

assignment1

Individual report for bth004

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

Content

1 Greedy Algorithm	3
1.1 Implementation	3
1.2 Pseudo Code	3
1.3 Verify Correctness	4
1.3.1 Test Data 1	4
1.3.2 Test Data 2	4
1.3.3 Test Data 3	5
1.4 Python Code	5
1.4.1 knapsack_greedy.py	5
1.4.2 TestDataGenerator.py	8
2 neighbourhood Search Algorithm	11
2.1 Implementation	11
2.2 Pseudo Code	11
2.3 Verify Correctness	12
2.3.1 Test Data 1	12
2.3.2 Test Data 2	12
2.3.3 Test Data 3	13
2.4 Python Code	13
2.4.1 knapsack_neighbour.py	13
2.4.2 TestDataGenerator.py	17
3 Tabu Search Algorithm	24
3.1 Implementation	24
3.2 Pseudo Code	25
3.3 Verify Correctness	26
3.3.1 Test Data 1	26
3.3.2 Test Data 2	26
3.3.3 Test Data 3	27
3.4 Python Code	27
3.4.1 knapsack_tabu.py	27
3.4.2 TestDataGenerator.py	32

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

```
1: class Item:
2:   properties: sequenceNo, value, weight, benefit
3: class Knapsack:
4:   properties: sequenceNo, capacity, residualCapacity, items
5:
6:  $items\_list = [Item(i+1, value, weight, \frac{value}{weight}) \text{ for } i \text{ in range}(n)]$ 
7:  $knapsacks\_list = [Knapsack(i+1, capacity) \text{ for } i \text{ in range}(m)]$ 
8:
9: call  $sort\_by\_benefit\_desc(items\_list)$ 
10: for  $item$  in  $items\_list$  do
11:   call  $sort\_by\_residual\_capacity\_asc(knapsacks\_list)$ 
12:   for  $knapsack$  in  $knapsacks\_list$  do
13:     if  $item$  not in  $knapsack$  &&  $knapsack.residualCapacity \geq item.weight$  then
14:       put  $item$  in  $knapsack$ 
15:     end if
16:   end for
17: end for
18: 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

```
1 from TestDataGenerator import TestDataGreedy
2
3 # class: items to put in snapsack
4 # sequenceNo: serial number of the item
5 # value: value of item
6 # weight: weight of item
7 # benefit: value per weight unit of item
```

```

8 class Item:
9     def __init__(self, sequenceNo, value, weight):
10         self.sequenceNo = sequenceNo
11         self.value = value
12         self.weight = weight
13         self.benefit = value / weight if weight != 0 else 0
14
15     def __str__(self):
16         return f"Item {self.sequenceNo}: Value = {self.value} Weight = {self.weight} Value
17             per Weight = {self.benefit:.2f}"
18
19 # class: knapsack which get in items
20 # sequenceNo: serial number of the knapsack
21 # capacity: capacity of knapsack
22 # residualCapacity: Residual capacity of knapsack
23 # items: items to put in the knapsack
24 class Knapsack:
25     def __init__(self, sequenceNo, capacity):
26         self.sequenceNo = sequenceNo
27         self.capacity = capacity
28         self.residualCapacity = self.capacity
29         self.items = []
30
31     def __str__(self):
32         return f"Knapsack {self.sequenceNo}: Capacity = {self.capacity} Residual capacity =
33             {self.residualCapacity} Items = {self.items}"
34
35 # function: put the item putted in the knapsack
36 def put_itemIn(self, item):
37     self.residualCapacity = self.residualCapacity - item.weight
38     self.items.append(item.sequenceNo)
39
40 # function: sort items_list based on Value per Weight
41 def sort_itemBenefit(items_list):
42     items_list.sort(key=lambda x: x.benefit, reverse=True)
43     return items_list
44
45 # function: sort knapsacks_list based on residualCapacity
46 def sort_knapsackResidualCapacity(knapsacks_list):
47     knapsacks_list.sort(key=lambda knapsack: knapsack.residualCapacity)
48     return knapsacks_list
49
50 # function: use a greedy algorithm to find the overall maximum value and item placement
51 # KNAPSACK_NUMBER: number of knapsacks
52 # knapsacks_list: knapsacks list
53 # ITEM_NUMBER: number of items

```

```

52 # items_list: items list
53 def greedy_knapsack(KNAPSACK_NUMBER,knapsacks_list,ITEM_NUMBER,items_list):
54     Z_valueSum = 0
55
56     if KNAPSACK_NUMBER != len(knapsacks_list) and ITEM_NUMBER != len(items_list):
57         print("knapsack or item quantity is wrong")
58         return None
59     else:
60         sort_itemBenefit(items_list)
61         for item in items_list:
62             sort_knapsackResidualCapacity(knapsacks_list)
63             for knapsack in knapsacks_list:
64                 if item.weight <= knapsack.residualCapacity:
65                     knapsack.put_itemIn(item)
66                     Z_valueSum = Z_valueSum + item.value
67                     break
68
69         return Z_valueSum
70
71 # function: receive test data and instantiate it
72 def creat_testData():
73     KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list =
74         TestDataGreedy.creat_testData1()
75
76     # create knapsacks
77     knapsacks_list = []
78     for i in range(KNAPSACK_NUMBER):
79         knapsack = Knapsack(i + 1, knapsacks_capacity_list[i])
80         knapsacks_list.append(knapsack)
81
82     # create items
83     items_list = []
84     for i in range(ITEM_NUMBER):
85         item = Item(i + 1, item_value_list[i], item_weight_list[i])
86         items_list.append(item)
87
88     return KNAPSACK_NUMBER, knapsacks_list, ITEM_NUMBER, items_list
89
90 def main():
91     # target
92     Z_valueSum = 0
93
94     # test
95     print("-----test data-----")
96     KNAPSACK_NUMBER,knapsacks_list,ITEM_NUMBER,items_list = creat_testData()
97     for knapsack in knapsacks_list:

```

