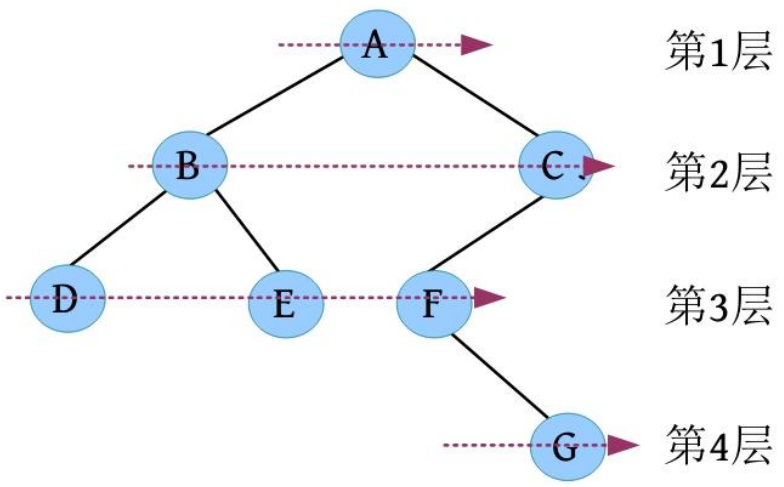
广度优先搜索：分支限界法

在图的应用中已讲过图的广度优先搜索，树上的广度优先搜索实际上就是层次遍历。首先遍历第 1 层，然后第 2 层……同一层按照从左向右的顺序访问，直到最后一层。一棵树如下图所示，首先遍历第 1 层 A；然后遍历第 2 层，从左向右遍历 B、C；再遍历第 3 层，从左向右遍历 D、E、F；再遍历第 4 层 G。



分支限界法通常**以广度优先或以最小耗费（最大效益）优先的方式搜索问题的解空间树。**首先将根节点加入活节点表中，接着从活节点表中取出根节点，使其成为当前扩展节点，一次性生成其所有孩子节点，判断对孩子节点是舍弃还是保留，**舍弃那些得不到可行解或最优解的节点**，将其余节点保留在活节点表中。再从活节点表中取出一个活节点作为当前扩展节点。重复上述过程，直到找到所需的解或活节点表为空时为止。每一个活节点最多只有一次机会成为扩展节点。

活节点表的实现通常有两种形式：一种是**普通的队列**，即**先进先出队列**；另一种是**优先级队列**，按照某种优先级决定哪个节点为当前扩展节点。

根据活节点表的不同，分支限界法分为以下两种：**队列式分支限界法**和**优先队列式分支限界法。**

分支限界法的解题过程如下：

（1）定义解空间。解空间的大小对搜索效率有很大的影响，首先要定义合适的解空间，确定解空间包括解的组织形式和显约束。解的组织形式规范为一个 **n 元组**$\{x_1,x_2,\ldots,x_n\}$，具体问题表达的含义不同。**显约束**是对**解分量的取值范围**的限定。

（2）确定解空间的组织结构。对解空间的组织结构通常用解空间树形象地表达，根据解空间树的不同，解空间分为**子集树**、**排列树**、**m 叉树**等。

（3）搜索解空间。分支限界法指按照**广度优先搜索策略**，一次性生成所有孩子节点，**根据约束函数和限界函数判定对孩子节点是舍弃还是保留**，如果保留，则将其依次放入活节点表中，活节点表是普通队列或优先队列。然后从活节点表中取出一个节点，继续扩展，直到找到所需的解或活节点表为空时为止。如果对该问题只求**可行解**，则只需设定**约束函数即可**；如果求**最优解**，则需要设定**约束函数和限界函数**。

在优先队列分支限界法中还有**一个关键问题**，即**优先级的设定**：**选择什么值作为优先级？如何定义优先级？因为优先级的设计直接决定算法的效率。**

一、队列式广度优先搜索

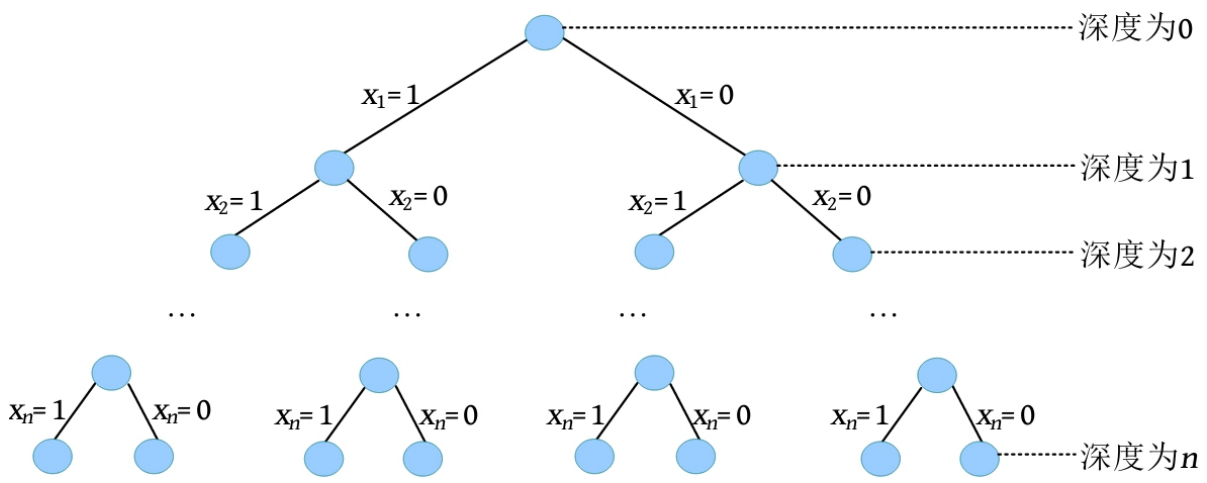
有 **n 个物品和 1 个背包**，每个物品 i 对应的价值都为 v_i、重量都为 w_i，背包的容量为 W（也可以将重量设定为体积）。每个物品只有一件，要么**装入**，要么**不装入**，**不可拆分**。如何选取物品装入背包，使背包所**装入物品的总价值最大**？

上述问题是典型的 **01 背包问题**，已经用回溯法求解过，在此先用普通队列式分支界限法求解，然后用优先队列式分支界限法求解，体会这两种算法的不同之处。

1. 算法设计

（1）定义问题的解空间。背包问题属于典型的 01 背包问题，问题的解是从 n 个物品中选择一些物品，使其在不超过容量的情况下价值最大。每个物品都有且只有两种状态：要么被装入背包，要么不被装入背包。那么是第 i 个物品被装入背包能够达到目标，还是不被装入能够达到目标呢？显然还不确定。因此，可以用变量 x_i 表示第 i 种物品是否被装入背包的状态，如果用“0”表示不被装入背包，用“1”表示被装入背包，则 x_i 的取值为 0 或 1。第 i 个物品被装入背包，$x_i=1$；不被装入背包，$x_i=0$。该问题解的形式是一个 n 元组，且每个分量的取值都为 0 或 1。由此可得，问题的**解空间为$\{x_1,x_2,\ldots,x_i,\ldots,x_n\}$，其中显约束 $x_i=0$ 或 1，i=1,2,3...n。**

