浙江工業大學

BTH004
Assignment1
Experiment Report

（原创）

Student ID	201619630101
Class	1601
Name	Chen Yihang
Teacher	Cheng Zhenbo, John
Date	2018/11/02

Part 1 - Problem description

Mathematically, the multiple knapsack problem can be formulated as follows. We let I = {1, . . . , n} be an index set over the n items, where item i ∈I have a value pi > 0 and a weight wi > 0. In addition, we let J = {1, . . . ,m} be an index set over the m knapsacks, where Wj > 0 denotes the weight capacity of knapsack j ∈ J. For item i ∈ I and knapsack j ∈ J, we let the binary decision variable xij determine whether to include item i in knapsack j: xij = 1 if item i is included in knapsack j, otherwise xij = 0.

The objective function of the problem is to maximize the utility of the chosen items, i.e

$$\boxed{\text{Maximize} \sum_{i \in I} \sum_{j \in J} p_i x_{ij}.}$$

For each of the knapsacks, the solution space is restricted by a weight capacity constraint, which states that the total weight of the selected items for that knapsack is not allowed to exceed the weight capacity of the knapsack. This is modeled by the following constraint set (one constraint for each of the m knapsacks):

$$\boxed{\sum_{i \in I} w_i x_{ij} \leq W_j, \quad j \in J.}$$

In addition, it needs to be explicitly modeled that an item is not allowed to be included in more than one of the knapsacks. This is modeled by the following constraint set (one constraint for each of the n items):

$$\boxed{\sum_{j \in J} x_{ij} \leq 1, \quad i \in I.}$$

Finally, the values of the decision variables are restricted by the constraint set:

$$\boxed{x_{ij} \in \{0, 1\}, \quad i \in I, j \in J.}$$

Part 2 -Algorithms

1. Greedy algorithm

1.1 Greedy algorithm implementation:

A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage [1] with the hope of finding a global optimum. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

In this multiple knapsack problem, I choose the greedy algorithm as following:

1)Calculate each item's value per weight bi = pi / wi, i = 0, 1,2, ..., n.

2)Sort the list of items by the size of bi.

3)Loop list of knapsacks, then loop list of sorted items, put the items into the current knapsacks. If the current knapsack's capacity is not enough, then see whether the next item could put into this knapsack.

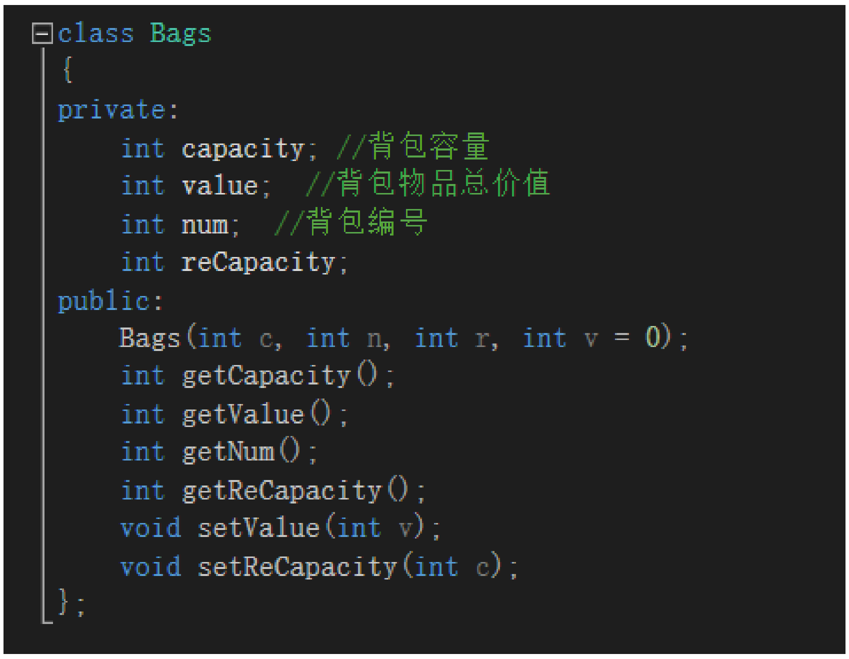
4)If the current knapsack can't load any items, then set next knapsack as current knapsack, loop the items which have not put into any knapsack.

5)If the items all load into knapsacks or the knapsacks can't load any items, end the loop.

