Введение в функциональное программирование – Лекция 9. Списочные гомоморфизмы. Третья теорема о гомоморфизмах. Технология MapReduce

Олег Смирнов
oleg.smirnov@gmail.com

18 ноября 2011 г.

Содержание

1 Задача о графе связей **2**

2 Параллелизация **4**

3 Технология MapReduce **5**

4 Списочные гомоморфизмы **6**

5 Теоремы о гомоморфизмах **7**

Цель лекции

- Списочные гомоморфизмы
- Третья теорема о гомоморфизмах
- MapReduce как система для параллелизации вычислений

1 Задача о графе связей

Пусть задана корпоративная структура некоторой компании в виде графа связей. Дуга от вершины A к вершине B означает, что сотрудник A подчиняется сотруднику B. Необходимо найти, кто в компании самый главный, т.е. директора.

Для графа без циклов, т.е. для дерева, задачу можно решить топологической сортировкой за $O(|V| + |E|)$ итераций, где $|V|$ – количество вершин, а $|E|$ – количество связей. Топологическую сортировку можно получить, например, из поиска в глубину.

А как быть, если в графе присутствуют циклы? В общем виде, пусть есть граф, где вершины – это люди, а дуги – некоторые связи между ними. Например, в социальной сети дуга может описывать отношение A (Алиса) "добавила в друзья" B (Боба). Задача: выявить наиболее "влиятельных" людей в такой сети. Эта задача имеет массу применений на практике. Например, в рекламе и маркетинге.

Очевидно, что "влиятельность" можно определять разными способами. Можно просто подсчитывать количество друзей. Предположим, что Алису добавило в друзья десять человек, а Боба – всего три. Но у каждого из троих друзей Боба есть ещё по сотне своих друзей. В этом случае наша формула посчитает Алису более "влиятельной", хотя на деле ситуация противоположная.

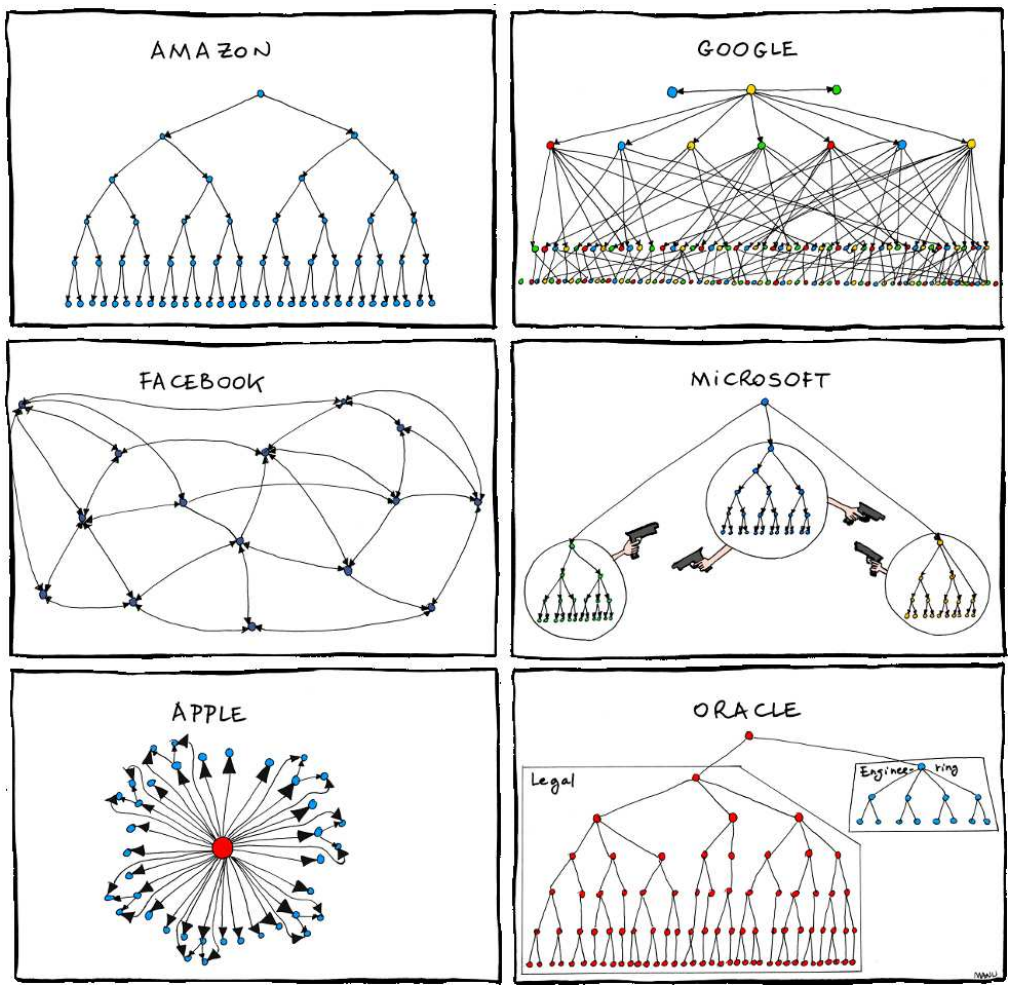
Идея: определим меру влиятельности рекурсивно. Т.е. для некоторого пользователя p определим влиятельность $I(p)$ пропорционально влиятельности его друзей, т.е.

