Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №8 по курсу «Дискретный анализ»

Студент: К. С. Саженов
Преподаватель: И. Н. Симахин
Группа: М8О-308Б-19
Дата:
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №8

Задача: Разработать жадный алгоритм решения задачи, определяемой своим вариантом. Доказать его корректность, оценить скорость и объём затрачиваемой оперативной памяти.

Реализовать программу на языке C или C++, соответсвующую построенному алгоритму. Формат входных и выходных данных описан в варианте задания.

Вариант 4: Откорм бычков. Бычкам дают пищевые добавки, чтобы ускорить их рост. Каждая добавка содержит некоторые из N действующих веществ. Соотношенияколичеств веществ в добавках могут отличаться. Воздействие добавки определяется как $c_1a_1 + c_2a_2 + \ + c_Na_N$, где a_i количество i-го вещества в добавке, c_i — неизвестный коэффициент, связанный с веществом и не зависящий от добавки. Чтобы найти неизвестные коэффициенты c_i, Биолог может измерить воздействие любой добавки, использовав один её мешок. Известна цена мешка каждой из $M(MN)$ различных добавок. Нужно помочь Биологу подобрать самый дешевый наобор добавок, позволяющий найти коэффициенты c_i. Возможно, соотношения веществ в добавках таковы, что определить коэффициенты нельзя.

Входные данные: В первой строке текста — целые числа M и N ; в каждой из следующих M строк записаны N чисел, задающих соотношение количеств веществ в ней, а за ними — цена мешка добавки. Порядок веществ во всех описаниях добавок один и тот же, все числа — неотрицательные целые не больше 50.

1 Описание

Жадный алгоритм [1] заключается в принятии на каждом этапе локально оптимальных решений, предполагая, что конечное решение также окажется оптимальным. Чтобы жадный алгоритм работал, необходимо, чтобы последовательные локальные оптимальные выборы по итогу давали оптимальное глобальное решение. К тому же, оптимальное решение задачи должно содержать в себе оптимальные решения подзадач. Для теоретических выкладок иногда используются матроиды: если показать, что объект является матроидом, то можно показать, что жадный алгоритм будет работать корректно. Свойства матроида примерно пересекаются с изложенными выше свойствами.

Этапы построения жадного алгоритма:

1. Привести задачау оптимизации к виду, когда после сделанного выбора остаётся решить только одну подзадачу.
2. Доказать, что всегда существует такое оптимальное решение исходной задачи, которое можно получить путём жадного выбора, так, что такой выбор всегда допустим.
3. Продемонстрировать оптимальную структуру, показав, что после жадного выбора остаётся подзадача, обладающая тем свойством, что при объединении оптимального решения подзадачи со сделанным жадным выбором приводит к глобальному оптимальному решению исходной задачи.

2 Исходный код

Заметим, что данная задача — не что иное, как нахождение решения системы линейных алгебраических уравнений(сокр. СЛАУ), но в более упрощенном виде.

Приведем матрицу к ступенчатому виду с помощью метода Гаусса. Алгоритм выглядит следующим образом:

1. Среди элементов 1-го столбца выбираем ненулевой, чья строка имеет наименьшую стоимость
2. Меняем ее местами с 1 строкой(если она сама не 1-ая)
3. Вычитаем первую строку из остальных строк, домножив её на величину, равную отношению первого элемента каждой из этих строк к первому элементу первой строки(то есть подбираем такой коэффициент, при котором, умноженная строка на этот коэффициент, обнулит свой 1-й элемент)
4. Забываем про 1 столбец и про 1 строку(как бы "обрезаем"матрицу сверху и слева). Но только при подсчете, то есть мы не удаляем их по-настоящему
5. Повторяем $1 - 4$ пункты, пока матрица не пустая

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <set>
#include <limits>

using data_type = double;
using cost_type = std::uint_fast32_t;

const cost_type MAX_MIN_COST = 50;

struct Row: std::vector<data_type>{
    std::size_t num;
    cost_type cost;
};

int main() {
    std::size_t m, n;
    std::cin >> m >> n;
    std::vector<Row> matrix(m);

    std::set<std::size_t> ans;

    for (std::size_t i(0); i < matrix.size(); ++i) {
```

```
25          auto& row = matrix[i];
26          row.num = i+1;
27          row.resize(n);
28          for (auto &el: row) {
29              std::cin >> el;
30          }
31          std::cin >> row.cost;
32      }
33
34      // just find similar upper-triangle matrix
35      for(std::size_t col(0); col < n; ++col){
36          cost_type min_cost = MAX_MIN_COST + 1;
37          std::size_t min_idx;
38          for (std::size_t row(col); row < m; ++row) {
39              static data_type epsilon = std::numeric_limits<data_type>::epsilon();
40              if(std::abs(matrix[row][col]) > epsilon && matrix[row].cost < min_cost){
41                  min_idx = row;
42                  min_cost = matrix[row].cost;
43              }
44          }
45          if(min_cost == MAX_MIN_COST + 1){
46              std::cout << -1 << std::endl;
47              return 0;
48          }
49
50          ans.insert(matrix[min_idx].num);
51
52          std::swap(matrix[col/*from which we started*/], matrix[min_idx/*found*/]);
53
54          for (std::size_t row(col + 1); row < m; ++row) { // change all "bottom"(for
                current) rows
55              data_type coefficient = matrix[row][col] / matrix[col][col]; // coefficient
                    for current row
56
57              for (std::size_t col2 = col; col2 < n; ++col2) { // start from our first "
                    not calculated" column
58                  matrix[row][col2] -= matrix[col][col2] * coefficient;
59              }
60          }
61      }
62
63  // std::copy(ans.begin(), ans.end(), std::ostream_iterator<std::size_t>(std::cout, "
        "));
64      for (const auto& elem: ans) {
65          std::cout << elem;
66          if(elem != *std::prev(ans.end())){
67              std::cout << ' ';
68          }
69      }
```

