Лабораторная работа № 8 по курсу дискретного анализа: Жадные алгоритмы

Выполнил студент группы М8О-308Б-20 *Ядров Артем.*

Условие

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++, соответствующую построенному алгоритму. Формат входных и выходных данных описан в варианте задания.

Бычкам дают пищевые добавки, чтобы ускорить их рост. Каждая добавка содержит некоторые из N действующих веществ. Соотношения количеств веществ в добавках могут отличаться. Воздействие добавки определяется как

$c_1 \times a_1 + c_2 \times a_2 + \ldots + c_n \times a_n$,

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 ($M \geq N$) различных добавок. Нужно помочь Биологу подобрать самый дешевый набор добавок, позволяющий найти коэффициенты c_i. Возможно, соотношения веществ в добавках таковы, что определить коэффициенты нельзя.

Метод решения

Если перефразировать задачу на язык математики, то нам дано M уравнений с N неизвестными ($M \geq N$), каждое из которых имеет стоимость $c_i \geq 0$. Очевидно, что для решения системы нам достаточно N уравнений. Тогда задача составить систему, удовлетворяющую следующим условиям:

$$\begin{cases} a_{1,1} \times c_i + a_{2,1} \times c_2 + \ldots + a_{n,1} * c_n \\ a_{1,2} \times c_i + a_{2,2} \times c_2 + \ldots + a_{n,2} * c_n \\ \ldots \\ a_{1,n} \times c_i + a_{2,n} \times c_2 + \ldots + a_{n,n} * c_n \\ \sum\limits_{k=1}^{n} cost_k - \text{минимальна} \end{cases}$$

Составим матрицу системы: $A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix}$

Чтобы система имела решение необходимо выполнение условия: $rg(A) = N$. Отмечу,

что добавление линейно зависимых строк в матрицу системы (еще одного уравнения в систему) только увеличит стоимость всей системы, т.к. цена каждого уравнения положительна. Поэтому легче просто не брать линейно зависимые строки.

Из математических выкладок понятно, что необходимо составить систему из N линейно независимых уравнений таким образом, чтобы их стоимость была минимальной. Для этого воспользуемся методом Гаусса: будем строить матрицу системы и приводить ее к верхнетреугольному виду. Напомню, что матрица называется верхнетреугольной, если имеет следующий вид: $A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ 0 & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ 0 & 0 & a_{3,3} & \cdots & a_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{n,n} \end{pmatrix}$ Построение матрицы будем осуществлять следующим образом: k-ую строку будем искать среди еще не включенных в систему уравнений таким образом, чтобы k-й коэффициент уравнения не был равен 0. Если таких уравнений несколько, то берем уравнение с минимальной стоимостью. Если такого уравнения не оказалось, то система не имеет решений.

Замечу, что построение верхнетреугольной матрицы дешевле, нежели приведение матрицы к простейшему виду, т.к. в данном случае мы для k-ой строки мы не обнуляем верхние $k-1$ строки.

Докажем правильность корректность алгоритма. Рассмотрим два случая: 1. Решения не существует. Тогда $rg(A) \leq N$. В нашем случае мы в определенный момент не сможем добавить строку на позицию $rg(A)+1$ и скажем, что решения нет.

2. Решение существует. Пусть $\exists\{a_1, a_2, a_n\} | \sum_{k=1}^{n} cost(a_k) < ans$, где ans - полученный нами ответ. