# 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

**call** *sort*($n$ 个物品, 根据物品单位重量价值从大到小排序)
**for** $j \leftarrow 1; j < n; j++$ **do**
  **for** $i \leftarrow 1; i < m; i++$ **do**
    **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**
      物品 $i$ 放入背包 $j$ 中
    **end if**
  **end for**
**end for**
**Output** $m$ 个背包的最大价值

## 1.3 代码实现

```cpp
#include <algorithm>
#include <iostream>

const int N = 1e5 + 5;

struct Object {
  int p;
  int w;
};

bool cmp_object(Object a, Object b) {
  return (double)a.p / (double)a.w > (double)b.p / (double)b.w;
}

int greedy(int n, int m, Object *objects, int *W) {
  std::sort(objects, objects + n, cmp_object);

  bool x[N];
  int W_used[N];
  memset(x, 0, sizeof(x));
  memset(W_used, 0, sizeof(W_used));
```

```cpp
  for (int j = 0; j < m; j++) {
    for (int i = 0; i < n; i++) {
      if (!x[i] && W[j] - W_used[j] >= objects[i].w) {
        x[i] = true;
        W_used[j] += objects[i].w;
      }
    }
  }

  int result = 0;
  for (int i = 0; i < n; i++) {
    if (x[i]) {
      result += objects[i].p;
    }
  }

  return result;
}

int main() {
  int n, m;
  Object objects[N];
  int W[N];

  // Input user data
  std::cin >> n >> m;
  for (int i = 0; i < n; i++) {
    std::cin >> objects[i].p >> objects[i].w;
  }
  for (int i = 0; i < m; i++) {
    std::cin >> W[i];
  }

  int result = greedy(n, m, objects, W);
  std::cout << result << std::endl;

  return 0;
}
```

# 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

**call** *greedyAlgorithm*(n 个物品, m 个背包)
*solution* $\leftarrow$ *getSolution*(m 个背包)
**while** 没有找到最优解 **do**
  *neighborhood* $\leftarrow$ *getNeighborhood*(solution)
  *bestNeighbor* $\leftarrow$ *evaluate*(neighborhood)
  **if** bestNeighbor 的价值大于 solution 的价值 **then**
    *solution* $\leftarrow$ *bestNeighbor*
  **else**
    **break**
  **end if**
**end while**

## 2.3 代码实现

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <deque>
#include <iostream>
```

```cpp
#include <list>
#include <map>
#include <unistd.h>
#include <vector>

#define in(container, object)                                          \
  (std::find(container.begin(), container.end(), object) != container.end())

using std::list;
using std::sort;
using std::vector;

const int N = 1e3 + 5;
const int INF = 0x3f3f3f3f;

struct Object {
  int p;
  int w;
  int index;
};

struct Knapsack {
  int capacity;
  int cost;
  int index;

  list<Object> objects;
};

bool operator==(const Object &a, const Object &b) {
  return a.p == b.p && a.w == b.w;
}

bool cmp_object(Object a, Object b) {
  return (double)a.p / (double)a.w > (double)b.p / (double)b.w;
}

void output_knapsacks(vector<Knapsack> &knapsacks) {
  // detail data output
  std::cout << "---------------knapsack info---------------" << std::endl;
  for (auto &knapsack : knapsacks) {
    std::cout << "Knapsack's index: " << knapsack.index << std::endl;
    std::cout << "Knapsack's capacity: " << knapsack.capacity << std::endl;
    std::cout << "Knapsack's cost: " << knapsack.cost << std::endl;
    for (auto &object : knapsack.objects) {
      std::cout << object.index << " " << object.p << " " << object.w
```

```
              << std::endl;
    }
  }
}

int calculate_result(vector<Object> &objects, vector<int> solution) {
  int result = 0;

  for (auto &object : objects) {
    if (solution[object.index] != -1) {
      result += object.p;
    }
  }

  return result;
}

int greedy(vector<Object> &objects, vector<Knapsack> &knapsacks,
           vector<bool> &flag) {
  sort(objects.begin(), objects.end(), cmp_object);

  for (auto &knapsack : knapsacks) {
    for (auto i = 0; i < objects.size(); i++) {
      if (!flag[objects[i].index] &&
          knapsack.capacity - knapsack.cost >= objects[i].w) {
        flag[objects[i].index] = true;
        knapsack.cost += objects[i].w;
        knapsack.objects.push_back(objects[i]);
      }
    }
  }

  int result = 0;

  for (auto &knapsack : knapsacks) {
    for (auto &object : knapsack.objects) {
      result += object.p;
    }
  }

  return result;
}

int neighborhood_search(vector<Object> &objects, vector<Knapsack> &knapsacks,
                        vector<bool> &flag) {
  auto result = greedy(objects, knapsacks, flag);
```

