Душкин Роман Викторович
Автор нескольких книг по функциональному программированию.

Задача о ранце

В статье предлагается рассмотрение одной из классических оптимизационных задач – задача о ранце. Даётся математическая формулировка задачи и её решения, предлагается реализация этого решения на функциональном языке программирования Haskell, а также рассматривается пример использования разработанной обобщённой функции для поиска решения конкретной проблемы.

Введение

Научно-технический прогресс и развитие методов прикладной математики дали начало новой дисциплине, которая занимается разработкой и применением методов нахождения оптимальных решений на основе математического и статистического моделирования и различных эвристических техник в разных областях человеческой деятельности. Данная дисциплина получила название «исследование операций» (в российских высших учебных заведениях может преподаваться под названием «методы оптимизации» или схожими).

Особый толчок к развитию методов оптимизации дала Вторая мировая война. Во время неё исследование операций стало широко применяться для планирования боевых действий. Например, для военно-воздушных сил союзников учёными были выработаны рекомендации, которые позволили увеличить эффективность бомбометания в четыре раза.

После войны группы специалистов по исследованию операций продолжили свою работу в вооружённых силах США и Великобритании. Публикация ряда результатов в от-

крытой печати вызвала всплеск общественного интереса к этому научному направлению. Возникла тенденция к применению методов исследования операций в коммерческой деятельности, в целях реорганизации производства, перевода промышленности на мирные рельсы. В СССР исследование операций также получило широкое распространение (вплоть до того, что советский математик и экономист Л.В. Канторович получил в 1975 году Нобелевскую премию по экономике за вклад в теорию оптимального распределения ресурсов).

Характерными особенностями исследования операций являются системный подход к поставленной проблеме и анализ. Системный подход – главный методологический принцип исследования операций. Любая задача, которая решается, должна рассматриваться с точки зрения влияния на функционирование системы в целом. Для исследования операций характерно то, что при решении каждой проблемы могут возникать новые задачи. Важная особенность исследования операций – стремление найти оптимальное решение поставленной задачи (принцип «оптимальности»). Однако на практике такое решение часто бывает невозможно найти по следующим причинам:

1) отсутствие методов, дающих возможность найти глобальное оптимальное решение задачи;

2) ограниченность существующих ресурсов (к примеру, ограниченность машинного времени ЭВМ), что делает невозможным реализацию точных методов оптимизации.

В таких случаях обычно ограничиваются поиском не оптимальных, а достаточно хороших с точки зрения практики решений. Приходится искать компромисс между эффективностью решений и затратами на их поиск. Исследование операций даёт инструмент для поиска таких компромиссов.

Исследование операций тесно связано с теорией управления, системным анализом, математическим программированием, теорией игр, теорией оптимальных решений, эвристическими подходами и методами искусственного интеллекта. В качестве канонических задач, которые решаются методами оптимизации, можно отметить задачу коммивояжёра, транспортную задачу, задачу об упаковке в контейнеры, задачу о ранце и др. О последней как раз и пойдёт речь в настоящей статье.

1. Классическая задача

Задача о ранце (бывает, что говорят «задача о рюкзаке») является классической задачей исследования операций и комбинаторной оптимизации. Задача получила своё название от максимизационной проблемы укладки как можно большего числа *нужных* вещей в рюкзак (ранец) при условии, что общий объём (или вес) всех предметов *ограничен*. Подобные задачи часто возникают в прикладной математике, экономике, криптографии.

В общем виде задачу можно сформулировать так: из неограниченного (или ограниченного) множества предметов со свойствами «стоимость» и «вес» требуется отобрать некое число предметов таким образом, чтобы получить максимальную суммарную стоимость при одновременном соблюдении ограничения на суммарный вес.

Само собой разумеется, что к данной классической формулировке задачи могут сводиться многие иные задачи разных размерностей. В качестве стоимости и веса могут ис-

пользоваться совершенно различные характеристики и даже их комбинации. В этом вопросе необходимо лишь наличие некоторого преобразования (функции), которая позволит свести требуемую задачу к классической.

В принципе существуют две формулировки классической задачи о ранце:

1) каждый предмет из множества можно выбирать неограниченное количество раз (пока есть возможность удовлетворять ограничение на вес);

2) каждый предмет можно использовать только один раз.

