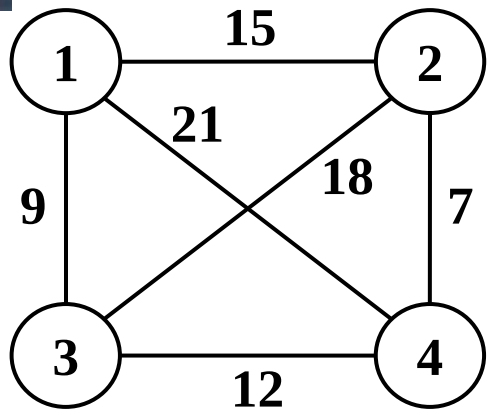
（1）队列式分支限界法的搜索过程是按照解空间树的层次逐层递进的。

（2）优先队列式分支限界法的搜索不是逐层展开的，而是按照节点的优先级跳跃式地选择扩展节点，使得搜索方向尽可能快地向最优解方向推进，从而减少搜索的节点数目，提高搜索效率。

13.2.2 旅行商问题

一个商品推销员要去*n*个城市（编号为1，2，…，*n*）推销商品，已知各城市之间的路程（或旅费）。该推销员从一个城市出发，途径每个城市一次，最后回到出发城市。推销员应该如何选择行进路线，才能使总的路程（或旅费）最短（或最少）。

例：右图所示的旅行商问题，若推销员选择从城市1出发，则最优的行进路线是：城市1->城市2->城市4->城市3->城市1。



最短（或最少）路程（或旅费）是43。

13.2.2 旅行商问题

【问题分析】

（1）确定问题的解空间

省略回路的终点，用向量$X=(x_1, x_2, \ldots, x_n)$表示旅行商问题的解向量，其中，$x_i(i=1, 2, \ldots, n)$表示哈密尔顿回路上的第$i$个顶点。

设$S=\{1, 2, \ldots, n\}$是编号为1，2……n的顶点构成的集合，则旅行商问题的显约束：

$$x_1 \in S \text{且} x_i \in S-\{x_1, x_2, \ldots, x_{i-1}\} \quad (i=2, 3, \ldots, n)$$

即哈密尔顿回路上的第i个顶点不能与前i-1个顶点重复。

旅行商问题的隐约束：

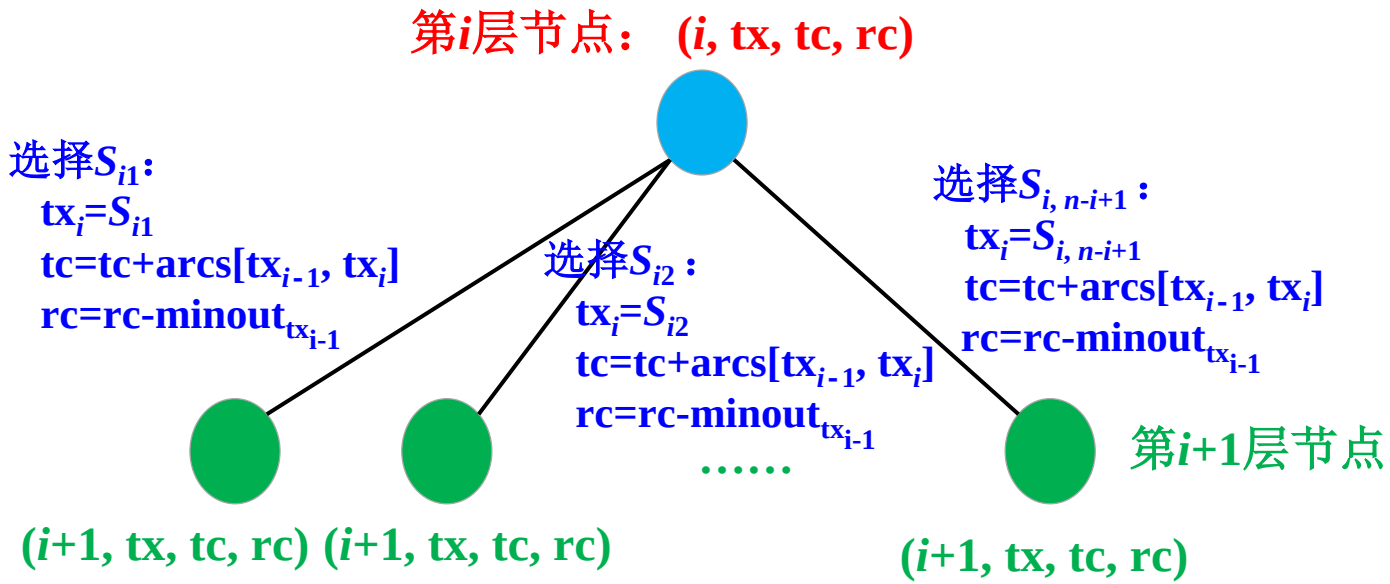
$$\forall i \in \{1, 2, \ldots, n-1\}, \ \text{arcs}[x_i, x_{i+1}] \neq \infty \text{且} \text{arcs}[x_n, x_1] \neq \infty$$

13.2.2 旅行商问题

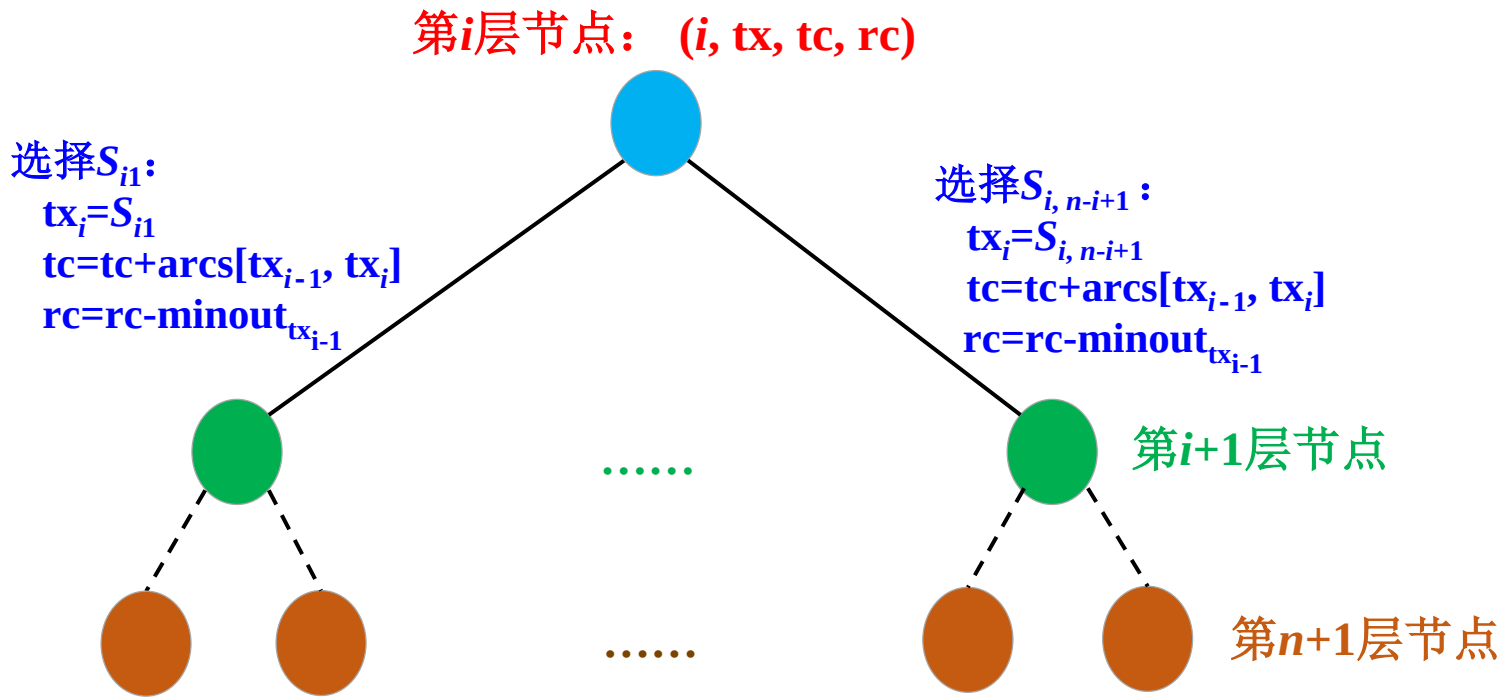
（2）确定节点的扩展规则

为解空间树的每个节点设置状态：**(*i*, tx, tc, rc)**，其中，*i*表示节点的层次；tx记录当前的解向量，即当前已经确定的路径；tc记录当前路径的长度；rc记录当前的最小出边和。