```

97         print(knapsack)
98     for item in items_list:
99         print(item)
100
101     print("-----result-----")
102     Z_valueSum = greedy_knapsack(KNAPSACK_NUMBER,knapsacks_list,ITEM_NUMBER,items_list)
103     knapsacks_list.sort(key=lambda knapsack: knapsack.sequenceNo)
104     for knapsack in knapsacks_list:
105         print(knapsack)
106
107     print(f"Z = {Z_valueSum}")
108
109 if __name__ == "__main__":
110     main()

```

1.4.2 TestDataGenerator.py

Code used to generate test data

```

1  import random
2
3  # class: create test data for greedy
4  class TestDataGreedy:
5      # function: create test data 1
6      @staticmethod
7      def creat_testData1():
8          # create knapsack
9          KNAPSACK_NUMBER = 2
10         knapsacks_capacity_list = [7,8]
11
12         # create items
13         ITEM_NUMBER = 6
14         item_value_list = [1.5, 3.0, 4.0, 2.5, 5.0, 7.5]
15         item_weight_list = [5, 4, 2, 3, 5, 5]
16
17         return
18         KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list
19
20     # function: create test data 2
21     @staticmethod
22     def creat_testData2():
23         # create knapsack
24         KNAPSACK_NUMBER = 4
25         knapsacks_capacity_list = [12, 11, 8, 7]
26
27         # create items

```



```

27     ITEM_NUMBER = 10
28     item_value_list = [8, 30, 12, 5, 6, 11, 2, 2, 1, 12]
29     item_weight_list = [4, 10, 7, 3, 4, 9, 8, 10, 9, 11]
30
31     return
        KNAPSACK_NUMBER, knapsacks_capacity_list, ITEM_NUMBER, item_value_list, item_weight_list
32
33     @staticmethod
34     def creat_testData3():
35         # create knapsack
36         KNAPSACK_NUMBER = 4
37         knapsacks_capacity_list = [10, 2, 1, 6]
38
39         # create items
40         ITEM_NUMBER = 6
41         item_value_list = [3, 4, 5, 5, 2, 3]
42         item_weight_list = [4, 5, 6, 7, 1, 3]
43
44     return
        KNAPSACK_NUMBER, knapsacks_capacity_list, ITEM_NUMBER, item_value_list, item_weight_list
45
46     @staticmethod
47     def creat_testData4():
48         # create knapsack
49         KNAPSACK_NUMBER = 5
50         knapsacks_capacity_list = [2, 6, 8, 17, 7]
51
52         # create items
53         ITEM_NUMBER = 9
54         item_value_list = [8, 25, 6, 34, 11, 42, 10, 33, 15]
55         item_weight_list = [5, 1, 6, 9, 2, 3, 8, 4, 5]
56
57     return
        KNAPSACK_NUMBER, knapsacks_capacity_list, ITEM_NUMBER, item_value_list, item_weight_list
58
59     @staticmethod
60     def creat_testData5():
61         # create knapsack
62         KNAPSACK_NUMBER = 5
63         knapsacks_capacity_list = [10, 7, 8, 15, 16]
64
65         # create items
66         ITEM_NUMBER = 18
67         item_value_list = [5, 6, 8, 2, 4, 9, 7, 3, 1, 6, 5, 10, 3, 7, 4, 8, 6, 5]
68         item_weight_list = [4, 5, 7, 2, 3, 8, 6, 2, 1, 5, 4, 9, 2, 6, 3, 7, 5, 4]
69

```

```

70     return
       KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list
71
72     # function: create test data with random values
73     def
       creat_testDataRandom(KNAPSACK_NUMBER,ITEM_NUMBER,RANDOM_WEIGHT_BASE,RANDOM_VALUE_BASE):
74     RANDOM_WEIGHT_BASE_HALF = RANDOM_WEIGHT_BASE//2
75     RANDOM_WEIGHT_BASE_2TIMES = RANDOM_WEIGHT_BASE*2
76     RANDOM_WEIGHT_BASE_3TIMES = RANDOM_WEIGHT_BASE*3
77     RANDOM_VALUE_BASE_3TIMES = RANDOM_VALUE_BASE*3
78     # create knapsacks
79     knapsacks_capacity_list = []
80     for i in range(1, KNAPSACK_NUMBER + 1):
81         capacity = random.randint(RANDOM_WEIGHT_BASE, RANDOM_WEIGHT_BASE_3TIMES)
82         knapsacks_capacity_list.append(capacity)
83
84     # create items
85     item_value_list = []
86     item_weight_list = []
87     for i in range(1, ITEM_NUMBER + 1):
88         value = random.randint(RANDOM_VALUE_BASE, RANDOM_VALUE_BASE_3TIMES)
89         weight = random.randint(RANDOM_WEIGHT_BASE_HALF, RANDOM_WEIGHT_BASE_2TIMES)
90         item_value_list.append(value)
91         item_weight_list.append(weight)
92
93     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*) from knapsack *i* to the other knapsack *ii*.
3. If the capacity of knapsack *ii* is not enough, move the item *jjj* from 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*

```
1: while No optimal solution found do
2:    $Z \leftarrow$  sum of items' values of solution
3:    $neighbourhood \leftarrow getNeighbourhood(solution)$ 
4:    $Z\_bestNeighbour, bestNeighbour \leftarrow evaluate(neighbourhood)$ 
5:   if  $Z\_bestNeighbour \geq Z$  then
6:      $solution \leftarrow bestNeighbour$ 
7:   else
8:     break
9:   end if
10: end while
11: 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_neighbour.py

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