Time complexity of greedy algorithm: O(n^2)

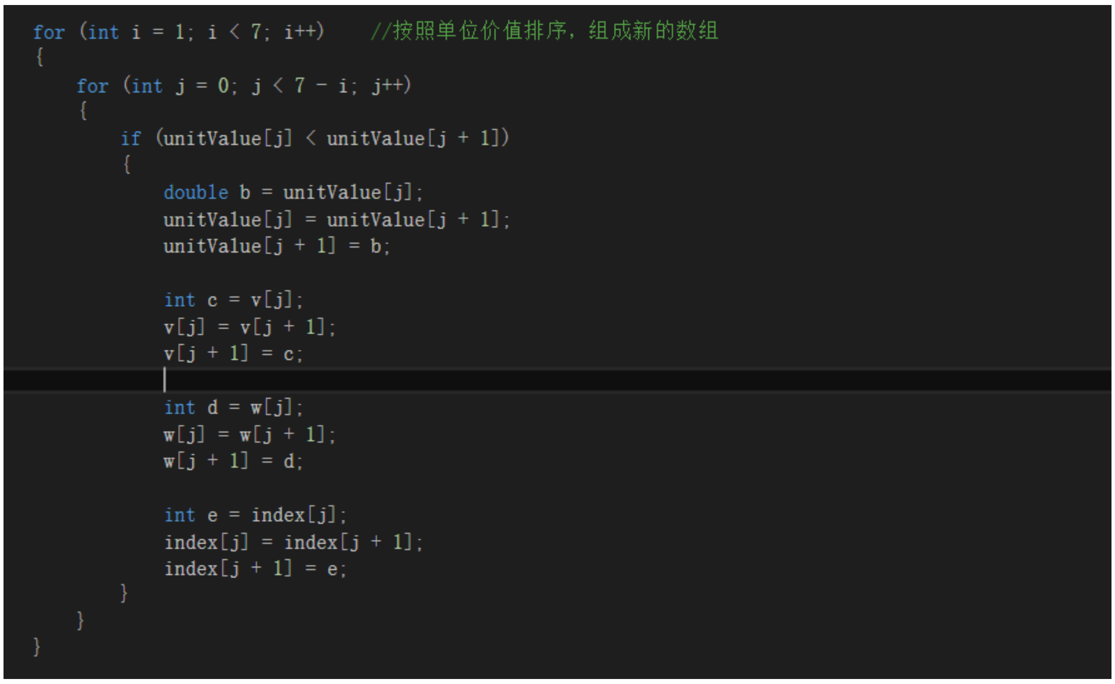
1.2 Code analysis(C++)

1.2.1 Define the Class Bags:



```
class Bags
{
private:
    int capacity; //背包容量
    int value;  //背包物品总价值
    int num;  //背包编号
    int reCapacity;
public:
    Bags(int c, int n, int r, int v = 0);
    int getCapacity();
    int getValue();
    int getNum();
    int getReCapacity();
    void setValue(int v);
    void setReCapacity(int c);
};
```

1.2.2 Calculate the value per weight and sort bi = pi / wi:



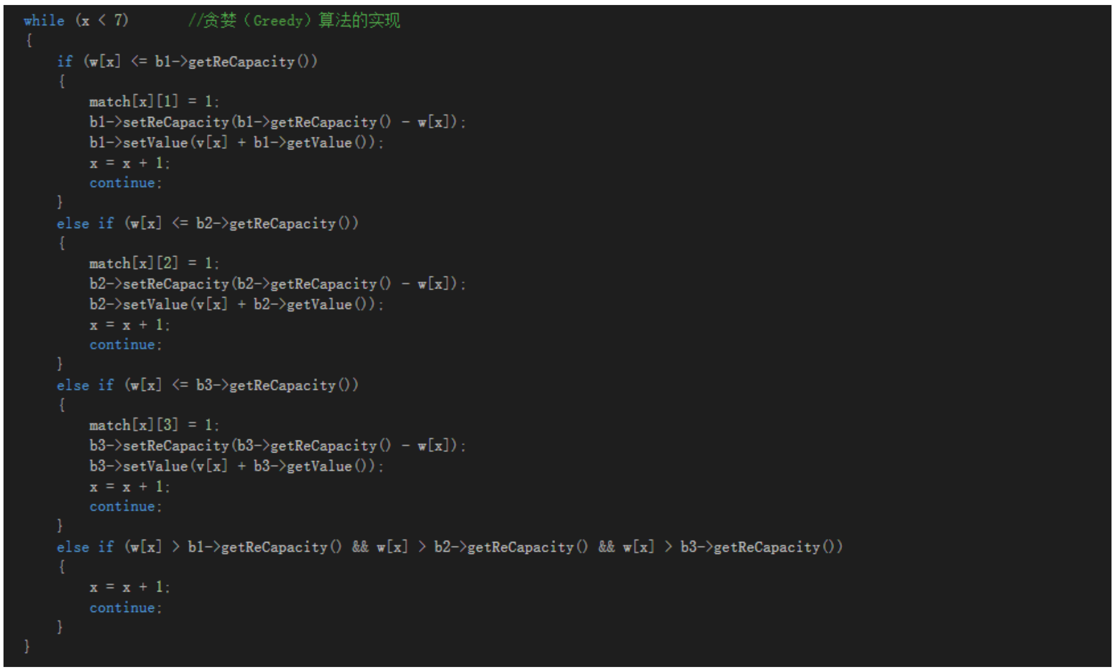
```
for (int i = 1; i < 7; i++)    //按照单位价值排序，组成新的数组
{
    for (int j = 0; j < 7 - i; j++)
    {
        if (unitValue[j] < unitValue[j + 1])
        {
            double b = unitValue[j];
            unitValue[j] = unitValue[j + 1];
            unitValue[j + 1] = b;

            int c = v[j];
            v[j] = v[j + 1];
            v[j + 1] = c;

            int d = w[j];
            w[j] = w[j + 1];
            w[j + 1] = d;

            int e = index[j];
            index[j] = index[j + 1];
            index[j + 1] = e;
        }
    }
}
```

