

assignment1 report



学 院： 计算机科学与技术学院、软件学院
专 业： 软件工程（中外合作）
学生姓名： 李肖
学 号： 202003340111

2022 年 11 月 26 日

1 贪心算法

1.1 贪心算法的实现

在这个题目中，我们先计算每个物品的单位重量价值，然后按照单位重量价值从大到小排序，然后从大到小依次放入背包中，直到背包装满为止。具体代码如下：

1.2 伪代码实现

Algorithm 1 Greedy algorithm

Require: n 个物品, m 个背包容量

Ensure: m 个背包的最大价值

```
1: call sort( $n$  个物品, 根据物品单位重量价值从大到小排序)
2: for  $j \leftarrow 1; j < n; j++$  do
3:   for  $i \leftarrow 1; i < m; i++$  do
4:     if 物品  $i$  没有被放入背包中 && 背包  $j$  剩余容量大于物品  $i$  的重量 then
5:       物品  $i$  放入背包  $j$  中
6:     end if
7:   end for
8: end for
9: Output  $m$  个背包的最大价值
```

1.3 代码实现

```
1  #include <algorithm>
2  #include <iostream>
3
4  const int N = 1e5 + 5;
5
6  struct Object {
7      int p;
8      int w;
9  };
10
11 bool cmp_object(Object a, Object b) {
12     return (double)a.p / (double)a.w > (double)b.p / (double)b.w;
13 }
14
15 int greedy(int n, int m, Object *objects, int *W) {
16     std::sort(objects, objects + n, cmp_object);
17
18     bool x[N];
19     int W_used[N];
20     memset(x, 0, sizeof(x));
21     memset(W_used, 0, sizeof(W_used));
```

```

22     for (int j = 0; j < m; j++) {
23         for (int i = 0; i < n; i++) {
24             if (!x[i] && W[j] - W_used[j] >= objects[i].w) {
25                 x[i] = true;
26                 W_used[j] += objects[i].w;
27             }
28         }
29     }
30
31     int result = 0;
32     for (int i = 0; i < n; i++) {
33         if (x[i]) {
34             result += objects[i].p;
35         }
36     }
37
38     return result;
39 }
40
41 int main() {
42     int n, m;
43     Object objects[N];
44     int W[N];
45
46     // Input user data
47     std::cin >> n >> m;
48     for (int i = 0; i < n; i++) {
49         std::cin >> objects[i].p >> objects[i].w;
50     }
51     for (int i = 0; i < m; i++) {
52         std::cin >> W[i];
53     }
54
55     int result = greedy(n, m, objects, W);
56     std::cout << result << std::endl;
57
58     return 0;
59 }

```

2 领域搜索算法

2.1 领域搜索算法的实现

在这个题目中，我们首先规定一个初始解。在这个题目里面，我们将贪心算法的结果作为初始解

然后我们规定一个输入变量，这里我们定义一个数组 `solution`，`solution[i]` 表示第 i 个物品放入哪个背包中，`solution[i] = -1` 表示第 i 个物品没有放入背包中，`solution[i] = j` 表示第 i 个物品放入了第 j 个背包中。

然后根据这个输入变量，我们可以得到在背包中的物品的总价值。

我们将贪心算法得到的一个 `solution` 数组作为初始解，然后我们对 `solution` 数组进行变异，产生领域。我这里使用了这么几种变异策略，对于所有没有塞入背包的物品 `object`

- 如果一个物品有空位直接塞进去
- 如果一个背包中的一个物品被移除了以后，再把 `object` 塞入能得到更大的价值，就把这个物品移除后再塞入 `object`
- 如果一个背包 `A` 中的一个物品可以被转移到别的背包里，那么就转移这个物品到别的背包后，再塞入 `object`
- 如果一个背包 `A` 中的一个物品 1 可以在背包 `B` 中的一个物品 2 被移除后放入，那么就把物品 2 移除后塞入物品 1，再把 `object` 塞入背包 1

然后我们对所有的领域进行评估，选择最优的领域作为下一步的输入变量，然后重复上述过程，直到有一次评估领域的时候得不到更优解结束。

2.2 伪代码实现

Algorithm 2 neighborhood search algorithm

Require: n 个物品, m 个背包

Ensure: m 个背包的最大价值

```
1: call greedyAlgorithm( $n$  个物品,  $m$  个背包)
2:  $solution \leftarrow getSolution(m$  个背包)
3: while 没有找到最优解 do
4:    $neighborhood \leftarrow getNeighborhood(solution)$ 
5:    $bestNeighbor \leftarrow evaluate(neighborhood)$ 
6:   if  $bestNeighbor$  的价值大于  $solution$  的价值 then
7:      $solution \leftarrow bestNeighbor$ 
8:   else
9:     break
10:  end if
11: end while
```

