assignment1

Individual report for bth004

Name:	YuKi
Student ID:	xxxxxxxxxxxxxx
Date: :	December 4, 2023

Content

1 Greedy Algorithm

1.1 Implementation

In this question, an object-oriented approach is adopted to construct items and knapsack into *Item* class and *Knapsack* class respectively.

Then proceed as follows:

1. Get the item list and knapsack list, encapsulate them into instances, and store them in the *Item* list and *Knapsack* list respectively.
2. For *Item* list, the unit weight value of each item is calculated and stored.
3. Sort the *Item* list according to unit weight value from largest to smallest.
4. Sort the *Knapsack* list according to the remaining capacity of the knapsack from small to large.
5. Each step, put the item with the largest unit weight value into the knapsack with the smallest remaining capacity. After placing it, repeat 4 (reorder the *Knapsack* list).
6. Repeat step 5 until all knapsacks are full and no more items can be placed in knapsacks.

1.2 Pseudo Code

Algorithm 1 Greedy Algorithm

Require: Value of n items, Weight of n items, Capacity of m knapsacks
Ensure: Maximum value sum Z, Item placement status

class Item:
  properties: sequenceNo, value, weight, benefit
class Knapsack:
  properties: sequenceNo, capacity, residualCapacity, items

items_list = [Item(i+1,value,weight,$\frac{value}{weight}$) for i in range(n)]
knapsacks_list = [Knapsack(i+1,capacity) for i in range(m)]

call *sort_by_benefit_desc*(*items_list*)
for *item* in *items_list* **do**
  call *sort_by_residual_capacity_asc*(*knapsacks_list*)
  for *knapsack* in *knapsacks_list* **do**
    if *item* not in *knapsack* && *knapsack.residualCapacity* $\geq$ *item.weight* **then**
      put *item* in *knapsack*
    end if
  end for
end for
Output Maximum value sum Z, Item placement status *knapsacks_list*

1.3 Verify Correctness

Verify correctness using test case

1.3.1 Test Data 1

Input:

```
------test data------
Knapsack 1: Capacity = 7  Residual capacity = 7  Items = []
Knapsack 2: Capacity = 8  Residual capacity = 8  Items = []
Item 1: Value = 1.5  Weight = 5  Value per Weight = 0.30
Item 2: Value = 3.0  Weight = 4  Value per Weight = 0.75
Item 3: Value = 4.0  Weight = 2  Value per Weight = 2.00
Item 4: Value = 2.5  Weight = 3  Value per Weight = 0.83
Item 5: Value = 5.0  Weight = 5  Value per Weight = 1.00
Item 6: Value = 7.5  Weight = 5  Value per Weight = 1.50
```

Output:

```
------result------
Knapsack 1: Capacity = 7  Residual capacity = 0  Items = [3, 6]
Knapsack 2: Capacity = 8  Residual capacity = 0  Items = [5, 4]
Z = 19.0
```

1.3.2 Test Data 2

Input:

```
------test data------
Knapsack 1: Capacity = 10  Residual capacity = 10  Items = []
Knapsack 2: Capacity = 2  Residual capacity = 2  Items = []
Knapsack 3: Capacity = 1  Residual capacity = 1  Items = []
Knapsack 4: Capacity = 6  Residual capacity = 6  Items = []
Item 1: Value = 3  Weight = 4  Value per Weight = 0.75
Item 2: Value = 4  Weight = 5  Value per Weight = 0.80
Item 3: Value = 5  Weight = 6  Value per Weight = 0.83
Item 4: Value = 5  Weight = 7  Value per Weight = 0.71
Item 5: Value = 2  Weight = 1  Value per Weight = 2.00
Item 6: Value = 3  Weight = 3  Value per Weight = 1.00
```

Output:

```
------result------
Knapsack 1: Capacity = 10  Residual capacity = 0  Items = [3, 1]
Knapsack 2: Capacity = 2  Residual capacity = 2  Items = []
Knapsack 3: Capacity = 1  Residual capacity = 0  Items = [5]
Knapsack 4: Capacity = 6  Residual capacity = 3  Items = [6]
Z = 13
```

1.3.3 Test Data 3

Input:

```
------test data------
Knapsack 1: Capacity = 10  Residual capacity = 10  Items = []
Knapsack 2: Capacity = 7  Residual capacity = 7  Items = []
Knapsack 3: Capacity = 8  Residual capacity = 8  Items = []
Knapsack 4: Capacity = 15  Residual capacity = 15  Items = []
Knapsack 5: Capacity = 16  Residual capacity = 16  Items = []
Item 1: Value = 5  Weight = 4  Value per Weight = 1.25
Item 2: Value = 6  Weight = 5  Value per Weight = 1.20
Item 3: Value = 8  Weight = 7  Value per Weight = 1.14
Item 4: Value = 2  Weight = 2  Value per Weight = 1.00
Item 5: Value = 4  Weight = 3  Value per Weight = 1.33
Item 6: Value = 9  Weight = 8  Value per Weight = 1.12
Item 7: Value = 7  Weight = 6  Value per Weight = 1.17
Item 8: Value = 3  Weight = 2  Value per Weight = 1.50
Item 9: Value = 1  Weight = 1  Value per Weight = 1.00
Item 10: Value = 6  Weight = 5  Value per Weight = 1.20
Item 11: Value = 5  Weight = 4  Value per Weight = 1.25
Item 12: Value = 10  Weight = 9  Value per Weight = 1.11
Item 13: Value = 3  Weight = 2  Value per Weight = 1.50
Item 14: Value = 7  Weight = 6  Value per Weight = 1.17
Item 15: Value = 4  Weight = 3  Value per Weight = 1.33
Item 16: Value = 8  Weight = 7  Value per Weight = 1.14
Item 17: Value = 6  Weight = 5  Value per Weight = 1.20
Item 18: Value = 5  Weight = 4  Value per Weight = 1.25
```

Output:

```
------result------
Knapsack 1: Capacity = 10  Residual capacity = 0  Items = [11, 18, 4]
Knapsack 2: Capacity = 7  Residual capacity = 0  Items = [8, 13, 5]
Knapsack 3: Capacity = 8  Residual capacity = 0  Items = [15, 1, 9]
Knapsack 4: Capacity = 15  Residual capacity = 0  Items = [2, 10, 17]
Knapsack 5: Capacity = 16  Residual capacity = 4  Items = [7, 14]
Z = 64
```