1.2.3 Loop the sorted items and then put them into the knapsack which still remain space to put the item, if not, check the other knapsack. This is how we implement the greedy algorithm:



```
while (x < 7)      //贪婪（Greedy）算法的实现
{
    if (w[x] <= b1->getReCapacity())
    {
        match[x][1] = 1;
        b1->setReCapacity(b1->getReCapacity() - w[x]);
        b1->setValue(v[x] + b1->getValue());
        x = x + 1;
        continue;
    }
    else if (w[x] <= b2->getReCapacity())
    {
        match[x][2] = 1;
        b2->setReCapacity(b2->getReCapacity() - w[x]);
        b2->setValue(v[x] + b2->getValue());
        x = x + 1;
        continue;
    }
    else if (w[x] <= b3->getReCapacity())
    {
        match[x][3] = 1;
        b3->setReCapacity(b3->getReCapacity() - w[x]);
        b3->setValue(v[x] + b3->getValue());
        x = x + 1;
        continue;
    }
    else if (w[x] > b1->getReCapacity() && w[x] > b2->getReCapacity() && w[x] > b3->getReCapacity())
    {
        x = x + 1;
        continue;
    }
}
```

1.3 Debugging of greedy algorithm:

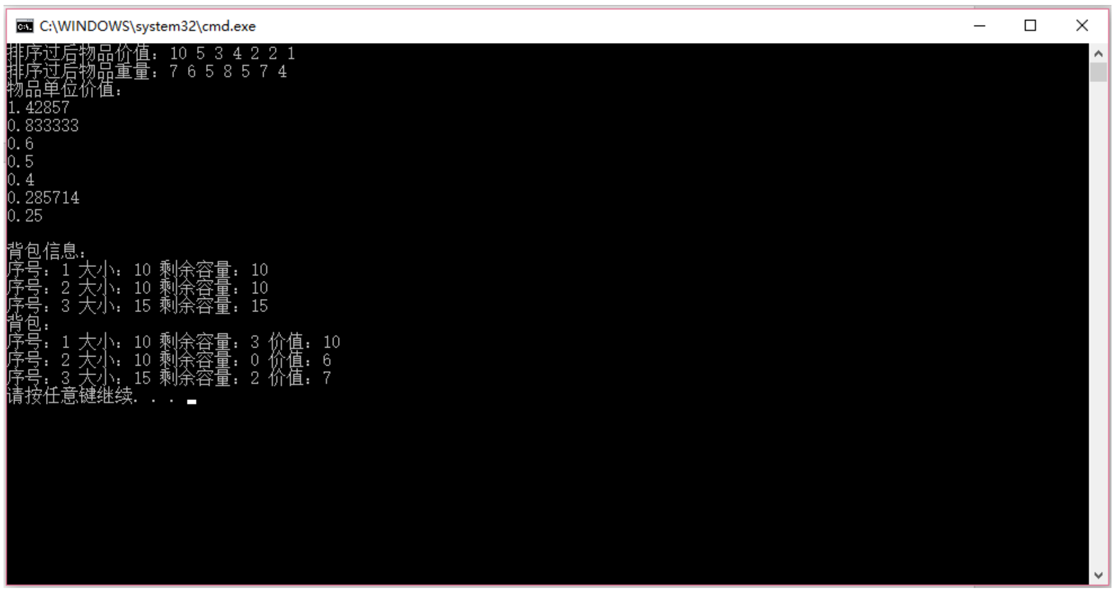
The length of the item list is 7.