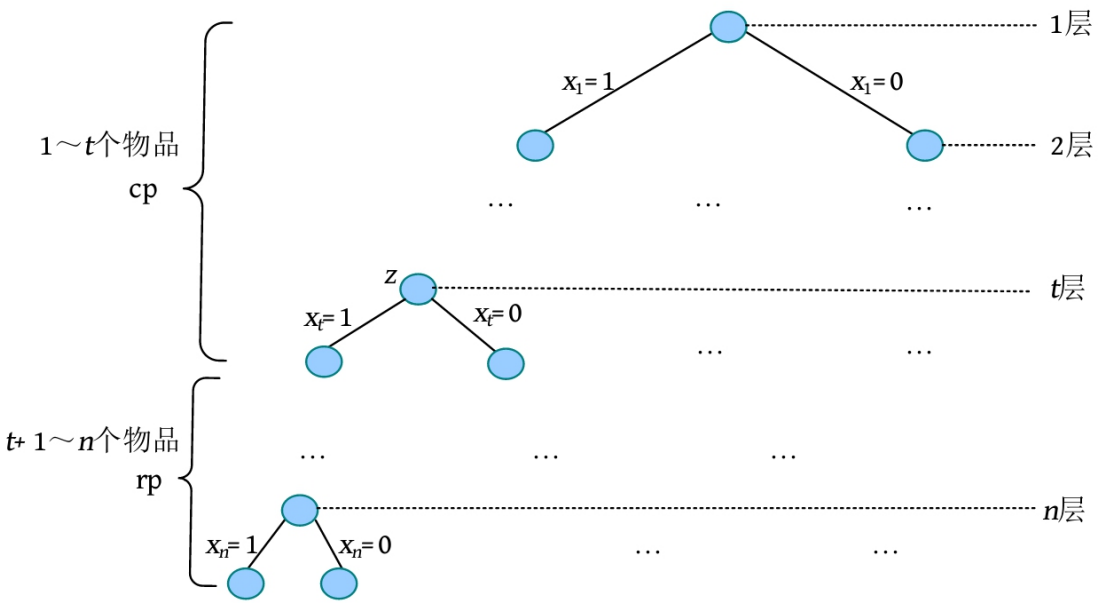
（2）确定解空间的组织结构。问题的解空间描述了 2^n 种可能的解，也可以说是 n 个元素组成的集合的所有子集个数。**解空间树为子集树**，解空间树的深度为问题的规模 n，如下图所示。



（3）搜索解空间。根据解空间的组织结构，对于任何一个中间节点 z（中间状态），从根节点到 z 节点的分支所代表的状态（是否装入背包）已确定，从 z 到其子孙节点的分支的状态待确定。也就是说，如果 z 在解空间树中所处的层次是 t，则说明从第 1 种物品到第 t-1 种物品的状态已确定，只需沿着 z 的分支扩展确定第 t 种物品的状态，前种物品的状态就确定了。在前 t 种物品的状态确定后，对当前已装入背包的物品的总重量用 cw 表示，对总价值用 cp 表示。

- **约束条件。**判断第 i 个物品被装入背包后总重量是否超出背包容量，如果超出，则为不可行解；否则为可行解。约束条件为 **cw+w[i]≤W**。其中 w[i]为第 i 个物品的重量，W 为背包容量。

- **限界条件。**已装入物品的价值高不一定就是最优的，因为还有剩余物品未确定。目前还不确定第 t+1 种物品到第 n 种物品的实际状态，因此只能**用估计值。**假设第 t+1 种物品到第 n 种物品都被装入背包，对**第 t+1 种物品到第 n 种物品的总价值用 rp 来表示**，因此 **cp+rp** 是所有从根出发经过中间**节点 z 的可行解的价值上界**，如下图所示。



如果价值上界小于当前搜索到的最优值（对最优值用 bestp 表示，初始值为 0），则说明从中间节点 z 继续向子孙节点搜索不可能得到一个比当前更优的可行解，没有继续搜索的必要；反之，继续向 z 的子孙节点搜索。

限界条件为 cp+rp≥bestp。

注意：回溯法中的背包问题，限界条件不带等号，因为 bestp 被初始化为 0，首次到达叶子时才会更新 bestp，因此只要有解，就必然存在至少一次到达叶子。而在**分支限界法中**，**只要 cp>bestp，就立即更新 bestp，如果在限界条件中不带等号，就会出现无法到达叶子的情况，比如解的最后一位是 0 时，例如(1,1,1,0)，就无法找到这个解向量。***因为在最后一位是 0 时，cp+rp=bestp，而不是 cp+rp>bestp，如果限界条件不带等号，就无法到达叶子，得不到解(1,1,1,0)。*该算法均设置了到叶子节点判断更新最优解和最优值。

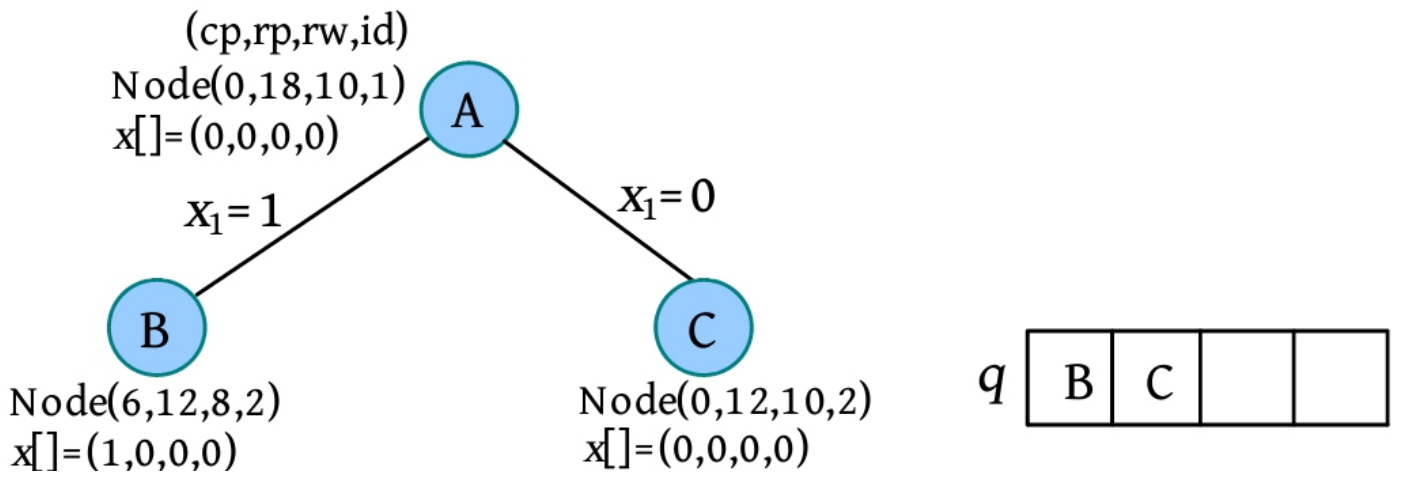
搜索过程：从根节点开始，以**广度优先的方式**进行搜索。根节点首先成为活节点，也是当前扩展节点。一次性生成所有孩子节点，由于在子集树中约定左分支上的值为“1”，因此沿着扩展节点的左分支扩展，则代表装入物品；由于在子集树中约定右分支上的值为“0”，因此沿着扩展节点的右分支扩展，则代表不装入物品。此时判断是否满足约束条件和限界条件，如果满足，则将其加入队列中；反之，舍弃。然后从队列中取出一个元素，作为当前扩展节点……直到搜索过程队列为空时为止。

2.图解

有**一个背包和 4 个物品**，每个物品的重量和价值都如下图所示，背包的**容量 W=10**。求在不超过背包容量的前提下，把哪些物品放入背包才能获得最大价值。

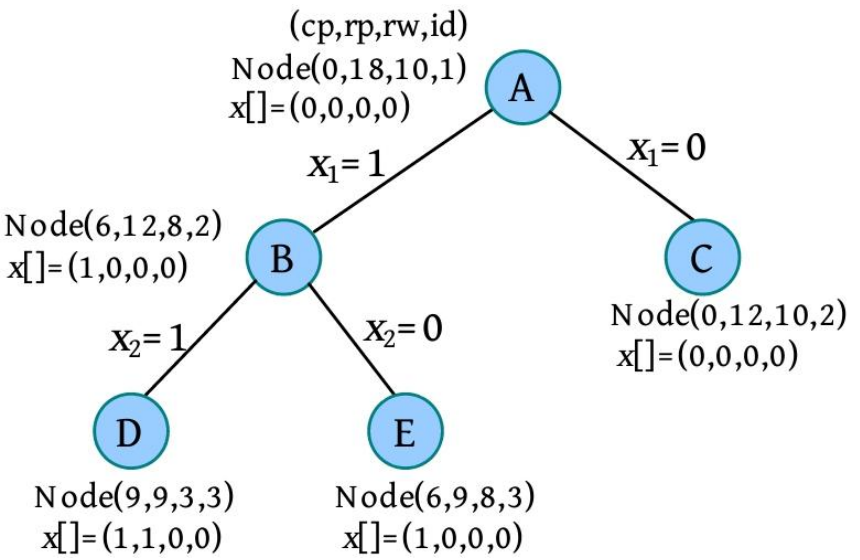
（1）初始化。sumw 和 sumv 分别用来统计所有物品的总重量和总价值。sumw=13，sumv=18，sumw>W，因此不能全部装完，需要搜索求解。初始化当前放入背包的物品价值 cp=0 ,当前剩余物品价值 rp=sumv ,当前剩余容量 rw=W ,当前处理物品序号为 1 且当前最优值 bestp=0。解向量 x[]=(0,0,0,0)，创建一个根节点 Node(cp,rp,rw,id)，将其标记为 A 并加入先进先出队列 q 中。cp 为装入背包的物品价值，rp 为剩余物品的总价值，rw 为剩余容量，id 为物品号，x[]为当前解向量，如下图所示。

