Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский университет «Высшая школа экономики»

Факультет компьютерных наук
Основная образовательная программа
Прикладная математика и информатика

ГРУППОВАЯ КУРСОВАЯ РАБОТА

Исследовательский проект на тему

“Методы решения задачи о ранце со многими ограничениями”

Выполнили студенты группы БПМИ-184, 3 курса,

- **Сахиуллин Ильдар Раушанович**
- **Вологодский Михаил Евгеньевич**

Руководитель КР:
Профессор, доцент, Посыпкин Михаил Анатольевич,

Москва 2021

Оглавление

Введение

Задачу о рюкзаке можно назвать одной из самых популярных в теории алгоритмов. В своей оригинальной формулировке она звучит так:

"Дано N предметов, предмет n_i имеет массу $w_i > 0$ и стоимость $p_i > 0$. Необходимо выбрать из этих предметов такой набор, чтобы их суммарная масса не превосходила вместимости рюкзака W, а суммарная стоимость при этом была максимальна."

У данной задачи есть одна очень важная особенность: она является NP-полной. Это значит, что для её решения в общем виде не существует быстрого (полиномиального) алгоритма. Вероятно, именно это и послужило причиной, по которой у задачи появилось очень много вариаций: неограниченный рюкзак, рюкзак 0-1, задача о размене и другие.

Мы же сосредоточим наше внимание на задаче о ранце со многими ограничениями:

"Найти максимум линейной функции при линейных ограничениях при условии, что переменные – бинарные, т.е. принимают значения либо 0, либо 1."

Данная задача также является NP-полной (даже при одном ограничении), но при этом имеет множество применений. В частности, в области беспроводных сетей передачи данных. По этой причине данную задачу важно уметь эффективно решать.

В этом проекте мы займёмся исследованием уже существующих методов решения задачи о рюкзаке со многими ограничениями, а также их имплементацией на языке программирования Python.

Основные цели и задачи

- Изучить 2-3 существующих методов решения задачи о ранце со многими ограничениями.
- Реализовать на каком-нибудь языке программирования хотя бы один из изученных методов.
- Провести вычислительный эксперимент для оценки эффективности реализованных методов.
- Сделать выводы из данных оценок эффективности.
- Сделать конечные выводы.

1. Формулировка задачи

Бинарная версия задачи о ранце со многими ограничениями (англ. *Multiple Knapsack Problem (MKP)*) поставлена следующим образом: есть n предметов и m рюкзаков ($m < n$),

$$p_j - \text{цена } j\text{-го предмета},$$
$$w_j - \text{вес } j\text{-го предмета},$$
$$c_i - \text{вместимость } i\text{-го рюкзака}.$$

Требуется выбрать m непересекающихся между собой множеств предметов таких, что их суммарная стоимость максимальна. При этом каждое из выбранных множеств может быть присвоено определённому рюкзаку, чья вместимость не меньше суммарной массы объектов выбранного множества. Более формально:

$$z = \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij} \to \max$$

$$\text{при условии, что } \sum_{j=1}^{n} w_j x_{ij} \le c_i, \quad i \in M = \{1, \dots, m\}.$$

$$\sum_{i=1}^{m} x_{ij} \le 1, \qquad j \in N = \{1, \dots, n\},$$

$$x_{ij} \in \{0, 1\}, \qquad i \in M,\ j \in N,$$

$$\text{где } x_{ij} = \begin{cases} 1, \text{ если предмет } j \text{ лежит в рюкзаке } i \\ 0 \text{ иначе} \end{cases}$$

Мы будем предполагать, что w_j - положительные целые числа. Также без потери общности будем предполагать, что

$$(1) \quad p_j, c_i \in \mathbb{Z}, \quad p_j > 0,\ c_i > 0$$
$$(2) \quad w_j \le \max_{i \in M} c_i \quad j \in N,$$
$$(3) \quad c_i \ge \min_{j \in N} w_j \quad i \in M,$$
$$(4) \quad \sum_{j=1}^{n} w_j > c_i \quad i \in M$$

В самом деле, если нарушено первое условие, то от дробей можно избавиться путём домножения всех переменных на одно и то же число, которое сокращает эти дроби. Отрицательные значения p_j и c_i при этом можно просто отбросить.