I input a list of value of items which is 3,4,1,2,5,2,10

I input a list of weight of the corresponding item which is 5,8,4,7,6,5,7

I input 3 knapsacks and bag1 has the volume 10, bag2 has the volume 10 and bag3 has the volume 15.

I get the following results:



```
排序过后物品价值: 10 5 3 4 2 2 1
排序过后物品重量: 7 6 5 8 5 7 4
物品单位价值:
1.42857
0.833333
0.6
0.5
0.4
0.285714
0.25

背包信息:
序号: 1 大小: 10 剩余容量: 10
序号: 2 大小: 10 剩余容量: 10
序号: 3 大小: 15 剩余容量: 15
背包:
序号: 1 大小: 10 剩余容量: 3 价值: 10
序号: 2 大小: 10 剩余容量: 0 价值: 6
序号: 3 大小: 15 剩余容量: 2 价值: 7
请按任意键继续. . .
```

1.4 Check up the algorithm:

In this case we have 7 items to put into 3 knapsacks. After the greedy algorithm we have three knapsacks with items. First we should check whether we can put more items into three knapsacks and the answer is no. Then we know that item 1 is in knapsack 1, item 2 is in knapsack 2, item 3 is in knapsack 3, item 4 is in knapsack 3, item 5 is not in the knapsack, item 6 is not in the knapsack, item 7 is in knapsack 2, and we could not find a better solution because we have already put the items which have higher value per weight into the knapsacks and there is no more room for the remaining items.

2.Improving search (neighborhood search) algorithm

2.1 Algorithm implementation:

A neighborhood (local) search algorithm is an improving search algorithm that searches for improving solutions in a neighborhood of a current solution. It guarantees local optima and reduces the risk of getting stuck in one "bad" local optima. A neighbor search algorithm can be restarted with different starting solutions. In my implementation, I start with a solution which solved in greedy algorithm and then to optimize it using two steps:

Step: 1 Change the items in different knapsacks to make capacity for items which haven't put into knapsacks.

1) Initialize j=0;

2) For item j, look for the items after him, calculate the weight difference, judge whether can put items that haven't put into knapsacks by swapping these two items;

3) Judge whether the total value is increase, if so, swap them;

4) If j < len (items), j=j+1,repeat the step 2).

Step: 2 Put the items in the knapsacks outside to make capacity for items that haven't put into knapsacks which have bigger value.

1) Initialize j=0;

2) For every item j, compare the value with the items which haven't put into knapsacks.

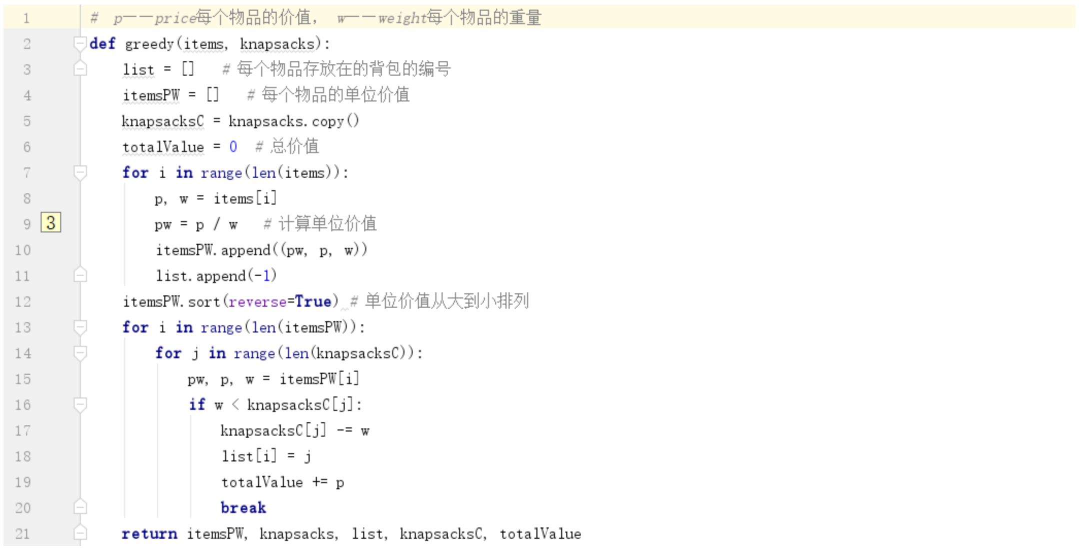
3) If item j 's value is smaller and there is enough capacity after put the item j out for items which haven't put into knapsacks, swap them.