节点的扩展规则：第*i*层在$S_i=S-\{tx_1, tx_2, \ldots, tx_{i-1}\}$中选择一个顶点作为哈密尔顿回路的第*i*个顶点（$i=1, 2, \ldots, n$）。



13.2.2 旅行商问题



- 叶子结点仅仅表示找到一条从起点开始、由n个顶点构成的简单路径，需要进一步判断是否可以形成回路。

13.2.2 旅行商问题

（3）确定剪枝函数

分支的扩展规则：在未选择的顶点中选择一个顶点。

约束条件：$arcs[x_i, x_{i+1}] \neq \infty$（$i=1, \ldots, n-1$）

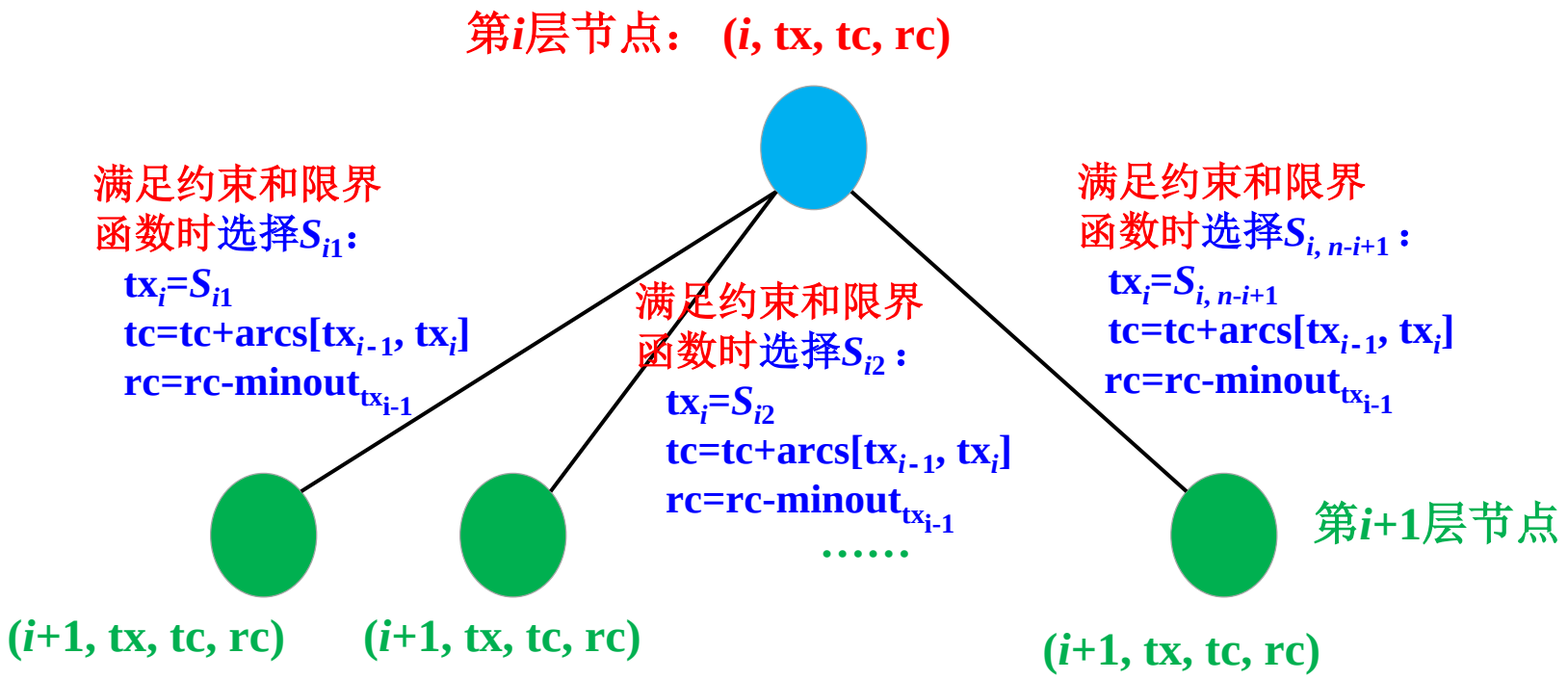
目标函数的优化目标：$min(\sum_{i=1}^{n-1} arcs[x_i, x_{i+1}] + arcs[x_n, x_1])$

剪枝函数：$arcs[tx_{i-1}, tx_i] \neq \infty$ （$i=2, \ldots, n$）（约束函数）

$bound(i) < minc$（$i=2, \ldots, n$）（限界函数）

其中，$bound(i)=tc+arcs[tx_{i-1},tx_i]+rc-minout_{tx_{i-1}}$表示选择$tx_i$作为第$i$个顶点形成的路径长度的下界；$minout_{tx_{i-1}}$表示顶点$tx_{i-1}$的最小出边的权值；minc表示当前最小的哈密尔顿回路的路径长度。

13.2.2 旅行商问题


第i层节点： (i, tx, tc, rc)
满足约束和限界函数时选择S_{i1}：
$tx_i=S_{i1}$
$tc=tc+arcs[tx_{i-1}, tx_i]$
$rc=rc-minout_{tx_{i-1}}$
满足约束和限界函数时选择S_{i2}：
$tx_i=S_{i2}$
$tc=tc+arcs[tx_{i-1}, tx_i]$
$rc=rc-minout_{tx_{i-1}}$
……
满足约束和限界函数时选择$S_{i, n-i+1}$：
$tx_i=S_{i, n-i+1}$
$tc=tc+arcs[tx_{i-1}, tx_i]$
$rc=rc-minout_{tx_{i-1}}$
第i+1层节点
(i+1, tx, tc, rc)
(i+1, tx, tc, rc)
(i+1, tx, tc, rc)

13.2.2 旅行商问题

【算法描述】

设n表示顶点数目；二维数组arcs表示邻接矩阵，其中，arcs[i][j]记录边（i，j）的权值；一维数组minout记录顶点的最小出边的权值，minout[i]记录顶点i的最小出边的权值。

结构体变量e表示解空间树的当前扩展节点，e.i表示节点层次，一维数组e.tx记录节点对应的路径，e.tx[i]记录路径上第i个顶点的编号，e.tc记录节点对应的路径长度，e.rc记录节点对应的剩余顶点的最小出边和；结构体变量e1表示扩展节点e的孩子节点。