```
  // Using vector to store the solution
  vector<int> solution(objects.size(), -1); // -1 stands for outside
  for (auto &knapsack : knapsacks) {
    for (auto &object : knapsack.objects) {
      solution[object.index] = knapsack.index;
    }
  }

  while (true) {
    // Generate neighbors
    vector<vector<int>> neighbors;
    for (auto i = 0; i < objects.size(); i++) {
      if (solution[objects[i].index] == -1) {
        // put the object into knapsack
        for (auto &knapsack : knapsacks) {
          if (knapsack.capacity - knapsack.cost >= objects[i].w) {
            auto neighbor = solution;
            neighbor[objects[i].index] = knapsack.index;
            neighbors.push_back(neighbor);
          }
        }
        // Find appropriate object in knapsacks to throw out and put the object
        for (auto &knapsack : knapsacks) {
          for (auto &object : knapsack.objects) {
            if (object.index != objects[i].index && object.w >= objects[i].w) {
              auto neighbor = solution;
              neighbor[object.index] = -1;
              neighbor[objects[i].index] = knapsack.index;
              neighbors.push_back(neighbor);
            }
          }
        }
        // Rotate
        for (auto ki = 0; ki < knapsacks.size(); ki++) {
          for (auto &object1 : knapsacks[ki].objects) {
            for (auto kj = 0; kj < knapsacks.size(); kj++) {
              auto ki_rest = knapsacks[ki].capacity - knapsacks[ki].cost;
              auto kj_rest = knapsacks[kj].capacity - knapsacks[kj].cost;
              if (object1.w <= kj_rest && objects[i].w <= ki_rest + object1.w &&
                  kj != ki) {
                auto neighbor = solution;
                neighbor[objects[i].index] = knapsacks[ki].index;
                neighbor[object1.index] = knapsacks[kj].index;
                neighbors.push_back(neighbor);
              } else {
```

```cpp
                for (auto &object2 : knapsacks[kj].objects) { // Throw object2
                  if (object1.w <= kj_rest + object2.w &&
                      objects[i].w <= ki_rest + object1.w &&
                      object2.p < objects[i].p && ki != kj) {
                    auto neighbor = solution;
                    neighbor[objects[i].index] = knapsacks[ki].index;
                    neighbor[object1.index] = knapsacks[kj].index;
                    neighbor[object2.index] = -1;
                    neighbors.push_back(neighbor);
                  }
                }
              }
            }
          }
        }
      }
    }

    // Find the best neighbor
    auto maxn_neighbor_result = -INF;
    for (auto &neighbor : neighbors) {
      int neighbor_result = calculate_result(objects, neighbor);

      if (neighbor_result > maxn_neighbor_result) {
        maxn_neighbor_result = neighbor_result;
        solution = neighbor;
      }
    }

    if (result < maxn_neighbor_result) {
      result = maxn_neighbor_result;
    } else {
      break;
    }

    // Update knapsacks
    for (auto &knapsack : knapsacks) {
      knapsack.objects.clear();
      knapsack.cost = 0;

      for (auto i = 0; i < objects.size(); i++) {
        if (solution[objects[i].index] == knapsack.index) {
          knapsack.objects.push_back(objects[i]);
          knapsack.cost += objects[i].w;
        }
      }
```

```
      if (knapsack.cost > knapsack.capacity) {
        std::cout << "Error!" << std::endl;
        output_knapsacks(knapsacks);
        return -1;
      }
    }
  }

  return result;
}

int main() {
  int n, m, p, w, W;
  vector<Object> objects;
  vector<Knapsack> knapsacks;

  // Input user data
  std::cin >> n >> m;
  for (int i = 0; i < n; i++) {
    std::cin >> p >> w;
    objects.push_back(Object{p, w, i});
  }
  for (int i = 0; i < m; i++) {
    std::cin >> W;
    knapsacks.push_back(Knapsack{W, 0, i});
  }

  vector<bool> flag(objects.size(), false);
  int result = neighborhood_search(objects, knapsacks, flag);
  std::cout << result << std::endl;