$$I(p) = \alpha(I(f_1) + I(f_2) + \ldots + I(f_k))$$

где $f_1 \cdot f_k$ – друзья p, а $\alpha > 0$ – некая константа.

Представив социальный граф в виде матрицы смежности $A \in \mathbb{R}^{n \times n}$, можно записать формулу:

$$I(p_i) = \alpha \sum_j A_{ij} I(p_j)$$



Корпоративные структуры

Легко видеть, что функция I принимает n возможных значений, т.е. является вектором $\vec{x} \in \mathbb{R}^n$. Пусть $\lambda = \frac{1}{\alpha}$. Тогда

$$\vec{x} = \alpha A\vec{x}$$

$$A\vec{x} = \lambda\vec{x}$$

Проще говоря, задача свелась к поиску собственного вектора $\vec{x}$ для матрицы смежности A для некоторого собственного числа $\lambda > 0$. Можно показать, что искомый вектор соответствует наибольшему из собственных чисел матрицы.

Эта задача называется "eigenvector centrality". В общем случае, "centrality" – это некий (возможно, частичный) порядок на множестве вершин гра-

фа, т.е. функция

$$\mu : V \to \mathbb{R}$$

Сумма всех связей вершины называется “degree centrality”. В нагруженном графе можно взять значение, обратно пропорциональное сумме весов связей – получится “closeness centrality” и т.д.

Разновидностью “eigenvector centrality” является алгоритм PageRank, который лежит в основе поискового движка компании Google [PBMW99]. В его модели используется некоторый гипотетический пользователь, который нажимает на ссылки в браузере в случайном порядке. “Важность” страницы определяется как вероятность попадания на неё этим “случайным” пользователем.

1. Интернет можно представить в виде матрицы смежности, где вершинами будут страницы, а рёбрами – ссылки между ними.

2. Такую матрицу смежности можно нормализовать, разделив значения в каждой строка на сумму значений этой строки, если она не равна нулю. Полученная матрица описывает марковский процесс для нашей модели со “случайным” пользователем.

3. Искомый вектор вероятностей – это “предел” марковского процесса или состояние равновесия.

В практической реализации в формулу добавляют ещё несколько членов: вероятность того, что пользователь остановится в какой-то момент времени (“dumping factor”) и т.п.:

$$PR(p_i) = \frac{1-d}{N} + d \sum \frac{PR(p_j)}{L(p_j)}$$

где A – матрица смежности *входящих* ссылок, а $L(p_i)$ – количество *исходящих* ссылок.

2 Параллелизация

Для вычисления “eigenvector centrality” существует ряд алгоритмов. Самый простой из них – это метод степенных итераций.

Идея: инициализируем вектор $\vec{x}$ каким-то значением (начальное распределение). На каждой итерации вычисляем и нормируем:

$$\vec{x_{k+1}} = \frac{A\vec{x_k}}{||A\vec{x_k}||}$$

Метод сходится, хотя достаточно медленно. Для точности в десять значащих цифр требуется порядка 140 итераций.

Очевидно, что для таких объёмов данных, как матрица смежности Интернет, потребуется параллелизация вычислений.

Можно заметить, что каждая степенная итерация состоит из умножения матрицы размера $n \times n$ на вектор размера $n \times 1$. Её можно представить как n умножений векторов $1 \times n$ на $n \times 1$, каждое из которых можно выполнять независимо. Два вектора можно представить в виде массива пар $2 \times n$, где каждую пару можно перемножить отдельно, а затем просуммировать все результаты.

Таким образом, функция вычисления будет иметь вид:

```
let h = mulPairs >> sumList
let mulPairs = map (fun (a, b) -> a * b)
let sumList = reduce (fun a b -> a + b) 0
```

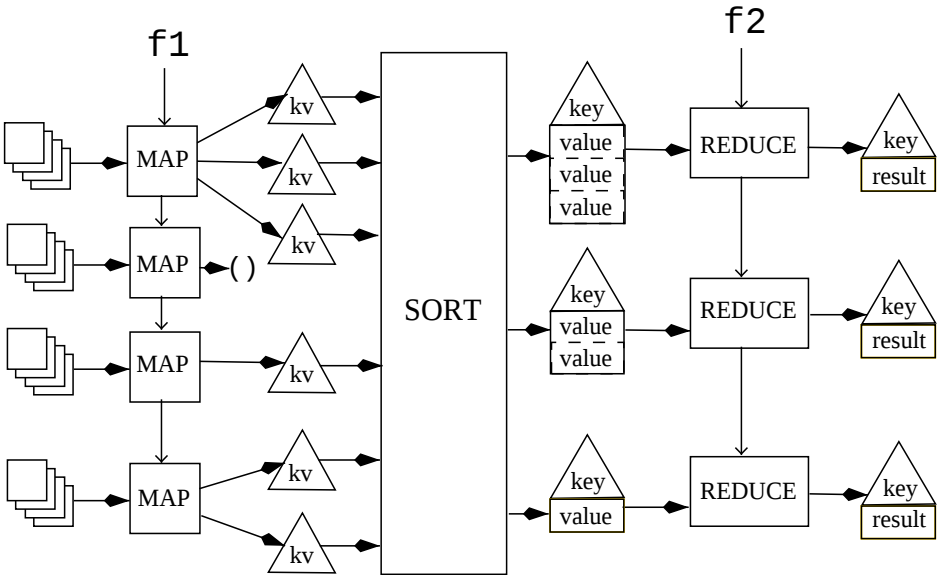
где *reduce* – это свёртка для ассоциативной операции $(+)$, т.е. неважно, левая или правая.

Рассмотрим другую задачу: для массива слов вычислить частоту употребления каждого слова. Предполагаем, что размер данных таков, что для решения опять требуется параллелизация. Эту задачу также можно представить в виде комбинации функций *map* и *reduce*:

- каждое слово w из исходного массива представляется в виде пары $(w, 1)$.
- пары сортируются, используя первый элемент в качестве ключа.
- все пары с одинаковым ключём подаются на вход функции, которая просто суммирует вторые элементы, вычисляя частоту употребления слов.

3 Технология MapReduce

Как оказалось, очень большое количество задач над большими объёмами данных можно представить в виде комбинации функций *map* и *reduce*. В 2004-м году два инженера компании Google выступили на конференции с докладом о технологии MapReduce [DGI04]. Они разработали систему, позволившую программистам отвлечься от побочных задач по управлению данными, обработке ошибок и т.д., и сосредоточиться на разработке алгоритмов. Входные данные поступают в систему в виде пар ключ-значение (k_1, v_1). Функция $f1$ (*mapper*) превращает кажду пару в



Система MapReduce

список пар-промежуточных результатов (k_2, v_2). Функция *sort* сортирует промежуточные пары по ключам k_2. На вход функции $f2$ (*reducer*) подаются пары вида $(k_2, list(v2))$, где второй элемент – это *список* значений, соотвестствующих ключу k_2. Её результат – список выходных значений v_3. Таким образом, задача программиста – это разработать две функции:

```
mapper (k1, v2) -> list((k2, v2))
reducer (k2, list(v2)) -> list(v3)
```

Технология *MapReduce* является проприетарной разработкой компании *Google*. Однако существует открытая реализация этой идеи в системе *Hadoop*. Именно кластеры *Hadoop* используются для обработки данных в таких компаниях, как *Yahoo*!, *Facebook*, *IBM* и другие.

Мы показали, что как минимум две задачи имеют структуру, подходящую для систему *MapReduce*. Это задача умножения матрицы на вектор и задача подсчёта частоты слов. Можно ли использовать *MapReduce* для произвольной задачи?

4 Списочные гомоморфизмы

Рассмотрим несколько функций:

- *map*

- $length$
- sum
- min
- all

Аргументом каждой из функций является список значений. Можно заметить, что если разбить список-аргумент на несколько частей, то каждую из этих функций можно вычислить от подсписка, а затем объединить вычисленные значения. Другими словами, эти функции можно представить в виде

$$h(x +\!\!+ y) = h\ x \odot h\ y \tag{1}$$

где $+\!\!+$ – операция конкатенации списков, а $\odot$ – некоторый бинарный оператор. Очевидно, что $\odot$ должен быть ассоциативным, т.к. ассоциативна конкатенация $+\!\!+$. Пример:

$$sum(x +\!\!+ y) = sum\ x + sum\ y$$

$$length(x +\!\!+ y) = length\ x + length\ y$$

С другой стороны, функцию вычисления наибольшего отсортированного префикса lsp нельзя представить в таком виде, т.к. $lsp\ x$ и $lsp\ y$ недостаточно для вычисления $lsp(x +\!\!+ y)$. Эта функция является *левосторонней*, т.к. может быть вычислена только *справа налево*. Аналогично, *правосторонней* функцией будем называть ту, которая вычисляется *слева направо*. Покажем, что если функция является одновременно и левосторонней и правосторонней, то она удовлетворяет условию 1. Такие функции называются *гомоморфизмами*, а описанное условие – третьей теоремой о гомоморфизмах.