Если не выполнено условие 2 или 3, то соответствующие переменные можно также исключить из рассмотрения.

Если рюкзак с номером i^* нарушает условие 4, то исходная задача имеет тривиальное решение:

$$x_{i^*j} = 1 \quad \text{для } j \in N,$$
$$x_{ij} = 0 \quad \text{для } i \in M \setminus \{i^*\}, j \in N.$$

Наконец, если $m > n$, то $(m-n)$ рюкзаков с наименьшей вместимостью могут быть просто исключены.

Для дальнейших выкладок также будем предполагать, что предметы отсортированы следующим образом:

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$$

2. Полиномиальный приблизительный алгоритм

2.1. Описание

Пусть X_s – подмножество элементов, порождающих $z\left(S(MKP)\right)$, где $S(MKP)$ - это так называемая задача *surrogate relaxation*:

$$\sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \to \max$$

$$\text{при условии, что } \sum_{i=1}^{m} \pi_i \sum_{j=1}^{n} w_j x_{ij} \leq \sum_{i=1}^{m} \pi_i c_i,$$

$$\sum_{i=1}^{m} x_{ij} \leq 1, \quad j \in N,$$

$$x_{ij} \in \{0;1\}, \quad i \in M,\ j \in N,$$

где $(\pi_1 \dots \pi_m)$ – положительный вектор множителей.

Оптимальный вектор множителей, то есть тот, на котором достигается минимальное значение $z\left(S(MKP,\pi)\right)$, определён следующим образом:

$$\pi_i = k, \text{ где } k \text{ – любая положительная константа } \forall i \in M.$$

Полиномиальный приблизительный алгоритм рассматривает предметы из X_s в порядке убывания их масс и пытается добавить каждый предмет в случайно выбранный рюкзак или, если он туда не помещается, – в один из оставшихся рюкзаков. Если предмет не может быть помещён ни в один из рюкзаков, алгоритм пытается для каждой пары рюкзаков обменять уже лежащие в них предметы с целью получения достаточного количества места для нового объекта.

Если все предметы из X_s использованы, оптимальное решение найдено. Иначе текущее решение может быть улучшено путём добавления как можно большего количества предметов из множества $N \backslash X_s$ в текущие рюкзаки.

Будем считать, что предметы отсортированы в соответствии с неравенством, введёного в описании формулировки исходной задачи, а рюкзаки – следующим образом:

$$c_1 \leq c_2 \leq \dots \leq c_m$$

```
def greedys(n, p, w, z, y, i, c):
  for j in range(n):
    if y[j] == -1 and w[j] <= c:
      y[j] = i
      c -= w[j]
      z += p[j]
  return z, y
```

Рис. 1: GREEDYS function

Первоначальное возможное решение получается путём вызова функции GREEDYS (см. рис. 1) для первого рюкзака, затем – для второго (для оставшихся предметов) и так далее. Таким образом функция GREEDYS вызывается m раз.

```
def MTHM(n, m, p, w, c):
  # initial solution
  z = 0
  y = [-1 for j in range(n)]
  for i in range(m):
    z, y = greedys(n, p, w, z, y, i, c[i])
```

Рис. 2: MTHM. Первоначальное решение.

После этого алгоритм улучшает решение через уже описанный процесс обмена предметами между рюкзаками. Когда все пары предметов были рассмотрены, алгоритм пытается исключить уже выбранные предметы и заменить их одним или более из тех, что ещё не были рассмотрены. Всё это делается с целью повышения максимальной суммарной ценности всех выбранных предметов из конечного решения.

```
# rearrangement
z = 0
c_mod = c.copy() # modified
i = 0
for j in range(n - 1, -1, -1):
  if y[j] > -1:
    l = -1
    for k in range(i, m):
      if w[j] <= c_mod[k]:
        l = k
        break
    if l == -1:
      for k in range(0, i):
        if w[j] <= c_mod[k]:
          l = k
          break
    if l == -1:
      y[j] = -1
    else:
      y[j] = l
      c_mod[l] -= w[j]
      z += p[j]
      if l < m:
        i = l + 1
      else:
        i = 0
for i in range(m):
  z, y = greedys(n, p, w, z, y, i, c_mod[i])
```

Рис. 3: МТНМ. Улучшение решения.

