有依赖背包问题

简化的问题

这种背包问题的物品间存在某种“依赖”的关系。也就是说，i依赖于j，表示若选物品i，则必须选物品j。为了简化起见，我们先设没有某个物品既依赖于别的物品，又被别的物品所依赖；另外，没有某件物品同时依赖多件物品。

算法

这个问题由 NOIP2006 金明的预算方案一题扩展而来。遵从该题的提法，将不依赖于别的物品的物品称为“主件”，依赖于某主件的物品称为“附件”。由这个问题的简化条件可知所有的物品由若干主件和依赖于每个主件的一个附件集合组成。

按照背包问题的一般思路，仅考虑一个主件和它的附件集合。可是，可用的策略非常多，包括：一个也不选，仅选择主件，选择主件后再选择一个附件，选择主件后再选择两个附件···无法用状态转移方程来表示如此多的策略。（事实上，设有 n 个附件，则策略有2^{n+1}个，为指数级。）

考虑到所有这些策略都是互斥的（也就是说，你只能选择一种策略），所以一个主件和它的附件集合实际上对应于分组背包中的一个物品组，每个选择了主件又选择了若干个附件的策略对应于这个物品组中的一个物品，其费用和价值都是这个策略中的物品的值的和。但仅仅是这一步转化并不能给出一个好的算法，因为物品组中的物品还是像原问题的策略一样多。 再考虑分组背包中的一句话： 可以对每组中的物品应用完全背包中“一个简单有效的优化”。 这提示我们，对于一个物品组中的物品，所有费用相同的物品只留一个价值最大的，不影响结果。 所以，我们可以对主件i的“附件集合”先进行一次01背包，得到费用依次为$0\cdots V-c[i]$所有这些值时相应的最大价值$f'[0\cdots V-c[i]]$。那么这个主件及它的附件集合相当于$V-c[i]+1$个物品的物品组，其中费用为$c[i]+k$的物品的价值为$f'[k]+w[i]$。也就是说原来指数级的策略中有很多策略都是冗余的，通过一次01背包后，将主件i转化为$V-c[i]+1$个物品的物品组，就可以直接应用分组背包的算法解决问题了。

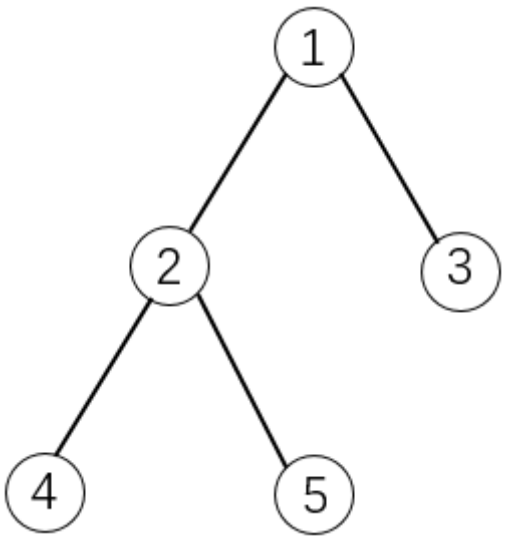
较一般的问题 更一般的问题是：依赖关系以图论中“森林”的形式给出（森林即多叉树的集合），也就是说，主件的附件仍然可以具有自己的附件集合，限制只是每个物品最多只依赖于一个物品（只有一个主件）且不出现循环依赖。 解决这个问题仍然可以用将每个主件及其附件集合转化为物品组的方式。唯一不同的是，由于附件可能还有附件，就不能将每个附件都看作一个一般的01背包中的物品了。若这个附件也有附件集合，则它必定要被先转化为物品组，然后用分组的背包问题解出主件及其附件集合所对应的附件组中各个费用的附件所对应的价值。 事实上，这是一种树形DP，其特点是每个父节点都需要对它的各个儿子的属性进行一次DP以求得自己的相关属性。这已经触及到了“泛化物品”的思想。看完泛化物品后，你会发现这个“依赖关系树”每一个子树都等价于一件泛化物品，求某节点为根的子树对应的泛化物品相当于求其所有儿子的对应的泛化物品之和。