```
1 import torch
2 from typing import Tuple
3 from TestDataGenerator import TestDataNeighbour
4
5 # function: to find the situation where items placed in which knapsack or not
6 # matrix: current items' situation matrix in snapsacks
7 def find_placedKnapsacks(matrix):
```

```

8     placedKnapsacks = [0] * matrix.size(1)
9     for i in range(matrix.size(0)):
10         for j in range(matrix.size(1)):
11             if matrix[i][j] == 1:
12                 placedKnapsacks[j] = i + 1
13     return placedKnapsacks
14
15 # function: use neighbor search algorithm to find the overall maximum value and item
16 # placement
17 # startSolution_matrix: current items' situation matrix in snapsacks
18 # KNAPSACK_NUMBER: number of knapsacks
19 # knapsacks_tensor: knapsacks capacity tensor
20 # ITEM_NUMBER: number of items
21 # items_matrix: items values and weights matrix
22 def
23     neighbourSearch_knapsack(startSolution_matrix:torch.Tensor,KNAPSACK_NUMBER:int,knapsacks_tensor:torch.Tensor)
24     if KNAPSACK_NUMBER == knapsacks_tensor.size()[0] == startSolution_matrix.size()[0] and
25         ITEM_NUMBER == items_matrix.size()[1] == startSolution_matrix.size()[1] :
26         Z_valueSum = torch.mm(startSolution_matrix, items_matrix.t())[:, 0].sum()
27         placedKnapsacks = find_placedKnapsacks(startSolution_matrix)
28         noBiggerNeighbor_flag = False
29         nextSolution_matrix = startSolution_matrix.clone()
30         nextSolution_placedKnapsacks = placedKnapsacks.copy()
31         neighbour_valueSum_max = Z_valueSum.clone()
32
33     # find solution which has maximum value sum
34     while noBiggerNeighbor_flag == False:
35         noBiggerNeighbor_flag = True
36         nowSolution_matrix = nextSolution_matrix.clone()
37         nowSolution_placedKnapsacks = nextSolution_placedKnapsacks.copy()
38         nowSolution_valueSum = neighbour_valueSum_max.clone()
39
40     # calculate the value sum of neighbours
41     if not torch.cuda.is_available():
42         raise RuntimeError("cuda not available when calculate the value sum of
43             neighbours")
44     neighbour_valueSum_max = torch.tensor(0.0).to('cuda')
45     for j in range(ITEM_NUMBER):
46         for i in range(KNAPSACK_NUMBER):
47
48             # item j is not put into any knapsack
49             if nowSolution_placedKnapsacks[j] == 0:
50
51                 # current weight of knapsack i
52                 weight_knapsackI = torch.matmul(nowSolution_matrix[i],
53                     items_matrix[1])

```

```

49     capacity_knapsackI = knapsacks_tensor[i]
50     value_itemJ = items_matrix[0][j]
51     weight_itemJ = items_matrix[1][j]
52
53     # put item j into knapsack i
54     if weight_knapsackI + weight_itemJ <= capacity_knapsackI:
55         neighbour_valueSum = nowSolution_valueSum + value_itemJ
56
57         # get the maximum value sum of neighbours
58         if neighbour_valueSum > neighbour_valueSum_max:
59             nextSolution_matrix = nowSolution_matrix.clone()
60             nextSolution_matrix[i][j] = 1.0
61             nextSolution_placedKnapsacks =
62                 nowSolution_placedKnapsacks.copy()
63             nextSolution_placedKnapsacks[j] = i+1
64             neighbour_valueSum_max = neighbour_valueSum
65         else:
66             # move item jj from knapsack i into knapsack ii
67             for jj in range(ITEM_NUMBER):
68                 if nowSolution_matrix[i][jj] == 1.0 and (weight_knapsackI +
69                     weight_itemJ - items_matrix[1][jj]) <= capacity_knapsackI:
70                     for ii in range(KNAPSACK_NUMBER):
71
72                         # current weight of knapsack ii
73                         weight_knapsackII =
74                             torch.matmul(nowSolution_matrix[ii],
75                                 items_matrix[1])
76                         capacity_knapsackII = knapsacks_tensor[ii]
77                         weight_itemJJ = items_matrix[1][jj]
78
79                         # put item jj into knapsack ii
80                         if weight_knapsackII + weight_itemJJ <
81                             capacity_knapsackII:
82                             neighbour_valueSum = nowSolution_valueSum +
83                                 value_itemJ
84
85                             # get the maximum value sum of neighbours
86                             if neighbour_valueSum > neighbour_valueSum_max:
87                                 nextSolution_matrix = nowSolution_matrix.clone()
88                                 nextSolution_matrix[i][j] = 1.0
89                                 nextSolution_matrix[i][jj] = 0.0
90                                 nextSolution_matrix[ii][jj] = 1.0
91                                 nextSolution_placedKnapsacks =
92                                     nowSolution_placedKnapsacks.copy()
93                                 nextSolution_placedKnapsacks[j] = i+1
94                                 nextSolution_placedKnapsacks[jj] = ii+1

```