一维数组x记录最优路径，x[i]表示路径上第i个顶点的编号；minc记录最优路径的长度。

13.2.2 旅行商问题

应用分支限界法求解旅行商问题的算法描述如下。

Step 1：初始化路径x为顶点的初始排列，路径的第一个顶点为起点；初始化最优路径的长度minc。

Step 2：记录所有顶点的最小出边权值于minout。

Step 3：初始化解空间树的根节点的孩子（解空间树第2层的节点），并入队。

Step 4：当队列不为空时，反复进行如下处理，否则结束算法。

Step 4.1 根据出队规则出队队头节点e。

13.2.2 旅行商问题

Step 4.2：判断节点e是否解空间树的叶子节点，即是否满足e.i>n，分为如下两种情况。

① 若e.i>n，表示找到一条从起点开始遍历所有顶点的简单路径，则判断路径的第n个顶点与起点是否相连且回路的路径长度是否小于minc，满足上述条件表明找到一条更优的哈密尔顿回路，更新x和minc。

② 若e.i≤n，则对j=e.i，e.i+1，…，n，检查顶点e.tx[j]是否满足约束函数arcs[e.tx[e.i-1]][e.tx[j]]≠∞及限界函数bound<minc（bound=e.tc+arcs[e.tx[e.i-1]][e.tx[j]]+e.rc-minout[e.tx[e.i-1]]），若满足，则构造孩子节点e1并入队。

13.2.2 旅行商问题

【算法实现——优先队列式分支限界法】

取节点的路径长度下界作为节点的优先级，路径长度下界越小的节点越先出队，因此，使用小顶堆作为优先队列。

```
#include "MinHeap.h"      //使用小顶堆作为优先队列，参见配套源码
const int INFINITE = INT_MAX;       //定义作为无穷的数据
struct STNode  {                    //解空间树节点类型
    int i;                          //节点层次
    int tx[MAX];                    //当前节点的路径
    int tc, rc;          //当前节点的路径长度，剩余顶点的最小出边和
    int lb;                         //当前节点的路径长度下界
```

13.2.2 旅行商问题

```
    //重载<=运算符用于小顶堆的元素比较
    bool operator<=(const STNode &s) const
    {  return lb <= s.lb;  }
    //重载>运算符用于小顶堆的元素比较
    bool operator>(const STNode &s) const
    {  return lb > s.lb;  }
    //重载赋值运算符用于小顶堆的元素赋值
    STNode& operator=(const STNode &s)  {
        if(this != &s)  {
            i = s.i;  tc = s.tc;  rc = s.rc;  lb = s.lb;
            for(int j = 0; j < MAX; j++)  tx[j] = s.tx[j];
        }
        return *this;
    }
};
```

13.2.2 旅行商问题

```
//优先队列式分支限界法求解旅行商问题
void TSP(int n, int arcs[][MAX], int start, int &minc, int * x)  {
    int * minout = new int[n];    //记录每个顶点的最小出边权值
    int rc = 0;                   //记录顶点的最小出边之和
    STNode e, e1;                 //解空间树的节点
    MinHeap<STNode> queue(MAX);   //定义小顶堆作为优先队列
    for(int i = 1; i <= n; i++)  x[i] = i;      //初始化路径
    x[1] = start;  x[start] = 1;      //起点start作为路径的第1个顶点
    minc = INFINITE;              //初始化最优哈密尔顿回路的长度
```

13.2.2 旅行商问题

```
for (int i = 1; i <= n; i++)  {          //寻找所有顶点的最小出边权值
      minout[i] = INFINITE;          //初始化顶点i的最小出边权值
      for(int j = 1; j <= n; j++)       //在邻接点中寻找权值最小的边
            if(arcs[i][j] != INFINITE && arcs[i][j] < minout[i])
                  minout[i] = arcs[i][j];
      if(minout[i] == INFINITE)  return; //不存在哈密尔顿回路
      rc += minout[i];                  //累加顶点的最小出边权值
}
e.i = 2;  e.tc = 0;  e.rc = rc;  e.lb = 0;          //初始化第2层节点
for(int i = 1; i <= n; i++)  e.tx[i] = x[i];      //初始化节点的路径
```

13.2.2 旅行商问题

```
queue.InsertElem(e);                    //节点e入队
while(!queue.IsEmpty()) {               //当队列不为空时循环
    queue.DeleteTop(e);                 //队头节点出队成为当前扩展节点
    if(e.i > n) {                       //找到一个叶子节点
        if(arcs[e.tx[n]][e.tx[1]] != INFINITE && e.tc +
            arcs[e.tx[n]][e.tx[1]] < minc) {
            //当前路径形成回路且更优
            minc = e.tc + arcs[e.tx[n]][e.tx[1]];   //更新路径长度
            for(int j = 1; j <= n; j++)  x[j] = e.tx[j];  //更新回路
        }
    }
```

13.2.2 旅行商问题

```
        else {
            for(int j = e.i; j <= n; j++) {
                if(arcs[e.tx[e.i - 1]][e.tx[j]] != INFINITE) {//满足约束函数
                    //计算节点对应的路径长度下界
                    int bound = e.tc + arcs[e.tx[e.i - 1]][e.tx[j]] + e.rc
                                - minout[e.tx[e.i - 1]];
                    if(bound < minc)  {          //满足限界函数
                        //构造孩子节点
                        e1.i = e.i + 1;
                        e1.tc = e.tc + arcs[e.tx[e.i - 1]][e.tx[j]];
                        e1.rc = e.rc - minout[e.tx[e.i - 1]];
                        e1.lb = bound;
                        for(int k = 1; k <= n; k++)  e1.tx[k] = e.tx[k];
                        e1.tx[e.i] = e.tx[j];  e1.tx[j] = e.tx[e.i];
                        queue.InsertElem(e1);          //节点e1入队
                    }
                }   }    }    }//End of while
}//End of TSP
```

13.2.3 流水作业调度

编号为1, 2, …, n的n个作业需要使用由两台机器M1和M2组成的流水线进行加工。每个作业都必须先经过M1、再经过M2方可完成加工。对于作业i(i=1, 2, …, n)，使用M1和M2加工的时间分别是a_i和b_i。要求寻找一个最优作业调度，即寻找这n个作业的最优加工次序，使得完成所有作业的加工所需的时间最少。