（2）扩展节点 A。队头元素 A 出队，该节点的 cp+rp≥bestp，满足限界条件，可以扩展。rw=10>goods[1].weight=2，剩余容量大于 1 号物品的重量，满足约束条件，可以被放入背包，cp=0+6=6，rp=18−6=12，rw=10−2=8，t=2，x[1]=1，解向量更新为 x[]=(1,0,0,0)，生成左孩子 B 并将其加入 q 队列，更新 bestp=6。再扩展右分支，cp=0，rp=18−6=12，cp+rp≥bestp=6，满足限界条件，不放入 1 号物品，cp=0，rp=12，rw=10，t=2，x[1]=0，解向量为 x[]=(0,0,0,0)，创建新节点 C 并将其加入 q 队列中，如下图所示。

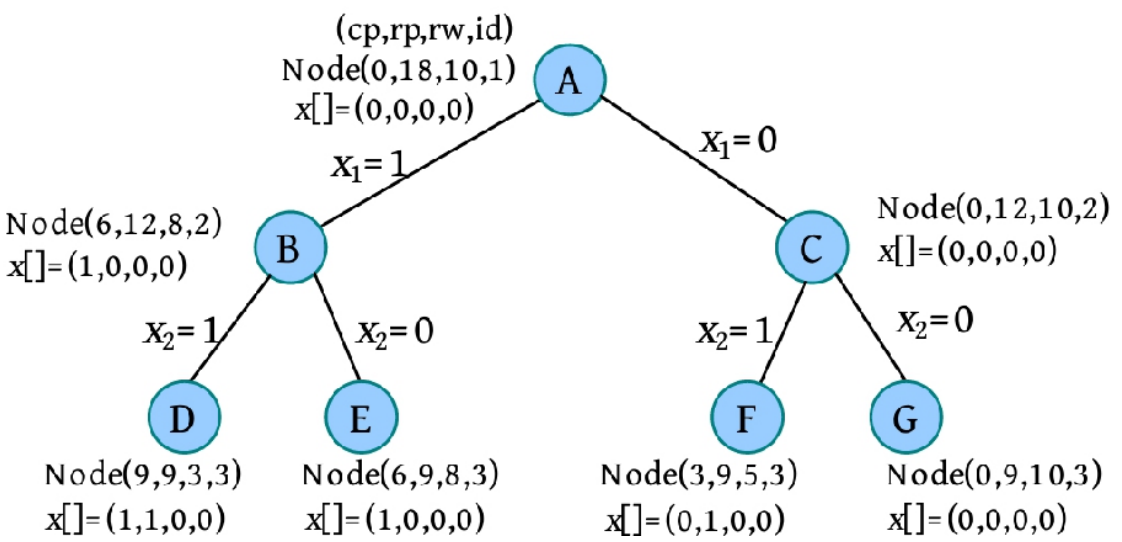


（3）扩展节点 B。队头元素 B 出队，该节点的 cp+rp≥bestp，满足限界条件，可以扩展。rw=8>goods[2]. weight=5，剩余容量大于 2 号物品的重量，满足约束条件，cp=6+3=9，rp=12−3=9，rw=8−5=3，t=3，x[2]=1，解向量更新为 x[]=1,1,0,0)，生成左孩子 D 并将其加入 q 队列中，更新 bestp=9。

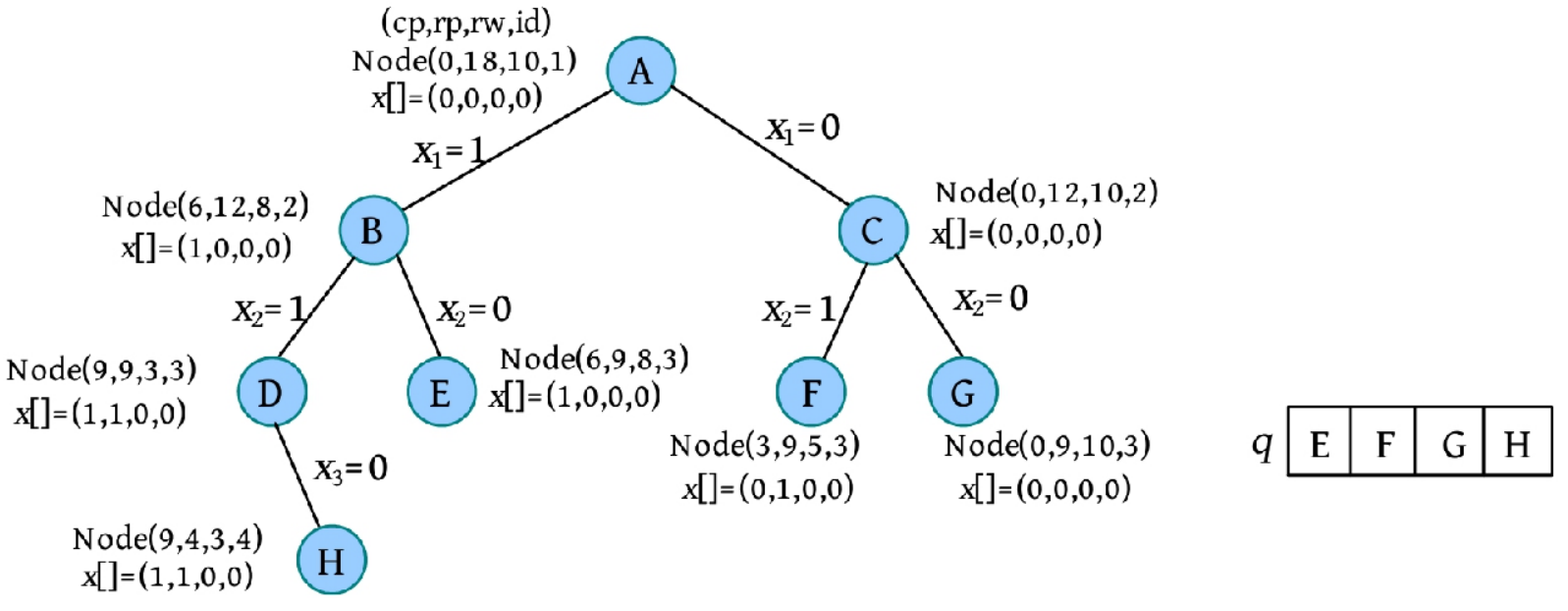
再扩展右分支，cp=6，rp=12−3=9，cp+rp≥bestp=9，满足限界条件，t=3，x[2]=0，解向量为 x[]=(1,0,0,0)，生成右孩子 E 并将其加入 q 队列中，如下图所示。