```

88             neighbour_valueSum_max = neighbour_valueSum
89         else:
90             # move item jjj from knapsack ii out
91             for jjj in range(ITEM_NUMBER):
92                 if nowSolution_matrix[ii][jjj] == 1.0 and
93                     (weight_knapsackII + weight_itemJJ -
94                      items_matrix[1][jjj]) <=
95                     capacity_knapsackII :
96                     neighbour_valueSum = nowSolution_valueSum +
97                         value_itemJ - items_matrix[0][jjj]
98
99                     # get the maximum value sum of neighbours
100                     if neighbour_valueSum >
101                         neighbour_valueSum_max:
102                         nextSolution_matrix =
103                             nowSolution_matrix.clone()
104                         nextSolution_matrix[i][j] = 1.0
105                         nextSolution_matrix[i][jj] = 0.0
106                         nextSolution_matrix[ii][jj] = 1.0
107                         nextSolution_matrix[ii][jjj] = 0.0
108                         nextSolution_placedKnapsacks =
109                             nowSolution_placedKnapsacks.copy()
110                         nextSolution_placedKnapsacks[j] = i+1
111                         nextSolution_placedKnapsacks[jj] = ii+1
112                         nextSolution_placedKnapsacks[jjj] = 0
113                         neighbour_valueSum_max =
114                             neighbour_valueSum
115
116             # next step
117             if neighbour_valueSum_max > nowSolution_valueSum:
118                 noBiggerNeighbor_flag = False
119
120             # return neighbour_valueSum_max,nextSolution_matrix
121             return nowSolution_valueSum,nowSolution_matrix
122
123     else:
124         print(f"KNAPSACK_NUMBER = {KNAPSACK_NUMBER},\nknapsacks_tensor =
125             {knapsacks_tensor.size()[0]},\nstartSolution_matrix.size()[0] =
126             {startSolution_matrix.size()[0]}")
127         print(f"ITEM_NUMBER = {ITEM_NUMBER},\nitems_matrix.size()[1] =
128             {items_matrix.size()[1]},\nstartSolution_matrix.size()[1] =
129             {startSolution_matrix.size()[1]}")
130         raise RuntimeError("knapsack or item quantity is wrong")
131
132 def main():
133     # target

```



```

122     Z_valueSum = 0
123
124     # test
125     print("-----test data-----")
126     test_data_generator = TestDataNeighbour()
127     startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix =
        test_data_generator.creat_testData5()
128     for i in range(KNAPSACK_NUMBER):
129         print(f"Knapsack {i+1}: capacity = {knapsacks_tensor[i]}")
130     for i in range(ITEM_NUMBER):
131         print(f"Item {i+1}: value = {items_matrix[0][i]}, weight = {items_matrix[1][i]}")
132
133     print("-----solution-----")
134     Z_valueSum,solution =
        neighbourSearch_knapsack(startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix)
135     solution_knapsacks = torch.mm(solution, items_matrix.t())
136     for i in range(KNAPSACK_NUMBER):
137         items_knapsackI = []
138         for j in range(ITEM_NUMBER):
139             if solution[i][j] != 0.0 :
140                 items_knapsackI.append(j+1)
141         print(f"Knapsack {i+1}: value = {solution_knapsacks[i][0]} Residual capacity =
            {knapsacks_tensor[i]-solution_knapsacks[i][1]} Items = {items_knapsackI}")
142     print(f"Z = {Z_valueSum.item()}")
143
144 if __name__ == "__main__":
145     main()

```

2.4.2 TestDataGenerator.py

Code used to generate test data

```

1  import random
2  import torch
3
4  # class: create test data for greedy
5  class TestDataGreedy:
6      # function: create test data 1
7      @staticmethod
8      def creat_testData1():
9          # create knapsack
10         KNAPSACK_NUMBER = 2
11         knapsacks_capacity_list = [7,8]
12
13         # create items
14         ITEM_NUMBER = 6

```

```

15     item_value_list = [1.5, 3.0, 4.0, 2.5, 5.0, 7.5]
16     item_weight_list = [5, 4, 2, 3, 5, 5]
17
18     return
        KNAPSACK_NUMBER, knapsacks_capacity_list, ITEM_NUMBER, item_value_list, item_weight_list
19
20     # function: create test data 2
21     @staticmethod
22     def creat_testData2():
23         # create knapsack
24         KNAPSACK_NUMBER = 4
25         knapsacks_capacity_list = [12, 11, 8, 7]
26
27         # create items
28         ITEM_NUMBER = 10
29         item_value_list = [8, 30, 12, 5, 6, 11, 2, 2, 1, 12]
30         item_weight_list = [4, 10, 7, 3, 4, 9, 8, 10, 9, 11]
31
32     return
        KNAPSACK_NUMBER, knapsacks_capacity_list, ITEM_NUMBER, item_value_list, item_weight_list
33
34     @staticmethod
35     def creat_testData3():
36         # create knapsack
37         KNAPSACK_NUMBER = 4
38         knapsacks_capacity_list = [10, 2, 1, 6]
39
40         # create items
41         ITEM_NUMBER = 6
42         item_value_list = [3, 4, 5, 5, 2, 3]
43         item_weight_list = [4, 5, 6, 7, 1, 3]
44
45     return
        KNAPSACK_NUMBER, knapsacks_capacity_list, ITEM_NUMBER, item_value_list, item_weight_list
46
47     @staticmethod
48     def creat_testData4():
49         # create knapsack
50         KNAPSACK_NUMBER = 5
51         knapsacks_capacity_list = [2, 6, 8, 17, 7]
52
53         # create items
54         ITEM_NUMBER = 9
55         item_value_list = [8, 25, 6, 34, 11, 42, 10, 33, 15]
56         item_weight_list = [5, 1, 6, 9, 2, 3, 8, 4, 5]
57

```