4) If j< len (items), j=j+1,repeat the step 2).

Time Complexity: O(n^2)

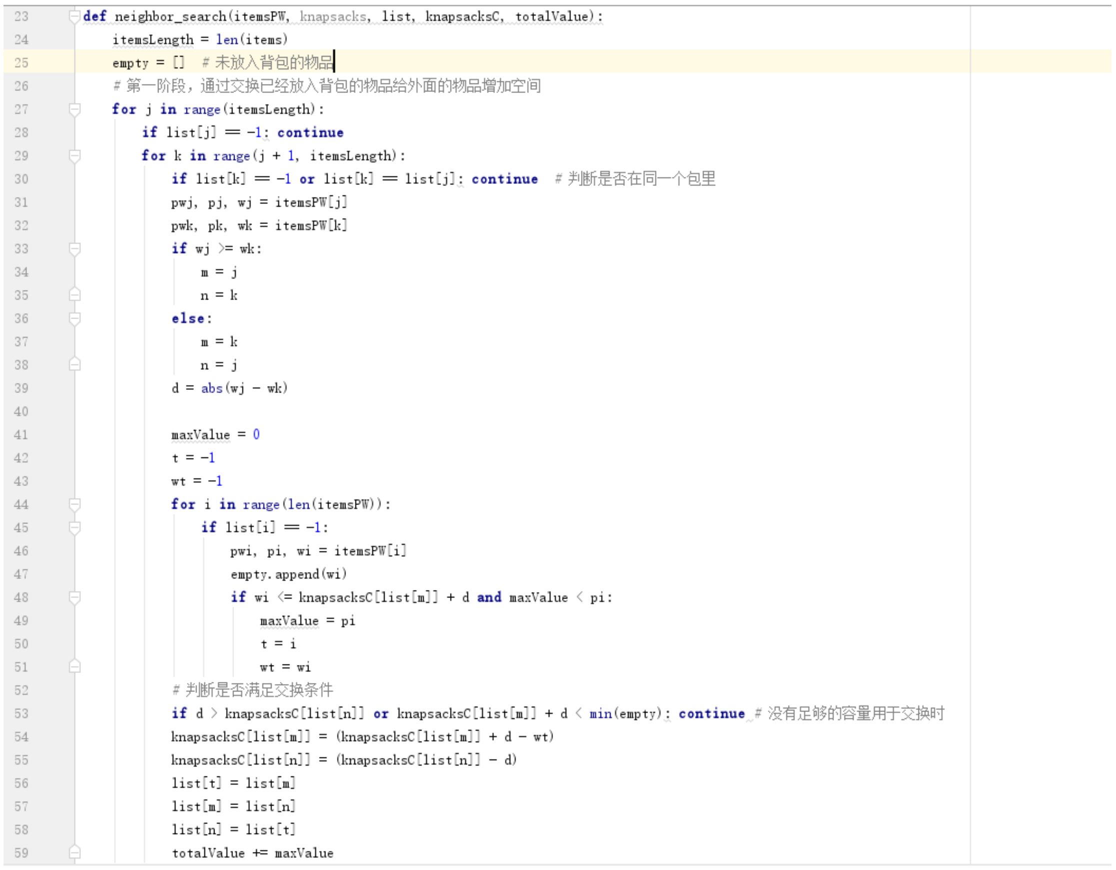
2.2 Code analysis(Python):

2.2.1 Define the greedy algorithm and we get the result using greedy algorithm:



```python
# p——price每个物品的价值， w——weight每个物品的重量
def greedy(items, knapsacks):
    list = []   # 每个物品存放在的背包的编号
    itemsPW = []   # 每个物品的单位价值
    knapsacksC = knapsacks.copy()
    totalValue = 0  # 总价值
    for i in range(len(items)):
        p, w = items[i]
        pw = p / w   # 计算单位价值
        itemsPW.append((pw, p, w))
        list.append(-1)
    itemsPW.sort(reverse=True)  # 单位价值从大到小排列
    for i in range(len(itemsPW)):
        for j in range(len(knapsacksC)):
            pw, p, w = itemsPW[i]
            if w < knapsacksC[j]:
                knapsacksC[j] -= w
                list[i] = j
                totalValue += p
                break
    return itemsPW, knapsacks, list, knapsacksC, totalValue
```