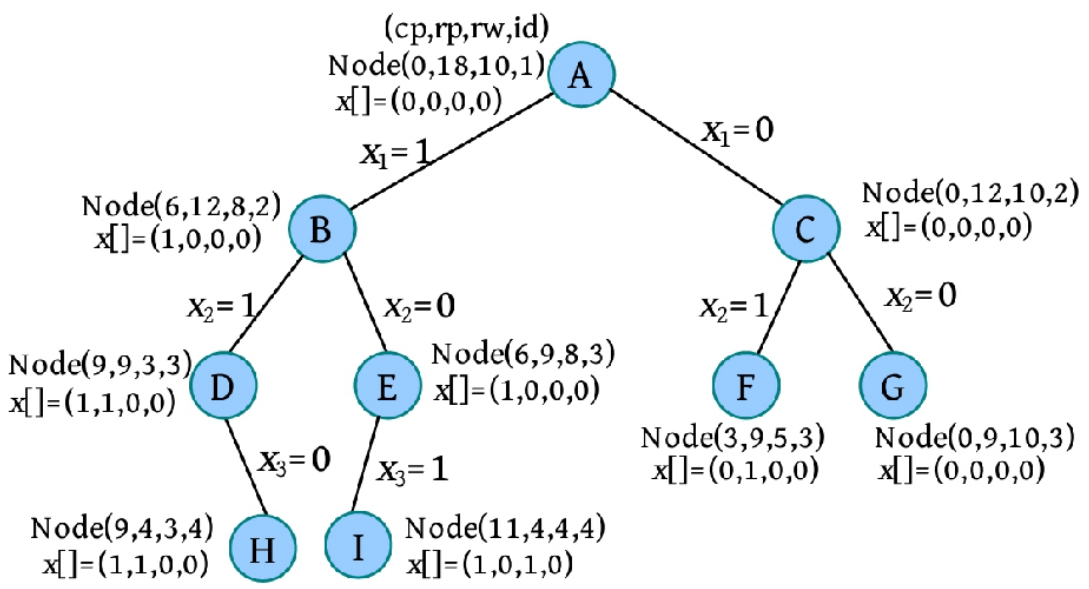
（4）扩展节点 C。队头元素 C 出队，该节点的 cp+rp≥bestp，满足限界条件，可以扩展。rw=10>goods[2].weight=5，剩余容量大于 2 号物品的重量，满足约束条件，cp=0+3=3，rp=12−3=9，rw=10−5=5，t=3，x[2]=1，解向量更新为 x[]=(0,1,0,0)，生成左孩子 F 并将其加入 q 队列中。再扩展右分支，cp=0，rp=12−3=9，cp+rp≥bestp=9，满足限界条件，rw=10，t=3，x[2]=0，解向量为 x[]=(0,0,0,0)，生成右孩子 G 并将其加入 q 队列中，如下图所示。



（5）扩展节点 D。队头元素 D 出队，该节点的 cp+rp≥bestp，满足限界条件，可以扩展。rw=3>goods[3]. weight=4，剩余容量小于 3 号物品的重量，不满足约束条件，舍弃左分支。再扩展右分支，cp=9，rp=9−5=4，cp+rp≥bestp=9，满足限界条件，t=4，x[3]=0，解向量为 x[]=(1,1,0,0)，生成右孩子 H 并将其加入 q 队列中，如下图所示。

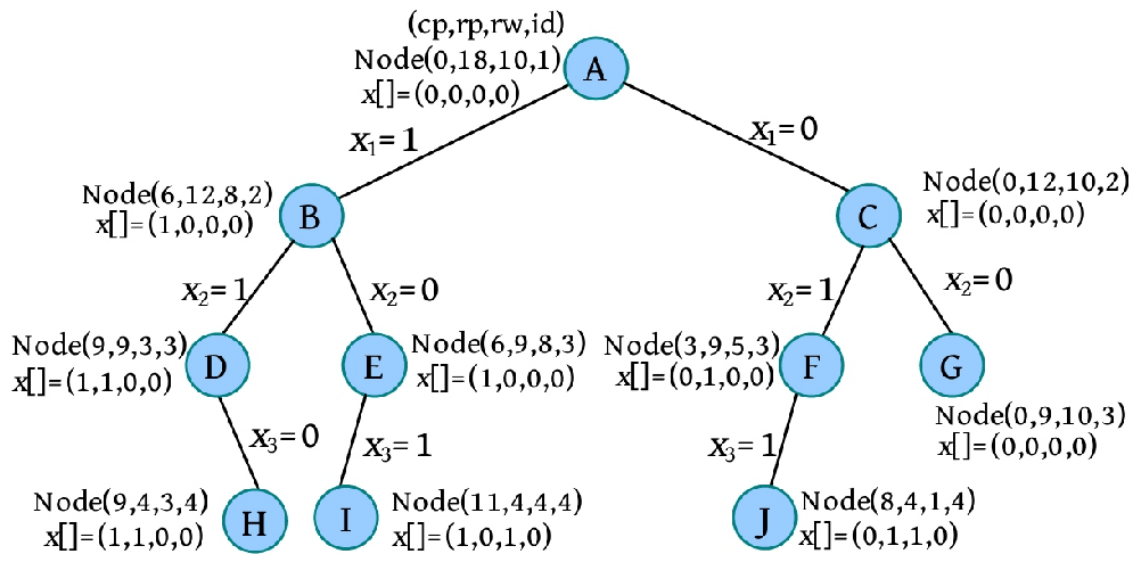


（6）扩展节点 E。队头元素 E 出队，该节点的 cp+rp≥bestp，满足限界条件，可以扩展。rw=8>goods[3].weight=4，剩余容量大于 3 号物品的重量，满足约束条件，cp=6+5=11，rp=9−5=4，rw=8−4=4，t=4，x[3]=1，解向量更新为 x[]=(1,0,1,0)，生成左孩子 I 并将其加入 q 队列中，更新 bestp=11。再扩展右分支，cp=6，rp=9−5=4，cp+rp<bestp=11，不满足限界条件，舍弃，如下图所示。



q | F | G | H | I

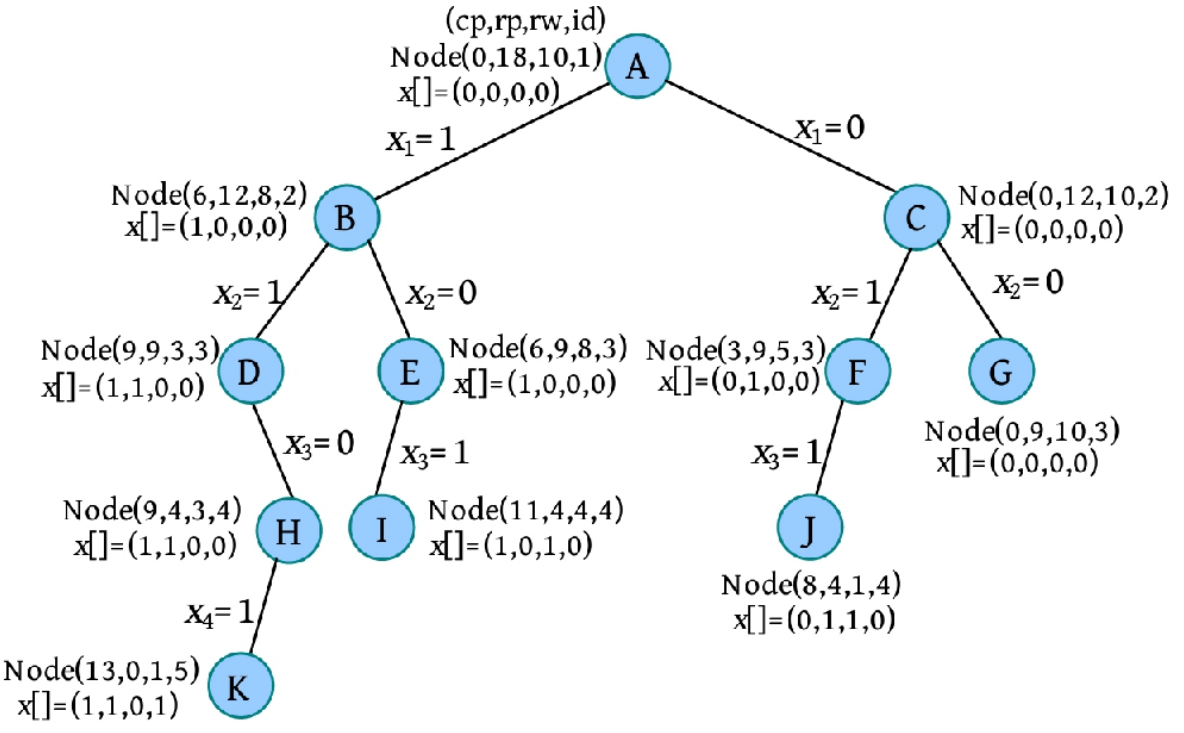
（7）扩展节点 F。队头元素 F 出队，该节点的 cp+rp≥bestp，满足限界条件，可以扩展。rw=5>goods[3].weight=4，剩余容量大于 3 号物品的重量，满足约束条件，cp=3+5=8，rp=9−5=4，rw=5−4=1，t=4，x[3]=1，解向量更新为 x[]=(0,1,1,0)，生成左孩子 J 并将其加入 q 队列中。再扩展右分支，cp=3，rp=9−5=4，cp+rp<bestp=11，不满足限界条件，舍弃，如下图所示。