2.3 代码实现

```
1 #include <algorithm>
2 #include <cmath>
3 #include <cstdio>
4 #include <deque>
5 #include <iostream>
```

```

6  #include <list>
7  #include <map>
8  #include <unistd.h>
9  #include <vector>
10
11 #define in(container, object) \
12     (std::find(container.begin(), container.end(), object) != container.end())
13
14 using std::list;
15 using std::sort;
16 using std::vector;
17
18 const int N = 1e3 + 5;
19 const int INF = 0x3f3f3f3f;
20
21 struct Object {
22     int p;
23     int w;
24     int index;
25 };
26
27 struct Knapsack {
28     int capacity;
29     int cost;
30     int index;
31
32     list<Object> objects;
33 };
34
35 bool operator==(const Object &a, const Object &b) {
36     return a.p == b.p && a.w == b.w;
37 }
38
39 bool cmp_object(Object a, Object b) {
40     return (double)a.p / (double)a.w > (double)b.p / (double)b.w;
41 }
42
43 void output_knapsacks(vector<Knapsack> &knapsacks) {
44     // detail data output
45     std::cout << "-----knapsack info-----" << std::endl;
46     for (auto &knapsack : knapsacks) {
47         std::cout << "Knapsack's index: " << knapsack.index << std::endl;
48         std::cout << "Knapsack's capacity: " << knapsack.capacity << std::endl;
49         std::cout << "Knapsack's cost: " << knapsack.cost << std::endl;
50         for (auto &object : knapsack.objects) {
51             std::cout << object.index << " " << object.p << " " << object.w

```

```

52         << std::endl;
53     }
54 }
55 }
56
57 int calculate_result(vector<Object> &objects, vector<int> solution) {
58     int result = 0;
59
60     for (auto &object : objects) {
61         if (solution[object.index] != -1) {
62             result += object.p;
63         }
64     }
65
66     return result;
67 }
68
69 int greedy(vector<Object> &objects, vector<Knapsack> &knapsacks,
70     vector<bool> &flag) {
71     sort(objects.begin(), objects.end(), cmp_object);
72
73     for (auto &knapsack : knapsacks) {
74         for (auto i = 0; i < objects.size(); i++) {
75             if (!flag[objects[i].index] &&
76                 knapsack.capacity - knapsack.cost >= objects[i].w) {
77                 flag[objects[i].index] = true;
78                 knapsack.cost += objects[i].w;
79                 knapsack.objects.push_back(objects[i]);
80             }
81         }
82     }
83
84     int result = 0;
85
86     for (auto &knapsack : knapsacks) {
87         for (auto &object : knapsack.objects) {
88             result += object.p;
89         }
90     }
91
92     return result;
93 }
94
95 int neighborhood_search(vector<Object> &objects, vector<Knapsack> &knapsacks,
96     vector<bool> &flag) {
97     auto result = greedy(objects, knapsacks, flag);

```

```

98
99 // Using vector to store the solution
100 vector<int> solution(objects.size(), -1); // -1 stands for outside
101 for (auto &knapsack : knapsacks) {
102     for (auto &object : knapsack.objects) {
103         solution[object.index] = knapsack.index;
104     }
105 }
106
107 while (true) {
108     // Generate neighbors
109     vector<vector<int>> neighbors;
110     for (auto i = 0; i < objects.size(); i++) {
111         if (solution[objects[i].index] == -1) {
112             // put the object into knapsack
113             for (auto &knapsack : knapsacks) {
114                 if (knapsack.capacity - knapsack.cost >= objects[i].w) {
115                     auto neighbor = solution;
116                     neighbor[objects[i].index] = knapsack.index;
117                     neighbors.push_back(neighbor);
118                 }
119             }
120             // Find appropriate object in knapsacks to throw out and put the object
121             for (auto &knapsack : knapsacks) {
122                 for (auto &object : knapsack.objects) {
123                     if (object.index != objects[i].index && object.w >= objects[i].w) {
124                         auto neighbor = solution;
125                         neighbor[object.index] = -1;
126                         neighbor[objects[i].index] = knapsack.index;
127                         neighbors.push_back(neighbor);
128                     }
129                 }
130             }
131             // Rotate
132             for (auto ki = 0; ki < knapsacks.size(); ki++) {
133                 for (auto &object1 : knapsacks[ki].objects) {
134                     for (auto kj = 0; kj < knapsacks.size(); kj++) {
135                         auto ki_rest = knapsacks[ki].capacity - knapsacks[ki].cost;
136                         auto kj_rest = knapsacks[kj].capacity - knapsacks[kj].cost;
137                         if (object1.w <= kj_rest && objects[i].w <= ki_rest + object1.w &&
138                             kj != ki) {
139                             auto neighbor = solution;
140                             neighbor[objects[i].index] = knapsacks[ki].index;
141                             neighbor[object1.index] = knapsacks[kj].index;
142                             neighbors.push_back(neighbor);
143                         } else {