```

58         return
           KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list
59
60     @staticmethod
61     def creat_testData5():
62         # create knapsack
63         KNAPSACK_NUMBER = 5
64         knapsacks_capacity_list = [10, 7, 8, 15,16]
65
66         # create items
67         ITEM_NUMBER = 18
68         item_value_list = [5, 6, 8, 2, 4, 9, 7, 3, 1, 6, 5, 10, 3, 7, 4, 8, 6, 5]
69         item_weight_list = [4, 5, 7, 2, 3, 8, 6, 2, 1, 5, 4, 9, 2, 6, 3, 7, 5, 4]
70
71         return
           KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list
72
73     # function: create test data with random values
74     def
       creat_testDataRandom(KNAPSACK_NUMBER,ITEM_NUMBER,RANDOM_WEIGHT_BASE,RANDOM_VALUE_BASE):
75         RANDOM_WEIGHT_BASE_HALF = RANDOM_WEIGHT_BASE//2
76         RANDOM_WEIGHT_BASE_2TIMES = RANDOM_WEIGHT_BASE*2
77         RANDOM_WEIGHT_BASE_3TIMES = RANDOM_WEIGHT_BASE*3
78         RANDOM_VALUE_BASE_3TIMES = RANDOM_VALUE_BASE*3
79         # create knapsacks
80         knapsacks_capacity_list = []
81         for i in range(1, KNAPSACK_NUMBER + 1):
82             capacity = random.randint(RANDOM_WEIGHT_BASE, RANDOM_WEIGHT_BASE_3TIMES)
83             knapsacks_capacity_list.append(capacity)
84
85         # create items
86         item_value_list = []
87         item_weight_list = []
88         for i in range(1, ITEM_NUMBER + 1):
89             value = random.randint(RANDOM_VALUE_BASE, RANDOM_VALUE_BASE_3TIMES)
90             weight = random.randint(RANDOM_WEIGHT_BASE_HALF, RANDOM_WEIGHT_BASE_2TIMES)
91             item_value_list.append(value)
92             item_weight_list.append(weight)
93
94         return
           KNAPSACK_NUMBER,knapsacks_capacity_list,ITEM_NUMBER,item_value_list,item_weight_list
95
96
97     # class: create test data for neighbour search or tabo search
98     class TestDataNeighbour:
99

```

```

100     # function: create test data 1
101     @staticmethod
102     def creat_testData1():
103         if not torch.cuda.is_available():
104             raise RuntimeError("CUDA not available when creating test data")
105
106         # Create knapsack
107         KNAPSACK_NUMBER = 2
108         capacities = torch.tensor([7, 8]).to('cuda')
109         knapsacks_tensor = capacities.float()
110
111         # Create items
112         ITEM_NUMBER = 6
113         values = torch.tensor([1.5, 3.0, 4.0, 2.5, 5.0, 7.5]).to('cuda')
114         weights = torch.tensor([5, 4, 2, 3, 5, 5]).to('cuda')
115         items_matrix = torch.stack((values, weights), dim=0).to('cuda')
116
117         startSolution_matrix = torch.tensor([[0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 0,
118             1]]).float().to('cuda')
119
120         return startSolution_matrix, KNAPSACK_NUMBER, knapsacks_tensor, ITEM_NUMBER,
121             items_matrix
122
123     # function: create test data 2
124     @staticmethod
125     def creat_testData2():
126         if not torch.cuda.is_available():
127             raise RuntimeError("cuda not available when create test data")
128
129         # create knapsack
130         KNAPSACK_NUMBER = 4
131         capacities = torch.tensor([12, 11, 8, 7]).to('cuda')
132         knapsacks_tensor = capacities.float()
133
134         # create items
135         ITEM_NUMBER = 10
136         values = torch.tensor([8, 30, 12, 5, 6, 11, 2, 2, 1, 12]).to('cuda')
137         weights = torch.tensor([4, 10, 7, 3, 4, 9, 8, 10, 9, 11]).to('cuda')
138         items_matrix = torch.stack((values, weights), dim=0).float().to('cuda')
139
140         startSolution_matrix = torch.tensor([[0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
141             [1, 0, 0, 1, 0, 0, 0, 0, 0, 0],
142             [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
143             [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
144             0]]).float().to('cuda')

```