1.4 Python Code

1.4.1 knapsack_greedy.py

Code for implementing a greedy algorithm to solve the knapsack problem

```python
from TestDataGenerator import TestDataGreedy

# class: items to put in snapsack
# sequenceNo: serial number of the item
# value: value of item
# weight: weight of item
# benefit: value per weight unit of item
```

```python
class Item:
    def __init__(self, sequenceNo, value, weight):
        self.sequenceNo = sequenceNo
        self.value = value
        self.weight = weight
        self.benefit = value / weight if weight != 0 else 0

    def __str__(self):
        return f"Item {self.sequenceNo}: Value = {self.value} Weight = {self.weight} Value per Weight = {self.benefit:.2f}"

# class: knapsack which get in items
# sequenceNo: serial number of the knapsack
# capacity: capacity of knapsack
# residualCapacity: Residual capacity of knapsack
# items: items to put in the knapsack
class Knapsack:
    def __init__(self, sequenceNo, capacity):
        self.sequenceNo = sequenceNo
        self.capacity = capacity
        self.residualCapacity = self.capacity
        self.items = []

    def __str__(self):
        return f"Knapsack {self.sequenceNo}: Capacity = {self.capacity} Residual capacity = {self.residualCapacity} Items = {self.items}"

    # function: put the item putted in the knapsack
    def put_itemIn(self,item):
        self.residualCapacity = self.residualCapacity - item.weight
        self.items.append(item.sequenceNo)

# funciton: sort items_list based on Value per Weight
def sort_itemBenefit(items_list):
    items_list.sort(key=lambda x: x.benefit, reverse=True)
    return items_list

# funciton: sort knapsacks_list based on residualCapacity
def sort_knapsackResidualCapacity(knapsacks_list):
    knapsacks_list.sort(key=lambda knapsack: knapsack.residualCapacity)
    return knapsacks_list

# function: use a greedy algorithm to find the overall maximum value and item placement
# KNAPSACK_NUMBER: number of knapsacks
# knapsacks_list: knapsacks list
# ITEM_NUMBER: number of items
```

```python
# items_list: items list
def greedy_knapsack(KNAPSACK_NUMBER,knapsacks_list,ITEM_NUMBER,items_list):
    Z_valueSum = 0

    if KNAPSACK_NUMBER != len(knapsacks_list) and ITEM_NUMBER != len(items_list):
        print("knapsack or item quantity is wrong")
        return None
    else:
        sort_itemBenefit(items_list)
        for item in items_list:
            sort_knapsackResidualCapacity(knapsacks_list)
            for knapsack in knapsacks_list:
                if item.weight <= knapsack.residualCapacity:
                    knapsack.put_itemIn(item)
                    Z_valueSum = Z_valueSum + item.value
                    break

        return Z_valueSum

# function: receive test data and instantiate it
def creat_testData():
    KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list =
        TestDataGreedy.creat_testData1()

    # create knapsacks
    knapsacks_list = []
    for i in range(KNAPSACK_NUMBER):
        knapsack = Knapsack(i + 1, knapsacks_capacity_list[i])
        knapsacks_list.append(knapsack)

    # create items
    items_list = []
    for i in range(ITEM_NUMBER):
        item = Item(i + 1, item_value_list[i], item_weight_list[i])
        items_list.append(item)

    return KNAPSACK_NUMBER, knapsacks_list, ITEM_NUMBER, items_list

def main():
    # target
    Z_valueSum = 0

    # test
    print("------test data------")
    KNAPSACK_NUMBER,knapsacks_list,ITEM_NUMBER,items_list = creat_testData()
    for knapsack in knapsacks_list:
```

```
            print(knapsack)
        for item in items_list:
            print(item)

        print("------result------")
        Z_valueSum = greedy_knapsack(KNAPSACK_NUMBER,knapsacks_list,ITEM_NUMBER,items_list)
        knapsacks_list.sort(key=lambda knapsack: knapsack.sequenceNo)
        for knapsack in knapsacks_list:
            print(knapsack)

        print(f"Z = {Z_valueSum}")

if __name__ == "__main__":
    main()
```

1.4.2 TestDataGenerator.py

Code used to generate test data

```
import random

# class: create test data for greedy
class TestDataGreedy:
    # function: create test data 1
    @staticmethod
    def creat_testData1():
        # create knapsack
        KNAPSACK_NUMBER = 2
        knapsacks_capacity_list = [7,8]

        # create items
        ITEM_NUMBER = 6
        item_value_list = [1.5, 3.0, 4.0, 2.5, 5.0, 7.5]
        item_weight_list = [5, 4, 2, 3, 5, 5]

        return KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list

    # function: create test data 2
    @staticmethod
    def creat_testData2():
        # create knapsack
        KNAPSACK_NUMBER = 4
        knapsacks_capacity_list = [12, 11, 8, 7]

        # create items
```

```
        ITEM_NUMBER = 10
        item_value_list = [8, 30, 12, 5, 6, 11, 2, 2, 1, 12]
        item_weight_list = [4, 10, 7, 3, 4, 9, 8, 10, 9, 11]

        return
            KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list

    @staticmethod
    def creat_testData3():
        # create knapsack
        KNAPSACK_NUMBER = 4
        knapsacks_capacity_list = [10,2,1,6]

        # create items
        ITEM_NUMBER = 6
        item_value_list = [3,4,5,5,2,3]
        item_weight_list = [4,5,6,7,1,3]

        return
            KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list

    @staticmethod
    def creat_testData4():
        # create knapsack
        KNAPSACK_NUMBER = 5
        knapsacks_capacity_list = [2,6,8,17,7]

        # create items
        ITEM_NUMBER = 9
        item_value_list = [8,25,6,34,11,42,10,33,15]
        item_weight_list = [5,1,6,9,2,3,8,4,5]

        return
            KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list

    @staticmethod
    def creat_testData5():
        # create knapsack
        KNAPSACK_NUMBER = 5
        knapsacks_capacity_list = [10, 7, 8, 15,16]

        # create items
        ITEM_NUMBER = 18
        item_value_list = [5, 6, 8, 2, 4, 9, 7, 3, 1, 6, 5, 10, 3, 7, 4, 8, 6, 5]
        item_weight_list = [4, 5, 7, 2, 3, 8, 6, 2, 1, 5, 4, 9, 2, 6, 3, 7, 5, 4]