q | G | H | I | J

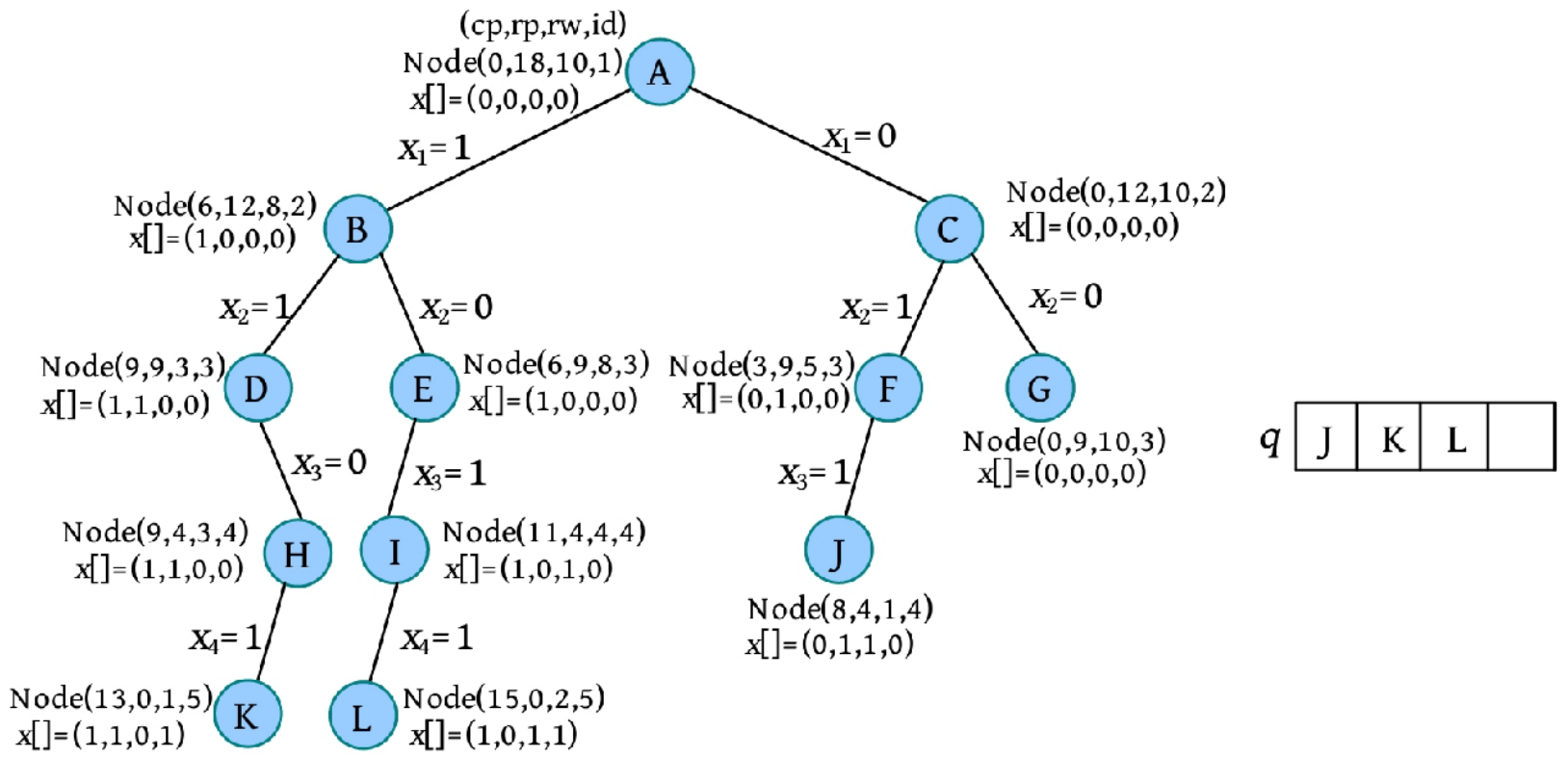
（8）扩展节点 G。队头元素 G 出队，该节点的 cp+rp<bestp=11，不满足限界条件，不扩展。

（9）扩展节点 H。队头元素 H 出队，该节点的 cp+rp≥bestp，满足限界条件，可以扩展。rw=3>goods[4].weight=2，剩余容量大于 4 号物品的重量，满足约束条件，令 cp=9+4=13，rp=4−4=0，rw=3−2=1，t=5，x[4]=1，解向量更新为 x[]=(1,1,0,1)，生成左孩子 K 并将其加入 q 队列中，更新 bestp=13。再扩展右分支，cp=9，rp=4−4=0，cp+rp<bestp，不满足限界条件，舍弃，如下图所示。



q | I | J | K |

（10）扩展节点 I。队头元素 I 出队，该节点的 cp+rp≥bestp，满足限界条件，可以扩展。rw=4>goods[4].weight=2，剩余容量大于 4 号物品的重量，满足约束条件，cp=11+4=15，rp=4-4=0，rw=4−2=2，t=5，x[4]=1，解向量更新为 x[]=(1,0,1,1)，生成左孩子 L 并将其加入 q 队列中，更新 bestp=15。再扩展右分支，cp=11，rp=4−4=0，cp+rp<bestp，不满足限界条件，舍弃，如下图所示。



（11）队头元素 J 出队，该节点的 cp+rp<bestp=15，不满足限界条件，不再扩展。

（12）队头元素 K 出队，扩展节点 K，t=5，已经处理完毕，cp<bestp，不是最优解。

（13）队头元素 L 出队，扩展节点 L，t=5，已经处理完毕，cp=bestp，是最优解，输出该解向量(1,0,1,1)。

（14）队列为空，算法结束。

3. 算法实现

（1）定义节点结构体。

```
struct Node{ //定义节点，记录当前节点的解信息
     int cp, rp;   //cp 背包的物品总价值，rp 剩余物品的总价值
     int rw;       //剩余容量
     int id;       //物品号
     bool x[N];    //解向量
     Node() { memset(x, 0, sizeof(x)); }//将解向量初始化为 0
     Node(int _cp, int _rp, int _rw, int _id){
          cp = _cp;
          rp = _rp;
          rw = _rw;
          id = _id;
     }
};
```

（2）定义物品结构体。在前面处理背包问题时，使用了两个一维数组 w[]、v[]分别存储物品的重量和价值，在此使用一个结构体数组来存储这些重量和价值。

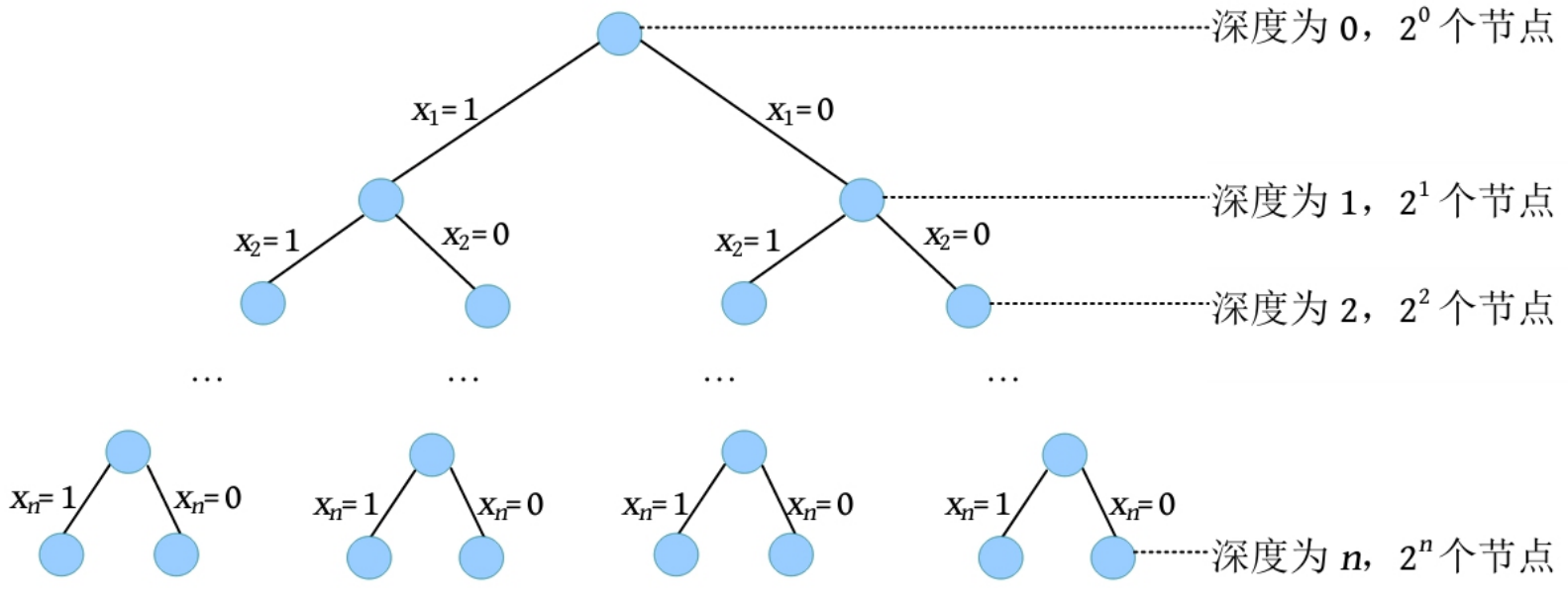
```
struct Goods{//物品
   int weight;//重量
   int value;//价值
}goods[N];
```

（3）搜索解空间。首先创建一个普通队列（先进先出），然后将根节点加入队列中，如果队列不空，则取出队头元素 livenode，得到当前处理的物品序号，如果当前处理的物品序号大于 n，则说明搜到最后一个物品了，不需要往下搜索。如果当前的背包没有剩余容量（已经装满）了，则不再扩展。如果当前放入背包的物品价值大于或等于最优值（livenode.cp≥bestp），则更新最优解和最优值。判断是否满足约束条件，满足则生成左孩子，判断是否更新最优值，左孩子入队，不满足约束条件则舍弃左孩子；判断是否满足限界条件，满足则生成右孩子，右孩子入队，不满足限界条件则舍弃右孩子。

```
int bfs(){//队列式分支限界法
   int t,tcp,trp,trw; //当前处理的物品序号 t、装入背包的物品价值 tcp、剩余容量 trw
   queue<Node> q; //创建一个普通队列（先进先出）
   q.push(Node(0,sumv,W,1)); //压入一个初始节点
   while(!q.empty()){
      Node livenode,lchild,rchild;//定义 3 个节点型变量
      livenode=q.front();//取出队头元素作为当前扩展节点 livenode
      q.pop(); //队头元素出队
      t=livenode.id;//当前处理的物品序号
      if(t>n||livenode.rw==0){
      if(livenode.cp>=bestp){//更新最优解和最优值
         for(int i=1;i<=n;i++)
            bestx[i]=livenode.x[i];
        bestp=livenode.cp;
      }
      continue;
   }
   if(livenode.cp+livenode.rp<bestp)//如果不满足，则不再扩展
      continue;
   tcp=livenode.cp; //当前背包中的价值
   trp=livenode.rp-goods[t].value; //不管当前物品装入与否，剩余价值都会减少
   trw=livenode.rw; //背包的剩余容量
   if(trw>=goods[t].weight){ //扩展左孩子，满足约束条件，可以放入背包
      lchild.rw=trw-goods[t].weight;
      lchild.cp=tcp+goods[t].value;
      lchild=Node(lchild.cp,trp,lchild.rw,t+1);//传递参数
      for(int i=1;i<t;i++)
         lchild.x[i]=livenode.x[i];//复制以前的解向量
      lchild.x[t]=true;
         if(lchild.cp>bestp)//比最优值大才更新
           bestp=lchild.cp;
         q.push(lchild);//左孩子入队
      }
      if(tcp+trp>=bestp){//扩展右孩子，满足限界条件，不放入背包
         rchild=Node(tcp,trp,trw,t+1);//传递参数
         for(int i=1;i<t;i++)
            rchild.x[i]=livenode.x[i];//复制以前的解向量
         rchild.x[t]=false;
         q.push(rchild);//右孩子入队
      }
   }
   return bestp;//返回最优值
}
```

4. 算法分析

时间复杂度：算法的运行时间取决于它在搜索过程中生成的节点数。而限界函数可以大大减少所生成的节点个数，避免无效搜索，加快搜索速度。左孩子需要判断约束函数，右孩子需要判断限界函数，那么在最坏情况下有多少个左孩子和右孩子呢？规模为 n 的子集树在最坏情况下的状态如下图所示。



总的节点个数为 $2^0+2^1+...+2^n=2^{n+1}-1$，减去树根节点再除以 2，就得到左右孩子的个数，左右孩子的个数=$(2^{n+1}-1-1)/2=2^n-1$。约束函数时间复杂度为 O(1)，限界函数时间复杂度为 O(1)。在最坏情况下有 $O(2^n)$个左孩子需要调用约束函数，有 $O(2^n)$个右孩子需要调用限界函数，所以计算背包问题的分支限界法的时间复杂度为 **$O(2^{n+1})$**。

空间复杂度：空间主要耗费在 Node 节点里面存储的变量和解向量上，因为最多有 $O(2^{n+1})$个节点，而每个节点的解向量都需要 O(n)个空间，所以空间复杂度为 **$O(n\times2^{n+1})$**。其实**让每个节点都记录解向量的办法是很笨的**，我们**可以用指针记录当前节点的左右孩子和父亲，到达叶子时逆向找其父亲，直到根节点，就得到了解向量，这样空间复杂度降为 O(n)。**

二、 优先队列式广度优先搜索

优先队列优化以当前节点的上界为优先值，把普通队列改成优先队列，这样就得到了优先队列式分支限界法。

1. 算法设计

优先级的定义为活节点代表的部分解所描述的已装入物品的价值上界，上界越大，优先级越高。活节点的价值上界 up=活节点的 cp+剩余物品装满背包剩余容量的最大价值 rp'。

约束条件：cw+w[i]≤W。

限界条件：up=cp+rp'≥bestp。

2. 图解

有一个背包和 4 个物品，每个物品的重量和价值如下图所示，背包的容量 W=10。求在不超过背包容量的前提下，把哪些物品放入背包才能获得最大价值。

goods[]	1	2	3	4
weight	2	5	4	2
value	6	3	5	4

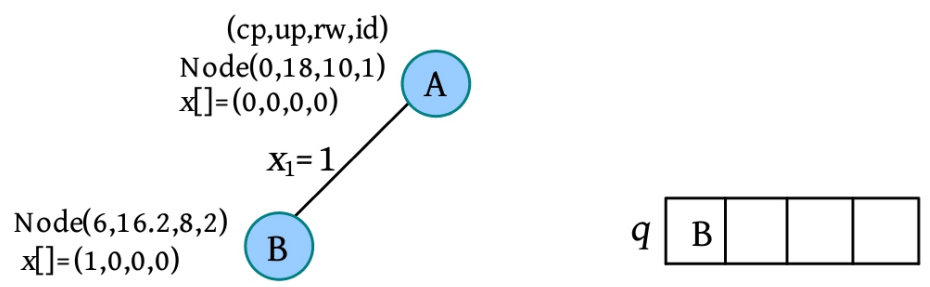
（1）初始化。sumw 和 sumv 分别用来统计所有物品的总重量和总价值。sumw=13，sumv=18，sumw>W，因此不能全部装完，需要搜索求解。

