BTH004 - Laboratory assignment 1— Multiple knapsacks problem and solutions

name: Zhao Shui–
student ID: 201806150329

November 2020

Part I

1 Greedy algorithm

1.1 mainly description for the algorithm

allocate knapsack for every items in the order of values per weight unit.

1.2 implementation step

step 1: Create a new list calculate all items' unit value(value per weight unit)in the items list and record its id, and then sort it in the order of unit value.

step 2: For every item in the sorted list, allocate knapsack for the item in the order. if there are enough room in this knapsack , then include it in the current knapsack. If all knapsacks don't have enough room for the item, then it will not be included in all knapsacks.

step 3: All items have been allocated knapsack once, algorithm them end.

1.3 two fit method are alternative: first fit and best fit

first fit: Once the current knapsack has enough room, then include in the knapsack immediately.

best fit: Search all knapsacks, and look for the best knapsack(which left capacity most closed to the item weight)

1.4 descriptions for some criteria

termination criteria: All items have been allocated knapsacks once, then end the algorithm.

1.5 pseudo code

```
def greedyForMultipleKnapsack_firstFit(knapsacks,items):

    valuesPerUnit = []
    leftCapacity = knapsacks.copy()

    for i in range(len(items)):
        valuesPerUnit.append([items[i][0]/items[i][1],i])

    #sort the array by the order of value per unit
    totalValue = 0
    for i in range(len(items)):
        for j in range(len(knapsacks)):
            #if find the knapsack with enough space:
                #include the item in this knapsack,add the value to the total value and reduce the knapsack space.
    return items,totalValue

def greedyForMultipleKnapsack_bestFit(knapsacks,items):
    valuesPerUnit = []
    leftCapacity = knapsacks.copy()
    for i in range(len(items)):
        valuesPerUnit.append([items[i][0]/items[i][1],i])
    #sort the array by the order of value per unit

    totalValue = 0
    for i in range(len(items)):
        knapIndex = -1
        for j in range(len(knapsacks)):
            #find the knapsack who have closest space for current item,
        if knapIndex != -1:
            totalValue += items[valuesPerUnit[i][1]][0]
            leftCapacity[knapIndex] -= items[valuesPerUnit[i][1]][1]
        items[valuesPerUnit[i][1]][2] = knapIndex

    return items,totalValue
```

