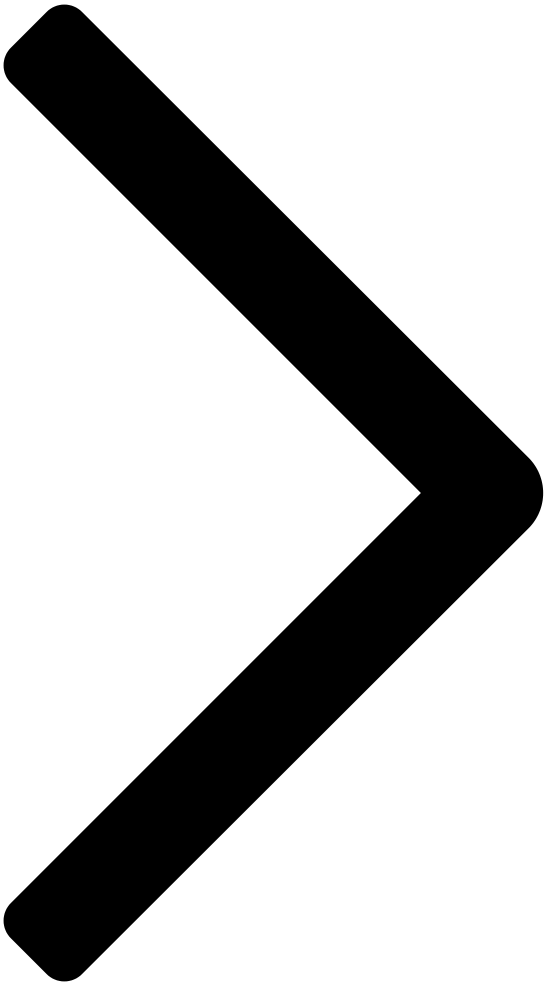

非整数0-1背包问题

sean
2021-02-20 2,032 阅读2分钟

0-1背包问题通常情况下物品的重量是整数的，采用动态规划可以解决，在解决物品重量非整数情况下的背包问题之前，我们先来回顾整数背包问题，并从中寻找解决非整数背包问题的方法。

问题定义：有n种物品和一个容量为cc的背包，第ii件物品的重量为$w_i$wi，价格为$v_i$vi,求出哪种物品组合放入背包使物品价值总和最大。

整数0-1背包问题

设$p(i,j)$p(i,j)表示在容量为j情况下，将物品i，$i+1$ ... ni，i + 1 ... n组合的放入背包的最优解的值

则其转移方程为

如果$j >= w_i$j >= wi ，$p(i,j) = max(p(i+1,j), p(i+1,j-w_i)+v_i)$
p(i,j) = max(p(i + 1,j), p(i + 1,j − wi) + vi)

如果$j < w_i$j < wi , $p(i,j) = p(i+1,j)$p(i,j) = p(i + 1,j)

可以理解为当$j < w_i$j < wi ,背包无法放下第ii件物品，所以其最优解和考虑放$i+1$i + 1到nn的物品到容量为j的背包的最优解相同。当$j >= w_i$j >= wi时，可以选择放和不放第ii件物品，当不放时背包价值和$p(i+1,j)$p(i + 1,j)相同，当放时背包价值为$p(i+1,j-w_i)$p(i + 1,j − wi)再加上第ii件物品的价值。

所以整数背包问题可以采用动态规划解决，开辟$n*c$n ∗ c的数组，从下往上不断更新数组的值，得到$p(i,j)$p(i,j)的值，最优解就是$p(0,c)$p(0,c)的值

```
public static int solution(int c, int[] v, int[] w){
    if(v.length != w.length){
        throw new IllegalArgumentException();
    }
    int n = v.length;int m = c+1;
    int[][] result = new int[n][m];
    for(int i = 0;i < m;i++){
        result[n-1][i] = i>=w[n-1]? v[n-1]:0;
    }
    for(int i = n-2;i>=0;i--){
        for(int j = 0;j < m;j++){
            if(j < w[i]){
                result[i][j] = result[i+1][j];
            }else {
                result[i][j] = Math.max(result[i+1][j], result[i+1][j-w[i]]+v[i]);
            }
        }
```

```
        }
        traceBack(c, v, w, result);
        return result[0][c];
    }
```

路径回溯，找到最优组合

```
    private static void traceBack(int c, int[]v, int[] w, int[][] p){
        int k = c;
        List<Integer> trace = new ArrayList<>();
        int i;
        for(i = 0;i < p.length-1;i++){
            if(p[i][k] == p[i+1][k-w[i]]+v[i]){
                k = k - w[i];
                trace.add(i+1);
            }
        }
        if(p[i][k] == v[i]){
            trace.add(i+1);
        }
        System.out.println(trace);
    }
```