Вычислительные эксперименты показывают, что процесс обмена, описанный выше, гораздо эффективнее, если каждый из ранцев содержит предметы с разной ценностью на единицу веса.

Собрав всё вместе, мы получаем конечную функцию MTHM, которая решает нашу задачу. Ниже представлены заключительные фрагменты данной функции. (Начало см. на рис. 3)

```
for j in range(n):
  if y[j] > -1:
    for k in range(j + 1, n):
      if -1 < y[k] and y[k] != y[j]:
        if w[j] > w[k]:
          h = j
          l = k
        else:
          h = k
          l = j
        d = w[h] - w[l]
        min_w = -1
        for ii in range(n):
          if y[ii] == -1:
            if min_w == -1 or min_w > w[ii]:
              min_w = w[ii]
        if d <= c_mod[y[l]] and c_mod[y[h]] + d >= min_w:
          max_p = -1
          for ii in range(n):
            if y[ii] == -1 and w[ii] <= c_mod[y[h]] + d:
              if max_p == -1 or max_p < p[ii]:
                max_p = p[ii]
                t = ii
          c_mod[y[h]] = c_mod[y[h]] + d - w[t]
          c_mod[y[l]] -= d
          y[t] = y[h]
          y[h] = y[l]
          y[l] = y[t]
          z += p[t]
```

Рис. 4: Реализация функции MTHM.

```
for j in range(n - 1, -1, -1):
  if y[j] > -1:
    c_temp = c_mod[y[j]] + w[j]
    Y = set()
    for k in range(n):
      if y[k] == -1 and w[k] <= c_temp:
        Y.add(k)
        c_temp -= w[k]
    sum_of_Y_elems = 0
    for el in Y:
      sum_of_Y_elems += p[el]
    if sum_of_Y_elems > p[j]:
      for el in Y:
        y[el] = y[j]
      c_mod[y[j]] = c_temp
      y[j] = -1
      z += sum_of_Y_elems - p[j]
return z, y
```

Рис. 5: Реализация функции MTHM.

2.2. Оценка эффективности

Ни один из шагов алгоритма MTHM не требует более, чем $O(n^2)$ времени. Это очевидно для шагов 1, 2 (рис. 2 и рис. 3), так как функция GREEDYS работает за $O(n)$. Также это очевидно для шага 4 (рис. 5). Что касается шага 3 (рис. 4), достаточно заметить, что обновление $\{w_u : y_u = 0\}$ и поиск t во внутреннем цикле выполняются, только когда в решение добавляется новый предмет. То есть в общей сложности $O(n)$ раз.

2.3. Пример работы алгоритма

Выше представлено 2 примера работы программы для различных входных данных. Далее представлено пояснение к выводу программы.

В качестве ответа алгоритм выдаёт пару значений: максимальная возможная суммарная стоимость предметов, которую можно получить, и массив-маска, в котором 0 и 1 на i-ой позиции означает, что предмет под номером i нужно положить в 1-й или во 2-й рюкзак соответственно. -1 на i-ой позиции говорит о том, что предмет i брать вообще не стоит.

```
n = 10
m = 2
p = [78, 35, 89, 36, 94, 75, 74, 79, 80, 16]
w = [18, 9, 23, 20, 59, 61, 70, 75, 76, 30]
c = [103, 156]

ans = MTHM(n, m, p, w, c)
print(ans)
```

```
(423, [1, 0, 1, 1, 0, 1, -1, -1, -1, 0])
```

Рис. 6: Пример 1 работы программы.

```
n = 9
m = 2
p = [80, 20, 60, 40, 60, 60, 65, 25, 30]
w = [40, 10, 40, 30, 50, 50, 55, 25, 40]
c = [100, 150]

ans = MTHM(n, m, p, w, c)
print(ans)
```

```
(350, [0, 0, 1, 1, -1, 0, 1, 1, -1])
```

Рис. 7: Пример 2 работы программы.

3. Bound-and-bound алгоритм

3.1. Вступление

Оптимальное решение для MKP обычно получается методом ветвей и границ (англ. *branch-and-bound method*). Динамическое программирование несильно практично для задач данного типа с точки зрения затрачиваемого времени и памяти. Алгоритмы MKP обычно подразделяют на 2 вида:

1. С маленьким значением n/m
2. С большим значением n/m

Далее мы сосредоточим наше внимание на алгоритмах второго типа.