```

```
        return
            KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list

    # function: create test data with random values
    def
        creat_testDataRandom(KNAPSACK_NUMBER,ITEM_NUMBER,RANDOM_WEIGHT_BASE,RANDOM_VALUE_BASE):
        RANDOM_WEIGHT_BASE_HALF = RANDOM_WEIGHT_BASE//2
        RANDOM_WEIGHT_BASE_2TIMES = RANDOM_WEIGHT_BASE*2
        RANDOM_WEIGHT_BASE_3TIMES = RANDOM_WEIGHT_BASE*3
        RANDOM_VALUE_BASE_3TIMES = RANDOM_VALUE_BASE*3
        # create knapsacks
        knapsacks_capacity_list = []
        for i in range(1, KNAPSACK_NUMBER + 1):
            capacity = random.randint(RANDOM_WEIGHT_BASE, RANDOM_WEIGHT_BASE_3TIMES)
            knapsacks_capacity_list.append(capacity)

        # create items
        item_value_list = []
        item_weight_list = []
        for i in range(1, ITEM_NUMBER + 1):
            value = random.randint(RANDOM_VALUE_BASE, RANDOM_VALUE_BASE_3TIMES)
            weight = random.randint(RANDOM_WEIGHT_BASE_HALF, RANDOM_WEIGHT_BASE_2TIMES)
            item_value_list.append(value)
            item_weight_list.append(weight)

        return
            KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list
```

2 neighbourhood Search Algorithm

2.1 Implementation

In this question, we first obtain an initial solution, and express and store it in the form of a matrix. Its row(ith) is the knapsack serial number(i+1), and its column(jth) is the item serial number(j+1).

Then we stipulate that solution[i][j]=1 means that item j is placed in knapsack i. Otherwise, solution[i][j]=0 means that item j is not placed in knapsack i.

Then For a solution $\hat{x}$, define a neighbourhood $N(\hat{x})$:

1. Put an item j that is not in knapsack into knapsack i
2. If the capacity of knapsack i is not enough, move item jj (weight greater than item j) form knapsack i to the other knapsack ii.
3. If the capacity of knapsack ii is not enough, move the item jjj form knapsack ii out so that the item jj can be moved into knapsack ii.

Then we use the initial solution to enter loop:

1. Perform small modifications on the solution to get the neighbour in the neighbourhood.
2. To evaluate all neighbours and select the best one.
3. If the best neighbour is better than the current solution, it will be used as the solution for the next iteration. If it is not better than the current solution, break out of the loop.

The solution after the loop ends is the optimal solution obtained by the neighbourhood search algorithm.

2.2 Pseudo Code

Algorithm 2 neighbourhood search algorithm

Require: Value of n items, Weight of n items, Capacity of m knapsacks, initial *solution*
Ensure: Maximum value sum Z, optimal *solution*

while No optimal *solution* found **do**
    $Z \leftarrow$ sum of items' values of *solution*
    $neighbourhood \leftarrow getNeighbourhood(solution)$
    $Z\_bestNeighbour, bestNeighbour \leftarrow evaluate(neighbourhood)$
    if $Z\_bestNeighbour \geq Z$ **then**
        $solution \leftarrow bestNeighbour$
    else
        break
    end if
end while
Output Maximum value sum Z, optimal *solution*

2.3 Verify Correctness

Verify correctness using test case. The three sets of test cases are the same as in the previous *1.3 Verify Correctness*

2.3.1 Test Data 1

Input:

```
------test data------
Knapsack 1: capacity = 7.0
Knapsack 2: capacity = 8.0
Item 1: value = 1.5, weight = 5.0
Item 2: value = 3.0, weight = 4.0
Item 3: value = 4.0, weight = 2.0
Item 4: value = 2.5, weight = 3.0
Item 5: value = 5.0, weight = 5.0
Item 6: value = 7.5, weight = 5.0
```

Output:

```
------solution------
Knapsack 1: value = 9.0  Residual capacity = 0.0   Items = [3, 5]
Knapsack 2: value = 10.0  Residual capacity = 0.0   Items = [4, 6]
Z = 19.0
```

2.3.2 Test Data 2

Input:

```
------test data------
Knapsack 1: capacity = 10.0
Knapsack 2: capacity = 2.0
Knapsack 3: capacity = 1.0
Knapsack 4: capacity = 6.0
Item 1: value = 3.0, weight = 4.0
Item 2: value = 4.0, weight = 5.0
Item 3: value = 5.0, weight = 6.0
Item 4: value = 5.0, weight = 7.0
Item 5: value = 2.0, weight = 1.0
Item 6: value = 3.0, weight = 3.0
```

Output:

```
------solution------
Knapsack 1: value = 10.0  Residual capacity = 0.0   Items = [3, 5, 6]
Knapsack 2: value = 0.0  Residual capacity = 2.0   Items = []
Knapsack 3: value = 0.0  Residual capacity = 1.0   Items = []
Knapsack 4: value = 4.0  Residual capacity = 1.0   Items = [2]
Z = 14.0
```

2.3.3 Test Data 3

Input:

```
------test data------
Knapsack 1: capacity = 10.0
Knapsack 2: capacity = 7.0
Knapsack 3: capacity = 8.0
Knapsack 4: capacity = 15.0
Knapsack 5: capacity = 16.0
Item 1: value = 5.0, weight = 4.0
Item 2: value = 6.0, weight = 5.0
Item 3: value = 8.0, weight = 7.0
Item 4: value = 2.0, weight = 2.0
Item 5: value = 4.0, weight = 3.0
Item 6: value = 9.0, weight = 8.0
Item 7: value = 7.0, weight = 6.0
Item 8: value = 3.0, weight = 2.0
Item 9: value = 1.0, weight = 1.0
Item 10: value = 6.0, weight = 5.0
Item 11: value = 5.0, weight = 4.0
Item 12: value = 10.0, weight = 9.0
Item 13: value = 3.0, weight = 2.0
Item 14: value = 7.0, weight = 6.0
Item 15: value = 4.0, weight = 3.0
Item 16: value = 8.0, weight = 7.0
Item 17: value = 6.0, weight = 5.0
Item 18: value = 5.0, weight = 4.0
```

Output:

```
------solution------
Knapsack 1: value = 13.0  Residual capacity = 0.0   Items = [8, 11, 18]
Knapsack 2: value = 9.0  Residual capacity = 0.0   Items = [1, 5]
Knapsack 3: value = 9.0  Residual capacity = 0.0   Items = [6]
Knapsack 4: value = 18.0  Residual capacity = 0.0   Items = [2, 10, 17]
Knapsack 5: value = 18.0  Residual capacity = 0.0   Items = [3, 12]
Z = 67.0
```