1.6 code & comment

```
def greedyForMultipleKnapsack_firstFit(knapsacks,items):
# input:
# @para knapsacks -> [1,2] represent 2 knapsacks with (weight) capacity 1,2
# @para items -> [[1,2,-1],[3,4,0]] represent 2 items
# items[0] means this item worth 1 unit, weigh 2 but not include in any kanpsack,
# while item[1] means this item worth 3 unit, weigh 4 unit and is included in knapsacks[0]
```

```
    print("\033[1;35mgreedy algorithm for multiple knapsacks --- first fit start->")
    print('input:')
    print('knapsacks: \n{}'.format(knapsacks))
    print('items: \n{}\n'.format(items))

    valuesPerUnit = []
    #used to record the value per weight unit for item and item index.
    leftCapacity = knapsacks.copy()
    #we will modify the list so copy a new knapsacks to avoid to modify the original
    for i in range(len(items)):
        #fill the lists mentioned above
        valuesPerUnit.append([items[i][0]/items[i][1],i])
    print('and the value per weight unit with the relevant item id are: \n{}'.format(valuesPerUnit))

    valuesPerUnit.sort(reverse=True)
    #descending sort by the value per unit
    print('after sort, it becomes: \n{}'.format(valuesPerUnit))
    print("\nnow we start allocate knapsack for each items in the order of the value per unit ->\033[0m")
    totalValue = 0
    # record the total values of these items have included
    for i in range(len(items)):
        #allocate knapsack for the items in the order of values per unit
        isInclude = False
        print("\033[1;32m-----------------------\n\033[0m")
        print("\033[1;32mnow allocate knapsack for items[{}]\n\033[0m".format(valuesPerUnit[i][1]))
        print("\033[1;36mknapsacks' condition:{}\n\033[0m".format(leftCapacity))
        for j in range(len(knapsacks)):
            #search rest space in knapsack for the item
            print("for knapsacks[{}], it still has  {}  weight left ".format(j,leftCapacity[j]))
            print("and items[{}] weight  {} ".format(valuesPerUnit[i][1],items[valuesPerUnit[i][1]][1]))
            input()
            if items[valuesPerUnit[i][1]][1] <= leftCapacity[j]:
                # if have enough space for the current item
                print("\033[1;34mit has enough space for the item, so we include it in this knapsack\n\033[0m")
                totalValue += items[valuesPerUnit[i][1]][0]
                # add the item value to the total
                leftCapacity[j] -= items[valuesPerUnit[i][1]][1]
                # it take up a part of space of the current knapsack,so we decrease the room
                items[valuesPerUnit[i][1]][2] = j
                # record its position
                isInclude = True
                break
            # current item have found its place,next item
            else:
                print("\033[1;33mknapsacks[{}] not has enough space
```

```
 for this item\n\033[0m".format(j))
        if not isInclude:
            print("\033[1;31mall knapsacks have no space for this item!\n\033[0m")
    print('\033[1;35mall items have been allocated!')
    print("\033[1;35m\n***************")
    print("<-algorithm end")

    print('{} {}\033[0m'.format(items,totalValue))
    return items,totalValue

def greedyForMultipleKnapsack_bestFit(knapsacks,items):
    print("\033[1;35mgreedy algorithm for multiple knapsacks ---best fit start->")
    print('input:')
    print('knapsacks: \n{}'.format(knapsacks))
    print('items: \n{}\n'.format(items))

    valuesPerUnit = []
    #used to record the value per weight unit for item and item index
    leftCapacity = knapsacks.copy()
    #we will modify the list so copy a new knapsacks to avoid to modify the original
    for i in range(len(items)):
        #fill the lists mentioned above
        valuesPerUnit.append([items[i][0]/items[i][1],i])
    print('and the value per weight unit with the relevant item id are: \n{}'.format(valuesPerUnit))

    valuesPerUnit.sort(reverse=True)
    #descending sort by the value per unit
    print('after sort, it becomes: \n{}'.format(valuesPerUnit))
    print("\nnow we start allocate knapsack for each items in the order of the value per unit ->\033[0m")

    totalValue = 0
    # record the total values of these items have included
    for i in range(len(items)):
        #allocate knapsack for the items in the order of values per unit
        difference = max(leftCapacity)
        # used to record the best fit diiference, initalize with the largest left rooms in these knapsacks
        knapIndex = -1
        # used to record the best fit diiference
        print("\033[1;32m-----------------------\n\033[0m")
        print("\033[1;32mnow allocate knapsack for items[{}]\n\033[0m".format(valuesPerUnit[i][1]))
        print("\033[1;36mknapsacks' condition:{}\n\033[0m".format(leftCapacity))
        for j in range(len(knapsacks)):
            #search rest space in knapsack for the item
            print("for knapsacks[{}], it still has  {}  weight left ".format(j,leftCapacity[j]))
            print("and items[{}] weight  {} ".format(valuesPerUnit[i][1],items[valuesPerUnit[i][1]][1]))
```

```
            input()
            newDifference = leftCapacity[j] - items[valuesPerUnit[i][1]][1]
            if newDifference >= 0:
                print("\033[1;36mknapsacks[{}] it has enough space for this item\n\033[0m".format(j))
                if newDifference < difference:
                    #if found a new better fit in knapsacks, choose it as the best fit
                    print("\033[1;36mand it's more appropriate for this item\n\033[0m")
                    knapIndex = j
                    difference = newDifference
                else:
                    print("\033[1;33mbut it's not better than current one\n\033[0m")
            else:
                print("\033[1;33mknapsacks[{}] not has enough space for this item\n\033[0m".format(j))

        if knapIndex != -1:
            print("\033[1;34mso we include it in knapsacks[{}](weigh {} capacity {})\n\033[0m".format(knapIndex,items[valuesPerUnit[i][1]][1],knapsacks[knapIndex]))
            totalValue += items[valuesPerUnit[i][1]][0]
            # add the item value to the total
            leftCapacity[knapIndex] -= items[valuesPerUnit[i][1]][1]
            # it take up a part of space of the current knapsack,so we decrease the room
        else:
            print("\033[1;31mall knapsacks have no space for this item!\n\033[0m")
        items[valuesPerUnit[i][1]][2] = knapIndex

    print('\033[1;35mall items have been allocated!')
    print("\033[1;35m\n****************")
    print("<-algorithm end")

    print('{} {}\033[0m'.format(items,totalValue))
    return items,totalValue
```