```

143         return
144         startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix
145
146     # function: create test data 3
147     @staticmethod
148     def creat_testData3():
149
150         if not torch.cuda.is_available():
151             raise RuntimeError("cuda not available when create test data")
152
153         # create knapsack
154         KNAPSACK_NUMBER = 4
155         capacities = torch.tensor([10,2,1,6]).to('cuda')
156         knapsacks_tensor = capacities.float()
157
158         # create items
159         ITEM_NUMBER = 6
160         values = torch.tensor([3,4,5,5,2,3]).to('cuda')
161         weights = torch.tensor([4,5,6,7,1,3]).to('cuda')
162         items_matrix = torch.stack((values,weights), dim=0).float().to('cuda')
163
164         startSolution_matrix = torch.tensor([[0, 0, 1, 0, 1, 1],
165                                             [0, 0, 0, 0, 0, 0],
166                                             [0, 0, 0, 0, 0, 0],
167                                             [0, 1, 0, 0, 0, 0]]).float().to('cuda')
168
169         return
170         startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix
171
172     # function: create test data 4
173     @staticmethod
174     def creat_testData4():
175
176         if not torch.cuda.is_available():
177             raise RuntimeError("cuda not available when create test data")
178
179         # create knapsack
180         KNAPSACK_NUMBER = 5
181         capacities = torch.tensor([2,6,8,17,7]).to('cuda')
182         knapsacks_tensor = capacities.float()
183
184         # create items
185         ITEM_NUMBER = 9
186         values = torch.tensor([8,25,6,34,11,42,10,33,15]).to('cuda')
187         weights = torch.tensor([5,1,6,9,2,3,8,4,5]).to('cuda')
188         items_matrix = torch.stack((values,weights), dim=0).float().to('cuda')

```

```

187
188     startSolution_matrix = torch.tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
189                                         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
190                                         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
191                                         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
192                                         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]).float().to('cuda')
193
194     return
195         startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix
196
197 @staticmethod
198 def creat_testData5():
199
200     if not torch.cuda.is_available():
201         raise RuntimeError("cuda not available when create test data")
202
203     # create knapsack
204     KNAPSACK_NUMBER = 5
205     capacities = torch.tensor([10, 7, 8, 15,16]).to('cuda')
206     knapsacks_tensor = capacities.float()
207
208     # create items
209     ITEM_NUMBER = 18
210     values = torch.tensor([5, 6, 8, 2, 4, 9, 7, 3, 1, 6, 5, 10, 3, 7, 4, 8, 6,
211                             5]).to('cuda')
212     weights = torch.tensor([4, 5, 7, 2, 3, 8, 6, 2, 1, 5, 4, 9, 2, 6, 3, 7, 5,
213                             4]).to('cuda')
214     items_matrix = torch.stack((values,weights), dim=0).float().to('cuda')
215
216     startSolution_matrix = torch.tensor([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
217     0, 0, 1],
218
219                                         [0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
220                                         0, 0],
221
222                                         [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
223                                         0, 0],
224
225                                         [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
226                                         1, 0],
227
228                                         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
229                                         0, 0]]).float().to('cuda')
230
231     return
232         startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix
233
234 @staticmethod
235 def
236     creat_testDataRandom(KNAPSACK_NUMBER,ITEM_NUMBER,RANDOM_WEIGHT_BASE,RANDOM_VALUE_BASE):

```

```

223 RANDOM_WEIGHT_BASE_HALF = RANDOM_WEIGHT_BASE//2
224 RANDOM_WEIGHT_BASE_2TIMES = RANDOM_WEIGHT_BASE*2
225 RANDOM_WEIGHT_BASE_3TIMES = RANDOM_WEIGHT_BASE*3
226 RANDOM_VALUE_BASE_3TIMES = RANDOM_VALUE_BASE*3
227 if not torch.cuda.is_available():
228     raise RuntimeError("cuda not available when create test data")
229
230 # create knapsack
231 capacities = torch.randint(low=RANDOM_WEIGHT_BASE, high=RANDOM_WEIGHT_BASE_3TIMES,
232                             size=(KNAPSACK_NUMBER,)).to('cuda')
233 knapsacks_tensor = capacities.float()
234
235 # create items
236 values = torch.randint(low=RANDOM_VALUE_BASE, high=RANDOM_VALUE_BASE_3TIMES,
237                         size=(ITEM_NUMBER,)).to('cuda')
238 weights = torch.randint(low=RANDOM_WEIGHT_BASE_HALF,
239                          high=RANDOM_WEIGHT_BASE_2TIMES, size=(ITEM_NUMBER,)).to('cuda')
240 items_matrix = torch.stack((values,weights), dim=0).float().to('cuda')
241
242 startSolution_matrix = torch.zeros((KNAPSACK_NUMBER, ITEM_NUMBER)).to('cuda')
243
244 return
245     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*

```

1: function updateTabuList(tabuList, solution)
2:   if tabuList not filled then
3:     remove tabuList[0]
4:   end if
5:   tabuList.append(solution)
6: function evaluate(neighbourhood)
7:   for neighbour in neighbourhood
8:     if neighbour not in tabuList then
9:        $Z\_Neighbour \leftarrow$  sum of items' values of neighbour
10:      if  $Z\_Neighbour \geq Z\_bestNeighbour$  then
11:         $Z\_bestNeighbour \leftarrow Z\_Neighbour$ 
12:         $bestNeighbour \leftarrow neighbour$ 
13:      end if
14:    end if
15:  end for
16:  return  $Z\_bestNeighbour, bestNeighbour$ 
17:
18:  $Z \leftarrow$  sum of items' values of solution
19:  $k \leftarrow n^2m$ 
20: tabuList  $\leftarrow [ ]$ 
21: while True do
22:   call updateTabuList(tabuList, solution)
23:   neighbourhood  $\leftarrow$  getNeighbourhood(solution)
24:    $Z\_bestNeighbour, bestNeighbour \leftarrow$  evaluate(neighbourhood)
25:   if  $Z\_bestNeighbour, bestNeighbour$  exist then
26:     if  $Z\_bestNeighbour \geq Z$  then
27:        $Z \leftarrow Z\_bestNeighbour$ 
28:        $solution\_globally\_optimal \leftarrow bestNeighbour$ 
29:     end if
30:     solution  $\leftarrow bestNeighbour$ 
31:   else
32:     break
33:   end if
34: end while
35:
36: 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