```

```

144         for (auto &object2 : knapsacks[kj].objects) { // Throw object2
145             if (object1.w <= kj_rest + object2.w &&
146                 objects[i].w <= ki_rest + object1.w &&
147                 object2.p < objects[i].p && ki != kj) {
148                 auto neighbor = solution;
149                 neighbor[objects[i].index] = knapsacks[ki].index;
150                 neighbor[object1.index] = knapsacks[kj].index;
151                 neighbor[object2.index] = -1;
152                 neighbors.push_back(neighbor);
153             }
154         }
155     }
156 }
157 }
158 }
159 }
160 }
161
162 // Find the best neighbor
163 auto maxn_neighbor_result = -INF;
164 for (auto &neighbor : neighbors) {
165     int neighbor_result = calculate_result(objects, neighbor);
166
167     if (neighbor_result > maxn_neighbor_result) {
168         maxn_neighbor_result = neighbor_result;
169         solution = neighbor;
170     }
171 }
172
173 if (result < maxn_neighbor_result) {
174     result = maxn_neighbor_result;
175 } else {
176     break;
177 }
178
179 // Update knapsacks
180 for (auto &knapsack : knapsacks) {
181     knapsack.objects.clear();
182     knapsack.cost = 0;
183
184     for (auto i = 0; i < objects.size(); i++) {
185         if (solution[objects[i].index] == knapsack.index) {
186             knapsack.objects.push_back(objects[i]);
187             knapsack.cost += objects[i].w;
188         }
189     }

```



```

190
191     if (knapsack.cost > knapsack.capacity) {
192         std::cout << "Error!" << std::endl;
193         output_knapsacks(knapsacks);
194         return -1;
195     }
196 }
197 }
198
199 return result;
200 }
201
202 int main() {
203     int n, m, p, w, W;
204     vector<Object> objects;
205     vector<Knapsack> knapsacks;
206
207     // Input user data
208     std::cin >> n >> m;
209     for (int i = 0; i < n; i++) {
210         std::cin >> p >> w;
211         objects.push_back(Object{p, w, i});
212     }
213     for (int i = 0; i < m; i++) {
214         std::cin >> W;
215         knapsacks.push_back(Knapsack{W, 0, i});
216     }
217
218     vector<bool> flag(objects.size(), false);
219     int result = neighborhood_search(objects, knapsacks, flag);
220     std::cout << result << std::endl;
221
222     return 0;
223 }

```

3 禁忌搜索

3.1 禁忌搜索的实现

禁忌搜索的实现和领域搜索算法的实现类似，只是在领域搜索算法的基础上加入了禁忌表，禁忌表中存储了一些不应该被选择的解。每次选择领域的时候，我们都会检查禁忌表，如果禁忌表中有这个解，那么就不选择这个解。

每次找到局部最优解的时候，就将全局最优解放入禁忌表里。

然后再用局部最优解和当前的全局最优解比较，如果局部最优解的价值更大，那么就更新

全局最优解。

其中禁忌表有一禁忌长度，一旦禁忌表中的元素超出了这个禁忌长度，那么就从禁忌表中删除最开头的元素。

然后就是一旦禁忌表最开头的元素可以赋值给 `solution` 变量，起到一个回溯的作用，因为有的时候会走到一个无论怎么走都得不到全局最优解的地方这个时候我们需要适当的回溯，因为这这个时候禁忌表中还存在这个 `solution` 附近的一个局部最优解，那么这次回溯以后不会找到这个 `solution` 附近的局部最优解，反而能够往别的局部最优解方向跳转，提高搜索到全局最优解的可能行。