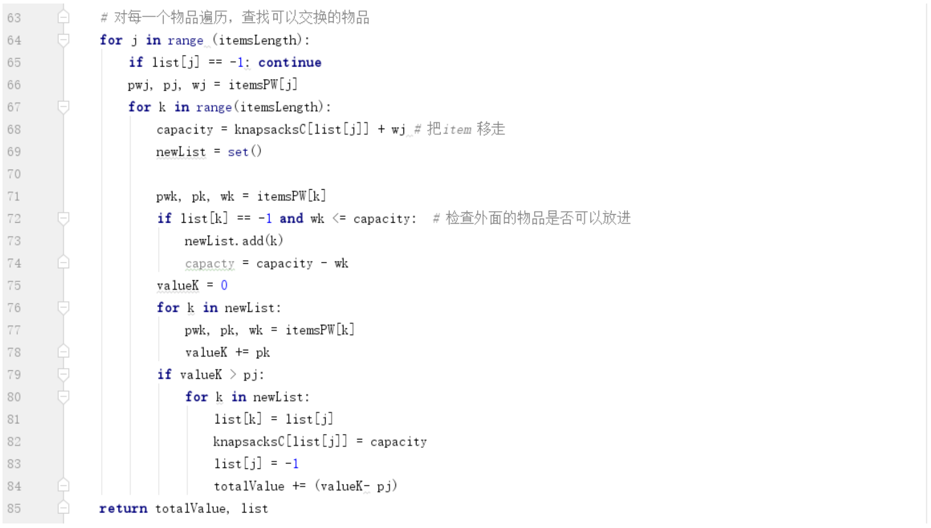
2.2.2 The neighbor search algorithm. In stage 1 we need to switch items between knapsacks to optimize the remaining empty space in order to put in more items outside the knapsacks.



```python
def neighbor_search(itemsPW, knapsacks, list, knapsacksC, totalValue):
    itemsLength = len(items)
    empty = []  # 未放入背包的物品
    # 第一阶段，通过交换已经放入背包的物品给外面的物品增加空间
    for j in range(itemsLength):
        if list[j] == -1: continue
        for k in range(j + 1, itemsLength):
            if list[k] == -1 or list[k] == list[j]: continue  # 判断是否在同一个包里
            pwj, pj, wj = itemsPW[j]
            pwk, pk, wk = itemsPW[k]
            if wj >= wk:
                m = j
                n = k
            else:
                m = k
                n = j
            d = abs(wj - wk)

            maxValue = 0
            t = -1
            wt = -1
            for i in range(len(itemsPW)):
                if list[i] == -1:
                    pwi, pi, wi = itemsPW[i]
                    empty.append(wi)
                    if wi <= knapsacksC[list[m]] + d and maxValue < pi:
                        maxValue = pi
                        t = i
                        wt = wi
            # 判断是否满足交换条件
            if d > knapsacksC[list[n]] or knapsacksC[list[m]] + d < min(empty): continue # 没有足够的容量用于交换时
            knapsacksC[list[m]] = (knapsacksC[list[m]] + d - wt)
            knapsacksC[list[n]] = (knapsacksC[list[n]] - d)
            list[t] = list[m]
            list[m] = list[n]
            list[n] = list[t]
            totalValue += maxValue
```