2.4 Python Code

2.4.1 knapsack_neigbour.py

Code for implementing a neighbourhood search algorithm to solve the knapsack problem

```python
import torch
from typing import Tuple
from TestDataGenerator import TestDataNeighbour

# function: to find the situation where items placed in which knapsack or not
# matrix: current items' situation matrix in snapsacks
def find_placedKnapsacks(matrix):
```

```
    placedKnapsacks = [0] * matrix.size(1)
    for i in range(matrix.size(0)):
        for j in range(matrix.size(1)):
            if matrix[i][j] == 1:
                placedKnapsacks[j] = i + 1
    return placedKnapsacks

# function: use neighbor search algorithm to find the overall maximum value and item placement
# startSolution_matrix: current items’ situation matrix in snapsacks
# KNAPSACK_NUMBER: number of knapsacks
# knapsacks_tensor: knapsacks capacity tensor
# ITEM_NUMBER: number of items
# items_matrix: items values and weights matrix
def neighbourSearch_knapsack(startSolution_matrix:torch.Tensor,KNAPSACK_NUMBER:int,knapsacks_tensor:torch.Tens
    if KNAPSACK_NUMBER == knapsacks_tensor.size()[0] == startSolution_matrix.size()[0] and ITEM_NUMBER == items_matrix.size()[1] == startSolution_matrix.size()[1] :
        Z_valueSum = torch.mm(startSolution_matrix, items_matrix.t())[:, 0].sum()
        placedKnapsacks = find_placedKnapsacks(startSolution_matrix)
        noBiggerNeighbor_flag = False
        nextSolution_matrix = startSolution_matrix.clone()
        nextSolution_placedKnapsacks = placedKnapsacks.copy()
        neighbour_valueSum_max = Z_valueSum.clone()

        # find solution which has maximum value sum
        while noBiggerNeighbor_flag == False:
            noBiggerNeighbor_flag = True
            nowSolution_matrix = nextSolution_matrix.clone()
            nowSolution_placedKnapsacks = nextSolution_placedKnapsacks.copy()
            nowSolution_valueSum = neighbour_valueSum_max.clone()

            # calculate the value sum of neighbours
            if not torch.cuda.is_available():
                raise RuntimeError("cuda not available when calculate the value sum of neighbours")
            neighbour_valueSum_max = torch.tensor(0.0).to(’cuda’)
            for j in range(ITEM_NUMBER):
                for i in range(KNAPSACK_NUMBER):

                    # item j is not put into any knapsack
                    if nowSolution_placedKnapsacks[j] == 0:

                        # current weight of knapsack i
                        weight_knapsackI = torch.matmul(nowSolution_matrix[i], items_matrix[1])
```

```
capacity_knapsackI = knapsacks_tensor[i]
value_itemJ = items_matrix[0][j]
weight_itemJ = items_matrix[1][j]

# put item j into knapsack i
if weight_knapsackI + weight_itemJ <= capacity_knapsackI:
    neighbour_valueSum = nowSolution_valueSum + value_itemJ

    # get the maximum value sum of neighbours
    if neighbour_valueSum > neighbour_valueSum_max:
        nextSolution_matrix = nowSolution_matrix.clone()
        nextSolution_matrix[i][j] = 1.0
        nextSolution_placedKnapsacks =
            nowSolution_placedKnapsacks.copy()
        nextSolution_placedKnapsacks[j] = i+1
        neighbour_valueSum_max = neighbour_valueSum
else:
    # move item jj from knapsack i into knapsack ii
    for jj in range(ITEM_NUMBER):
        if nowSolution_matrix[i][jj] == 1.0 and (weight_knapsackI +
            weight_itemJ - items_matrix[1][jj]) <= capacity_knapsackI:
            for ii in range(KNAPSACK_NUMBER):

                # current weight of knapsack ii
                weight_knapsackII =
                    torch.matmul(nowSolution_matrix[ii],
                    items_matrix[1])
                capacity_knapsackII = knapsacks_tensor[ii]
                weight_itemJJ = items_matrix[1][jj]

                # put item jj into knapsack ii
                if weight_knapsackII + weight_itemJJ <
                    capacity_knapsackII:
                    neighbour_valueSum = nowSolution_valueSum +
                        value_itemJ

                    # get the maximum value sum of neighbours
                    if neighbour_valueSum > neighbour_valueSum_max:
                        nextSolution_matrix = nowSolution_matrix.clone()
                        nextSolution_matrix[i][j] = 1.0
                        nextSolution_matrix[i][jj] = 0.0
                        nextSolution_matrix[ii][jj] = 1.0
                        nextSolution_placedKnapsacks =
                            nowSolution_placedKnapsacks.copy()
                        nextSolution_placedKnapsacks[j] = i+1
                        nextSolution_placedKnapsacks[jj] = ii+1
```

```python
                                                neighbour_valueSum_max = neighbour_valueSum
                                        else:
                                            # move item jjj from knapsack ii out
                                            for jjj in range(ITEM_NUMBER):
                                                if nowSolution_matrix[ii][jjj] == 1.0 and (weight_knapsackII + weight_itemJJ - items_matrix[1][jjj]) <= capacity_knapsackII :
                                                    neighbour_valueSum = nowSolution_valueSum + value_itemJ - items_matrix[0][jjj]

                                                    # get the maximum value sum of neighbours
                                                    if neighbour_valueSum > neighbour_valueSum_max:
                                                        nextSolution_matrix = nowSolution_matrix.clone()
                                                        nextSolution_matrix[i][j] = 1.0
                                                        nextSolution_matrix[i][jj] = 0.0
                                                        nextSolution_matrix[ii][jj] = 1.0
                                                        nextSolution_matrix[ii][jjj] = 0.0
                                                        nextSolution_placedKnapsacks = nowSolution_placedKnapsacks.copy()
                                                        nextSolution_placedKnapsacks[j] = i+1
                                                        nextSolution_placedKnapsacks[jj] = ii+1
                                                        nextSolution_placedKnapsacks[jjj] = 0
                                                        neighbour_valueSum_max = neighbour_valueSum

            # next step
            if neighbour_valueSum_max > nowSolution_valueSum:
                noBiggerNeighbor_flag = False

        # return neighbour_valueSum_max,nextSolution_matrix
        return nowSolution_valueSum,nowSolution_matrix

    else:
        print(f"KNAPSACK_NUMBER = {KNAPSACK_NUMBER},\nknapsacks_tensor = {knapsacks_tensor.size()[0]},\nstartSolution_matrix.size()[0] = {startSolution_matrix.size()[0]}")
        print(f"ITEM_NUMBER = {ITEM_NUMBER},\nitems_matrix.size()[1] = {items_matrix.size()[1]},\nstartSolution_matrix.size()[1] = {startSolution_matrix.size()[1]}")
        raise RuntimeError("knapsack or item quantity is wrong")

def main():
    # target
```

```
    Z_valueSum = 0

    # test
    print("------test data------")
    test_data_generator = TestDataNeighbour()
    startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix = test_data_generator.creat_testData5()
    for i in range(KNAPSACK_NUMBER):
        print(f"Knapsack {i+1}: capacity = {knapsacks_tensor[i]}")
    for i in range(ITEM_NUMBER):
        print(f"Item {i+1}: value = {items_matrix[0][i]}, weight = {items_matrix[1][i]}")

    print("------solution------")
    Z_valueSum,solution = neighbourSearch_knapsack(startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix
    solution_knapsacks = torch.mm(solution, items_matrix.t())
    for i in range(KNAPSACK_NUMBER):
        items_knapsackI = []
        for j in range(ITEM_NUMBER):
            if solution[i][j] != 0.0 :
                items_knapsackI.append(j+1)
        print(f"Knapsack {i+1}: value = {solution_knapsacks[i][0]} Residual capacity = {knapsacks_tensor[i]-solution_knapsacks[i][1]} Items = {items_knapsackI}")
    print(f"Z = {Z_valueSum.item()}")

if __name__ == "__main__":
    main()
```