例如，已知4个作业在机器M1和M2上加工的时间如下表所示，这4个作业的最优调度是1，4，3，2，需要的最少加工时间为36。

作业编号	1	2	3	4
M1加工时间	5	10	9	7
M2加工时间	7	5	9	8

13.2.3 流水作业调度

【问题分析】

◆如何计算一个作业调度的加工时间？

M1对作业的加工是连续的，其完成所有作业的加工时间等于M1加工这些作业的实际时间之和。

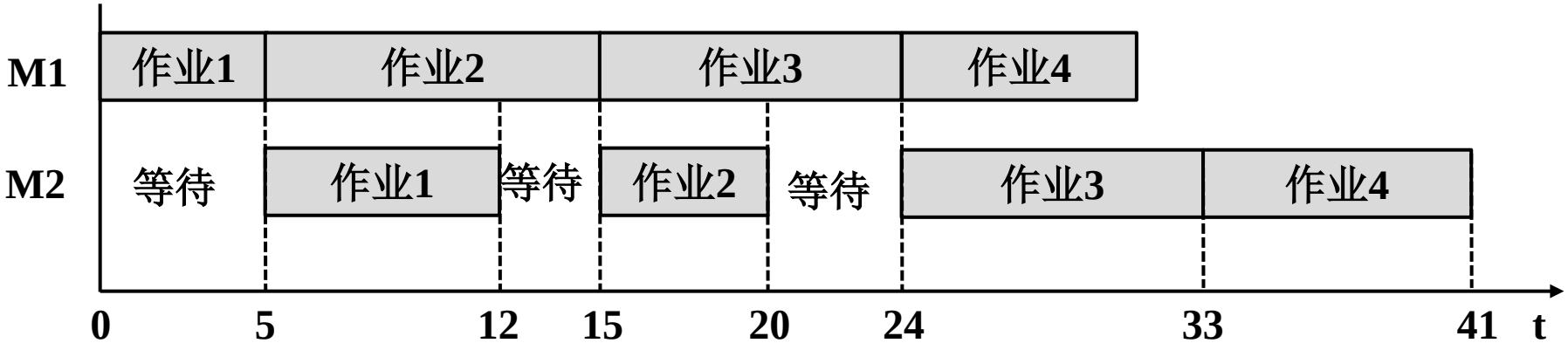
M2完成所有作业的加工时间等于M2加工这些作业的实际时间与等待时间之和。

所有作业总是先由M1加工再由M2加工，因此，一个作业调度的加工时间就是M2完成所有作业的加工时间。

13.2.3 流水作业调度

作业编号	1	2	3	4
M1加工时间	5	10	9	7
M2加工时间	7	5	9	8

若采用1，2，3，4的调度次序加工作业，M1和M2加工的时间线如下图所示：



这一调度的加工时间就是M2完成所有作业的加工时间41。

13.2.3 流水作业调度

（1）确定问题的解空间

解向量：$X=(x_1, x_2, \ldots, x_n)$，其中，$x_i(i=1, 2, \ldots, n)$表示第$i$个被加工的作业。

设$S=\{1, 2, \ldots, n\}$是编号为$1, 2, \ldots, n$的作业构成的集合，则流水作业调度问题的显约束：

$$x_1 \in S \text{且} x_i \in S-\{x_1, x_2, \ldots, x_{i-1}\} \quad (i=2, 3, \ldots, n)$$

隐约束：无。

解空间树类型：排列树。

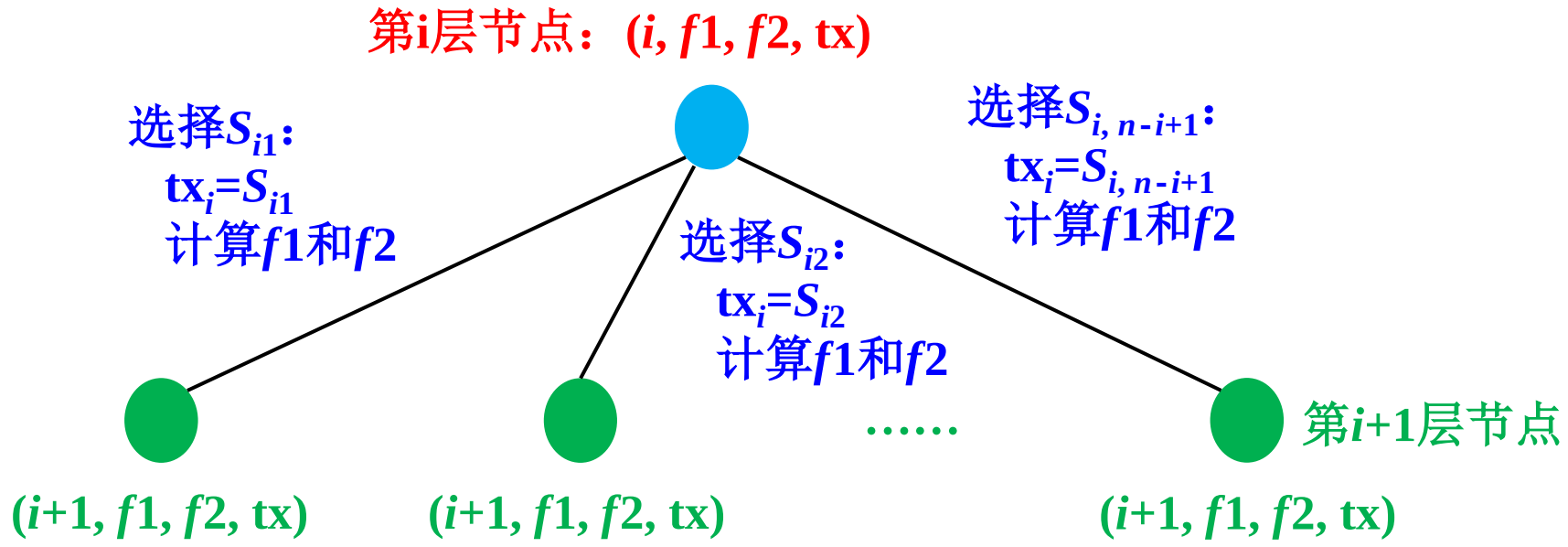
13.2.3 流水作业调度

（2）确定节点的扩展规则

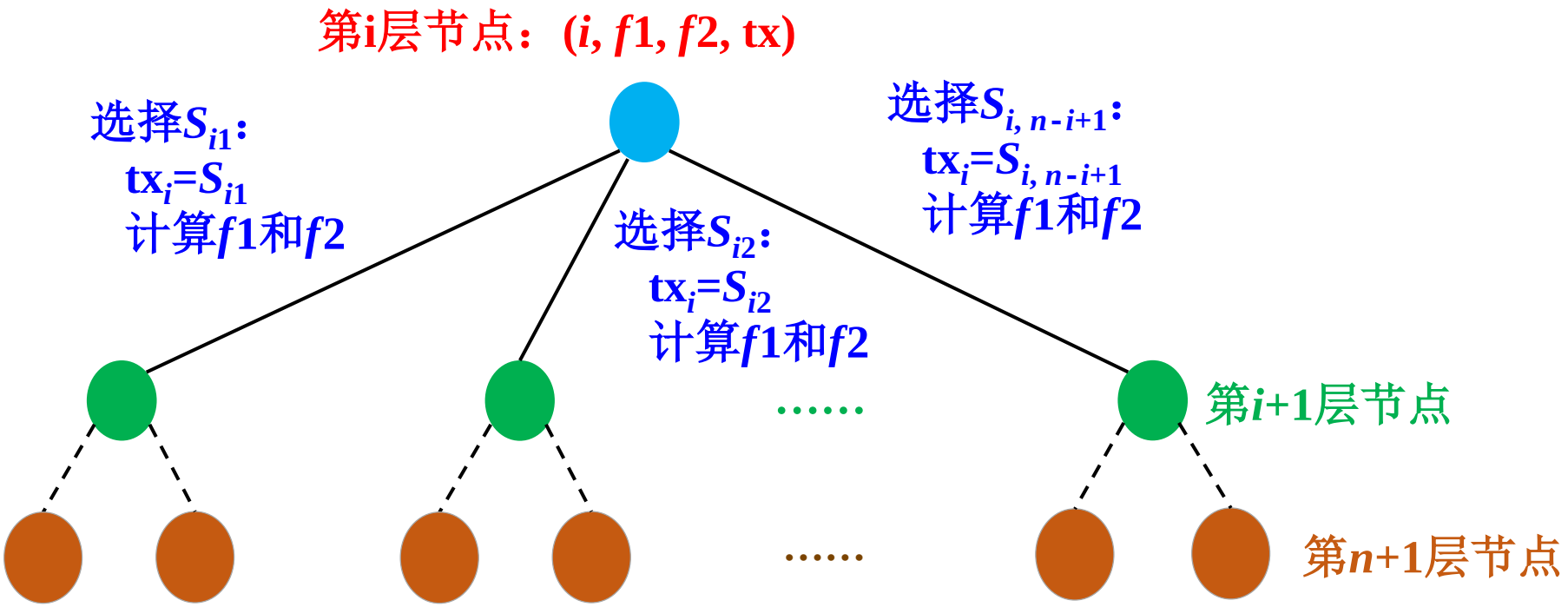
为解空间树的每个节点设置状态：**(*i*, *f*1, *f*2, tx)**，其中，*i*表示节点的层次；*f*1记录M1加工完成已调度作业的时间；*f*2记录M2加工完成已调度作业的时间；tx记录当前的作业调度（tx_1~tx_{i-1}是已调度的作业，tx_i~tx_n是尚未调度的作业）。