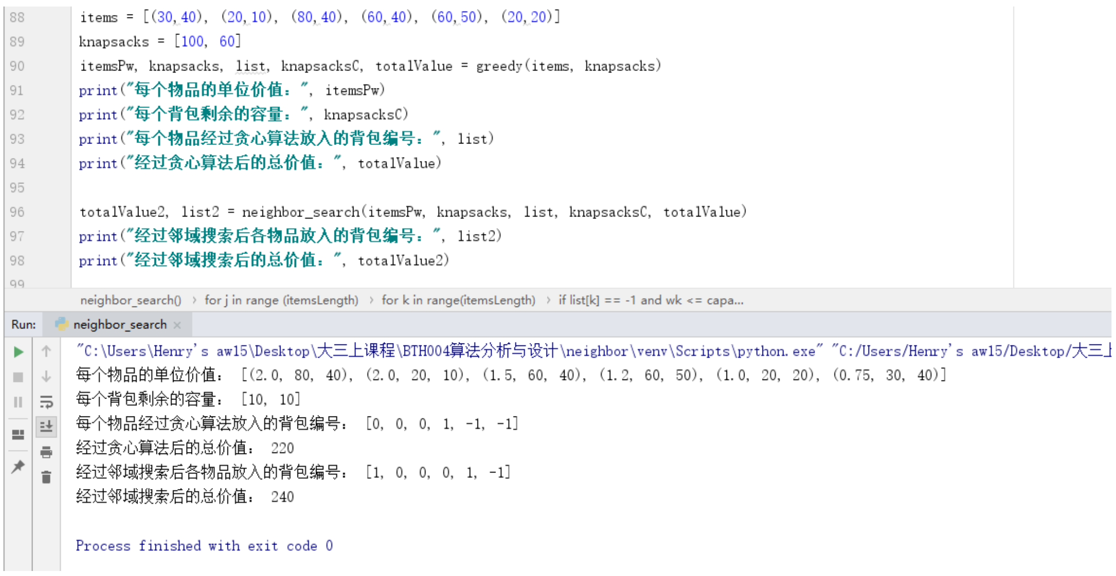
2.2.3 In Stage 2 we need to traversal all the items in the knapsacks and move those items that already in the knapsacks and have less value than the items outside out of the knapsack and put an item which has a higher value per weight into the knapsack if there is enough room.



```python
# 对每一个物品遍历，查找可以交换的物品
for j in range (itemsLength):
    if list[j] == -1: continue
    pwj, pj, wj = itemsPW[j]
    for k in range(itemsLength):
        capacity = knapsacksC[list[j]] + wj # 把item 移走
        newList = set()

        pwk, pk, wk = itemsPW[k]
        if list[k] == -1 and wk <= capacity:  # 检查外面的物品是否可以放进
            newList.add(k)
            capacty = capacity - wk
        valueK = 0
        for k in newList:
            pwk, pk, wk = itemsPW[k]
            valueK += pk
        if valueK > pj:
            for k in newList:
                list[k] = list[j]
                knapsacksC[list[j]] = capacity
                list[j] = -1
                totalValue += (valueK- pj)
return totalValue, list
```

2.3 Debugging of neighbor search algorithm:



```python
items = [(30,40), (20,10), (80,40), (60,40), (60,50), (20,20)]
knapsacks = [100, 60]
itemsPw, knapsacks, list, knapsacksC, totalValue = greedy(items, knapsacks)
print("每个物品的单位价值：", itemsPw)
print("每个背包剩余的容量：", knapsacksC)
print("每个物品经过贪心算法放入的背包编号：", list)
print("经过贪心算法后的总价值：", totalValue)

totalValue2, list2 = neighbor_search(itemsPw, knapsacks, list, knapsacksC, totalValue)
print("经过邻域搜索后各物品放入的背包编号：", list2)
print("经过邻域搜索后的总价值：", totalValue2)
```

neighbor_search() › for j in range (itemsLength) › for k in range(itemsLength) › if list[k] == -1 and wk <= capa...
Run: neighbor_search

```
"C:\Users\Henry's aw15\Desktop\大三上课程\BTH004算法分析与设计\neighbor\venv\Scripts\python.exe" "C:/Users/Henry's aw15/Desktop/大三上
每个物品的单位价值： [(2.0, 80, 40), (2.0, 20, 10), (1.5, 60, 40), (1.2, 60, 50), (1.0, 20, 20), (0.75, 30, 40)]
每个背包剩余的容量： [10, 10]
每个物品经过贪心算法放入的背包编号： [0, 0, 0, 1, -1, -1]
经过贪心算法后的总价值： 220
经过邻域搜索后各物品放入的背包编号： [1, 0, 0, 0, 1, -1]
经过邻域搜索后的总价值： 240

Process finished with exit code 0
```

2.5 Check up the algorithm:

From the debugging result we can find that when using the greedy algorithm, we get the total value of 220 and we put the item 1, 2, 3 into the first knapsack and the item 4 into the second knapsack. Then we try the neighbor search algorithm and we get a total value of 240 which is higher than the former one. We put item 1, 5 into the first knapsack and we put item 2, 3, 4 into the second knapsack. After the neighbor search algorithm we've got a better solution.

3. Tabu-search

3.1. Problem description

A problem with the neighborhood search heuristics is that they eventually will get stuck in a locally optimal solution. A locally optimal solution is a solution that cannot be improved by moving to any of the solutions that are sufficiently close in the search space (i.e., in the neighborhood), but there are solutions farther away that are better.