  return 0;
}
```

# 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

**call** *greedyAlgorithm*(n 个物品, m 个背包)
$solution \leftarrow getSolution$(m 个背包)
$iteration \leftarrow 1000$
$tabuList \leftarrow [\quad]$
$neighborhood \leftarrow [\quad]$
$tabuLen \leftarrow$n/2
**while** 没有找到最优解 **do**
    **while** 还能生成新的解 **do**
        $neighbor \leftarrow generateNeighbor$(solution)
        **if** neighbor not in tabuList **then**
            $neighborhood \leftarrow neighborhood + neighbor$
        **end if**
    **end while**
    $bestNeighbor \leftarrow evaluate$(neighborhood)
    **if** bestNeighbor 的价值大于 solution 的价值 **then**
        $solution \leftarrow bestNeighbor$
        $tabuList \leftarrow addTabuList$(tabuList, bestNeighbor, tabuLen)
        **if** tabuList.size() $>$ tabuLen **then**
            solution = tabuList[0]
            tabuList.popFront()
        **end if**
    **end if**
**end while**

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <deque>
#include <iostream>
#include <list>
#include <map>
#include <unistd.h>
#include <vector>

#define in(container, object)                                          \
  (std::find(container.begin(), container.end(), object) != container.end())

using std::deque;
using std::list;
using std::sort;
using std::vector;

const int N = 1e3 + 5;
```

```cpp
const int INF = 0x3f3f3f3f;

struct Object {
  int p;
  int w;
  int index;
};

struct Knapsack {
  int capacity;
  int cost;
  int index;

  list<Object> objects;
};

bool operator==(const Object &a, const Object &b) {
  return a.p == b.p && a.w == b.w;
}

bool cmp_object(Object a, Object b) {
  return (double)a.p / (double)a.w > (double)b.p / (double)b.w;
}

void output_knapsacks(vector<Knapsack> &knapsacks) {
  // detail data output
  std::cout << "---------------knapsack info---------------" << std::endl;
  for (auto &knapsack : knapsacks) {
    std::cout << "Knapsack's index: " << knapsack.index << std::endl;
    std::cout << "Knapsack's capacity: " << knapsack.capacity << std::endl;
    std::cout << "Knapsack's cost: " << knapsack.cost << std::endl;
    for (auto &object : knapsack.objects) {
      std::cout << object.index << " " << object.p << " " << object.w
                << std::endl;
    }
  }
}

int calculate_result(vector<Object> &objects, vector<int> solution) {
  int result = 0;

  for (auto &object : objects) {
    if (solution[object.index] != -1) {
      result += object.p;
    }
  }
```

```

  return result;
}

int greedy(vector<Object> &objects, vector<Knapsack> &knapsacks,
           vector<bool> &flag) {
  sort(objects.begin(), objects.end(), cmp_object);

  for (auto &knapsack : knapsacks) {
    for (auto i = 0; i < objects.size(); i++) {
      if (!flag[objects[i].index] &&
          knapsack.capacity - knapsack.cost >= objects[i].w) {
        flag[objects[i].index] = true;
        knapsack.cost += objects[i].w;
        knapsack.objects.push_back(objects[i]);
      }
    }
  }

  int result = 0;

  for (auto &knapsack : knapsacks) {
    for (auto &object : knapsack.objects) {
      result += object.p;
    }
  }

  return result;
}

int tabu_search(vector<Object> &objects, vector<Knapsack> &knapsacks,
                vector<bool> &flag) {
  auto result = greedy(objects, knapsacks, flag);

  // Using vector to store the solution
  vector<int> solution(objects.size(), -1); // -1 stands for outside
  for (auto &knapsack : knapsacks) {
    for (auto &object : knapsack.objects) {
      solution[object.index] = knapsack.index;
    }
  }

  deque<vector<int>> tabu_list;
  tabu_list.push_back(solution);
  auto tabu_length = objects.size() / 2;
  auto iteration = 1000; // iteration times
```

```
  while (iteration--) {
    // Generate neighbors
    vector<vector<int>> neighbors;
    for (auto i = 0; i < objects.size(); i++) {
      if (solution[objects[i].index] == -1) {
        // put the object into knapsack
        for (auto &knapsack : knapsacks) {
          if (knapsack.capacity - knapsack.cost >= objects[i].w) {
            auto neighbor = solution;
            neighbor[objects[i].index] = knapsack.index;
            if (!in(tabu_list, neighbor)) {
              neighbors.push_back(neighbor);
            }
          }
        }

        // Find appropriate object in knapsacks to throw out and put the object
        for (auto &knapsack : knapsacks) {
          for (auto &object : knapsack.objects) {
            auto k_rest = knapsack.capacity - knapsack.cost;