2.4.2 TestDataGenerator.py

Code used to generate test data

```
import random
import torch

# class: create test data for greedy
class TestDataGreedy:
    # function: create test data 1
    @staticmethod
    def creat_testData1():
        # create knapsack
        KNAPSACK_NUMBER = 2
        knapsacks_capacity_list = [7,8]

        # create items
        ITEM_NUMBER = 6
```

```python
        item_value_list = [1.5, 3.0, 4.0, 2.5, 5.0, 7.5]
        item_weight_list = [5, 4, 2, 3, 5, 5]

        return
            KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list

    # function: create test data 2
    @staticmethod
    def creat_testData2():
        # create knapsack
        KNAPSACK_NUMBER = 4
        knapsacks_capacity_list = [12, 11, 8, 7]

        # create items
        ITEM_NUMBER = 10
        item_value_list = [8, 30, 12, 5, 6, 11, 2, 2, 1, 12]
        item_weight_list = [4, 10, 7, 3, 4, 9, 8, 10, 9, 11]

        return
            KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list

    @staticmethod
    def creat_testData3():
        # create knapsack
        KNAPSACK_NUMBER = 4
        knapsacks_capacity_list = [10,2,1,6]

        # create items
        ITEM_NUMBER = 6
        item_value_list = [3,4,5,5,2,3]
        item_weight_list = [4,5,6,7,1,3]

        return
            KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list

    @staticmethod
    def creat_testData4():
        # create knapsack
        KNAPSACK_NUMBER = 5
        knapsacks_capacity_list = [2,6,8,17,7]

        # create items
        ITEM_NUMBER = 9
        item_value_list = [8,25,6,34,11,42,10,33,15]
        item_weight_list = [5,1,6,9,2,3,8,4,5]

```

```
        return
            KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list

    @staticmethod
    def creat_testData5():
        # create knapsack
        KNAPSACK_NUMBER = 5
        knapsacks_capacity_list = [10, 7, 8, 15,16]

        # create items
        ITEM_NUMBER = 18
        item_value_list = [5, 6, 8, 2, 4, 9, 7, 3, 1, 6, 5, 10, 3, 7, 4, 8, 6, 5]
        item_weight_list = [4, 5, 7, 2, 3, 8, 6, 2, 1, 5, 4, 9, 2, 6, 3, 7, 5, 4]

        return
            KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list

    # function: create test data with random values
    def
        creat_testDataRandom(KNAPSACK_NUMBER,ITEM_NUMBER,RANDOM_WEIGHT_BASE,RANDOM_VALUE_BASE):
        RANDOM_WEIGHT_BASE_HALF = RANDOM_WEIGHT_BASE//2
        RANDOM_WEIGHT_BASE_2TIMES = RANDOM_WEIGHT_BASE*2
        RANDOM_WEIGHT_BASE_3TIMES = RANDOM_WEIGHT_BASE*3
        RANDOM_VALUE_BASE_3TIMES = RANDOM_VALUE_BASE*3
        # create knapsacks
        knapsacks_capacity_list = []
        for i in range(1, KNAPSACK_NUMBER + 1):
            capacity = random.randint(RANDOM_WEIGHT_BASE, RANDOM_WEIGHT_BASE_3TIMES)
            knapsacks_capacity_list.append(capacity)

        # create items
        item_value_list = []
        item_weight_list = []
        for i in range(1, ITEM_NUMBER + 1):
            value = random.randint(RANDOM_VALUE_BASE, RANDOM_VALUE_BASE_3TIMES)
            weight = random.randint(RANDOM_WEIGHT_BASE_HALF, RANDOM_WEIGHT_BASE_2TIMES)
            item_value_list.append(value)
            item_weight_list.append(weight)

        return
            KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list


# class: create test data for neighbour search or tabo search
class TestDataNeighbour:

```

```
    # function: create test data 1
    @staticmethod
    def creat_testData1():
        if not torch.cuda.is_available():
            raise RuntimeError("CUDA not available when creating test data")

        # Create knapsack
        KNAPSACK_NUMBER = 2
        capacities = torch.tensor([7, 8]).to('cuda')
        knapsacks_tensor = capacities.float()

        # Create items
        ITEM_NUMBER = 6
        values = torch.tensor([1.5, 3.0, 4.0, 2.5, 5.0, 7.5]).to('cuda')
        weights = torch.tensor([5, 4, 2, 3, 5, 5]).to('cuda')
        items_matrix = torch.stack((values, weights), dim=0).to('cuda')

        startSolution_matrix = torch.tensor([[0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 0, 1]]).float().to('cuda')

        return startSolution_matrix, KNAPSACK_NUMBER, knapsacks_tensor, ITEM_NUMBER, items_matrix

    # function: create test data 2
    @staticmethod
    def creat_testData2():
        if not torch.cuda.is_available():
            raise RuntimeError("cuda not available when create test data")

        # create knapsack
        KNAPSACK_NUMBER = 4
        capacities = torch.tensor([12, 11, 8, 7]).to('cuda')
        knapsacks_tensor = capacities.float()

        # create items
        ITEM_NUMBER = 10
        values = torch.tensor([8, 30, 12, 5, 6, 11, 2, 2, 1, 12]).to('cuda')
        weights = torch.tensor([4, 10, 7, 3, 4, 9, 8, 10, 9, 11]).to('cuda')
        items_matrix = torch.stack((values,weights), dim=0).float().to('cuda')

        startSolution_matrix = torch.tensor([[0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
                                             [1, 0, 0, 1, 0, 0, 0, 0, 0, 0],
                                             [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
                                             [0, 0, 0, 0, 1 , 0, 0, 0, 0, 0]]).float().to('cuda')