节点的扩展规则：第$i(i=1, \ldots, n)$层在$S_i=S-\{tx_1, tx_2, \ldots, tx_{i-1}\}$中选择一个作业作为作业调度的第*i*个作业。

13.2.3 流水作业调度


第i层节点：(i, f1, f2, tx)
选择S_{i1}：
$tx_i=S_{i1}$
计算f1和f2
选择S_{i2}：
$tx_i=S_{i2}$
计算f1和f2
……
选择$S_{i,n-i+1}$：
$tx_i=S_{i,n-i+1}$
计算f1和f2
第i+1层节点
(i+1, f1, f2, tx)
(i+1, f1, f2, tx)
(i+1, f1, f2, tx)

13.2.3 流水作业调度



- 叶子节点表示找到一个作业调度方案。

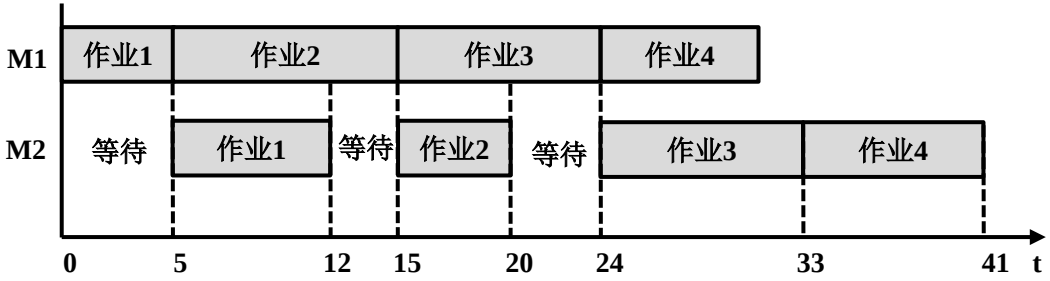
13.2.3 流水作业调度

➢如何计算孩子节点的*f*1和*f*2？

若作业调度的第*i*个作业选择了作业*j*(即$tx_i=j$)，则

对于*f*1，$f1=f1+a_j$

对于*f*2，分为两种情况：



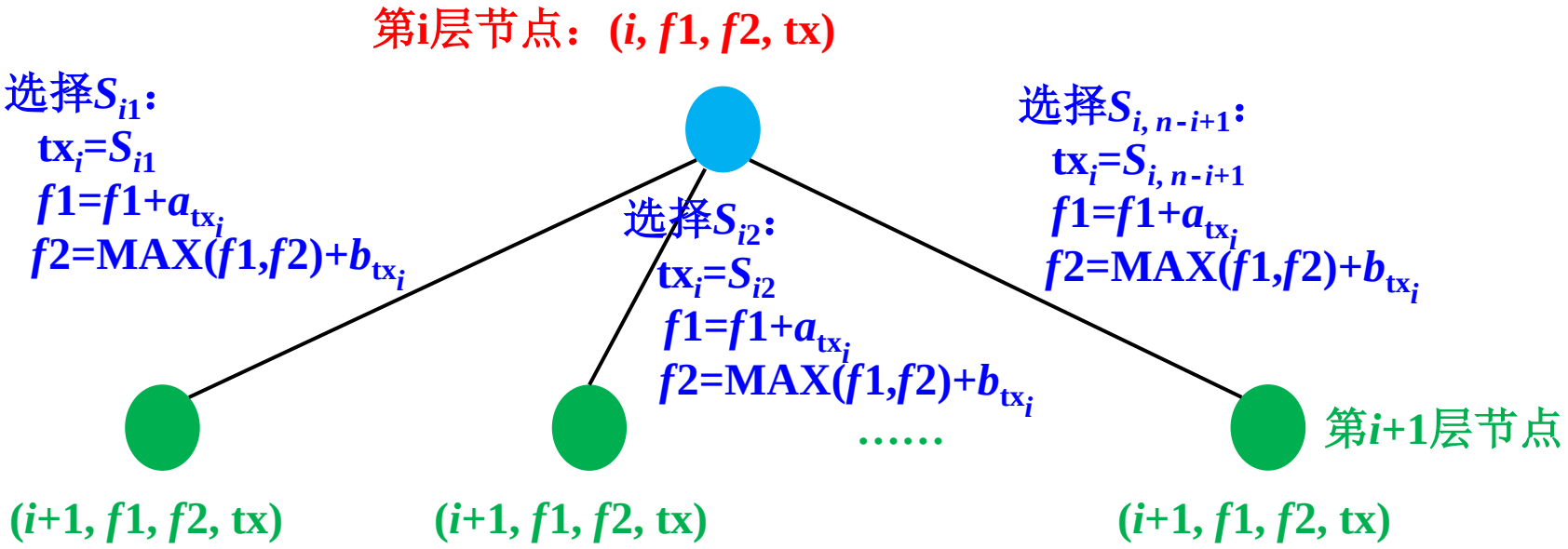
① M1加工完作业*j*时，M2处于等待状态，即*f*1>*f*2，则$f2=f1+b_j$。

② M1加工完作业*j*时，M2处于加工状态，即*f*1<*f*2，则$f2=f2+b_j$。

综上：$f2=MAX(f1, f2)+b_j$

注意：在计算扩展节点的*f*1和*f*2时，必须先计算*f*1，后计算*f*2，计算顺序不能颠倒。

13.2.3 流水作业调度


第i层节点：(i, f1, f2, tx)
选择S_{i1}：
$tx_i=S_{i1}$
$f1=f1+a_{tx_i}$
$f2=MAX(f1,f2)+b_{tx_i}$
选择S_{i2}：
$tx_i=S_{i2}$
$f1=f1+a_{tx_i}$
$f2=MAX(f1,f2)+b_{tx_i}$
……
选择$S_{i,n-i+1}$：
$tx_i=S_{i,n-i+1}$
$f1=f1+a_{tx_i}$
$f2=MAX(f1,f2)+b_{tx_i}$
第i+1层节点
(i+1, f1, f2, tx)
(i+1, f1, f2, tx)
(i+1, f1, f2, tx)