3.2 伪代码实现

Algorithm 3 tabu search algorithm

Require: n 个物品, m 个背包

Ensure: m 个背包的最大价值

```
1: call greedyAlgorithm( $n$  个物品,  $m$  个背包)
2:  $solution \leftarrow getSolution(m$  个背包)
3:  $iteration \leftarrow 1000$ 
4:  $tabuList \leftarrow [ ]$ 
5:  $neighborhood \leftarrow [ ]$ 
6:  $tabuLen \leftarrow n/2$ 
7: while 没有找到最优解 do
8:   while 还能生成新的解 do
9:      $neighbor \leftarrow generateNeighbor(solution)$ 
10:    if neighbor not in  $tabuList$  then
11:       $neighborhood \leftarrow neighborhood + neighbor$ 
12:    end if
13:  end while
14:   $bestNeighbor \leftarrow evaluate(neighborhood)$ 
15:  if  $bestNeighbor$  的价值大于  $solution$  的价值 then
16:     $solution \leftarrow bestNeighbor$ 
17:     $tabuList \leftarrow addTabuList(tabuList, bestNeighbor, tabuLen)$ 
18:    if  $tabuList.size() > tabuLen$  then
19:       $solution = tabuList[0]$ 
20:       $tabuList.popFront()$ 
21:    end if
22:  end if
23: end while
```

```
1  #include <algorithm>
2  #include <cmath>
3  #include <cstdio>
4  #include <deque>
5  #include <iostream>
6  #include <list>
7  #include <map>
8  #include <unistd.h>
9  #include <vector>
10
11 #define in(container, object) \
12     (std::find(container.begin(), container.end(), object) != container.end())
13
14 using std::deque;
15 using std::list;
16 using std::sort;
17 using std::vector;
18
19 const int N = 1e3 + 5;
```

```

20  const int INF = 0x3f3f3f3f;
21
22  struct Object {
23      int p;
24      int w;
25      int index;
26  };
27
28  struct Knapsack {
29      int capacity;
30      int cost;
31      int index;
32
33      list<Object> objects;
34  };
35
36  bool operator==(const Object &a, const Object &b) {
37      return a.p == b.p && a.w == b.w;
38  }
39
40  bool cmp_object(Object a, Object b) {
41      return (double)a.p / (double)a.w > (double)b.p / (double)b.w;
42  }
43
44  void output_knapsacks(vector<Knapsack> &knapsacks) {
45      // detail data output
46      std::cout << "-----knapsack info-----" << std::endl;
47      for (auto &knapsack : knapsacks) {
48          std::cout << "Knapsack's index: " << knapsack.index << std::endl;
49          std::cout << "Knapsack's capacity: " << knapsack.capacity << std::endl;
50          std::cout << "Knapsack's cost: " << knapsack.cost << std::endl;
51          for (auto &object : knapsack.objects) {
52              std::cout << object.index << " " << object.p << " " << object.w
53                  << std::endl;
54          }
55      }
56  }
57
58  int calculate_result(vector<Object> &objects, vector<int> solution) {
59      int result = 0;
60
61      for (auto &object : objects) {
62          if (solution[object.index] != -1) {
63              result += object.p;
64          }
65      }

```

```

66
67     return result;
68 }
69
70 int greedy(vector<Object> &objects, vector<Knapsack> &knapsacks,
71           vector<bool> &flag) {
72     sort(objects.begin(), objects.end(), cmp_object);
73
74     for (auto &knapsack : knapsacks) {
75         for (auto i = 0; i < objects.size(); i++) {
76             if (!flag[objects[i].index] &&
77                 knapsack.capacity - knapsack.cost >= objects[i].w) {
78                 flag[objects[i].index] = true;
79                 knapsack.cost += objects[i].w;
80                 knapsack.objects.push_back(objects[i]);
81             }
82         }
83     }
84
85     int result = 0;
86
87     for (auto &knapsack : knapsacks) {
88         for (auto &object : knapsack.objects) {
89             result += object.p;
90         }
91     }
92
93     return result;
94 }
95
96 int tabu_search(vector<Object> &objects, vector<Knapsack> &knapsacks,
97                vector<bool> &flag) {
98     auto result = greedy(objects, knapsacks, flag);
99
100     // Using vector to store the solution
101     vector<int> solution(objects.size(), -1); // -1 stands for outside
102     for (auto &knapsack : knapsacks) {
103         for (auto &object : knapsack.objects) {
104             solution[object.index] = knapsack.index;
105         }
106     }
107
108     deque<vector<int>> tabu_list;
109     tabu_list.push_back(solution);
110     auto tabu_length = objects.size() / 2;
111     auto iteration = 1000; // iteration times