题目描述

有N个物品和一个容量是V的背包。

物品之间具有依赖关系，且依赖关系组成一棵树的形状。如果选择一个物品，则必须选择它的父节点。

如下图所示：



如果选择物品5，则必须选择物品1和2。这是因为2是5的父节点，1是2的父节点。

每件物品的编号是i，体积是v_i，价值是w_i，依赖的父节点编号是p_i。物品的下标范围是$1 \cdots N$。

求解将哪些物品装入背包，可使物品总体积不超过背包容量，且总价值最大。

输出最大价值。

输入格式

第一行有两个整数N，V，用空格隔开，分别表示物品个数和背包容量。

接下来有N行数据，每行数据表示一个物品。

第i行有三个整数v_i, w_i, p_i，用空格隔开，分别表示物品的体积、价值和依赖的物品编号。

如果$pi = -1$，表示根节点。 数据保证所有物品构成一棵树。

输出格式

输出一个整数，表示最大价值。

数据范围

$1 \le N, V \le 100$

$1 \le v_i, w_i \le 100$

父节点编号范围：

1. 内部结点：$1 \le pi \le N$;
2. 根节点$pi = -1$;

输入样例

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

输出样例：

```
11
```

In [1]:
```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
#define Max(a, b) ((a > b) ? (a) : (b))
extern "C" {
    const int N = 110;

    int n, m;
    int h[N];  //head的缩写，以i为起点第一条边储存的位置
    int e[N], ne[N], idx=0;  //ne父节点下标，
    int v[N], w[N];   //i物品对应的价值，花费
    int f[N][N];    //答案

    //边x-y存到结构中
    void add(int x, int y) {  //参数：父节点下标，当前节点下标，*链式前向星,编号为y的依赖点为x
        e[idx] = y;         //这条边的终点是y
        ne[idx] = h[x];     //h[x]中存的是一x为起点的所有边中第一条边存的地方，本句代码是将新的边插到最前面
        h[x] = idx;       //将新的第一条边赋值给h[x],然后idx加一
        //printf("h[%d]=%d\n",x,idx);
        idx+=1;
    }

    void dfs(int x) {
        for (int i = h[x]; i != -1; i = ne[i]) {
            int y = e[i];  //孩子节点
            dfs(y);         //dfs孩子节点
            for (int j = m - v[x]; j >= 0; j--) {   //01背包，或者包含孩子节点，
                for (int k = 0; k <= j; k++) {   //注意，这个是从前向后的，是不超过背包容积的最大价值
                    // printf("dfs-->%d  %d\n",x,y);
                    //前面dfs(y)返回之后，f[y][]就有对应的值了。
                    f[x][j] = Max(f[x][j], f[x][j - k] + f[y][k]);  //这个必须是二维的，因为迭代的顺序是按照树的dfs顺序。不是数组的下标顺序
                }
            }
        }
        //到头了，没有孩子了，下面的两个for循环是一个整体，完成一个m->0的循环，填上对应的值
        for (int i = m; i >= v[x]; i--) {  //对每个叶节点做01背包
            f[x][i] = f[x][i - v[x]] + w[x];
        }
        for (int i = 0; i < v[x]; i++) {
            f[x][i] = 0;
        }
    }
    void ylpack(){
        freopen("dp04beibao07_01.in","r",stdin);
        memset(h, -1, sizeof h);  //head,初始化为-1
        scanf("%d %d",&n,&m); //物品个数,背包容量
        int root;  //用来记录根节点
        for (int i = 1; i <= n; i++) {
            int p;
            scanf("%d %d %d",&v[i],&w[i],&p);  //体积,价值,依赖点
            if (p == -1) {    //根节点
                root = i;
            } else {
                add(p, i);    //如果不是根节点,用链式前向星存储
            }
        }
        ///*
        for (int i = 0; i <= n; i++) {
            printf("%3d ",i);
        }printf("\t<------i\n");
        for (int i = 0; i <= n; i++) {
            printf("%3d ",h[i]);
        }printf("\t<------h[i],没有以3,4,5为起点的边,这是一棵树,以i为起点的边集中第一条边的存储位置\n");
        for (int i = 0; i <= n; i++) {
            printf("%3d ",e[i]);
        }printf("\t<------to[i]\n");
        for (int i = 0; i <= n; i++) {
            printf("%3d ",ne[i]);
        }printf("\t<------next[i]\n");
        //*/
        dfs(root);
        printf("%d",f[root][m]);
    }
}
#endif
```

Out[1]:

In [2]:
```
ylpack();
```

```
  0   1   2   3   4   5         <------i
 -1   1   3  -1  -1  -1         <------h[i],没有以3,4,5为起点的边,这是一棵树,以i为起点的边集中第一条边的存储位置
  2   3   4   5   0   0         <------to[i]
 -1   0  -1   2   0   0         <------next[i]
11
```

Out[2]: (void) nullptr

h[1]=1,to[1]=3-->1/3,next[1]=0;to[0]=2-->1/2,next[0]=-1

h[2]=3,to[3]=5-->2/5,next[3]=2;to[2]=4-->2/4,next[2]=-1

小结

拿金明的预算方案来说，通过引入“物品组”和“依赖”的概念可以加深对这题的理解，还可以解决它的推广问题。用物品组的思想考虑那题中极其特殊的依赖关系：物品不能既作主件又作附件，每个主件最多有两个附件，可以发现一个主件和它的两个附件等价于一个由四个物品组成的物品组，这便揭示了问题的某种本质。