```

```
        return startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix

    # function: create test data 3
    @staticmethod
    def creat_testData3():

        if not torch.cuda.is_available():
            raise RuntimeError("cuda not available when create test data")

        # create knapsack
        KNAPSACK_NUMBER = 4
        capacities = torch.tensor([10,2,1,6]).to('cuda')
        knapsacks_tensor = capacities.float()

        # create items
        ITEM_NUMBER = 6
        values = torch.tensor([3,4,5,5,2,3]).to('cuda')
        weights = torch.tensor([4,5,6,7,1,3]).to('cuda')
        items_matrix = torch.stack((values,weights), dim=0).float().to('cuda')

        startSolution_matrix = torch.tensor([[0, 0, 1, 0, 1, 1],
                                             [0, 0, 0, 0, 0, 0],
                                             [0, 0, 0, 0, 0, 0],
                                             [0, 1, 0, 0, 0, 0]]).float().to('cuda')

        return startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix

    # function: create test data 4
    @staticmethod
    def creat_testData4():

        if not torch.cuda.is_available():
            raise RuntimeError("cuda not available when create test data")

        # create knapsack
        KNAPSACK_NUMBER = 5
        capacities = torch.tensor([2,6,8,17,7]).to('cuda')
        knapsacks_tensor = capacities.float()

        # create items
        ITEM_NUMBER = 9
        values = torch.tensor([8,25,6,34,11,42,10,33,15]).to('cuda')
        weights = torch.tensor([5,1,6,9,2,3,8,4,5]).to('cuda')
        items_matrix = torch.stack((values,weights), dim=0).float().to('cuda')
```

```

        startSolution_matrix = torch.tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0],
                                             [0, 0, 0, 0, 0, 0, 0, 0, 0],
                                             [0, 0, 0, 0, 0, 0, 0, 0, 0],
                                             [0, 0, 0, 0, 0, 0, 0, 0, 0],
                                             [0, 0, 0, 0, 0, 0, 0, 0, 0]]).float().to('cuda')

        return startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix

    @staticmethod
    def creat_testData5():

        if not torch.cuda.is_available():
            raise RuntimeError("cuda not available when create test data")

        # create knapsack
        KNAPSACK_NUMBER = 5
        capacities = torch.tensor([10, 7, 8, 15,16]).to('cuda')
        knapsacks_tensor = capacities.float()

        # create items
        ITEM_NUMBER = 18
        values = torch.tensor([5, 6, 8, 2, 4, 9, 7, 3, 1, 6, 5, 10, 3, 7, 4, 8, 6, 5]).to('cuda')
        weights = torch.tensor([4, 5, 7, 2, 3, 8, 6, 2, 1, 5, 4, 9, 2, 6, 3, 7, 5, 4]).to('cuda')
        items_matrix = torch.stack((values,weights), dim=0).float().to('cuda')

        startSolution_matrix = torch.tensor([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1],
                                             [0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                                             [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                                             [0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
                                             [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]).float().to('cuda')

        return startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix

    @staticmethod
    def creat_testDataRandom(KNAPSACK_NUMBER,ITEM_NUMBER,RANDOM_WEIGHT_BASE,RANDOM_VALUE_BASE):
```

```python
        RANDOM_WEIGHT_BASE_HALF = RANDOM_WEIGHT_BASE//2
        RANDOM_WEIGHT_BASE_2TIMES = RANDOM_WEIGHT_BASE*2
        RANDOM_WEIGHT_BASE_3TIMES = RANDOM_WEIGHT_BASE*3
        RANDOM_VALUE_BASE_3TIMES = RANDOM_VALUE_BASE*3
        if not torch.cuda.is_available():
            raise RuntimeError("cuda not available when create test data")

        # create knapsack
        capacities = torch.randint(low=RANDOM_WEIGHT_BASE, high=RANDOM_WEIGHT_BASE_3TIMES,
            size=(KNAPSACK_NUMBER,)).to('cuda')
        knapsacks_tensor = capacities.float()

        # create items
        values = torch.randint(low=RANDOM_VALUE_BASE, high=RANDOM_VALUE_BASE_3TIMES,
            size=(ITEM_NUMBER,)).to('cuda')
        weights = torch.randint(low=RANDOM_WEIGHT_BASE_HALF,
            high=RANDOM_WEIGHT_BASE_2TIMES, size=(ITEM_NUMBER,)).to('cuda')
        items_matrix = torch.stack((values,weights), dim=0).float().to('cuda')

        startSolution_matrix = torch.zeros((KNAPSACK_NUMBER, ITEM_NUMBER)).to('cuda')

        return
            startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix
```

3 Tabu Search Algorithm

3.1 Implementation

The tabu-search algorithm, which are based on *neighbourhood search algorithm*, creates a k-length tabu list that holds k most recent solutions which are tabu for subsequent search. And if the tabu list is filled, then tabu list will be remove the first solution.

So in this question, its neighbourhood definition is exactly the same as *neighbourhood search algorithm*. While, we take the maximum size of the neighbourhood as the k.

For never getting stuck in a locally optimal solution, we need to record the globally optimal solution. Then we use the initial solution to enter loop:

1. Update tabu list.
2. Perform small modifications on the solution to get the neighbour in the neighbourhood.
3. To evaluate all neighbours which are not in tabu list and select the best one.
4. If the best neighbour is better than the globally optimal solution and, it will be saved as the globally optimal solution.
5. Use the best neighbour as the solution for the next iteration.
6. If the best neighbour found in step 2 do not exist, that is, all neighbours are all in the tabu list, break out of the loop.

The globally optimal solution after the loop ends is the optimal solution obtained by the tabu search algorithm.