（2）按价值重量比非递增排序。排序后的结果如下图所示。为了程序处理方便，把排序后的数据存储在 w[]和 v[]数组中。后面的程序在该数组上操作即可，如下图所示。

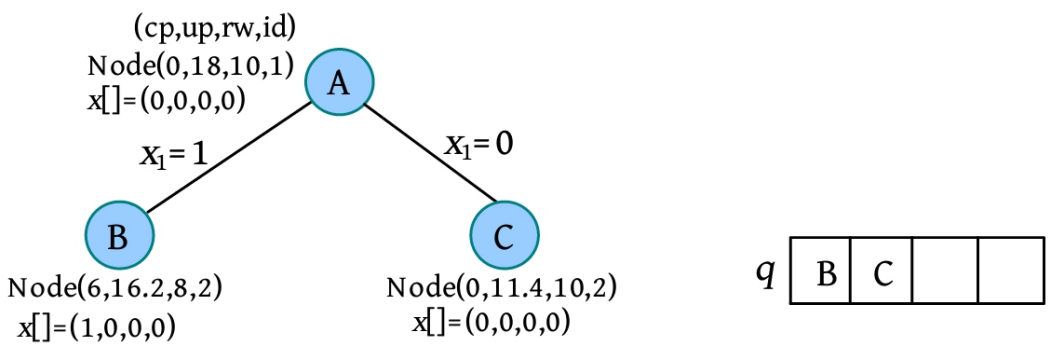
	1	2	3	4
w[]	2	2	4	5
v[]	6	4	5	3

（3）创建根节点 A。初始化当前放入背包的物品重量 cp=0，当前价值上界 up=sumv，当前剩余容量 rw=W，当前处理物品序号为 1，当前最优值 bestp=0。最优解初始化为 x[]=(0,0,0,0)，创建一个根节点 Node(cp,up,rw,id)，标记为 A，加入优先队列 q 中，如下图所示。

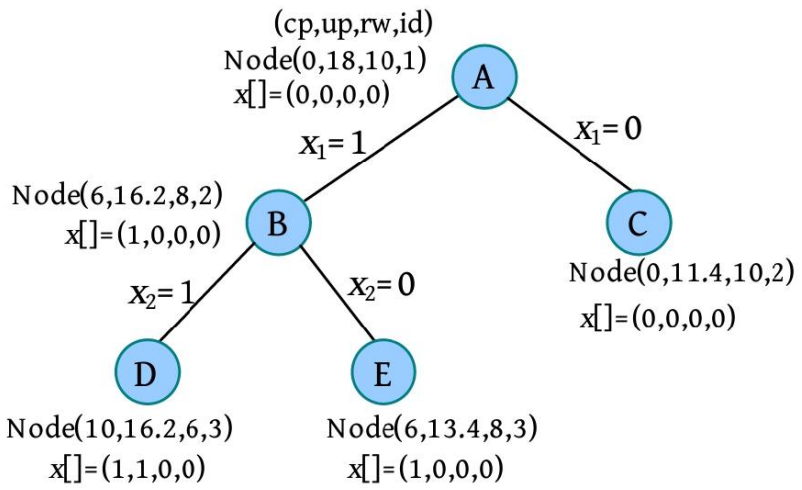
（4）扩展节点 A。队头元素 A 出队，该节点的 up≥bestp，满足限界条件，可以扩展。rw=10>w[1]=2，剩余容量大于 1 号物品的重量，满足约束条件，可以放入背包，生成左孩子，令 cp=0+6=6，rw=10-2=8。那么上界怎么算呢？up=cp+rp'=cp+剩余物品装满背包剩余容量的最大价值 rp'。剩余容量还有 8，可以装入 2、3 号物品，装入后还有剩余容量 2，只能装入 4 号物品的一部分，装入的价值为剩余容量×单位重量价值，即 2×3/5=1.2，rp'=4+5+1.2=10.2，up=cp+rp'= 16.2。在此需要注意，背包问题属于 01 背包问题，物品要么装入，要么不装入，是不可以分割的，这里为什么还会有部分装入的问题呢？很多读者看到这里都有这样的疑问，在此不是真的部分装入了，只是算上界而已。令 t=2，x[1]=1，解向量更新为 x[]=(1,0,0,0)，创建新节点 B 并将其加入 q 队列中，更新 bestp=6，如下图所示。

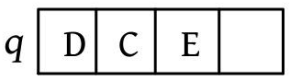


再扩展右分支，cp=0，rw=10，剩余容量可以装入 2、3 号物品，装入后还有剩余容量 4，只能装入 4 号物品的一部分，装入的价值为剩余容量　单位重量价值，即 4×3/5=2.4，rp'=4+5+2.4= 11.4，up=cp+rp'=11.4，up>bestp，满足限界条件，令 t=2，x[1]=0，解向量更新为 x[]=（0,0,0,0），生成右孩子 C 并将其加入 q 队列中，如下图所示。

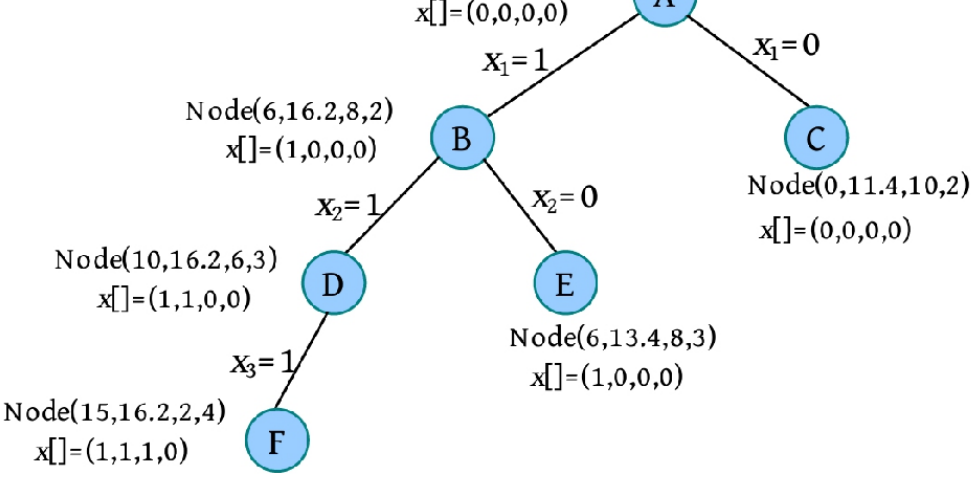


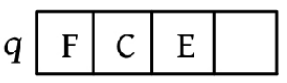
（5）扩展节点 B。队头元素 B 出队，该节点的 up≥bestp，满足限界条件，可以扩展。剩余容量 rw=8>w[2]=2，大于 2 号物品的重量，满足约束条件，令 cp=6+4=10，rw=8−2=6，up=cp+rp'=10+5+2×3/5=16.2，t=3，x[2]=1，解向量更新为 x[]=(1,1,0,0)，生成左孩子 D 并将其加入 q 队列中，更新 bestp=10。再扩展右分支，cp=6，rw=8，剩余容量可以装入 3 号物品，4 号物品部分装入，up=cp+rp'=6+5+3×4/5=13.4，up>bestp，满足限界条件，令 t=3，x[2]=0，解向量为 x[]=(1,0,0,0)，生成右孩子 E 并将其加入 q 队列中。注意：q 为优先队列，其实是用堆实现的，如果不想搞清楚，则只需知道每次 up 值最大的节点出队即可，如下图所示。



