Бинарный поиск

Бинарный поиск – быстрый алгоритм, который находит элемент с определенным значением, но только в массивах, элементы которого упорядочены.

При использовании этого метода, заданная последовательность делится на две равные части, и поиск осуществляется в одной из них, далее эта часть вновь делится на две равные, и так до тех пор, пока обнаружится наличие или отсутствие искомого элемента.

# Рекурсивная реализация бинарного поиска

def binary\_search\_recursive(array, element, start, end):

# Выходим из рекурсии, если start > end

if start > end:

return -1

# Находим средний элемент, если искомый элемент равен среднему,

# то выводим его

mid = (start + end) // 2

if element == array[mid]:

return mid

#Если нет, мы проверяем, больше ли элемент или меньше среднего

if element < array[mid]:

# Поиск в левой половине

return binary\_search\_recursive(array, element, start, mid-1)

else:

# Поиск в правой половине

return binary\_search\_recursive(array, element, mid+1, end)

Транспортная задача

Транспортная задача – это задача линейного программирования об оптимальном плане перевозок грузов из пунктов отправления в пункты потребления, с минимальными затратами на перевозки.

Существует множество методов решения данной задачи, вот некоторые из них: метод минимального элемента, метод северо-западного угла и метод потенциалов.

Метод минимального элемента

Суть метода заключается в том, что из всей таблицы выбирают наименьшую стоимость и в клетку, которая ей соответствует, помещают меньшее из чисел.

Затем из рассмотрения исключают либо строку, соответствующую поставщику, запасы которого полностью израсходованы, либо столбец, соответствующий потребителю, потребности которого полностью удовлетворены, либо и строку и столбец, если израсходованы запасы поставщика и удовлетворены потребности потребителя.

Из оставшейся части таблицы стоимостей снова выбирают наименьшую стоимость, и процесс распределения запасов продолжают, пока все запасы не будут распределены, а потребности удовлетворены.

# Необходима функция нахождения индексов минимального элемента матрицы

def ij(c\_min):

c = np.inf

for i in range(c\_min.shape[0]):

for j in range(c\_min.shape[1]):

if (c\_min[i, j] !=0) and (c\_min[i, j]<c):

c = c\_min[i, j]

i\_, j\_ = i, j

return i\_, j\_

# Функция минимального элемента

def M\_min(a\_, b\_, c\_, print\_ = False):

a = np.copy(a\_)

b = np.copy(b\_)

c = np.copy(c\_)

# Проверяем условие замкнутости: если не замкнута - меняем соотвествующие векторы и матрицу трансп. расходов

if a.sum() > b.sum():

b = np.hstack((b, [a.sum() - b.sum()]))

c = np.hstack((c, np.zeros(len(a)).reshape(-1, 1)))

elif a.sum() < b.sum():

a = np.hstack((a, [b.sum() - a.sum()]))

c = np.vstack((c, np.zeros(len(b))))

m = len(a)

n = len(b)

x = np.zeros((m, n), dtype=int) # создаем матрицу для x и заполняем нулями

funk = 0

while True:

c\_min = np.zeros((m,n))

for i in range(m):

for j in range(n):

c\_min[i, j] = (c[i, j]\*min(a[i], b[j])) # составляем матрицу суммарных расходов

i, j = ij(c\_min) # определяем индексы минимального элемента составленной матрицы суммарных расходов

x\_ij = int(min(a[i], b[j]))

x[i, j] = x\_ij # добавляем элемент x\_ij в матрицу x

funk += int(c\_min[i, j]) # добавляем x\_ij в итоговую функцию

a[i] -= x\_ij #

b[j] -= x\_ij # обновляем векторы a и b

if print\_:

print('c\_min:')

print(c\_min.astype(int))

print('a: ', a)

print('b: ', b)

print()

if len(c\_min[c\_min>0])==1: # повторяем до сходимости метода

break

return x, funk # возращаем матрицу x и целевую функцию

Метод северо-западного угла

В этом случае сначала заполняется клетка в верхнем левом углу, затем следующая клетка справа и т. д., пока не заполнится вся строка. Затем мы переходим ко второй строке и снова заполняем ее слева направо и так далее.