例题：

HDU 3449 Consumer (http://acm.hdu.edu.cn/showproblem.php?pid=3449)
Luogu 1064 金明的预算方案 (https://www.luogu.org/problemnew/show/P1064)

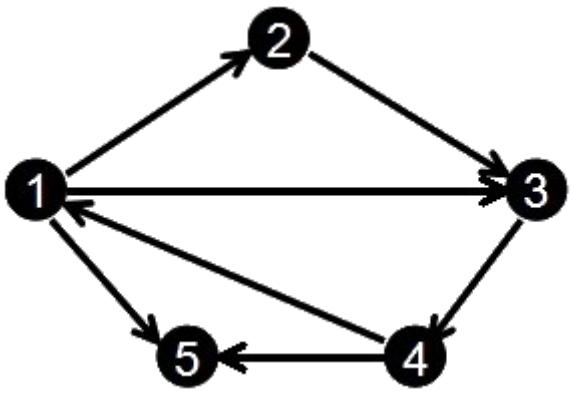
链式前向星

前向星是一种特殊的边集数组，把边集数组中的每一条边按照起点从小到大排序，如果起点相同就按照终点从小到大排序，并记录下以某个点为起点的所有边在数组中的起始位置和存储长度，那么前向星就构造好了。

用 len[i] 来记录所有以 i 为起点的边在数组中的存储长度。

用 head[i] 记录以 i 为边集在数组中的第一个存储位置。

那么对于下图：



输入边的顺序为：

```
1 2
2 3
3 4
1 3
4 1
1 5
4 5
```

那么排完序后就得到：

```
编号:   1   2   3   4   5   6   7
起点u:  1   1   1   2   3   4   4
终点v:  2   3   5   3   4   1   5
```

得到：

```
head[1] = 1    len[1] = 3
head[2] = 4    len[2] = 1
head[3] = 5    len[3] = 1
head[4] = 6    len[4] = 2
```

但是利用前向星会有排序操作，如果用快排时间至少为$O(n\log(n))$。还记得上课时说的那个基数排序？

如果用链式前向星，就可以避免排序。

建立边结构体为:

```
struct Edge{
    int next;
    int to;
    int w;
};
```

其中 edge[i].to 表示第 i 条边的终点, edge[i].next 表示与第 i 条边同起点的下一条边的存储位置, edge[i].w 为边权值.

另外还有一个数组 head[] ,它是用来表示以 i 为起点的第一条边存储的位置,实际上这里的第一条边存储的位置其实在以 i 为起点的所有边的最后输入的那个编号.

head[] 数组一般初始化为 -1 ,对于加边的 add 函数是这样的:

```
void add(int u,int v,int w){   //添加边，u是起点，v是终点，w是权值
    edge[cnt].w = w;           //权值
    edge[cnt].to = v;          //终点
    edge[cnt].next = head[u];  //第u条边同起点的下一条边的位置，其实这里是将新的边插入到前一条边的前面
    head[u] = cnt++;           //cnt作为新的一条同起点的边的第一条边，赋值给head[u]
}
```

初始化 cnt = 0 ,这样,现在我们还是按照上面的图和输入来模拟一下:

```
edge[0].to = 2;     edge[0].next = -1;      head[1] = 0;
edge[1].to = 3;     edge[1].next = -1;      head[2] = 1;
edge[2].to = 4;     edge[2],next = -1;      head[3] = 2;
edge[3].to = 3;     edge[3].next = 0;       head[1] = 3;
edge[4].to = 1;     edge[4].next = -1;      head[4] = 4;
edge[5].to = 5;     edge[5].next = 3;       head[1] = 5;
edge[6].to = 5;     edge[6].next = 4;       head[4] = 6;
```

很明显, head[i] 保存的是以 i 为起点的所有边中编号最大的那个，而把这个当作顶点 i 的第一条起始边的位置。

这样在遍历时是倒着遍历的，也就是说与输入顺序是相反的，不过这样不影响结果的正确性。

比如以上图为例，以节点 1 为起点的边有 3 条，它们的编号分别是 0,3,5 ，而 head[1] = 5

在遍历以 u 节点为起始位置的所有边的时候是这样的:

```
for(int i=head[u];~i;i=edge[i].next)
```

那么就是说先遍历编号为 5 的边，也就是 head[1] ，然后就是 edge[5].next ，也就是编号 3 的边，然后继续 edge[3].next ，也就是编号 0 的边，可以看出是逆序的。

In [ ]: