Федеральное государственное бюджетное образовательное учереждение
высшего
профессионального образования
«Уфимский государственный авиационный технический университет»

Пояснительная записка к курсовой работе
по дисциплине: «Методы Оптимизации»
на тему: «Задача о рюкзаке: Задача об отправке грузов».

Выполнил:
Пахтусов Н. Г., ПРО-306
Проверила:
Валеева А. Ф.

Уфа, 2015 г.

Содержание

Введение

Для множества задач в прикладной математике нахождение решения прямым перебором за приемлимое время невозможно. К таким задачам относится, например, класс NP-полных задач.

Одной из задач этого класса является так называемая «Задача о рюкзаке». Задача о рюкзаке – одна из NP-задач комбинаторной оптимизации. Своё название она получила от максимизационной задачи укладки как можно большего числа ценных вещей в рюкзак при условии, что вместимость рюкзака ограничена. С различными вариациями задачи о ранце можно столкнуться в экономике, прикладной математике, криптографии, генетике и логистике.

В работе рассмативется одна из разновидностей этой задачи – «Задача об отправке грузов».

1. Постановка задачи

Пусть существует некоторое *количество* авиалайнеров и некоторое *количество* контейнеров. У каждого контейнера есть свой *вес*, а у авиалайнеров есть *ограничение по суммарному весу* контейнеров.

Пусть также различна *выгода* от отправки различными авиалайнерами одного и того же контейнера.

Задача состоит в том, чтобы перевезти контейнеры на авиалайнерах с *максимальной выгодой*.

2. Математическая модель

Пусть $I = \{1, \cdots, n\}$ – авиалайнеры, $J = \{1, \cdots, m\}$ – контейнеры.

p_{ij} – доход от доставки авиалайнером i контейнера j.

w_j – вес контейнера j.

c_i – вместимость авиалайнера i.

$x_{ij} \in \{0, 1\}$ – количество контейнеров j в авиалайнере i.

Таким образом, необходимо найти:

$$\sum_{i=0}^{n} \sum_{j=0}^{m} p_{ij} x_{ij} \to max$$

При ограничениях:

$$\sum_{i=0}^{n} x_{ij} \leq 1, j \in J$$

$$\sum_{j=0}^{m} w_j x_{ij} \leq c_i, i \in I.$$

3. Методы решения задачи

Для решения задач о рюкзаке используется несколько различных эвристических и оптимизационных. Рассмотрим некоторые из них.

3.1 Полный перебор

Временная сложность алгоритма O(N!), т.е он работоспособен для небольших значений N. С ростом N задача становится неразрешимой данным методом за приемлемое время.

3.2 Метод ветвей и границ

Метод ветвей и границ (англ. *branch and bound*) – общий алгоритмический метод для нахождения оптимальных решений различных задач оптимизации, особенно дискретной и комбинаторной оптимизации. По существу, метод является вариацией полного перебора с отсевом подмножеств допустимых решений, заведомо не содержащих оптимальных решений.

Общая идея метода может быть описана на примере поиска минимума функции `f(x)` на множестве допустимых значений переменной `x`. Функция `f` и переменная x могут быть произвольной природы. Для метода ветвей и границ необходимы две процедуры: ветвление и нахождение оценок (границ).

Процедура ветвления состоит в разбиении множества допустимых значений переменной x на подобласти (подмножества) меньших размеров. Процедуру можно рекурсивно применять к подобластям. Полученные подобласти образуют дерево, называемое деревом поиска или деревом ветвей и границ. Узлами этого дерева являются построенные подобласти (подмножества множества значений переменной `x`).

Процедура нахождения оценок заключается в поиске верхних и нижних границ для решения задачи на подобласти допустимых значений переменной `x`.

В основе метода ветвей и границ лежит следующая идея: если нижняя граница значений функции на подобласти `A` дерева поиска больше, чем верхняя граница на какой-либо ранее просмотренной подобласти `B`, то `A` может быть исключена из дальнейшего рассмотрения (правило отсева).

Если нижняя граница для узла дерева совпадает с верхней границей, то это значение является минимумом функции и достигается на соответствующей подобласти.

3.3 Жадный алгоритм

Жадный алгоритм (англ. *Greedy algorithm*) – алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Согласно жадному алгоритму предметы сортируются по убыванию стоимости единицы веса каждого. В рюкзак последовательно складываются самые дорогие за единицу веса предметы из тех, что помещаются внутри.

Сложность сортировки предметов $O(N \log_2(N))$. Далее происходит перебор всех `N` элементов.

3.4 Генетический алгоритм

Содержимое рюкзака представляется в виде хромосом или бинарных строк, i-й бит которых равен единице в случае наличия предмета в рюкзаке, нулю – в случае его отсутствия. Задается целевая функция S – вместимость рюкзака.

Отбор осуществляется следующим образом.

Выбирается произвольная хромосома. Пусть $L_{max} = max(S, S'' - S)$ максимальное расхождение между целевой функцией и хромосомой. S'' суммарный вес всех предметов, входящих в рюкзачный вектор. S' – вес рюкзака при выбранной хромосоме.

Если $S' \leq S$, о хромосома оценивается числом

$q = 1 - \sqrt{\frac{|S'-S|}{S}}$

Иначе:

$q = 1 - \sqrt{\frac{|S'-S|}{L_{max}}}$

По этому числу осуществляется отбор хромосом.

Алгоритм прерывается после заданного числа итераций.

Генетический алгоритм не гарантирует нахождения оптимального решения, однако показывает хорошие результаты за меньшее время по сравнению с другими алгоритмами.

4. Алгоритм решения задачи

Для решения задачи был выбран «**Жадный алгоритм**».

Введём некоторые структуры данных, которые будут использоваться в алгоритме:

- `Cargo` – структура, использующаяся для представления одного груза, с полями обозначающими информацию о том, был ли он выбран и его вес w_j.

- `Knapsack` – структура, использующаяся для представления одного вместилища, с полем-списком, хранящим некоторые значения $j \in J$ и означающим принадлежность элемента j данному контейнеру и полем-значением, означающим оставшийся запас возможного веса c_i.

- `Cost` – структура, использующаяся для хранения стоимости. Она хранит следующие значения: $i \in I$ и $j \in J$, которые означают i вместилище и j груз, а также выгоду отправки p_{ij}.

Таким образом, алгоритм будет состоять из следующих шагов:

1. Сформировать из входных данных массивы структур `Cargo`, `Knapsack` и `Cost` длин i, j и $i \times j$ соответственно.

2. Отсортировать массив структур `Cost` по стоимости в порядке убывания.

3. Повторять для каждого `Cost` из массива:

```
1. i = информация из Cost о номере вместилища;
2. j = информация из Cost о номере груза;
3. Если Cargo[j] ещё не использовался:
4.     Если w = вес Cargo[j] =< p = оставшееся место Knapsack[i]:
5.         положить j в список принадлежности Knapsack[i];
6.         уменьшить оставшееся место p в Knapsack[i] на w;
7.         пометить Cost[j] как использовавшийся
8.    Иначе:
9.         продолжить цикл;
10. Иначе:
11.    продолжить цикл;
```

4. Напечатать результат.

5. Программная реализация алгоритма

Для реализации был выбран язык программирования **Haskell**.

Введём некоторые абстракции над типами данных и создадим необходимые программные представления для структур `Cargo`, `Knapsack` и `Cost`:

```
type W = Int
type P = Int
type I = Int
type J = Int
type Cargo = (J, Bool, W)
type Knapsack = (I, [J], W)
type Cost = (I, J, P)
```

Создадим некоторые вспомогательные функции:

- Функции для удобного взятия первого, второго и третьего элемента кортежа соответственно:

```
mfst (x, _, _) = x
msnd (_, x, _) = x
mthd (_,_,x) = x
```

- Функция для обновления элемента с индексом `i` на элемент `el`:

```
substitude i el xs = take i xs ++ [el] ++ drop (i + 1) xs
```

- Функция для создания списка элементов `Cargo` из входного списка весов `w`:

```
makeCargo = makeCargo' 0
        where
            makeCargo'::Int -> [W] -> [Cargo]
            makeCargo' _ [] = []
            makeCargo' n (w:ws) = (n, False, w) : (makeCargo' (n + 1) ws)
```

- Функция для создания списка элементов `Knapsack` из входного списка весов `c`:

```
makeKnapsack = makeKnapsack' 0
        where
            makeKnapsack'::Int -> [C] -> [Knapsack]
            makeKnapsack' _ [] = []
            makeKnapsack' n (c:cs) = (n, [], c) : (makeKnapsack' (n + 1) cs)
```

- Функция для создания списка `Cost` из матрицы выгоды:

```
makeCost::[[P]] -> [Cost]
makeCost = makeCostI 0
            where
        makeCostI _ [] = []
        makeCostI i (x:xs) = (makeCostIJ i 0 x) ++ (makeCostI (i + 1) xs)
                where
            makeCostIJ i j [] = []
            makeCostIJ i j (w:ws) = (i, j, w) : (makeCostIJ i (j + 1) ws)
```

Реализуем основной алгоритм:

```
findSoluton cargo ks ((i,j,p):cs)
                    | msnd (cargo !! j) = findSoluton cargo ks cs
                    | (mthd (ks !! i)) > mthd (cargo !! j) = let
                            (_, _, cargoW) = cargo !! j
                            newCargo = substitude j (0, True, 0) cargo
                            (ki, clst, w) = ks!!i
                            newKnapsack = substitude i (ki, (j:clst), (w - cargoW)) ks
                            in findSoluton newCargo newKnapsack cs
                    | otherwice = findSoluton cargo ks cs
```