```
70      std::cout << std::endl;
71
72      return 0;
73  }
```

3 Консоль

```
~/university/2 course/diskran/lab8/cmake-build-debug on master
$ cmake ../
--The C compiler identification is AppleClang 13.0.0.13000029
--The CXX compiler identification is AppleClang 13.0.0.13000029
--Detecting C compiler ABI info
--Detecting C compiler ABI info -done
--Check for working C compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/
-skipped
--Detecting C compile features
--Detecting C compile features -done
--Detecting CXX compiler ABI info
--Detecting CXX compiler ABI info -done
--Check for working CXX compiler: /Applications/Xcode.app/Contents/Developer/Toolchain
-skipped
--Detecting CXX compile features
--Detecting CXX compile features -done
--Configuring done
--Generating done
--Build files have been written to: /Users/k.sazhenov/university/2 course/diskran/lab8

~/university/2 course/diskran/lab8/cmake-build-debug on master
$ cmake --build . --clean-first
[ 50%] Building CXX object CMakeFiles/lab8.dir/main.cpp.o
[100%] Linking CXX executable lab8
[100%] Built target lab8

~/university/2 course/diskran/lab8/cmake-build-debug on master
$ ./lab8
5 3
1 1 1 1
4 2 1 3
4 2 1 5
1 2 3 1
4 6 8 2
1 2 4

~/university/2 course/diskran/lab8/cmake-build-debug on master
$
```

4 Тест производительности

Тест производительности представляет собой сравнение с наивным решением этой задачи, в котором перебираются все возможные подсистемы уравнений, а потом выбирается решение с наименьшей стоимостью. В данном примере есть 4 файла, содержащие 10, 20, 30 и 40 уравнений(строк) с 10, 20, 30 и 40 коэффициентами соответственно.

```
~/university/2 course/diskran/lab8/cmake-build-debug on master
$ ./lab8_bench < input10.txt
Greed time: 6.4e-05 sec
Naive time: 8.7e-05 sec
~/university/2 course/diskran/lab8/cmake-build-debug on master
$ ./lab8_bench < input20.txt
Greed time: 0.00023 sec
Naive time: 0.002684 sec
~/university/2 course/diskran/lab8/cmake-build-debug on master
$ ./lab8_bench < input30.txt
Greed time: 0.00017 sec
Naive time: 0.0097 sec
~/university/2 course/diskran/lab8/cmake-build-debug on master
$ ./lab8_bench < input40.txt
Greed time: 0.00104 sec
Naive time: 27.4821 sec
```

Как видно, на малых тестах время работы жадного алгоритма сравнима с временем работы наивного алгоритма, однако, начиная, где-то с 40-ка уравнений наивный алгоритм начинает значительно отставать по времени работы от жадного. Сложность наивного алгоритма $O(m * 2^n)$, где m — кол-во добавок, а n — кол-во коэффициентов этих добавок. Также хочется отметить, что наивный алгоритм затрачивает гораздо больше памяти, чем жадный алгоритм, поскольку ему необходимо хранить исходную сстему уравнений и обрабатываемую на текущем шаге подсистему.

5 Выводы

Стоит сразу отметить, что если глобальная оптимальность алгоритма имеет место практически всегда, его обычно предпочитают другим методам, таким как динамическое программирование. К тому же, очевидно, что ситуация, когда жадный алгоритм это такой алгоритм, который на каждом шаге делает локально наилучший выбор в надежде, что итоговое решение будет оптимальным – частая. Примечательно, что, в каком-то смысле, жадные алгоритмы – частный случай динамического программирования, за исключением того, что в жадном алгоритме выбор делается сразу – до решения подзадач –, а в динамическом программировании наоборот – выбор происходит после решения подзадач. В общем и целом жадные алгоритмы, как и динамическое программирование, упрощает жизнь практикам, которые эти алгоритмы придумывают, т.к. в основном такие алгоритмы довольно просты в реализации. Несмотря на простоту приминения, для каждой задачи требуется весьма сложное доказательство применимости жадного алгоритма для её решения. Область приминения жадных алгоритмов широка: кодирование Хаффмана для сжатия данных, алгоритмы аллокации в ОС, многие алгоритмы на графах.

Список литературы

[1] Т. Кормен, Ч Лейзерсон, Р Ривест, К Штайн. *Алгоритмы: построение и анализ, 3-е издание.* — Издательский дом «Вильямс», 2013. Перевод с английского: ООО «И.Д. Вильямс» — 1328 с. (ISBN 978-5-8459-1794-2 (рус.))