            if (object.index != objects[i].index &&
                object.w + k_rest >= objects[i].w) {
              auto neighbor = solution;
              neighbor[object.index] = -1;
              neighbor[objects[i].index] = knapsack.index;
              if (!in(tabu_list, neighbor)) {
                neighbors.push_back(neighbor);
              }
            }
          }
        }

        // Rotate
        for (auto ki = 0; ki < knapsacks.size(); ki++) {
          for (auto &object1 : knapsacks[ki].objects) {
            for (auto kj = 0; kj < knapsacks.size(); kj++) {
              auto ki_rest = knapsacks[ki].capacity - knapsacks[ki].cost;
              auto kj_rest = knapsacks[kj].capacity - knapsacks[kj].cost;
              if (object1.w <= kj_rest && objects[i].w <= ki_rest + object1.w &&
                  ki != kj) {
                auto neighbor = solution;
                neighbor[objects[i].index] = knapsacks[ki].index;
                neighbor[object1.index] = knapsacks[kj].index;
                if (!in(tabu_list, neighbor)) {
                  neighbors.push_back(neighbor);
                }
```

```
            } else {
              for (auto &object2 : knapsacks[kj].objects) { // Throw object2
                if (object1.w <= kj_rest + object2.w &&
                    objects[i].w <= ki_rest + object1.w && ki != kj) {
                  auto neighbor = solution;
                  neighbor[objects[i].index] = knapsacks[ki].index;
                  neighbor[object1.index] = knapsacks[kj].index;
                  neighbor[object2.index] = -1;
                  if (!in(tabu_list, neighbor)) {
                    neighbors.push_back(neighbor);
                  }
                }
              }
            }
          }
        }
      }
    } else {
      for (auto &knapsack : knapsacks) {
        if (knapsack.index != solution[objects[i].index] &&
            knapsack.capacity - knapsack.cost >= objects[i].w) {
          auto neighbor = solution;
          neighbor[objects[i].index] = knapsack.index;
          if (!in(tabu_list, neighbor)) {
            neighbors.push_back(neighbor);
          }
        }
      }
    }
  }

  // Find the best neighbor
  auto maxn_neighbor_result = -INF;
  for (auto &neighbor : neighbors) {
    int neighbor_result = calculate_result(objects, neighbor);

    if (neighbor_result > maxn_neighbor_result) {
      maxn_neighbor_result = neighbor_result;
      solution = neighbor;
    }
  }
  if (result < maxn_neighbor_result) {
    result = maxn_neighbor_result;
  }
  if (!in(tabu_list, solution)) {
    tabu_list.push_back(solution);
```

```
      if (tabu_list.size() > tabu_length) {
        solution = tabu_list.front();
        tabu_list.pop_front();
      }
    }

    // Update knapsacks
    for (auto &knapsack : knapsacks) {
      knapsack.objects.clear();
      knapsack.cost = 0;

      for (auto i = 0; i < objects.size(); i++) {
        if (solution[objects[i].index] == knapsack.index) {
          knapsack.objects.push_back(objects[i]);
          knapsack.cost += objects[i].w;
        }
      }

      if (knapsack.cost > knapsack.capacity) {
        std::cout << "Error!" << std::endl;
        output_knapsacks(knapsacks);
        return -1;
      }
    }
  }

  // Generate the best knapsacks
  for (auto &knapsack : knapsacks) {
    knapsack.objects.clear();
    knapsack.cost = 0;

    for (auto i = 0; i < objects.size(); i++) {
      if (solution[objects[i].index] == knapsack.index) {
        knapsack.objects.push_back(objects[i]);
        knapsack.cost += objects[i].w;
      }
    }

    std::cout << "Knapsack " << knapsack.index << " used " << knapsack.cost
              << " of " << knapsack.capacity << std::endl;

    if (knapsack.cost > knapsack.capacity) {
      std::cout << "Error!" << std::endl;
      output_knapsacks(knapsacks);
      return -1;
    }
```

```
  }

  return result;
}

int main() {
  int n, m, p, w, W;
  vector<Object> objects;
  vector<Knapsack> knapsacks;

  // Input user data
  std::cin >> n >> m;
  for (int i = 0; i < n; i++) {
    std::cin >> p >> w;
    objects.push_back(Object{p, w, i});
  }
  for (int i = 0; i < m; i++) {
    std::cin >> W;
    knapsacks.push_back(Knapsack{W, 0, i});
  }

  vector<bool> flag(objects.size(), false);
  int result = tabu_search(objects, knapsacks, flag);
  std::cout << "Result: " << result << std::endl;

  return 0;
}
```

# 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
```