```
1 import torch
2 from typing import Tuple
3 from TestDataGenerator import TestDataNeighbour
4
5 # function: to find the situation where items placed in which knapsack or not
6 # matrix: current item's situation matrix in snapsacks
7 def find_placedKnapsacks(matrix):
```

```

8     placedKnapsacks = [0] * matrix.size(1)
9     for i in range(matrix.size(0)):
10         for j in range(matrix.size(1)):
11             if matrix[i][j] == 1:
12                 placedKnapsacks[j] = i + 1
13     return placedKnapsacks
14
15 # function: use tabo search algorithm to find the overall maximum value and item placement
16 # TABOLIST_LEN: The length of tabo list,store placedKnapsacks
17 # startSolution_matrix: current item's situation matrix in snapsacks
18 # KNAPSACK_NUMBER: number of knapsacks
19 # knapsacks_tensor: knapsacks capacity tensor
20 # ITEM_NUMBER: number of items
21 # items_matrix: items values and weights matrix
22 def
23     taboSearch_knapsack(TABOLIST_LEN:list,startSolution_matrix:torch.Tensor,KNAPSACK_NUMBER:int,knapsacks_tens
24     if KNAPSACK_NUMBER == knapsacks_tensor.size()[0] == startSolution_matrix.size()[0] and
25         ITEM_NUMBER == items_matrix.size()[1] == startSolution_matrix.size()[1] :
26         Z_valueSum = torch.mm(startSolution_matrix, items_matrix.t())[:, 0].sum()
27         placedKnapsacks = find_placedKnapsacks(startSolution_matrix)
28         tabo_list = []
29         noBiggerNeighbor_flag = False
30         nextSolution_matrix = startSolution_matrix.clone()
31         nextSolution_placedKnapsacks = placedKnapsacks.copy()
32         neighbour_valueSum_max = Z_valueSum.clone()
33
34         # find solution which has maximum value sum
35         while noBiggerNeighbor_flag == False:
36             noBiggerNeighbor_flag = True
37             nowSolution_matrix = nextSolution_matrix.clone()
38             nowSolution_placedKnapsacks = nextSolution_placedKnapsacks.copy()
39             nowSolution_valueSum = neighbour_valueSum_max.clone()
40             if len(tabo_list) == TABOLIST_LEN:
41                 tabo_list.pop(0)
42                 tabo_list.append(nowSolution_placedKnapsacks)
43
44             # calculate the value sum of neighbours
45             if not torch.cuda.is_available():
46                 raise RuntimeError("cuda not available when calculate the value sum of
47                     neighbours")
48             neighbour_valueSum_max = torch.tensor(0.0).to('cuda')
49             for j in range(ITEM_NUMBER):
50                 for i in range(KNAPSACK_NUMBER):
51
52                     # item j is not put into any knapsack
53                     if nowSolution_placedKnapsacks[j] == 0:

```

```

51
52     # current weight of knapsack i
53     weight_knapsackI = torch.matmul(nowSolution_matrix[i],
54                                     items_matrix[1])
55     capacity_knapsackI = knapsacks_tensor[i]
56     value_itemJ = items_matrix[0][j]
57     weight_itemJ = items_matrix[1][j]
58
59     # put item j into knapsack i
60     if weight_knapsackI + weight_itemJ <= capacity_knapsackI:
61         neighbour_valueSum = nowSolution_valueSum + value_itemJ
62
63     # get the maximum value sum of neighbours
64     if neighbour_valueSum > neighbour_valueSum_max:
65         neighbour_placedKnapsacks = nowSolution_placedKnapsacks.copy()
66         neighbour_placedKnapsacks[j] = i+1
67
68     # check if it is in tabo list
69     if neighbour_placedKnapsacks not in tabo_list:
70         nextSolution_matrix = nowSolution_matrix.clone()
71         nextSolution_matrix[i][j] = 1.0
72         nextSolution_placedKnapsacks =
73             nowSolution_placedKnapsacks.copy()
74         nextSolution_placedKnapsacks[j] = i+1
75         neighbour_valueSum_max = neighbour_valueSum
76     else:
77         # move item jj from knapsack i into knapsack ii
78         for jj in range(ITEM_NUMBER):
79             if nowSolution_matrix[i][jj] == 1.0 and (weight_knapsackI +
80                 weight_itemJ - items_matrix[1][jj]) <= capacity_knapsackI:
81                 for ii in range(KNAPSACK_NUMBER):
82
83                     # current weight of knapsack ii
84                     weight_knapsackII =
85                         torch.matmul(nowSolution_matrix[ii],
86                                     items_matrix[1])
87                     capacity_knapsackII = knapsacks_tensor[ii]
88                     weight_itemJJ = items_matrix[1][jj]
89
90                     # put item jj into knapsack ii
91                     if weight_knapsackII + weight_itemJJ <
92                         capacity_knapsackII:
93                         neighbour_valueSum = nowSolution_valueSum +
94                             value_itemJ
95
96                     # get the maximum value sum of neighbours

```