3.2. Описание алгоритма

Задача MKP была разрешена с помощью усовершенствования уже существующего алгоритма ветвей и границ путём подсчёта в каждом узле решений не только верхней границы, но и нижней. Алгоритм для MKP был получен Martello и Toth путём выведения числовой схемы, в которой каждый узел дерева решений порождает две ветви: либо присваивая предмет j рюкзаку i, либо исключая j из i. Далее будем считать, что в S_k $(k = 1, \ldots, m)$ содержатся те предметы, которые присвоены рюкзаку с номером k или исключены из него.

Пусть $S = \{S_1, \ldots, S_m\}$. На каждой итерации i - это текущий рюкзак, и алгоритм добавляет в него следующий выбранный предмет j с помощью эвристики. Переход к следующему рюкзаку (то есть к $i + 1$) наступает, когда в текущий рюкзак i больше не может быть добавлено ни одного предмета. Поэтому на i-ой итерации рюкзаки $1, \ldots, i-1$ полностью заполнены, рюкзак i заполнен частично, а рюкзаки под номерами $i + 1, \ldots, m$ пустые.

Верхние границы $U = U(S)$ вычисляются с помощью *surrogate relaxation* (см. предыдущий раздел) в функции UPPER. Нижние границы $L = L(S)$ и соответствующие ей эвристические решения $\widetilde{x}$ вычисляются функцией
LOWER, которая находит оптимальное решение для текущего рюкзака, затем исключает предметы, которые в него входят, и находит оптимальное решение для следующего рюкзака и так далее. В обеих функциях в списке параметров i – это номер рюкзака, а $\hat{x}_{kj}$ $(k = 1, \ldots, i;\ j = 1, \ldots, n)$ – текущее решение.

```
def upper(n, m, p, w, c, x_caret, S, i):
  sum_1 = 0
  sum_2 = 0
  for j in S[i]:
    sum_1 += w[j] * x_caret[i][j]
  for k in range(i + 1, m):
    sum_2 += c[k]
  c_ = c[i] - sum_1 + sum_2

  N = list()
  for j in range(n):
    all_x_equal_to_0 = True
    for k in range(0, i + 1):
      if x_caret[k][j] != 0:
        all_x_equal_to_0 = False
        break
    if all_x_equal_to_0 == True:
      N.append(j)

  w_ = []
  p_ = []
  for el in N:
    w_.append(w[el])
    p_.append(p[el])
  z = dynamic_method(N, len(N), w_, p_, c_, False)

  sum_3 = 0
  for k in range(0, i + 1):
    for j in S[k]:
      sum_3 += p[j] * x_caret[k][j]
  U = sum_3 + z

  return U
```

Рис. 8: Функция UPPER

```
def lower(n, m, p, w, c, x_caret, S, i):
  L = 0
  for k in range(0, i + 1):
    for j in S[k]:
      L += p[j] * x_caret[k][j]

  N_ = list()
  for j in range(n):
    all_x_equal_to_0 = True
    for k in range(0, i + 1):
      if x_caret[k][j] != 0:
        all_x_equal_to_0 = False
        break
    if all_x_equal_to_0 == True:
      N_.append(j)

  N = set(N_) - set(S[i])

  sum_1 = 0
  for j in S[i]:
    sum_1 += w[j] * x_caret[i][j]
  c_ = c[i] - sum_1
  k = i
  x_tilda = [[0 for j in range(n)] for r in range(m)] # m x n
  for r in range(m):
        for j in range(n):
          x_tilda[r][j] = x_caret[r][j]

  while k <= m - 1:
    w_ = []
    p_ = []
    for el in N:
      w_.append(w[el])
      p_.append(p[el])
    z, used_indexes = dynamic_method(list(N), len(N), w_, p_, c_, True)
    for el in used_indexes:
      x_tilda[k][el] = 1

    L += z
    N_ = set(N_) - set(used_indexes)
    N = N_
    k += 1
    if k != m:
      c_ = c[k]

  return L, x_tilda
```