3.2 Pseudo Code

Algorithm 3 Tabu search algorithm

Require: Value of n items, Weight of n items, Capacity of m knapsacks, initial *solution*
Ensure: Maximum value sum Z, globally optimal *solution_globally_optimal*

```
function updateTabuList(tabuList, solution)
    if tabuList not filled then
        remove tabuList[0]
    end if
    tabuList.append(solution)
function evaluate(neighbourhood)
    for neighbour in neighbourhood
        if neighbour not in tabuList then
            Z_Neighbour ← sum of items' values of neighbour
            if Z_Neighbour ≥ Z_bestNeighbour then
                Z_bestNeighbour ← Z_Neighbour
                bestNeighbour ← neighbour
            end if
        end if
    end for
    return Z_bestNeighbour, bestNeighbour

Z ← sum of items' values of solution
k ← n^2 m
tabuList ← [   ]
while True do
    call updateTabuList(tabuList, solution)
    neighbourhood ← getNeighbourhood(solution)
    Z_bestNeighbour, bestNeighbour ← evaluate(neighbourhood)
    if Z_bestNeighbour, bestNeighbour exist then
        if Z_bestNeighbour ≥ Z then
            Z ← Z_bestNeighbour
            solution_globally_optimal ← bestNeighbour
        end if
        solution ← bestNeighbour
    else
        break
    end if
end while

Output Maximum value sum Z, globally optimal solution_globally_optimal
```

3.3 Verify Correctness

Verify correctness using test case. The three sets of test cases are the same as in the previous *1.3 Verify Correctness*

3.3.1 Test Data 1

Input:

```
------test data------
Knapsack 1: capacity = 7.0
Knapsack 2: capacity = 8.0
Item 1: value = 1.5, weight = 5.0
Item 2: value = 3.0, weight = 4.0
Item 3: value = 4.0, weight = 2.0
Item 4: value = 2.5, weight = 3.0
Item 5: value = 5.0, weight = 5.0
Item 6: value = 7.5, weight = 5.0
```

Output:

```
------solution------
Knapsack 1: value = 9.0  Residual capacity = 0.0   Items = [3, 5]
Knapsack 2: value = 10.0  Residual capacity = 0.0   Items = [4, 6]
Z = 19.0
```

3.3.2 Test Data 2

Input:

```
------test data------
Knapsack 1: capacity = 10.0
Knapsack 2: capacity = 2.0
Knapsack 3: capacity = 1.0
Knapsack 4: capacity = 6.0
Item 1: value = 3.0, weight = 4.0
Item 2: value = 4.0, weight = 5.0
Item 3: value = 5.0, weight = 6.0
Item 4: value = 5.0, weight = 7.0
Item 5: value = 2.0, weight = 1.0
Item 6: value = 3.0, weight = 3.0
```

Output:

```
------solution------
Knapsack 1: value = 10.0  Residual capacity = 0.0   Items = [3, 5, 6]
Knapsack 2: value = 0.0  Residual capacity = 2.0   Items = []
Knapsack 3: value = 0.0  Residual capacity = 1.0   Items = []
Knapsack 4: value = 4.0  Residual capacity = 1.0   Items = [2]
Z = 14.0
```

3.3.3 Test Data 3

Input:

```
------test data------
Knapsack 1: capacity = 10.0
Knapsack 2: capacity = 7.0
Knapsack 3: capacity = 8.0
Knapsack 4: capacity = 15.0
Knapsack 5: capacity = 16.0
Item 1: value = 5.0, weight = 4.0
Item 2: value = 6.0, weight = 5.0
Item 3: value = 8.0, weight = 7.0
Item 4: value = 2.0, weight = 2.0
Item 5: value = 4.0, weight = 3.0
Item 6: value = 9.0, weight = 8.0
Item 7: value = 7.0, weight = 6.0
Item 8: value = 3.0, weight = 2.0
Item 9: value = 1.0, weight = 1.0
Item 10: value = 6.0, weight = 5.0
Item 11: value = 5.0, weight = 4.0
Item 12: value = 10.0, weight = 9.0
Item 13: value = 3.0, weight = 2.0
Item 14: value = 7.0, weight = 6.0
Item 15: value = 4.0, weight = 3.0
Item 16: value = 8.0, weight = 7.0
Item 17: value = 6.0, weight = 5.0
Item 18: value = 5.0, weight = 4.0
```

Output:

```
------solution------
Knapsack 1: value = 13.0  Residual capacity = 0.0   Items = [8, 11, 18]
Knapsack 2: value = 9.0  Residual capacity = 0.0   Items = [1, 5]
Knapsack 3: value = 9.0  Residual capacity = 0.0   Items = [6]
Knapsack 4: value = 18.0  Residual capacity = 0.0   Items = [2, 10, 17]
Knapsack 5: value = 18.0  Residual capacity = 0.0   Items = [3, 12]
Z = 67.0
```