3.2. Algorithm description

Tabu-search algorithm is based on neighbor search algorithm to avoid loop. However, in the mutiple knapsack problem, I have used the “bestlist” to record the best solutions in the previous neighbor search algorithm, and in this case I have interated all the situations so the bestlist is global optimum solution. We do not need to do another tabu-search algorithm in this case.

3.3.Source code

```
def moveAndAdd(visited, Wj, Wi, w, Pi):
    keyCopy = list(visited.keys())   # 返回所有可用键
    keys = copy.deepcopy(keyCopy)
    i = len(Wi)
    tempPrice = 0   # 目前的价值
    bestPrice = 0   # 最高价值
    bestList = { }   # 存储最好的情况
    unvisited = []
    for i in range(i):
        if i not in visited:
            unvisited.append(i)
    if len(unvisited) != 0:
        for i in range(len(keyCopy)):
            q = keyCopy[i]
            value = visited.get(q)
            visited.pop(q)   # pop() 函数用于移除列表中的一个元素（默认最后一个元素），并且返回该
元素的值。
            keys.remove(q)
            unvisited.append(q)
            now_W = Wj[value] - w[value] + Wi[q]   # 可放入的重量
            for k in range(len(unvisited) - 1):
                for j in range(k + 1, len(unvisited)):
                    now_WCopy = now_W   # 可放入的重量
                    if j > len(unvisited):
                        continue
                    if now_WCopy > Wi[unvisited[k]] and now_WCopy > Wi[unvisited[j]]:
                        if Wi[unvisited[k]] + Wi[unvisited[j]] <= now_WCopy:
                            tempPrice = Pi[unvisited[k]] + Pi[unvisited[j]] - Pi[q]
                            if tempPrice > bestPrice:
                                bestPrice = tempPrice
                                visited[unvisited[k]] = value
                                visited[unvisited[j]] = value
                                for q in visited:
                                    bestList[q] = visited[q]
                                unvisited.remove(unvisited[k])
                                j -= 1
                                unvisited.remove(unvisited[j])
                            else:
                                visited[q] = value
                                keys.append(q)
                                unvisited.remove(q)
                        else:
                            if Pi[unvisited[j]] > Pi[unvisited[k]]:
                                tempPrice = Pi[unvisited[j]] - Pi[q]
                                if tempPrice > bestPrice:
                                    bestPrice = tempPrice
                                    visited[unvisited[j]] = value
                                    bestList = list(visited.keys())
                                    keys.append(unvisited[j])
                                    unvisited.remove(unvisited[j])
                                else:
                                    visited[q] = value
                                    keys.append(q)
                                    unvisited.remove(q)
                            else:
                                tempPrice = Pi[unvisited[k]] - Pi[q]
                                if tempPrice > bestPrice:
                                    bestPrice = tempPrice
                                    visited[unvisited[k]] = value
                                    bestList = list(visited.keys())
                                    visited[unvisited[k]] = value
                                    bestList = list(visited.keys())
                                    keys.append(unvisited[k])
                                    unvisited.remove(unvisited[k])
                                else:
                                    visited[q] = value
                                    keys.append(q)
                                    unvisited.remove(q)
                    elif now_WCopy > Wi[unvisited[k]] and now_WCopy < Wi[unvisited[j]]:
                        tempPrice = Pi[unvisited[k]] - Pi[q]
                        if tempPrice > bestPrice:
                            bestPrice = tempPrice
                            visited[unvisited[k]] = value
                            bestList = list(visited.keys())
                            keys.append(unvisited[k])
                            unvisited.remove(unvisited[k])
                        else:
                            visited[q] = value
                            keys.append(q)
                            unvisited.remove(q)
                    elif now_WCopy > Wi[unvisited[j]] and now_WCopy<Wi[unvisited[k]]:
                        tempPrice = Pi[unvisited[j]] - Pi[q]
                        if tempPrice > bestPrice:
                            bestPrice = tempPrice
                            visited[unvisited[j]] = value
                            bestList = list(visited.keys())
                            keys.append(unvisited[j])
                            unvisited.remove(unvisited[j])
                        else:
                            visited[q] = value
                            keys.append(q)
                            unvisited.remove(q)
                    else:
                        visited[q] = value
                        keys.append(q)
                        unvisited.remove(q)
    totalPrice = caculatePrice(Pi,bestList)
    return totalPrice, bestList
```

3.4.Debugging

```
After swapping the knapsack:
Value: 180
{0: 1, 1: 0, 2: 0}
After rotation:
Value: 210
{1: 0, 2: 0, 3: 1, 0: 1}
------------------------------------------------------------------
Final result:
(210, {1: 0, 2: 0, 3: 1, 0: 1})
```