Сама но себе задача о ранце является NP-полной задачей, то есть такой, время работы алгоритма для решения которой существенно зависит от размера входных данных, при этом если предоставить алгоритму некоторые дополнительные сведения, то он сможет за время, не превосходящее некоторого многочлена от размера входных данных, решить задачу. Дополнительные сведения в данном случае называются «свидетелем решения».

В свою очередь, это означает, что задачу можно решить при помощи методов динамического программирования. Данный раздел исследования операций позволяет решать задачи с оптимальной структурой и перекрывающимися подзадачами, при этом метод решения основан на постепенном приближении к конечному результату посредством его получения через уже решённые подзадачи. Слово «программирование» в наименовании дисциплины не имеет ничего общего с написанием кода, поскольку обозначает оптимальную последовательность действий для поиска решения задачи (можно сравнить с такими терминами, как «линейное программирование», «математическое программирование»).

Итак, формулировка задачи о ранце с возможностью неограниченного выбора звучит так. По заданному набору из n предметов со стоимостями $v_1, v_2 \dots v_n$ и весами w_1, $w_2 \dots w_n$ необходимо найти такой поднабор (с учётом того, что можно брать любой предмет неограниченное число раз), что его стоимость будет максимальной среди всех поднаборов с общим весом не более W.

Решение такой задачи несложно. Пусть K_w – максимальная стоимость, которую можно набрать при весе не более w. Следующее рекуррентное соотношение позволяет найти решение:

1) $K_0 = 0$ (при весе не более 0 максимальная стоимость равна 0) – базис рекурсии;

2) $K_w = \max\left\{K_{w-w_i} + v_i\right\}_{i=1}^{n}, \quad w_i \le w,$

где n – размер набора – шаг рекурсии.

С другой стороны, задача о ранце с возможностью использовать любой предмет из набора не более одного раза формулируется сле-

дующим образом. Пусть $K_{w,i}$ – максимальная стоимость, которую можно набрать при весе не более w среди i первых предметов. Следующее рекуррентное соотношение находит решение:

1) $K_{0,1} = 0,\ 0 \le i \le n;$

2) $K_{w,0} = 0,\ 0 \le w \le W;$

3) $K_{w,i} = \max\left\{K_{w,\,i-1}, K_{w-w_i,\,i-1} + v_i\right\},$

$0 \le w \le W,\ w_i \le w;$

4) $K_{w,i} = K_{w,i-1}$, если $w_i > w$ (невозможно добавить элемент этого веса).

Собственно, на рис. 1 идеографически проиллюстрирована такая задача.

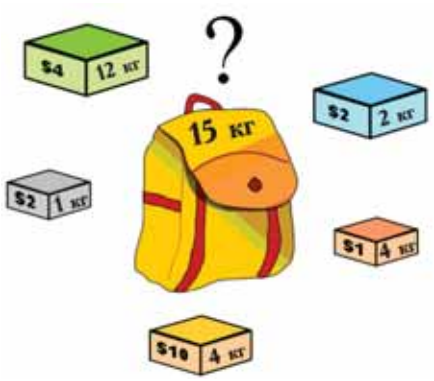

Рис. 1. Иллюстрация задачи о ранце – исходные данные

2. Реализация решения на языке Haskell

Классическая математическая задача так и будет оставаться «интересной абстракцией», если не реализовать алгоритм для решения, что называется, в коде. Как обычно, для иллюстрации методов решения описанной задачи можно воспользоваться функциональным подходом и реализовать возможный алгоритм на языке программирования Haskell.

Реализацию алгоритма необходимо начать с небольшого проектирования тех программных сущностей, которые должны быть определены в программе. Дело в том, что математическое описание задачи говорит о произвольных объектах, которые могут фигурировать в процессе упаковки в ранец. Для таких объектов необходимо иметь две операции – получение стоимости и получение веса. На основании этих операций можно произвести вычисления и найти решение.

Итак, язык Haskell имеет механизм для абстракции подобных требований – это классы, описывающие необходимые методы (подробно о классах в языке Haskell можно ознакомиться в статье: Объектно-ориентированное и функциональное программирование // Потенциал, 2007, №2). Так что описать класс для объектов, которые могут быть упакованы в ранец, можно так:

```
class KnapsackItem a where
  cost   :: a -> Integer
  weight :: a -> Integer
```

Любой тип данных, имеющий экземпляр класса **KnapsackItem**, можно будет использовать в функциях, которые производят поиск решения в задаче о ранце. Конечно, и соответствующие функции необходимо реализовать с учётом этого требования. Но это будет сделать уже совсем несложно.