```

```

112 while (iteration--) {
113     // Generate neighbors
114     vector<vector<int>> neighbors;
115     for (auto i = 0; i < objects.size(); i++) {
116         if (solution[objects[i].index] == -1) {
117             // put the object into knapsack
118             for (auto &knapsack : knapsacks) {
119                 if (knapsack.capacity - knapsack.cost >= objects[i].w) {
120                     auto neighbor = solution;
121                     neighbor[objects[i].index] = knapsack.index;
122                     if (!in(tabu_list, neighbor)) {
123                         neighbors.push_back(neighbor);
124                     }
125                 }
126             }
127
128             // Find appropriate object in knapsacks to throw out and put the object
129             for (auto &knapsack : knapsacks) {
130                 for (auto &object : knapsack.objects) {
131                     auto k_rest = knapsack.capacity - knapsack.cost;
132                     if (object.index != objects[i].index &&
133                         object.w + k_rest >= objects[i].w) {
134                         auto neighbor = solution;
135                         neighbor[object.index] = -1;
136                         neighbor[objects[i].index] = knapsack.index;
137                         if (!in(tabu_list, neighbor)) {
138                             neighbors.push_back(neighbor);
139                         }
140                     }
141                 }
142             }
143
144             // Rotate
145             for (auto ki = 0; ki < knapsacks.size(); ki++) {
146                 for (auto &object1 : knapsacks[ki].objects) {
147                     for (auto kj = 0; kj < knapsacks.size(); kj++) {
148                         auto ki_rest = knapsacks[ki].capacity - knapsacks[ki].cost;
149                         auto kj_rest = knapsacks[kj].capacity - knapsacks[kj].cost;
150                         if (object1.w <= kj_rest && objects[i].w <= ki_rest + object1.w &&
151                             ki != kj) {
152                             auto neighbor = solution;
153                             neighbor[objects[i].index] = knapsacks[ki].index;
154                             neighbor[object1.index] = knapsacks[kj].index;
155                             if (!in(tabu_list, neighbor)) {
156                                 neighbors.push_back(neighbor);
157                             }
158                         }
159                     }
160                 }
161             }
162         }
163     }
164 }

```

```

158         } else {
159             for (auto &object2 : knapsacks[kj].objects) { // Throw object2
160                 if (object1.w <= kj_rest + object2.w &&
161                     objects[i].w <= ki_rest + object1.w && ki != kj) {
162                     auto neighbor = solution;
163                     neighbor[objects[i].index] = knapsacks[ki].index;
164                     neighbor[object1.index] = knapsacks[kj].index;
165                     neighbor[object2.index] = -1;
166                     if (!in(tabu_list, neighbor)) {
167                         neighbors.push_back(neighbor);
168                     }
169                 }
170             }
171         }
172     }
173 }
174 }
175 } else {
176     for (auto &knapsack : knapsacks) {
177         if (knapsack.index != solution[objects[i].index] &&
178             knapsack.capacity - knapsack.cost >= objects[i].w) {
179             auto neighbor = solution;
180             neighbor[objects[i].index] = knapsack.index;
181             if (!in(tabu_list, neighbor)) {
182                 neighbors.push_back(neighbor);
183             }
184         }
185     }
186 }
187 }
188
189 // Find the best neighbor
190 auto maxn_neighbor_result = -INF;
191 for (auto &neighbor : neighbors) {
192     int neighbor_result = calculate_result(objects, neighbor);
193
194     if (neighbor_result > maxn_neighbor_result) {
195         maxn_neighbor_result = neighbor_result;
196         solution = neighbor;
197     }
198 }
199 if (result < maxn_neighbor_result) {
200     result = maxn_neighbor_result;
201 }
202 if (!in(tabu_list, solution)) {
203     tabu_list.push_back(solution);