13.2.3 流水作业调度

（3）确定剪枝函数

分支的扩展规则：在未调度的作业中选择一个作业。

约束条件：无

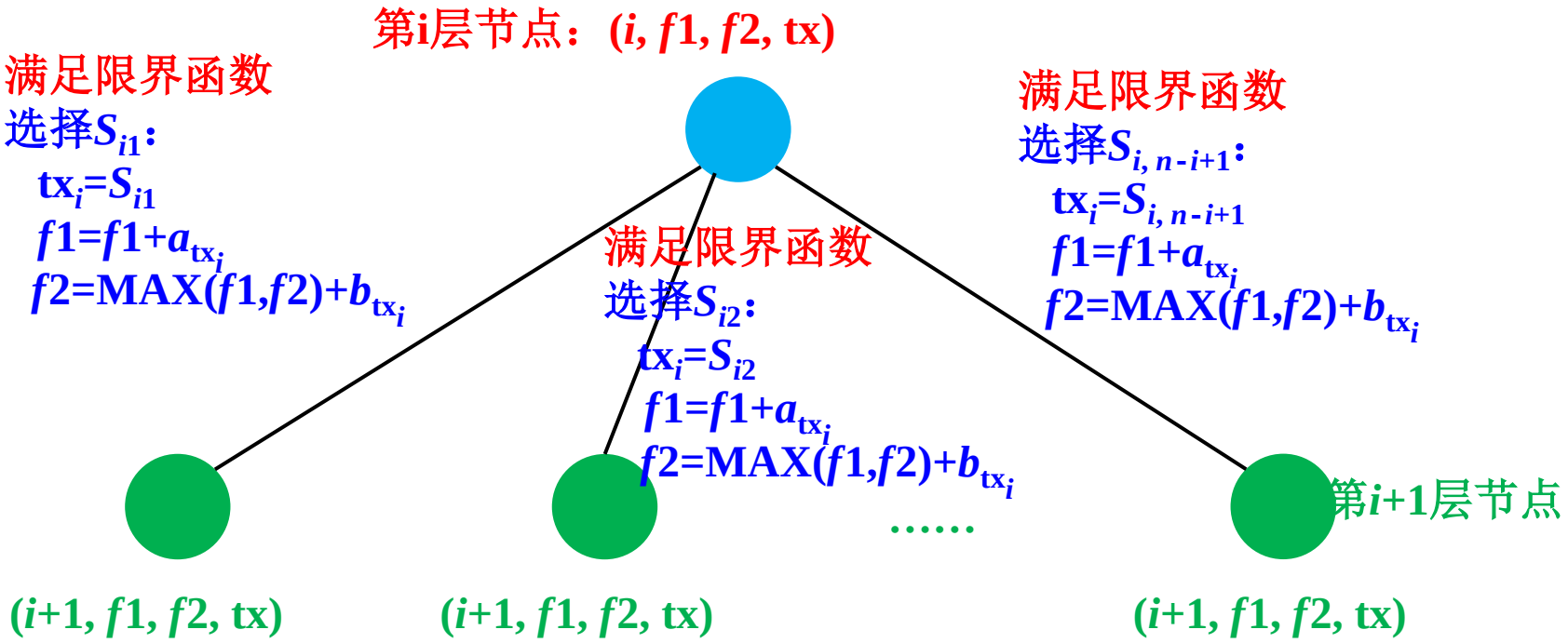
目标函数的优化目标：$min(\sum_{i=1}^{n} \mathbf{time}_{\mathbf{tx}_i})$，$\mathbf{time}_{\mathbf{tx}_i}$表示作业$\mathbf{tx}_i$的加工时间

剪枝函数：无约束函数

bound(*i*)<mint（限界函数）

其中，**bound(*i*)**表示对第***i***个作业做出选择之后，所形成的作业调度的加工时间下界；**mint**表示最优调度的加工时间。

13.2.3 流水作业调度


第i层节点：(i, f1, f2, tx)
满足限界函数
选择S_{i1}：
$tx_i=S_{i1}$
$f1=f1+a_{tx_i}$
$f2=MAX(f1,f2)+b_{tx_i}$
满足限界函数
选择S_{i2}：
$tx_i=S_{i2}$
$f1=f1+a_{tx_i}$
$f2=MAX(f1,f2)+b_{tx_i}$
……
满足限界函数
选择$S_{i,n-i+1}$：
$tx_i=S_{i,n-i+1}$
$f1=f1+a_{tx_i}$
$f2=MAX(f1,f2)+b_{tx_i}$
第i+1层节点
(i+1, f1, f2, tx)
(i+1, f1, f2, tx)
(i+1, f1, f2, tx)

13.2.3 流水作业调度

◆ 如何计算作业调度的加工时间下界？

设当前扩展节点为解空间树的第i层节点，作业调度中已确定的作业是tx_1~tx_{i-1}，若第i个作业选择作业j（即$tx_i=j$），则这一选择所形成的作业调度的加工时间下界：

$$bound(i)=f2'+rt$$

$f1=f1+a_j$
$f2=MAX(f1, f2)+b_j$
$f2'=f2$

其中，$f2'$表示M2加工完成作业tx_1~tx_i（作业调度中的第1~i个作业）的时间；rt表示M2加工完成剩余的作业tx_{i+1}~tx_n（第i+1~n个作业）最少需要的时间。

显然，$rt=b_{tx_{i+1}}+b_{tx_{i+2}}+\ldots+b_{tx_n}$，即这些作业在M1上完成加工后，无需任何等待即可在M2上加工。

13.2.3 流水作业调度

作业编号	1	2	3	4
M1加工时间	5	10	9	7
M2加工时间	7	5	9	8

当前扩展节点*e*（第2层）

*e*的孩子结点*e*1

e.i=2

e.f1=10

e.f2=15

e.tx=(2, 1, 3, 4)

若选择作业4

作为第2个作业

e1.i=e.i+1=3

e1.f1=e.f1+a_4=17

e1.f2=max(e1.f1, e.f2)+b_4=25

e1.tx=(2, 4, 3, 1)

bound(2)=*e*1.*f*2+b_3+b_1=41

13.2.3 流水作业调度

【算法描述】

设n表示作业数目；一维数组a记录机器M1加工作业的时间，a[i]为M1加工作业i的时间；一维数组b记录机器M2加工作业的时间，b[i]为M2加工作业i的时间。

结构体变量e表示解空间树的当前扩展节点，e.i表示节点层次，e.f1表示M1加工完成已调度作业的时间，e.f2表示M2加工完成已调度作业的时间，一维数组e.tx记录当前的作业调度，e.tx[i]表示作业调度中第i个作业的编号；结构体变量e1表示扩展节点e的孩子节点。

一维数组x记录最优的作业调度，x[i]表示调度中第i个作业的编号；mint记录最优调度的加工时间。

13.2.3 流水作业调度

分支限界法求解流水作业调度问题的算法描述如下。

Step 1：初始化最优调度的加工时间**mint**。

Step 2：初始化根节点，并入队。

Step 3：当队列不为空时，反复进行如下处理，否则结束算法。

Step 3.1：根据出队规则出队队头节点**e**。

Step 3.2：对**j=e.i，e.i+1，…，n**，进行如下处理。

Step 3.2.1：将**e.tx[j]**作为作业调度中的第**i**个作业，建立对应的孩子节点**e1**。

13.2.3 流水作业调度

Step 3.2.2：判断**e1**是否解空间树的叶子节点，即是否满足**e1.i>n**，分为如下两种情况。

① 若**e1.i>n**，表明形成了一个完整的作业调度，则判断该调度的加工时间是否小于**mint**，若小于则找到了一个加工时间更少的作业调度，更新**mint**和**x**。

② 若**e1.i≤n**，则判断扩展该节点是否满足限界函数，满足则将其入队。

13.2.3 流水作业调度

【算法实现——优先队列式分支限界法】

取节点的加工时间下界作为节点的优先级，加工时间下界越小的节点越先出队。因此，使用小顶堆作为优先队列。

```
#include "MinHeap.h"    //使用小顶堆作为优先队列，参见配套源码
const int INFINITE = INT_MAX;    //定义作为无穷的数据
struct STNode  {          //解空间树节点类型
    int i;                 //节点层次
    int f1, f2;            //已调度的作业在M1和M2上的加工时间
    int lb;                //节点相应的加工时间下界
    int tx[MAX];           //当前的作业调度
```

13.2.3 流水作业调度

```
    //重载<=运算符用于小顶堆的元素比较
    bool operator<=(const STNode &s) const
    {  return lb <= s.lb;  }
    //重载>运算符用于小顶堆的元素比较
    bool operator>(const STNode &s) const
    {  return lb > s.lb;  }
   //重载赋值运算符用于小顶堆的元素赋值
    STNode& operator=(const STNode &s)  {
        if (this != &s)  {
            i = s.i;  f1 = s.f1;   f2 = s.f2;  lb = s.lb;
            for(int j = 0; j < MAX; j++)  tx[j] = s.tx[j];
        }
        return *this;
    }
};
```