```

90         if neighbour_valueSum > neighbour_valueSum_max:
91             neighbour_placedKnapsacks =
92                 nowSolution_placedKnapsacks.copy()
93             neighbour_placedKnapsacks[j] = i+1
94             neighbour_placedKnapsacks[jj] = ii+1
95             if neighbour_placedKnapsacks not in tabo_list:
96                 nextSolution_matrix =
97                     nowSolution_matrix.clone()
98                 nextSolution_matrix[i][j] = 1.0
99                 nextSolution_matrix[i][jj] = 0.0
100                 nextSolution_matrix[ii][jj] = 1.0
101                 nextSolution_placedKnapsacks =
102                     nowSolution_placedKnapsacks.copy()
103                 nextSolution_placedKnapsacks[j] = i+1
104                 nextSolution_placedKnapsacks[jj] = ii+1
105                 neighbour_valueSum_max = neighbour_valueSum
106             else:
107                 # move item jjj from knapsack ii out
108                 for jjj in range(ITEM_NUMBER):
109                     if nowSolution_matrix[ii][jjj] == 1.0 and
110                         (weight_knapsackII + weight_itemJJ -
111                          items_matrix[1][jjj]) <=
112                          capacity_knapsackII :
113                         neighbour_valueSum = nowSolution_valueSum +
114                             value_itemJ - items_matrix[0][jjj]
115
116                 # get the maximum value sum of neighbours
117                 if neighbour_valueSum >
118                     neighbour_valueSum_max:
119                     neighbour_placedKnapsacks =
120                         nowSolution_placedKnapsacks.copy()
121                     neighbour_placedKnapsacks[j] = i+1
122                     neighbour_placedKnapsacks[jj] = ii+1
123                     if neighbour_placedKnapsacks not in
124                         tabo_list:
125                         nextSolution_matrix =
126                             nowSolution_matrix.clone()
127                         nextSolution_matrix[i][j] = 1.0
128                         nextSolution_matrix[i][jj] = 0.0
129                         nextSolution_matrix[ii][jj] = 1.0
130                         nextSolution_matrix[ii][jjj] = 0.0
131                         nextSolution_placedKnapsacks =
132                             nowSolution_placedKnapsacks.copy()
133                         nextSolution_placedKnapsacks[j] = i+1
134                         nextSolution_placedKnapsacks[jj] =
135                             ii+1

```

```

123                                     nextSolution_placedKnapsacks[jjj] = 0
124                                     neighbour_valueSum_max =
                                         neighbour_valueSum
125
126                                     # next step
127                                     if neighbour_valueSum_max > nowSolution_valueSum:
128                                         noBiggerNeighbor_flag = False
129
130                                     # return neighbour_valueSum_max,nextSolution_matrix
131                                     return nowSolution_valueSum,nowSolution_matrix
132
133     else:
134         print(f"KNAPSACK_NUMBER = {KNAPSACK_NUMBER},\nknapsacks_tensor =
                {knapsacks_tensor.size()[0]},\nstartSolution_matrix.size()[0] =
                {startSolution_matrix.size()[0]}")
135         print(f"ITEM_NUMBER = {ITEM_NUMBER},\nitems_matrix.size()[1] =
                {items_matrix.size()[1]},\nstartSolution_matrix.size()[1] =
                {startSolution_matrix.size()[1]}")
136         raise RuntimeError("knapsack or item quantity is wrong")
137
138 def main():
139     # target
140     Z_valueSum = 0
141
142     # test
143     print("-----test data-----")
144     test_data_generator = TestDataNeighbour()
145     startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,items_matrix =
        test_data_generator.creat_testData1()
146     for i in range(KNAPSACK_NUMBER):
147         print(f"Knapsack {i+1}: capacity = {knapsacks_tensor[i]}")
148     for i in range(ITEM_NUMBER):
149         print(f"Item {i+1}: value = {items_matrix[0][i]}, weight = {items_matrix[1][i]}")
150     print("-----solution-----")
151     TABOLIST_LEN = ITEM_NUMBER*ITEM_NUMBER*KNAPSACK_NUMBER
152     Z_valueSum,solution =
        taboSearch_knapsack(TABOLIST_LEN,startSolution_matrix,KNAPSACK_NUMBER,knapsacks_tensor,ITEM_NUMBER,item
153     # print(f"Solution:\n{solution}")
154     solution_knapsacks = torch.mm(solution, items_matrix.t())
155     for i in range(KNAPSACK_NUMBER):
156         items_knapsackI = []
157         for j in range(ITEM_NUMBER):
158             if solution[i][j] != 0.0 :
159                 items_knapsackI.append(j+1)
160         print(f"Knapsack {i+1}: value = {solution_knapsacks[i][0]} Residual capacity =
                {knapsacks_tensor[i]-solution_knapsacks[i][1]} Items = {items_knapsackI}")

```

```
161
162     print(f"Z = {Z_valueSum.item()}")
163 if __name__ == "__main__":
164     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