1.7 correctness check & test

As you see in the code above, a lot of print statement are in the code section. They are used to show the process of the program.

```
greedy algorithm for multiple knapsacks --- first fit start->
input:
knapsacks:
[5, 1, 2]
items:
[[2, 3, -1], [5, 3, -1], [2, 2, -1], [8, 5, -1], [1, 1, -1]]

and the value per weight unit with the relevant item id are:
[[0.6666666666666666, 0], [1.6666666666666667, 1], [1.0, 2], [1.6, 3], [1.0, 4]]
after sort, it becomes:
[[1.6666666666666667, 1], [1.6, 3], [1.0, 4], [1.0, 2], [0.6666666666666666, 0]]

now we start allocate knapsack for each items in the order of the value per unit ->
------------------------

now allocate knapsack for items[1]

knapsacks' condition:[5, 1, 2]

for knapsacks[0], it still has  5  weight left
and items[1] weight  3
```

That means we can observe the process of the program, that's the most direct way of correctness check.
for current example, the input of this algorithm will be:

```
knapsacks[0] not has enough space for this item

for knapsacks[1], it still has  1  weight left
and items[0] weight  3


knapsacks[1] not has enough space for this item

for knapsacks[2], it still has  0  weight left
and items[0] weight  3


knapsacks[2] not has enough space for this item

all knapsacks have no space for this item!

all items have been allocated!

****************
<-algorithm end
[[2, 3, -1], [5, 3, 0], [2, 2, 2], [8, 5, -1], [1, 1, 0]] 8
```

2 Neighbourhood search

2.1 mainly description for the algorithm

Choose a suitable solution as the start solution, then find its best "neighbourhood" from all neighbourhood as the solution for the next iteration.When the best neighbourhood is even not better than the current one, then the algorithm end and output the current solution.

2.2 implementation step

step 1: Choose a suitable solution as the start solution.

step 2: while there are better neighbourhood existing, then choose the neighbourhood as the solution for next iteration. Redo this step continually.

step 3: If the best neighbourhood is even not better than the current solution, the algorithm end and output the current solution.

2.3 descriptions for some criteria

neighbourhood definition: On the base of the current solution, move one item to a certain knapsack. After removing a lowest value per unit item from this knapsack(if need), if there is enough space for this new item, then it is neighbourhood of the current solution.

termination criteria: If the best neighbourhood is even not better than the current solution, the algorithm end and output the current solution.

2.4 pseudo code

```
def neighbourhood(knapsacks,items):
    # search neighbourhoods of current solution
    #the definition of neighbour is: move one item's from a
    knapsack (or not included) to another knapsack
    Fx = []
    for i in range(len(items)):
        for j in range(len(knapsacks)):
            if items[i][2] != j:
                # search the neighbours but not include itself
                # move items[[Xn[i]][1]] to knapsack j
                # calculate value for current solution.
                # if the total weight out of the capacity, then
    give up an item with the lowest value per unit.
                #then add this solution to the Fx
    return Fx

def bestNeighbourhood(Fx,knapsacks,items):
    # find the best neighbour with the largest value in all
    neighbours


def neighbourhoodSearchForMultipleKnapsack(knapsacks,items):

    currentBestValue = 0
    # record the total value with current solution
    for item in items:
        if item[2] != -1:
            currentBestValue += item[0]

```

```python
    bestNeighbourhoodValue = currentBestValue + 1 # record the best values of all neighbourhoods
    firstR = True
    count = 0
    bestSolution = []
    for item in items:
        bestSolution.append(item[:])

    while currentBestValue < bestNeighbourhoodValue:
        currentBestValue = bestNeighbourhoodValue
        if firstR:
            firstR = False
            currentBestValue -= 1
        items = []
        for item in bestSolution:
            items.append(item[:])
        nbhs = neighbourhood(knapsacks, items)
        bestSolution,bestNeighbourhoodValue = bestNeighbourhood(nbhs, knapsacks, items)
        count+=1

    return items,currentBestValue
```