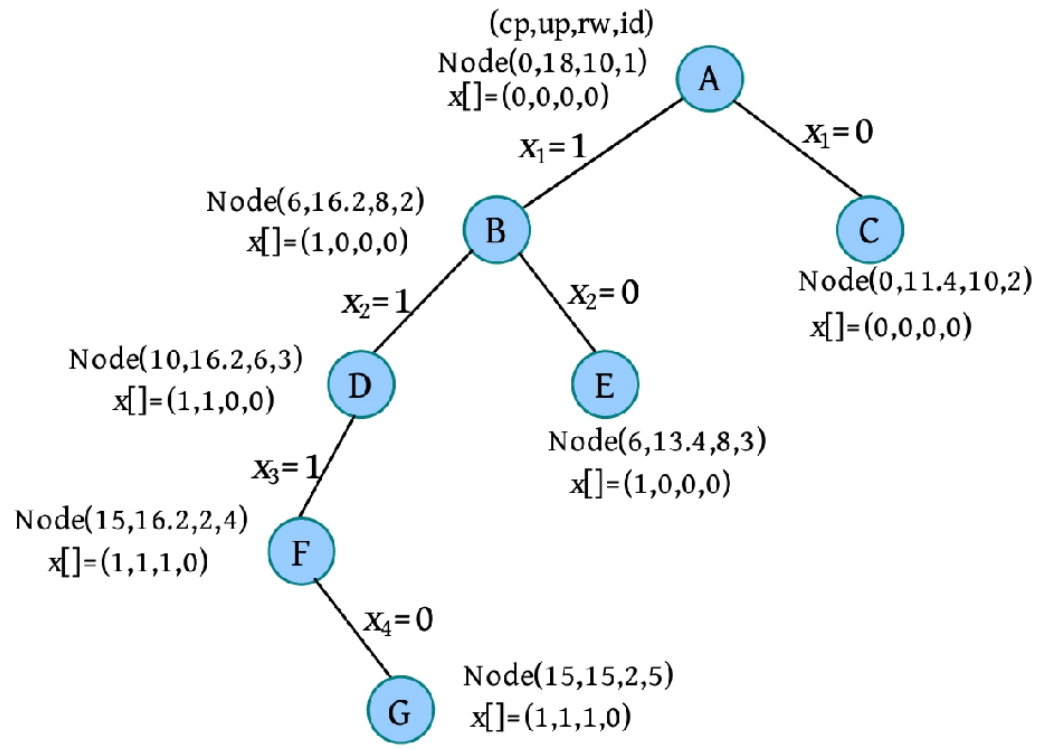


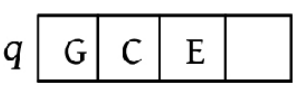
（6）扩展节点 D。队头元素 D 出队，该节点的 up≥bestp，满足限界条件，可以扩展。剩余容量 rw=6>w[3]=4，大于 3 号物品的重量，满足约束条件，令 cp=10+5=15，rw=6−4=2，up=cp+rp'=10+5+2×3/5=16.2，t=4，x[3]=1，解向量更新为 x[]=(1,1,1,0)，生成左孩子 F 并将其加入 q 队列中，更新 bestp=15。再扩展右分支，cp=10，rw=8，剩余容量可以装入 4 号物品，up=cp+rp'=10+3=13，up<bestp，不满足限界条件，舍弃右孩子，如下图所示。





（7）扩展节点 F。队头元素 F 出队，该节点的 up≥bestp，满足限界条件，可以扩展。剩余容量 rw=2<w[4]=5，不满足约束条件，舍弃左孩子。再扩展右分支，cp=15，rw=2，虽然有剩余容量，但物品已经处理完毕，已没有物品可以装入，up=cp+rp'=15+0=15，up≥bestp，满足限界条件，令 t=5，x[4]=0，解向量为 x[]=(1,1,1,0)，生成右孩子 G 并将其加入 q 队列中，如下图所示。





（8）扩展节点 G。队头元素 G 出队，该节点的 up≥bestp，满足限界条件，可以扩展。t=5，已经处理完毕，bestp=cp=15，是最优解，解向量为 x[]=(1,1,1,0)。注意：虽然解是(1,1,1,0)，但对应的物品原来的序号是 1、4、3。节点 G 出队。

（9）队头元素 E 出队，该节点的 up<bestp，不满足限界条件，不再扩展。

（10）队头元素 C 出队，该节点的 up<bestp，不满足限界条件，不再扩展。

（11）队列为空，算法结束。

3. 算法实现

（1）定义节点和物品结构体。

```
struct Node{//定义节点，记录当前节点的解信息
   int cp; //已装入背包的物品价值
   double up; //价值上界
   int rw; //背包剩余容量
   int id; //物品号
   bool x[N];
   Node() {}
   Node(int _cp,double _up,int _rw,int _id){
      cp=_cp;
      up=_up;
      rw= rw;
      id=_id;
      memset(x, 0, sizeof(x));
    }
};
struct Goods{ //物品结构体
   int weight;//重量
   int value;//价值
}goods[N];
```

（2）定义辅助结构体和排序优先级（从大到小排序）。

```
struct Object{//辅助物品结构体，用于按单位重量价值（价值/重量比）排序
   int id; //序号
   double d;//单位重量价值
}S[N];

bool cmp(Object a1,Object a2){//排序优先级，按照物品的单位重量价值由大到小
   return a1.d>a2.d;
}
```

（3）定义队列的优先级。

```
bool operator <(const Node &a, const Node &b){//队列优先级，up 越大越优先
   return a.up<b.up;
}
```

（4）计算节点的上界。

```
double Bound(Node tnode){
   double maxvalue=tnode.cp;//已装入背包的物品价值
   int t=tnode.id;//排序后序号
   double left=tnode.rw;//剩余容量
   while(t<=n&&w[t]<=left){
      maxvalue+=v[t];
      left-=w[t++];
   }
   if(t<=n)
      maxvalue+=double(v[t])/w[t]*left;
   return maxvalue;
}
```

（5）优先队列分支限界法。

```
int priorbfs(){
   int t,tcp,trw;//当前处理的物品序号 t、当前装入背包的物品价值 tcp、当前剩余容量 trw
   double tup; //当前价值上界 tup
   priority_queue<Node> q; //创建一个优先队列
   q.push(Node(0,sumv,W,1));//初始化，将根节点加入优先队列中
   while(!q.empty()){
      Node livenode, lchild, rchild;//定义三个节点型变量
      livenode=q.top();//取出队头元素作为当前扩展节点 livenode
      q.pop(); //队头元素出队
      t=livenode.id;//当前处理的物品序号
      if(t>n||livenode.rw==0){
         if(livenode.cp>=bestp){//更新最优解和最优值
            for(int i=1;i<=n;i++)
               bestx[i]=livenode.x[i];
            bestp=livenode.cp;
         }
         continue;
      }
      if(livenode.up<bestp)//如果不满足，则不再扩展
         continue;
      tcp=livenode.cp; //当前背包中的价值
      trw=livenode.rw; //背包的剩余容量
      if(trw>=w[t]){ //扩展左孩子，满足约束条件，可以放入背包
         lchild.cp=tcp+v[t];
         lchild.rw=trw-w[t];
         lchild.id=t+1;
         tup=Bound(lchild); //计算左孩子的上界
         lchild=Node(lchild.cp,tup,lchild.rw,lchild.id);
         for(int i=1;i<=n;i++)//复制以前的解向量
            lchild.x[i]=livenode.x[i];
         lchild.x[t]=true;
         if(lchild.cp>bestp)//比最优值大才更新
            bestp=lchild.cp;
         q.push(lchild);//左孩子入队
      }
      rchild.cp=tcp;
      rchild.rw=trw;
      rchild.id=t+1;
      tup=Bound(rchild);//计算右孩子的上界
      if(tup>=bestp){ //扩展右孩子，满足限界条件，不放入
         rchild=Node(tcp,tup,trw,t+1);
         for(int i=1;i<=n;i++)//复制以前的解向量
            rchild.x[i]=livenode.x[i];
         rchild.x[t]=false;
         q.push(rchild);//右孩子入队
       }
}
return bestp;//返回最优值
```

4. 算法分析

虽然在算法复杂度数量级上，优先队列的分支限界法算法和普通队列的算法相同，但从图解可以看出，采用优先队列式的分支限界法算法生成的节点数更少，找到最优解的速度更快。