3.4 Python Code

3.4.1 knapsack_tabu.py

Code for implementing a tabu search algorithm to solve the knapsack problem

```python
import torch
from typing import Tuple
from TestDataGenerator import TestDataNeighbour

# function: to find the situation where items placed in which knapsack or not
# matrix: current item's situation matrix in snapsacks
def find_placedKnapsacks(matrix):
```

```
    placedKnapsacks = [0] * matrix.size(1)
    for i in range(matrix.size(0)):
        for j in range(matrix.size(1)):
            if matrix[i][j] == 1:
                placedKnapsacks[j] = i + 1
    return placedKnapsacks

# function: use tabo search algorithm to find the overall maximum value and item placement
# TABOLIST_LEN: The length of tabo list,store placedKnapsacks
# startSolution_matrix: current item's situation matrix in snapsacks
# KNAPSACK_NUMBER: number of knapsacks
# knapsacks_tensor: knapsacks capacity tensor
# ITEM_NUMBER: number of items
# items_matrix: items values and weights matrix
def
     taboSearch_knapsack(TABOLIST_LEN:list,startSolution_matrix:torch.Tensor,KNAPSACK_NUMBER:int,knapsacks_tens
    if KNAPSACK_NUMBER == knapsacks_tensor.size()[0] == startSolution_matrix.size()[0] and
         ITEM_NUMBER == items_matrix.size()[1] == startSolution_matrix.size()[1] :
        Z_valueSum = torch.mm(startSolution_matrix, items_matrix.t())[:, 0].sum()
        placedKnapsacks = find_placedKnapsacks(startSolution_matrix)
        tabo_list = []
        noBiggerNeighbor_flag = False
        nextSolution_matrix = startSolution_matrix.clone()
        nextSolution_placedKnapsacks = placedKnapsacks.copy()
        neighbour_valueSum_max = Z_valueSum.clone()

        # find solution which has maximum value sum
        while noBiggerNeighbor_flag == False:
            noBiggerNeighbor_flag = True
            nowSolution_matrix = nextSolution_matrix.clone()
            nowSolution_placedKnapsacks = nextSolution_placedKnapsacks.copy()
            nowSolution_valueSum = neighbour_valueSum_max.clone()
            if len(tabo_list) == TABOLIST_LEN:
                tabo_list.pop(0)
            tabo_list.append(nowSolution_placedKnapsacks)

            # calculate the value sum of neighbours
            if not torch.cuda.is_available():
                raise RuntimeError("cuda not available when calculate the value sum of
                     neighbours")
            neighbour_valueSum_max = torch.tensor(0.0).to('cuda')
            for j in range(ITEM_NUMBER):
                for i in range(KNAPSACK_NUMBER):

                    # item j is not put into any knapsack
                    if nowSolution_placedKnapsacks[j] == 0:
```

```

# current weight of knapsack i
weight_knapsackI = torch.matmul(nowSolution_matrix[i], items_matrix[1])
capacity_knapsackI = knapsacks_tensor[i]
value_itemJ = items_matrix[0][j]
weight_itemJ = items_matrix[1][j]

# put item j into knapsack i
if weight_knapsackI + weight_itemJ <= capacity_knapsackI:
    neighbour_valueSum = nowSolution_valueSum + value_itemJ

    # get the maximum value sum of neighbours
    if neighbour_valueSum > neighbour_valueSum_max:
        neighbour_placedKnapsacks = nowSolution_placedKnapsacks.copy()
        neighbour_placedKnapsacks[j] = i+1

        #  check if it is in tabo list
        if neighbour_placedKnapsacks not in tabo_list:
            nextSolution_matrix = nowSolution_matrix.clone()
            nextSolution_matrix[i][j] = 1.0
            nextSolution_placedKnapsacks = nowSolution_placedKnapsacks.copy()
            nextSolution_placedKnapsacks[j] = i+1
            neighbour_valueSum_max = neighbour_valueSum
else:
    # move item jj from knapsack i into knapsack ii
    for jj in range(ITEM_NUMBER):
        if nowSolution_matrix[i][jj] == 1.0 and (weight_knapsackI + weight_itemJ - items_matrix[1][jj]) <= capacity_knapsackI:
            for ii in range(KNAPSACK_NUMBER):

                # current weight of knapsack ii
                weight_knapsackII = torch.matmul(nowSolution_matrix[ii], items_matrix[1])
                capacity_knapsackII = knapsacks_tensor[ii]
                weight_itemJJ = items_matrix[1][jj]

                # put item jj into knapsack ii
                if weight_knapsackII + weight_itemJJ < capacity_knapsackII:
                    neighbour_valueSum = nowSolution_valueSum + value_itemJ

                    # get the maximum value sum of neighbours
```

```
                    if neighbour_valueSum > neighbour_valueSum_max:
                        neighbour_placedKnapsacks = nowSolution_placedKnapsacks.copy()
                        neighbour_placedKnapsacks[j] = i+1
                        neighbour_placedKnapsacks[jj] = ii+1
                        if neighbour_placedKnapsacks not in tabo_list:
                            nextSolution_matrix = nowSolution_matrix.clone()
                            nextSolution_matrix[i][j] = 1.0
                            nextSolution_matrix[i][jj] = 0.0
                            nextSolution_matrix[ii][jj] = 1.0
                            nextSolution_placedKnapsacks = nowSolution_placedKnapsacks.copy()
                            nextSolution_placedKnapsacks[j] = i+1
                            nextSolution_placedKnapsacks[jj] = ii+1
                            neighbour_valueSum_max = neighbour_valueSum
                else:
                    # move item jjj from knapsack ii out
                    for jjj in range(ITEM_NUMBER):
                        if nowSolution_matrix[ii][jjj] == 1.0 and (weight_knapsackII + weight_itemJJ - items_matrix[1][jjj]) <= capacity_knapsackII :
                            neighbour_valueSum = nowSolution_valueSum + value_itemJ - items_matrix[0][jjj]

                            # get the maximum value sum of neighbours
                            if neighbour_valueSum > neighbour_valueSum_max:
                                neighbour_placedKnapsacks = nowSolution_placedKnapsacks.copy()
                                neighbour_placedKnapsacks[j] = i+1
                                neighbour_placedKnapsacks[jj] = ii+1
                                if neighbour_placedKnapsacks not in tabo_list:
                                    nextSolution_matrix = nowSolution_matrix.clone()
                                    nextSolution_matrix[i][j] = 1.0
                                    nextSolution_matrix[i][jj] = 0.0
                                    nextSolution_matrix[ii][jj] = 1.0
                                    nextSolution_matrix[ii][jjj] = 0.0
                                    nextSolution_placedKnapsacks = nowSolution_placedKnapsacks.copy()
                                    nextSolution_placedKnapsacks[j] = i+1
                                    nextSolution_placedKnapsacks[jj] = ii+1
```

```
                                                            nextSolution_placedKnapsacks[jjj] = 0
                                                            neighbour_valueSum_max = neighbour_valueSum

            # next step
            if neighbour_valueSum_max > nowSolution_valueSum:
                noBiggerNeighbor_flag = False

        # return neighbour_valueSum_max,nextSolution_matrix
        return nowSolution_valueSum,nowSolution_matrix

    else:
        print(f"KNAPSACK_NUMBER = {KNAPSACK_NUMBER},\nknapsacks_tensor = {knapsacks_tensor.size()[0]},\nstartSolution_matrix.size()[0] = {startSolution_matrix.size()[0]}")
        print(f"ITEM_NUMBER = {ITEM_NUMBER},\nitems_matrix.size()[1] = {items_matrix.size()[1]},\nstartSolution_matrix.size()[1] = {startSolution_matrix.size()[1]}")
        raise RuntimeError("knapsack or item quantity is wrong")

def main():
    # target
    Z_valueSum = 0

    # test
    print("------test data------")
    test_data_generator = TestDataNeighbour()
    startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix = test_data_generator.creat_testData1()
    for i in range(KNAPSACK_NUMBER):
        print(f"Knapsack {i+1}: capacity = {knapsacks_tensor[i]}")
    for i in range(ITEM_NUMBER):
        print(f"Item {i+1}: value = {items_matrix[0][i]}, weight = {items_matrix[1][i]}")
    print("------solution------")
    TABOLIST_LEN = ITEM_NUMBER*ITEM_NUMBER*KNAPSACK_NUMBER
    Z_valueSum,solution = taboSearch_knapsack(TABOLIST_LEN,startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,item
    # print(f"Solution:\n{solution}")
    solution_knapsacks = torch.mm(solution, items_matrix.t())
    for i in range(KNAPSACK_NUMBER):
        items_knapsackI = []
        for j in range(ITEM_NUMBER):
            if solution[i][j] != 0.0 :
                items_knapsackI.append(j+1)
        print(f"Knapsack {i+1}: value = {solution_knapsacks[i][0]} Residual capacity = {knapsacks_tensor[i]-solution_knapsacks[i][1]} Items = {items_knapsackI}")
```

```python

    print(f"Z = {Z_valueSum.item()}")
if __name__ == "__main__":
    main()
```

3.4.2 TestDataGenerator.py

Code used to generate test data, it is exactly the same as the code in *2.4.2 TestDataGenerator*, so it is omitted