```

```

204     if (tabu_list.size() > tabu_length) {
205         solution = tabu_list.front();
206         tabu_list.pop_front();
207     }
208 }
209
210 // Update knapsacks
211 for (auto &knapsack : knapsacks) {
212     knapsack.objects.clear();
213     knapsack.cost = 0;
214
215     for (auto i = 0; i < objects.size(); i++) {
216         if (solution[objects[i].index] == knapsack.index) {
217             knapsack.objects.push_back(objects[i]);
218             knapsack.cost += objects[i].w;
219         }
220     }
221
222     if (knapsack.cost > knapsack.capacity) {
223         std::cout << "Error!" << std::endl;
224         output_knapsacks(knapsacks);
225         return -1;
226     }
227 }
228 }
229
230 // Generate the best knapsacks
231 for (auto &knapsack : knapsacks) {
232     knapsack.objects.clear();
233     knapsack.cost = 0;
234
235     for (auto i = 0; i < objects.size(); i++) {
236         if (solution[objects[i].index] == knapsack.index) {
237             knapsack.objects.push_back(objects[i]);
238             knapsack.cost += objects[i].w;
239         }
240     }
241
242     std::cout << "Knapsack " << knapsack.index << " used " << knapsack.cost
243         << " of " << knapsack.capacity << std::endl;
244
245     if (knapsack.cost > knapsack.capacity) {
246         std::cout << "Error!" << std::endl;
247         output_knapsacks(knapsacks);
248         return -1;
249     }

```



```

250     }
251
252     return result;
253 }
254
255 int main() {
256     int n, m, p, w, W;
257     vector<Object> objects;
258     vector<Knapsack> knapsacks;
259
260     // Input user data
261     std::cin >> n >> m;
262     for (int i = 0; i < n; i++) {
263         std::cin >> p >> w;
264         objects.push_back(Object{p, w, i});
265     }
266     for (int i = 0; i < m; i++) {
267         std::cin >> W;
268         knapsacks.push_back(Knapsack{W, 0, i});
269     }
270
271     vector<bool> flag(objects.size(), false);
272     int result = tabu_search(objects, knapsacks, flag);
273     std::cout << "Result: " << result << std::endl;
274
275     return 0;
276 }

```

4 算法正确性证明

4.1 输入数据 1

```
8 5
12 1
38 6
18 3
39 7
14 3
27 8
24 8
8 7
11 7 6 6 6
```

greedy algorithm 结果

```
Knapsack 0 used 10 of 11
Knapsack 1 used 7 of 7
Knapsack 2 used 3 of 6
Knapsack 3 used 0 of 6
Knapsack 4 used 0 of 6
Result: 121
```

neighborhood search algorithm 结果

```
Knapsack 0 used 11 of 11
Knapsack 1 used 7 of 7
Knapsack 2 used 3 of 6
Knapsack 3 used 6 of 6
Knapsack 4 used 0 of 6
Result: 129
```

tabu search algorithm 结果

```
Knapsack 0 used 9 of 11
Knapsack 1 used 7 of 7
Knapsack 2 used 3 of 6
Knapsack 3 used 6 of 6
Knapsack 4 used 3 of 6
Result: 148
```

4.2 输入数据 2

```
8 4
30 10
12 7
5 3
6 4
11 9
2 8
2 10
1 9
12 11 8 7
```

greedy algorithm 结果

```
Knapsack 0 used 10 of 12
Knapsack 1 used 10 of 11
Knapsack 2 used 4 of 8
Knapsack 3 used 0 of 7
Result: 53
```

neighborhood search algorithm 结果

```
Knapsack 0 used 10 of 12
Knapsack 1 used 11 of 11
Knapsack 2 used 4 of 8
Knapsack 3 used 7 of 7
Result: 55
```

tabu search algorithm 结果

```
Knapsack 0 used 9 of 12
Knapsack 1 used 11 of 11
Knapsack 2 used 4 of 8
Knapsack 3 used 7 of 7
Result: 64
```

4.3 输入数据 3

```
8 5
25 1
6 6
34 9
11 2
42 3
10 8
33 4
15 5
2 6 8 17 7
```

greedy algorithm 结果

```
Knapsack 0 used 1 of 2
Knapsack 1 used 5 of 6
Knapsack 2 used 4 of 8
Knapsack 3 used 14 of 17
Knapsack 4 used 6 of 7
Result: 166
```

neighborhood search algorithm 结果

```
Knapsack 0 used 1 of 2
Knapsack 1 used 5 of 6
Knapsack 2 used 8 of 8
Knapsack 3 used 14 of 17
Knapsack 4 used 4 of 7
Result: 170
```

tabu search algorithm 结果

```
Knapsack 0 used 0 of 2
Knapsack 1 used 6 of 6
Knapsack 2 used 8 of 8
Knapsack 3 used 17 of 17
Knapsack 4 used 7 of 7
Result: 176
```