整数0-1背包问题的改进

例如背包的容量为10，5件物品的重量分别为2，2，6，5，4，对应价值分别是6，3，5，4，6，则$p(i,j)$p(i,j)数组的更新情况如下

weight	value	1	2	3	4
2	6	0	6	6	9
2	3	0	3	3	6
6	5	0	0	0	6
5	4	0	0	0	6
4	6	0	0	0	6

当背包的容量很大（即c的值特别大），则这个数组将会异常的庞大，并且数组的每一个数都需要更新，算法需要的计算时间较多。

所以可以进行适当的改进，从上面的表格来看，我们无需记录数组种的每一个数，只需要记录下每行的跳跃点即可，比如最后一行从第4列开始跳跃，我们只需记录{(0,0),(4,6)}即可,倒数第二行只需记录{(0,0),(4,6),(9,10)}即可。接下来的问题是如何更新每行的跳跃点。

p[5] = {(0,0),(4,6)}

将p[5]的跳跃点加上下一个需要放入的物品的重量和价值，则可以得到q[5] = p[5]+(5,4) = {(5,4),(9,10)},

将p[5]和q[5]合并得到{(0,0),(4,6),(5,4),(9,10)}，之后去除重量大却价值小的点，如(5,4)点，放入背包的物品总重量大于4，价值却只有4小于6，所以通过比较(5,4)和(4,6)需要去掉(5,4)得到p[4] = {(0,0),(4,6),(9,10)}

非整数0-1背包问题

非整数0-1背包问题可以转化为整数0-1背包问题，如果非整数可以用保留三位小数来表示的话，那么可以将非整数背包问题的所有值乘上1000,全部转为整数，采用整数背包问题解决，但是这样必须牺牲一定的精度，而且会增加开辟数组的大小（乘上1000，数组的列肯定超过1000以上），这对计算时间也有很大影响，因为需要更新每一个数组中的元素。

有效的解决方法和上述对于整数0-1背包问题的改进是一样的，而且发现上述的跳跃点并不要求是整数，对于实数一样适用，因此可以采用更新跳跃点的动态规划的方式解决非整数0-1背包问题。

```
public static double solution(double c, double[] v, double[] w){
    if(v.length != w.length){
        throw new IllegalArgumentException();
    }
    double[][] p = new double[10000][2];
    int n = v.length;
    p[0][0] = 0;p[0][1] = 0;
    int left = 0, right = 0,next = 1;
    int[] head = new int[n+2];
    head[n+1] = 0;
    head[n] = 1;
    for(int i = n-1; i >= 0;i--){
        int k = left;
        for(int j = left; j <= right;j++){
            if(p[j][0] + w[i] > c){
                break;
            }
            double nw = p[j][0] + w[i];
            double nv = p[j][1] + v[i];
            //放入比nw小的跳跃点，因为重量小的价值无论大小
            for(;k<=right && p[k][0] < nw;k++,next++){
                p[next][0] = p[k][0];
                p[next][1] = p[k][1];
            }
            //如果重量相等，取价值大的跳跃点
            if(k <= right && p[k][0] == nw){
                if(p[k][1] > nv){
                    nv = p[k][1];
                }
                k++;
            }
            //放入更新的跳跃点
            if(nv > p[next-1][1]){
                p[next][0] = nw;
                p[next][1] = nv;
                next++;
            }
                /*去除比更新的跳跃点重量大却价值小的点，
                  由于是每一次更新完之后结果都是重量和价值都是递增的跳跃点排列
                  一旦出现价值超过当前的点，那后续的点一定都是超过的*/
            for(;k <= right && p[k][1] <= nv;k++);
        }
```

```
        //将后续的点放入
        for (;k <= right; k++,next++){
            p[next][0] = p[k][0];
            p[next][1] = p[k][1];
        }
        left = right+1;right = next - 1;head[i] = next;
    }
    traceBack(v, w, p, head);
    return p[next-1][1];
}
```

路径回溯，找到最优组合

```
private static void traceBack(double[] v, double[] w, double[][] p, int[] head){
    List<Integer> trace = new ArrayList<>();
    int k = head[0]-1;
    int n = w.length;
    for(int i = 1;i <= n;i++){
        int left = head[i+1];
        int right = head[i]-1;
        for(int j = left;j<=right;j++){
            if(p[j][0] + w[i-1] ==  p[k][0] && p[j][1] + v[i-1] == p[k][1]){
                k = j;
                trace.add(i);
                break;
            }
        }
    }
    System.out.println(trace);
}
```

如果不考虑放入哪些物品（即不考虑路径回溯），只关心最优解的值，可以只存放当前的跳跃点集和下一步的跳跃点集，无需记录每一步的跳跃点集

```
public static double solution2(double c, double[] v, double[] w){
    if(v.length != w.length){
        throw new IllegalArgumentException();
    }
    int n = v.length;
    List<double[]> p = new ArrayList<>();
    p.add(new double[]{0, 0});
    for(int i = n - 1; i >= 0;i--){
```

```
            int k = 0;
            List<double[]> q = new ArrayList<>();
            for(double[] element: p){
                if(w[i] + element[0] > c){
                    break;
                }
                double nw = w[i] + element[0];
                double nv = v[i] + element[1];
                for(;k < p.size() && p.get(k)[0] < nw;k++){
                    q.add(p.get(k));
                }
                if(k < p.size() && p.get(k)[0] == nw){
                    if(p.get(k)[1] > nv){
                        nv = p.get(k)[1];
                    }
                    k++;
                }
                if(nv > q.get(q.size()-1)[1]){
                    q.add(new double[]{nw, nv});
                }
                for(;k < p.size() && p.get(k)[1] < nv;k++);
            }
            for(;k < p.size();k++){
                q.add(p.get(k));
            }
            p = q;
        }
        return p.get(p.size()-1)[1];
    }
```