2.5 code & comment

```python
def neighbourhood(knapsacks,items):
    # search neighbourhoods of current solution
    #the definition of neighbour is: move one item's from a knapsack (or not included) to another knapsack
    Fx = []
    for i in range(len(items)):
        for j in range(len(knapsacks)):
            if items[i][2] != j:
                # search the neighbours but not include itself
                temp = items[i][2]
                items[i][2] = j
                # move items[[Xn[i]][1]] to knapsack j

                totalValue = 0
                # calculate value for current solution.
                for item in items:
                    if item[2] != -1:
                        totalValue += item[0]
                totalWeight = 0
                for item in items:
                    if item[2] == j :
                        totalWeight += item[1]
                if knapsacks[j] >= totalWeight:
                    Xtemp = []
                    for item in items:
                        Xtemp.append([item[0],item[1],item[2]])
                    Fx.append([Xtemp,totalValue])
                else:
                    index = i
                    minVal = items[i][0]
                    for k in range(len(items)):
                        if items[k][2] == j and items[k][0] <
```

```python
minVal:
                            index = k
                            minVal = items[k][0]
                    totalValue -= items[index][0]
                    if index != i and totalWeight - items[index][1] <= knapsacks[j]:
                        items[index][2] = -1
                        Xtemp = []
                        for item in items:
                            Xtemp.append([item[0],item[1],item[2]])
                        Fx.append([Xtemp,totalValue])

                items[i][2] = temp
    return Fx

def bestNeighbourhood(Fx,knapsacks,items):
    # find the best neighbour with the largest value in all neighbours
    if len(Fx) == 0:
        return [],0

    bestIndex = -1
    maxTotalValue = 0
    for i in range(len(Fx)):
        if Fx[i][1] > maxTotalValue:
            maxTotalValue = Fx[i][1]
            bestIndex = i
    return Fx[bestIndex][0],maxTotalValue

def neighbourhoodSearchForMultipleKnapsack(knapsacks,items):
    print("\033[1;35mneighbourhood search algorithm for multiple knapsacks --- start->")
    print('input:')
    print('knapsacks: \n{}'.format(knapsacks))
    print('items: \n{}\n\033[0m'.format(items))

    currentBestValue = 0
    # record the total value with current solution
    for item in items:
        if item[2] != -1:
            currentBestValue += item[0]

    bestNeighbourhoodValue = currentBestValue + 1 # record the best values of all neighbourhoods
    firstR = True
    print("\033[1;36mwe choose the input items as our start solution. it's: \n{}\033[0m".format(items))
    input()
    count = 0
    bestSolution = []
    for item in items:
        bestSolution.append(item[:])

    while currentBestValue < bestNeighbourhoodValue:
        #termination condition: current value equals the best value of all of its neighbours
        currentBestValue = bestNeighbourhoodValue
```

```
        if firstR:
            firstR = False
            currentBestValue -= 1
        items = []
        for item in bestSolution:
            items.append(item[:])
        print("\033[1;32miteration {}:".format(count))
        print("------------------------\n\033[0m")
        print("\033[1;36mthe current solution is: \n{}".format(bestSolution))
        print("for this solution, the total value is: {}\n\033[0m".format(currentBestValue))
        nbhs = neighbourhood(knapsacks, items)
        print("and all it's feasible {} neighbours are:".format(len(nbhs)))
        input()
        for nbh in nbhs:
            print("solution is: \n{}".format(nbh[0]))
            print("value is {}\n".format(nbh[1]))
        bestSolution,bestNeighbourhoodValue = bestNeighbourhood(nbhs, knapsacks, items)
        print("\033[1;34mso we choose the neighbour \n{}\nas our next solution, value is {}\033[0m".format(bestSolution, bestNeighbourhoodValue))
        count+=1
        input()
    else:
        print("\033[1;33mbut we have found that even the best neighbour is not better than the current solution, so we end the loop\n\033[0m")
    print("\033[1;35m\n****************\033[0m")
    print("\033[1;35m<-algorithm end\n\033[0m")

    return items,currentBestValue
```