13.2.3 流水作业调度

```
void swap(int &a, int &b)        //交换a和b
{    int t = a;  a = b;  b = t;    }

int bound(STNode &e, int n, int * b) {  //求节点e相应的加工时间下界
        int sum = 0;
        for(int i = e.i; i <= n; i++) //累加未调度作业在M2上的加工时间
                sum += b[e.tx[i]];
        return e.f2 + sum;              //计算并返回加工时间下界
}
```

13.2.3 流水作业调度

```
//应用分支限界法求解流水作业调度问题
void JobSchedule(int n, int * a, int * b, int &mint, int * x)  {
    STNode e, e1;                    //解空间树的节点
    MinHeap<STNode> queue(MAX);  //定义小顶堆作为优先队列
    mint = INFINITE;                          //初始化最优调度的加工时间
    //初始化根节点
    e.i = 1;  e.f1 = 0;  e.f2 = 0;
    for(int j = 1; j <= n; j++)  e.tx[j] = j;  //初始化作业调度的排列
    e.lb = bound(e, n, b);                    //计算根节点的加工时间下界
    queue.InsertElem(e);                      //节点e入队
```

13.2.3 流水作业调度

```
while(!queue.IsEmpty()) {                    //当队列不为空时循环
    queue.DeleteTop(e);                      //队头节点出队成为当前扩展节点
    for(int j = e.i; j <= n; j++) {
        //构造孩子节点
        e1.i = e.i + 1;
        for(int k = 1; k <= n; k++)  e1.tx[k] = e.tx[k];
        swap(e1.tx[e.i], e1.tx[j]);
        e1.f1 = e.f1 + a[e1.tx[e.i]];  //计算M1加工完成第1~i个作业的时间
        //计算M2加工完成第1~i个作业的时间
        e1.f2 = (e1.f1 > e.f2 ? e1.f1 : e.f2) + b[e1.tx[e.i]];
        e1.lb = bound(e1, n, b);      //计算节点e1相应的加工时间下界
```

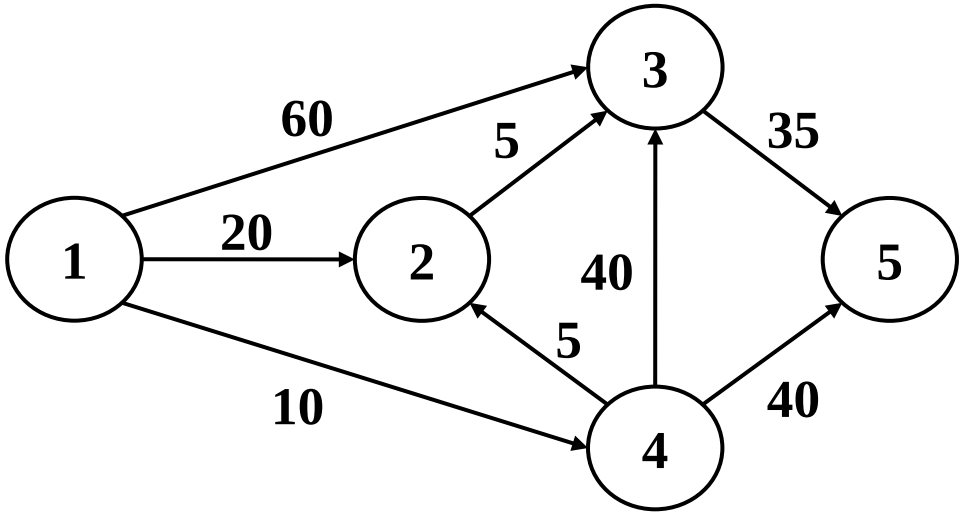
13.2.3 流水作业调度

```
            if(e1.i > n) {     //找到一个叶子节点，即确定了一个作业调度
                if(e1.f2 < mint)  {          //比较求最优解
                    mint = e1.f2;          //更新最优调度的加工时间
                    for(int k = 1; k <= n; k++)  x[k] = e1.tx[k];
                }
            }
            else if(e1.lb < mint)
                queue.InsertElem(e1);//满足限界函数，将孩子节点e1入队
        }//End of for
    }//End of while
}
```

13.2.4 单源点最短路径问题

给定一个有向网*G*（所有弧的权均为非负值）与源点*v*，求从*v*到*G*中其余各顶点的最短路径。

例如，在下图所示的有向网中寻找从顶点1到其他顶点的最短路径。



13.2.4 单源点最短路径问题

【问题分析】

（1）确定问题的解空间

用向量$P=(p_1, p_2, \ldots, p_n)$表示问题的解向量，其中，$p_i$（$i=1, 2, \ldots, n$）表示从源点到顶点$i$的最短路径上顶点$i$的前驱。

用向量$D=(d_1, d_2, \ldots, d_n)$记录每个顶点的最短路径长度。

设$V=\{1, 2, \ldots, n\}$是由编号为1, 2, …, n的顶点构成的集合，s表示源点的编号，则显约束：

$$p_s=-1且\forall i \in V-\{s\}，\ p_i \in V且p_i \neq i$$

采用图的邻接矩阵表示法存储有向网中顶点与边的数据，且设arcs表示邻接矩阵，则隐约束：

$$\forall i \in V-\{s\}，\ arcs[p_i, i] \neq \infty$$

13.2.4 单源点最短路径问题

（2）确定节点的扩展规则

为解空间树的每个节点设置状态：**(vno, dist)**，其中，**vno**表示顶点编号；**dist**记录顶点**vno**当前的最短路径长度。

在解空间树的分支节点处，只会向该分支节点对应顶点的邻接点扩展。

设$e=(i, d_i)$为当前扩展节点，节点e的扩展规则：

$\forall j \in V-\{s\}$，若$arcs[i, j] \neq \infty$，则从节点e扩展得到顶点编号为j的孩子节点。

13.2.4 单源点最短路径问题

（3）确定剪枝函数

设$e=(i, d_i)$为当前扩展节点。

节点的扩展规则已经体现了问题的隐约束，因此，节点扩展的约束函数：

$$\forall j \in V-\{s\}，\ \mathrm{arcs}[i, j] \neq \infty$$

若以节点e对应的顶点i作为前驱，其邻接点j可以得到更短的路径，则该扩展是有意义的。因此，节点扩展的限界函数：

$$e.d_i+\mathrm{arcs}[i, j]<d_j$$

其中，d_j表示顶点i的邻接点j的当前最短路径长度。

13.2.4 单源点最短路径问题

【算法描述】

设n表示顶点数目；start表示源点的编号；二维数组arcs记录有向网的邻接矩阵。

一维数组path记录顶点在最短路径上的前驱；一维数组dist记录顶点的最短路径长度。

结构体变量e表示解空间树的当前扩展节点，e.vno表示顶点编号，e.dist记录顶点的当前最短路径长度；结构体变量e1表示扩展节点e的孩子节点。

13.2.4 单源点最短路径问题

应用分支限界法求解单源点最短路径问题的算法如下。

Step 1：初始化数组**path**和**dist**。

Step 2：初始化解空间树根节点，并入队。

Step 3：当队列不为空时，反复进行如下处理，否则结束算法。

Step 3.1：根据出队规则出队队头节点**e**。

Step 3.2：对**j=1，2，…，n**，若顶点**j**是顶点**e.vno**的邻接点，即**arcs[e.vno][j]≠∞**，则判断以顶点**e.vno**作为前驱，从源点**start**到顶点**j**的路径是否比原来更短，即判断是否满足**e.dist+arcs[e.vno][j]<dist[j]**，若满足，则更新**path[j]**和**dist[j]**，并构造孩子节点**e1**入队。