5 Теоремы о гомоморфизмах

Определение: списочным гомоморфизмом называется функция h такая, что

$$h(x +\!\!+ y) = h\ x \odot h\ y$$

где $\odot$ – ассоциативная операция, а $h\ []$ – единица для $\odot$, т.к. $[]$ является единицей для $+\!\!+$. Т.е.

$$h = hom\ (\odot)\ f\ e$$

где e – начальное значение (единица), $[\cdot]\ a = [a]$ и $h \circ [\cdot] = f$.

Например, можно определить:

$$sum = hom\ (+)\ id\ 0$$

$$length = hom\ (+)\ one\ 0, \text{ где } one\ a = 1$$

Левосторонняя функция h относительно бинарной операции $\oplus$ определяется так:

$$h([a] +\!\!+\ y) = a \oplus h\ y$$

Проще говоря, функцию h можно записать в виде правой свёртки:

$$h = foldr\ (\oplus)\ e, \text{ где } h[] = e$$

Например, для lsp:

$$a \oplus [] = [a]$$

$$a \oplus (b : x) = a : b : x, \text{ если } a \leqslant b$$

$$a \oplus (b : x) = [a], \text{ иначе}$$

Тогда

$$lsp = foldr\ (\oplus)\ []$$

Симметрично, *правосторонняя* функция h определяется как

$$h(x +\!\!+\ [a]) = h\ x \otimes a$$

$$h = foldl\ (\otimes)\ e$$

Важно заметить, что операции $\oplus$ и $\otimes$ не обязаны быть ассоциативными. Кроме того, можно показать, что

$$hom\ (\odot)\ id\ e \text{ – это } \textit{редукция}$$

$$hom\ (+\!\!+\ )\ ([\cdot] \circ f)\ [] \text{ – это } \textit{проекция} \text{ map f}$$

Теорема 1. Каждый гомоморфизм h может быть представлен в виде композиции редукции и проекции:

$$hom\ (\odot)\ f\ e = hom\ (\odot)\ id\ e \circ map\ f$$

Теорема 2. Каждый гомоморфизм является правосторонней и левосторонней функцией:

$$hom\ (\odot)\ f\ e = foldr\ (\oplus)\ e, \text{ где } a \oplus s = f\ a \odot s$$

$$hom\ (\odot)\ f\ e = foldl\ (\oplus)\ e, \text{ где } r \otimes a = r \odot f\ a$$

Теорема 3. Если функция h – одновременно *левосторонняя* и *правосторонняя*, то h – гомоморфизм.

Схема доказательства:

1. Лемма 1: Для любой функции h существует, возможно, частично вычислимая функция g такая, что $h \circ g \circ h = h$.

2. Лемма 2: Если из $h\ v = h\ x \wedge h\ w = h\ y$ следует $h\ (v +\!\!+\ w) = h\ (x +\!\!+\ y)$ для любых списков v, w, x, y, то h – гомоморфизм. Доказывается через определение оператора $\odot$: $t \odot u = h(g\ t +\!\!+\ g\ u)$

3. Доказательство теоремы: рассматриваем h поочерёдно как левостороннюю и правостороннюю функцию. Показываем, что она удовлетворяет условию второй леммы, а, следовательно, является гомоморфизмом.

Практическим примером третьей теоремы является сортировка слиянием (Mergesort) [GG95].

Понятие списочных гомоморфизмов было введено профессором информатики и директором компьютерной лаборатории Оксфордского университета Ричардом Бёрдом [Bir87].

Ричард Бёрд
(род. 1943)

Существует ряд исследований, обобщающих понятие списочных гомоморфизмов на древовидные структуры.

Список литературы

[Bir87] R. S. Bird. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*, pages 5–42, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[DGI04] Jeffrey Dean, Sanjay Ghemawat, and Google Inc. Mapreduce: simplified data processing on large clusters. In *In OSDI'04:*

Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation. USENIX Association, 2004.

[GG95] Jeremy Gibbons and Jeremy Gibbons. The third homomorphism theorem, 1995.

[MMHT09] Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. *SIGPLAN Not.*, 44:177–185, January 2009.

[MMM+] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Automatic inversion generates divide-and-conquer parallel programs.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web, 1999.

[RL08] Ralf and Lämmel. Google's mapreduce programming model — revisited. *Science of Computer Programming*, 70(1):1 – 30, 2008.