Рис. 9: Функция LOWER

Наконец, стоит обратить внимание, что для рюкзнака с номером m в функции MTM не определено своё текущее решение, так как, зная $\hat{x}_{kj}$ для $k = 1, \dots m-1$, решение, выдаваемое функцией LOWER (рис. 9) для рюкзака m, оптимально. Поэтому изначально удобно отсортировать рюкзаки в порядке неубывания их вместимости:

$$c_1 \leq c_2 \leq \cdots \leq c_m.$$

Ввиду своей объёмности ниже представлена лишь часть функции MTM. Для полной версии можно обратиться к исходному коду.

```
def MTM(n, m, p, w, c):
  # initialize
  S = [[] for k in range(m)]
  x_caret = [[0 for j in range(n)] for k in range(m)]
  x = [[0 for j in range(n)] for k in range(m)]
  z = 0
  i = 0
  U = upper(n, m, p, w, c, x_caret, S, i)
  UB = U
  return_to_2_step = True

  # heuristic
  while return_to_2_step:
    L, x_tilda = lower(n, m, p, w, c, x_caret, S, i)
    go_to_backtrack = False
    if L > z:
      z = L
      for k in range(m):
        for j in range(n):
          x[k][j] = x_caret[k][j]
      for k in range(i, m):
        for j in range(n):
          if x_tilda[k][j] == 1:
            x[k][j] = 1
      if z == UB:
        return z, x
      if z == U:
        go_to_backtrack = True

    # define a new current solution
    if not go_to_backtrack:
      worse_solution_flag = False
      while i != m - 1 and not worse_solution_flag:
        I = set()
        for l in range(n):
          if x_tilda[i][l] == 1:
            I.add(l)
        while len(I) > 0:
          j = min(I)
          I -= {j}
          if j not in S[i]:
            S[i].append(j)
          x_caret[i][j] = 1
          U = upper(n, m, p, w, c, x_caret, S, i)
          if U <= z:
            worse_solution_flag = True
            break
        i += 1
      if not worse_solution_flag:
        i = m - 2
      else:
        i -= 1
```

Рис. 10: Функция MTM (неполная версия)

Пример работы алгоритма

Рассмотрим пример работы вышеописанного метода. В данном случае на вход программы передаются 10 предметов с различными ценностями и массами, заданными в массивах p и w соответственно. Также заданы 2 рюкзака с вместимостями 103 и 156.

```
n = 10
m = 2
p = [78, 35, 89, 36, 94, 75, 74, 79, 80, 16]
w = [18, 9, 23, 20, 59, 61, 70, 75, 76, 30]
c = [103, 156]
print(MTM(n, m, p, w, c))
```

```
(452, [[1, 0, 1, 0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 1, 0, 0, 0, 1, 0]])
```

Рис. 11: Пример работы алгоритма

Как мы видим, программа выдаёт ответ для каждого рюкзака в отдельности. То есть результатом работы является кортеж и 2 элементов, где 1-й элемент – максимальная суммарная ценность предметов, которую можно получить, а 2-й – массив длины m, состоящий из масок для каждого из рюкзаков – массивов из 0 и 1 (1 на i-ой позиции означает, что i-й элемент принадлежит данному рюкзаку, а 0 – означает, что нет).

Результаты

В результате проделанной работы мы изучили 2 метода задачи о ранце со многими ограничениями (МКР):

- Полиномиальный приблизительный алгоритм
- Bound-and-bound метод

Первый из них довольно прост в реализации и сравнительно быстр по времени работы, однако даёт лишь приблизительную оценку максимальной суммарной стоимости.

Второй, с другой стороны, значительно сложнее в написании, зато даёт оптимальный ответ на любых входных данных.

Также были произведены аналитические оценки эффективности данных методов, из которых можно заключить, что поставленная перед нами задача точными методами (как, например, $bound - and - bound$) не решается за полиномиальное время. Однако зачастую может быть достаточно приближённой оценки. В этом случае полиномиальный алгоритм может быть гораздо полезнее, так как он работает значительно быстрее.

Список источников

[1] Knapsack Problem (Wikipedia)

[2] The multidimensional 0-1 knapsack problem: An Overview.

[3] Knapsack Problems. Algorithms and computer implementations

Приложения

[1] Программный код для описанных методов