В определении указанного класса имеется одна тонкость. Функции **cost** («стоимость») и **weight** («вес») возвращают целое число неограниченного размера. В классической формулировке задачи ничего не говорится о типе свойств «стоимость» и «вес», а

потому их типами могут быть целые числа, действительные числа, да и вообще произвольные типы, для которых определены некоторые операции: для стоимости – сложение, для веса – сравнение (это следует из формул решения задачи, представленных в предыдущем разделе).

К сожалению, не так просто реализовать данную возможность на языке Haskell, поскольку стандарт языка Haskell-98 не позволяет сделать это удобным способом. Тем не менее, некоторые расширения языка позволяют написать не менее эффективное определение. Например:

```
class KnapsackItem a where
  cost   :: Num b => a -> b
  weight :: Ord b => a -> b
```

Или:

```
class Num b => KnapsackItem a b where
  cost   :: a -> b
  weight :: a -> b
```

В первом из этих случаев в сигнатуры методов вводятся ограничения на типы возвращаемых значений (при этом методы могут вообще возвращать значения различных типов – одинаковое обозначение **b** не должно смущать). Метод **cost** должен возвращать значение, над которым можно производить арифметические операции (класс **Num**). Соответственно, метод **weight** возвращает значение, которое можно сравнивать (класс **Ord**). Но стандарт Haskell-98 не разрешает вводить ограничения на типы в сигнатуры методов, ограничение может быть только на параметрическую переменную класса.

Во втором случае класс **KnapsackItem** параметризует два типа, причём на второй наложено ограничение – он должен иметь экземпляр класса **Num** (что автоматически влечёт наличие экземпляра класса **Ord**). Но опять же стандарт Haskell-98 не разрешает использовать многопараметрические классы. Это возможно только в расширениях компилятора GHC.

Использование любого из указанных подходов влечёт необходимость не только подключения того или иного расширения языка Haskell, но и дополнительного удовлетворения требований при написании функций. Таким образом, изначальное определение класса подчиняется стандарту, но вместе с тем позволяет написать простые и понятные функции. Также это определение класса **KnapsackItem** вполне достаточно и не сказывается на общности рассуждений при решении задач.

Теперь можно реализовать непосредственную функцию, которая будет производить поиск решения задачи о ранце. Ради удобства такая функция будет выбирать по одному предмету из заданного списка элементов (неограниченный выбор элементов можно сделать из той же функции, убрав вызов функции удаления элемента из списка **delete**). Собственно, такая функция выглядит следующим образом:

```
knapsack 0 _  = 0
knapsack _ [] = 0
```

```
knapsack w vs | null l    = 0
              | otherwise = maximum l
  where
    l = [knapsack (w - weight v) (delete v vs) + cost v |
         v <- vs,
         weight v <= w]
```

Как видно, два первых клоза определяют базис рекурсии, при этом отслеживать пустой список элементов также надо (математическое описание задачи это подразумевает). Третий клоз наиболее интересен, поскольку определяет шаг рекурсии, и в нём производится динамический перебор и поиск решения. Два выражения охраны определяют результат функции в случае, если список возможных значений, полученных в рекурсивном вызове, пуст и не пуст. В первом случае результатом всё так же должен быть 0, во втором необходимо выбрать максимальное значение, как то предписывает математическое рекуррентное соотношение.

Выражение **l**, которое вынесено в локальное определение в оптимизационных целях (в теле функции оно встречается два раза), как раз и запускает процесс рекурсивного вызова функции **knapsack** для всех элементов входного списка, чьи веса не больше заданного веса. Вычисление основано на генераторе списка (подробно о генерации списков можно ознакомиться в статье: Магические квадраты. Решение переборных задач на языке Haskell // Потенциал, 2007, №6).

Ту же самую функцию можно написать и с префиксным определением локального выражения **l**. В методических целях будет интересно рассмотреть такой вариант:

```
knapsack 0 _   = 0
knapsack _ []  = 0
knapsack w vs = let l = [knapsack (w - weight v)
(delete v vs) + cost v|
                         v <- vs,
                         weight v <= w]
                in if (null l)
                      then 0
                      else maximum l
```

Понятно, что этот вариант ничем не отличается от предыдущего, кроме своей формы.