Недостатком данного метода, является его низкая эффективность. Сформированный с его помощью план в большинстве случаев не является оптимальным.

def sev\_zap(a\_, b\_, c\_):

a = np.copy(a\_)

b = np.copy(b\_)

c = np.copy(c\_)

# Проверяем условие замкнутости:

if a.sum() > b.sum():

b = np.hstack((b, [a.sum() - b.sum()]))

c = np.hstack((c, np.zeros(len(a)).reshape(-1, 1)))

elif a.sum() < b.sum():

a = np.hstack((a, [b.sum() - a.sum()]))

c = np.vstack((c, np.zeros(len(b))))

m = len(a)

n = len(b)

i = 0

j = 0

funk = 0

x = np.zeros((m, n), dtype=int)

while (i<m) and (j<n): # повторяем цикл до сходимости метода

x\_ij = min(a[i], b[j]) # проверяем минимальность a\_i и b\_j

funk += c[i, j]\*min(a[i], b[j]) # записываем в итоговую функцию элемент трат

a[i] -= x\_ij #

b[j] -= x\_ij # обновляем векторы a и b

x[i, j] = x\_ij # добавляем элемент x\_ij в матрицу x

if a[i]>b[j]: # делаем сдвиги при выполнении условий

j += 1

elif a[i]<b[j]:

i += 1

else:

i += 1

j += 1

return x, funk # возращаем матрицу x и целевую функцию

Метод потенциалов

Данный метод является наиболее оптимальным и подходящим для большинства случаев транспортных задача.

Его суть заключается в нахождении опорного плана и проверки его на оптимальность, то есть на минимальные затраты на перевозку грузов. Если план оптимален, то решение найдено, в противном случае происходит улучшение плана столько раз, сколько потребуется, пока не будет найден оптимальный план.

# Для метода потенциалов потребуется матрица дельт

# На вход она получает x - матрицу одного из опорных методов

def delta(a, b, c, x):

# Проверяем условие замкнутости:

if a.sum() > b.sum():

b = np.hstack((b, [a.sum() - b.sum()]))

c = np.hstack((c, np.zeros(len(a)).reshape(-1, 1)))

elif a.sum() < b.sum():

a = np.hstack((a, [b.sum() - a.sum()]))

c = np.vstack((c, np.zeros(len(b))))

m = len(a)

n = len(b)

u = np.zeros(m)

v = np.zeros(n)

for i in range(m):

for j in range(n):

if x[i, j] != 0: # если элемент матрицы x не равен 0, расчитываем для данных индексов векторы u и v

if v[j] != 0:

u[i] = c[i, j]-v[j]

else:

v[j] = c[i, j]-u[i]

delta = np.zeros((m, n))

for i in range(m):

for j in range(n):

delta[i, j] = u[i] + v[j] - c[i, j] # расчитываем элемент дельта матрицы

return delta

# Функция возвращает матрицу системы ограничений

def prepare(a, b):

m = len(a)

n = len(b)

h = np.diag(np.ones(n))

v = np.zeros((m, n))

v[0] = 1

for i in range(1, m):

h = np.hstack((h, np.diag(np.ones(n))))

k = np.zeros((m, n))

k[i]=1

v = np.hstack((v, k))

return np.vstack((h, v)).astype(int), np.hstack((b,a))

# Метод потенциалов

def potenz(a\_, b\_, c\_):

a = np.copy(a\_)

b = np.copy(b\_)

c = np.copy(c\_)

# Проверяем условие замкнутости:

if a.sum() > b.sum():

b = np.hstack((b, [a.sum() - b.sum()]))

c = np.hstack((c, np.zeros(len(a)).reshape(-1, 1)))

elif a.sum() < b.sum():

a = np.hstack((a, [b.sum() - a.sum()]))

c = np.vstack((c, np.zeros(len(b))))

m = len(a)

n = len(b)

A\_eq, b\_eq = prepare(a, b)

res = linprog(c.reshape(1, -1), A\_eq=A\_eq, b\_eq=b\_eq, bounds=(0, None), method='simplex')

return res.x.astype(int).reshape(m, n), res.fun.astype(int) # возращаем матрицу x и целевую функцию