2.6 correctness check & test

Like above saying that we can observe the process of the program. we choose the output of the greedy algorithm as the start solution.

```
neighbourhood search algorithm for multiple knapsacks --- start->
input:
knapsacks:
[5, 1, 2]
items:
[[2, 3, -1], [5, 3, 0], [2, 2, 2], [8, 5, -1], [1, 1, 0]]

we choose the input items as our start solution. it's:
[[2, 3, -1], [5, 3, 0], [2, 2, 2], [8, 5, -1], [1, 1, 0]]
```

and for each iteration, it will find neighbourhoods of current solution

```
iteration 0:
-----------------------

the current solution is:
[[2, 3, -1], [5, 3, 0], [2, 2, 2], [8, 5, -1], [1, 1, 0]]
for this solution, the total value is: 8

and all it's feasible 4 neighbours are:


solution is:
[[2, 3, -1], [5, 3, 0], [2, 2, 0], [8, 5, -1], [1, 1, -1]]
value is 7

solution is:
[[2, 3, -1], [5, 3, -1], [2, 2, 2], [8, 5, 0], [1, 1, -1]]
value is 10

solution is:
```

and its output is

```
-----------------------

the current solution is:
[[2, 3, -1], [5, 3, -1], [2, 2, 2], [8, 5, 0], [1, 1, 1]]
for this solution, the total value is: 11

and all it's feasible 0 neighbours are:


so we choose the neighbour
[]
as our next solution, value is 0


but we have found that even the best neighbour is not better than the current solution, so
we end the loop


****************
<-algorithm end

([[2, 3, -1], [5, 3, -1], [2, 2, 2], [8, 5, 0], [1, 1, 1]], 11)
```

Part II

3 Tabu-search

3.1 mainly description for the algorithm

On the base of the neighbourhood search, we record the best solution when we search the neighbours.The difference is when we found that best neighbour is not better than the current one, we continue the algorithm until it reached the iteration times we provided. Also a tabu list will be used, we add the current solution into the list, and when the list is full, we replace the oldest solution in the list with the current one. all solutions in tabu list will be removed from the neighbours list.

3.2 implementation step

step 1: Choose a suitable solution as the start solution.

step 2: for each iteration, choose the best from those neighbourhoods are not inlcuded in the tabu list as the solution for next iteration. Redo this step continually.

step 3: If the iteration times equals the given times, or there are no feasible alternative neighbours, end the algorithm.

3.3 descriptions for some criteria

neighbourhood definition: On the base of the current solution, move one item to a certain knapsack. After removing a lowest value per unit item from this knapsack(if need), if there is enough space for this new item, then it is neighbourhood of the current solution.

tabu list length: It was designed as a parameter that passed by user, that means we can choose to change the tabu list length to get different results.

termination criteria: If the iteration times equals the given times, or there are no feasible alternative neighbours, end the algorithm.

3.4 pseudo code

```
def tabuSearch(knapsacks,items,iterTimes,tabuLength):

    currentBestValue = 0
    # record the total value with current solution
    bestNeighbourhoodValue = 0
    # record the best values of all neighbourhoods
    for item in items:
        if item[2] != -1:
            currentBestValue += item[0]
    bestNeighbourhoodValue = currentBestValue
    bestSolution = []
    for item in items:
        bestSolution.append(item[:])
    tabuList = []
    for i in range(iterTimes):
        solution = []
        for item in bestSolution:
            #deep copy
            solution.append(item[:])
        if i < tabuLength:
            tabuList.append(solution)
        else:
            tabuList[i % tabuLength] = solution
        nbhs = neighbourhood(knapsacks, bestSolution)
```

```
        for tabuSolution in tabuList:
            # remove solutions in tabu list
            for nbh in nbhs:
                if tabuSolution in nbh:
                    nbhs.remove(nbh)
        bestSolution,bestNeighbourhoodValue = bestNeighbourhood(nbhs, knapsacks, items)
        if currentBestValue < bestNeighbourhoodValue:
            items = []
            for item in bestSolution:
                items.append(item[:])
            currentBestValue = bestNeighbourhoodValue

    return items,currentBestValue
```