Всё хорошо, только определённая любым из представленных вариантов функция **knapsack** не имеет большого практического значения. Это связано с тем, что она возвращает лишь максимально возможную стоимость упакованных в ограниченный ранец элементов. Но в прикладных задачах обычно необходимо знать не только такую максимальную стоимость, но и тот набор элементов, который приводит к такой стоимости. Для решения этой задачи функцию придётся слегка преобразовать – она будет возвращать пару, первым значением которой является максимальная стоимость, а вторым – набор элементов:

```
knapsack' 0 vs = (0, [])
knapsack' _ [] = (0, [])
knapsack' w vs = getMaxCost [((fst prevKS) + cost v,
v:(snd prevKS)) |
                              v <- vs,
                              let prevKS = knapsack'
                              (w – weight  v)(delete v vs),
                               weight v <= w]
  where
    getMaxCost []           = (0, [])
    getMaxCost ((c, v):vs) = let (c', v')  = getMaxCost vs
                              in if (c > c')
                                    then (c, v)
                                    else (c', v')
```

Как видно, из-за того, что результатом выполнения функции стала пара элементов, определение сделалось сложнее. Всё по причине того, что пару необходимо собирать непосредственно в генераторе списка (а для этого её необходимо разбирать, но для этого есть стандартные функции **fst** и **snd**), а над каждым элементом пары надо производить свои операции. Над первым – сложение весов, над вторым – добавление элемента в список. Это также влечёт необходимость реализации собственной функции поиска максимума, поскольку стандартная **maximum** работает только со списками сравниваемых элементов. Локальная функция **getMaxCost** ищет максимальную стоимость в списке пар, в которых стоимость стоит на первом месте.

Ну и, собственно, при помощи этих функций теперь можно решать задачи о ранце. Например, та задача, которая показана на рис. 1, кодируется следующими программными сущностями:

```
data KI = KI Integer Integer
  deriving (Eq, Show)

instance KnapsackItem KI where
  cost   (KI c w) = c
  weight (KI c w) = w

example = [KI 4 12, KI 2 2, KI 2 1, KI 1 1, KI 10 4]
```

Здесь тип **KI** содержит в себе два поля – одно для стоимости, второе для веса. Для соответствия требованиям на типы значений, использующихся в функциях **knapsack** и **knapsack'**, необходимо определить экземпляр класса **KnapsackItem**. Функция **example** просто определяет список элементов, которые изображены на рис. 1.

Теперь можно запустить функции на проверку:

```
> knapsack 15 example
15
```

```
>knapsack' 15 example
(15, [KI 10 4, KI 1 1, KI 2 1, KI 2 2])
```

Заключение

Как видно, язык функционального программирования Haskell в очередной раз показал свою близость к математике и простоту в использовании для решения задач. Полученные функции могут быть использованы для различных целей – достаточно лишь закодировать исходные данные в соответствии с требованиями. Так, к примеру, автор использовал функцию **knapsack'** для оптимального распределения музыкальных альбомов в формате MP3 на нескольких компакт-дисках (задача о ранце применялась рекурсивно) для неутомительного прослушивания, при этом критерием стоимости выступала достаточно сложная формула, принимающая во внимание размер альбома, его жанр, субъективную оценку слушателя и некоторые иные факторы.

Таким образом, язык Haskell вполне успешно можно использовать в качестве инструмента быстрой разработки небольших утилит, предназначенных для решения какой-то одной несложной задачи.

Автор выражает признательность своим коллегам – Антонюку Д.А., Отту А.Я., которые помогли своими дельными советами при подготовке статьи к публикации. Также автор с благодарностью примет комментарии и замечания на адрес электронной почты **roman.dushkin@gmail.com**, на который также можно присылать запросы исходных кодов, перечисленных в статье.

Юмор Юмор Юмор Юмор Юмор Юмор

Безошибочное действие

– Я разработал систему, – говорит изобретатель, представляя свой аппарат, – устанавливающую личность человека по его голосу. Принцип действия несложен: анализатор измеряет спектральные характеристики звука, а компьютер по этим данным определяет, кому этот голос принадлежит.

– Давайте попробуем, – предлагает заинтересовавшийся человек. – Что мне нужно сказать?

– Вы должны чётко и ясно назвать свои имя и фамилию.