13.2.4 单源点最短路径问题

【算法实现——优先队列式分支限界法】

取节点对应顶点的最短路径长度作为节点的优先级，最短路径长度越小的节点越先出队。因此，使用小顶堆作为优先队列。

```
#include "MinHeap.h" //使用小顶堆作为优先队列，参考配套源码
const int INFINITE = INT_MAX;       //定义作为无穷的数据
struct STNode  {                    //解空间树节点类型
    int vno;                        //顶点编号
    int dist;                       //最短路径长度
    //重载<=运算符用于小顶堆的元素比较
    bool operator<=(const STNode &s) const
    {   return dist <= s.dist;   }
```

13.2.4 单源点最短路径问题

```
    //重载>运算符用于小顶堆的元素比较
    bool operator>(const STNode &s) const
    {    return dist > s.dist;    }
    //重载赋值运算符用于小顶堆的元素赋值
    STNode& operator=(const STNode &s)  {
        if (this != &s)
        {    vno = s.vno;    dist = s.dist;    }
        return *this;
    }
};
```

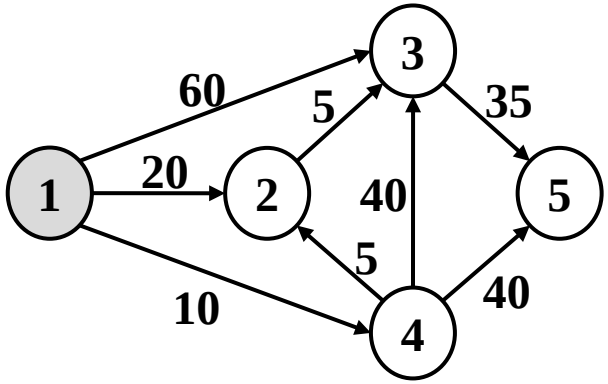
13.2.4 单源点最短路径问题

```
//应用分支限界法求解单源点最短路径问题
void ShortestPath(int n, int start, int arcs[][MAX], int * dist, int * path) {
    STNode e, e1;                          //解空间树的节点
    MinHeap<STNode> queue(MAX);            //定义优先队列
    for (int i = 1; i <= n; i++)           //初始化数组path和dist
    {  path[i] = -1;  dist[i] = INFINITE;  }
    e.vno = start;          e.dist = 0;    //初始化根节点
    queue.InsertElem(e);                   //节点e入队
    while(!queue.IsEmpty())   {            //当队列不为空时循环
        queue.DeleteTop(e);        //队头节点出队成为当前扩展节点
```

13.2.4 单源点最短路径问题

```
        for(int j = 1; j <= n; j++) {    //寻找顶点e.vno的邻接点
            if(j != start && arcs[e.vno][j] != INFINITE && e.dist +
                arcs[e.vno][j] < dist[j]) {  //满足约束函数和限界函数
                path[j] = e.vno;                    //更新前驱顶点
                dist[j] = e.dist + arcs[e.vno][j];  //更新最短路径长度
                e1.vno = j;  e1.dist = dist[j];     //构造孩子节点e1
                queue.InsertElem(e1);               //孩子节点e1入队
            }
        }//End of for
    }//End of while
}
```

13.2.4 单源点最短路径问题



A(1,0) (顶点编号, 路径长度)

1→2
0+20<∞

1→3
0+60<∞

1→4
0+10<∞

path[2]=1
dist[2]=20
B(2,20)

path[3]=1
dist[3]=60
C(3,60)

path[4]=1
dist[4]=10
D(4,10)

2→3
20+5>20

3→5
60+35>50

4→2
10+5<20

4→3
10+40<60

4→5
10+40<∞

path[2]=4
dist[2]=15
E(2,15)

path[3]=4
dist[3]=50
F(3,50)

path[5]=4
dist[5]=50
G(5,50)

2→3
15+5<50

3→5
50+35>50

path[3]=2
dist[3]=20
H(3,20)

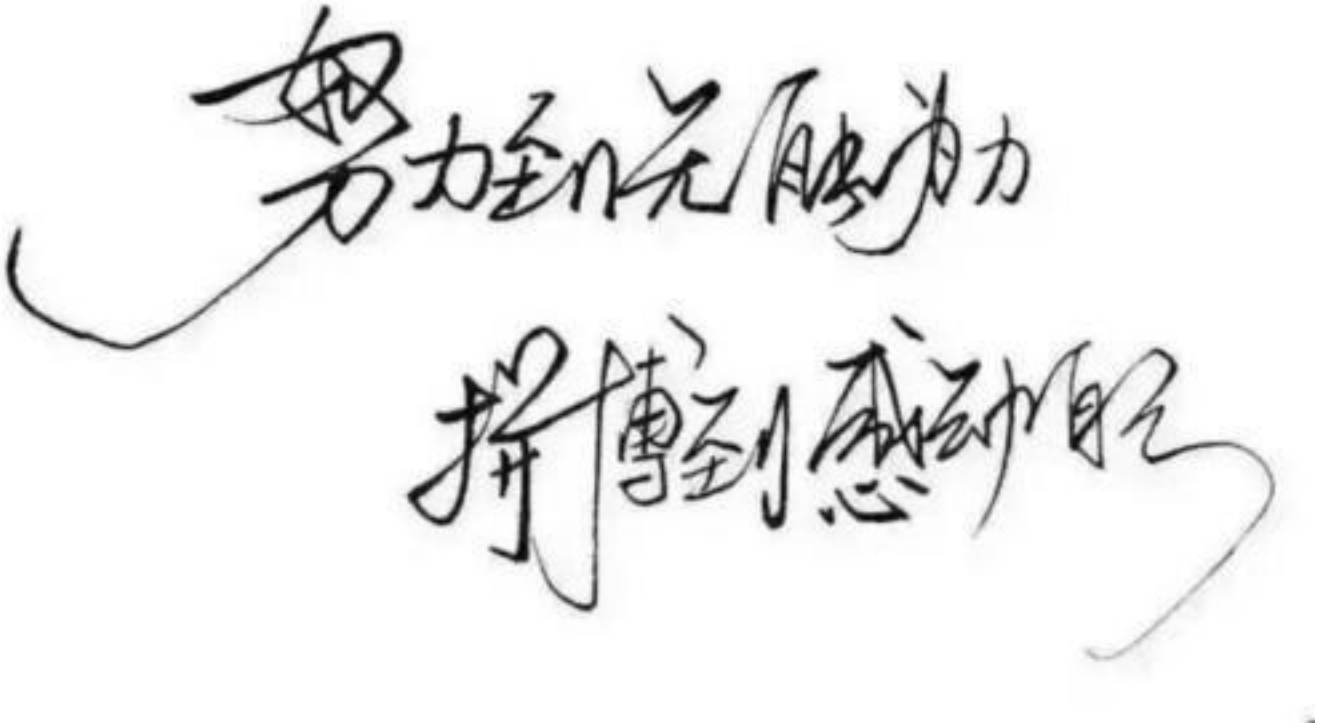
3→5
20+35>50

编号	1	2	3	4	5
path	-1	4	2	1	4
dist	∞	15	20	10	50

13.3 小结

13.3 小结

- 分支限界法类似于回溯法，通过搜索问题的解空间树寻找问题的解，但其求解目标与搜索方法不同于回溯法。分支限界法通常用于求解问题的一个可行解或最优解，使用广度优先的方法搜索解空间树。
- 根据活节点成为扩展节点的不同方式，分支限界法分为队列式分支限界法和优先队列式分支限界法。队列式分支限界法使用完全的广度优先策略搜索解空间树；而优先队列式分支限界法是结合最小耗费或最大效益的策略搜索解空间树，可以推动搜索更快地向可能产生最优解的分支前进。
- 由于在解空间树上的搜索存在跳跃性，因此分支限界法需要为节点提供比回溯法更多的空间存储搜索过程中的解。


努力到无能为力
拼搏到感动自己