3.5 code & comment

```
def tabuSearch(knapsacks,items,iterTimes,tabuLength):
    print("\033[1;35mneighbourhood search algorithm for multiple knapsacks ---tabu search start->")
    print('input:')
    print('knapsacks: \n{}'.format(knapsacks))
    print('items: \n{}'.format(items))
    print('iteration times: \n{}'.format(iterTimes))
    print('length of tabu list: \n{}\n\033[0m'.format(tabuLength))

    currentBestValue = 0
    # record the total value with current solution
    bestNeighbourhoodValue = 0
    # record the best values of all neighbourhoods
    for item in items:
        if item[2] != -1:
            currentBestValue += item[0]
    bestNeighbourhoodValue = currentBestValue
    bestSolution = []
    for item in items:
        bestSolution.append(item[:])

    print("\033[1;36mwe choose the input items as our start solution. it's: \n{}\033[0m".format(items))
    input()

    tabuList = []
    for i in range(iterTimes):
        print("\033[1;32miteration {}:".format(i+1))
        print("------------------------\n\033[0m")
        print("and the current solution is: \n{}".format(bestSolution))
        print("for this solution, the total value is: {}\n\033[0m".format(bestNeighbourhoodValue))

        solution = []
        for item in bestSolution:
            #deep copy
            solution.append(item[:])
        if i < tabuLength:
            tabuList.append(solution)
```

```
        else:
            tabuList[i % tabuLength] = solution
        print("update the tabu list")
        print("and now the list is :\n")
        for tabu in tabuList:
            print("\033[1;31m{}\033[0m".format(tabu))
        nbhs = neighbourhood(knapsacks, bestSolution)
        for tabuSolution in tabuList:
            # remove solutions in tabu list
            for nbh in nbhs:
                if tabuSolution in nbh:
                    nbhs.remove(nbh)
        print("\nand all it's feasible(enough capacity and also not in tabulist) {} neighbours are:".format(len(nbhs)))
        input()
        if len(nbhs) == 0:
            print("\033[1;31mthere are no solution for next iteration. so we end the loop\033[0m")
            break;
        for nbh in nbhs:
            print("solution is: \n{}".format(nbh[0]))
            print("value is {}\n".format(nbh[1]))
        input()
        bestSolution,bestNeighbourhoodValue = bestNeighbourhood(nbhs, knapsacks, items)
        print("\033[1;34mso we choose the neighbour \n{}\nas our next solution, value is {}\033[0m".format(bestSolution, bestNeighbourhoodValue))

        print("\033[1;36mthe best solution now is: \n{}\nthe total value is: {}\n".format(items,currentBestValue))

        if currentBestValue < bestNeighbourhoodValue:
            items = []
            for item in bestSolution:
                items.append(item[:])
            currentBestValue = bestNeighbourhoodValue
            print("\033[1;34mso we update the best solution, now the best value is {}\n\033[0m".format(currentBestValue))
        else:
            print("\033[1;33mso we do not update the best solution\033[0m")
    print("\033[1;35m\n****************\033[0m")
    print("\033[1;35m<-algorithm end\n\033[0m")

    return items,currentBestValue
```

3.6 correctness check & test

Like above saying that we can observe the process of the program. we choose the output of the greedy algorithm as the start solution.

```
neighbourhood search algorithm for multiple knapsacks ---tabu search start->
input:
knapsacks:
[5, 1, 2]
items:
[[2, 3, -1], [5, 3, 0], [2, 2, 2], [8, 5, -1], [1, 1, 0]]
iteration times:
15
length of tabu list:
2

we choose the input items as our start solution. it's:
[[2, 3, -1], [5, 3, 0], [2, 2, 2], [8, 5, -1], [1, 1, 0]]
```

in one iteration, the tabu list is

```
iteration 2:
------------------------

and the current solution is:
[[2, 3, -1], [5, 3, -1], [2, 2, 2], [8, 5, 0], [1, 1, -1]]
for this solution, the total value is: 10

update the tabu list
and now the list is :

[[2, 3, -1], [5, 3, 0], [2, 2, 2], [8, 5, -1], [1, 1, 0]]
[[2, 3, -1], [5, 3, -1], [2, 2, 2], [8, 5, 0], [1, 1, -1]]

and all it's feasible(enough capacity and also not in tabulist) 1 neighbours are:


solution is:
[[2, 3, -1], [5, 3, -1], [2, 2, 2], [8, 5, 0], [1, 1, 1]]
value is 11
```

and finally its output is

```
iteration 3:
------------------------

and the current solution is:
[[2, 3, -1], [5, 3, -1], [2, 2, 2], [8, 5, 0], [1, 1, 1]]
for this solution, the total value is: 11

update the tabu list
and now the list is :

[[2, 3, -1], [5, 3, -1], [2, 2, 2], [8, 5, 0], [1, 1, 1]]
[[2, 3, -1], [5, 3, -1], [2, 2, 2], [8, 5, 0], [1, 1, -1]]

and all it's feasible(enough capacity and also not in tabulist) 0 neighbours are:


there are no solution for next iteration. so we end the loop

****************
<-algorithm end

([[2, 3, -1], [5, 3, -1], [2, 2, 2], [8, 5, 0], [1, 1, 1]], 11)
```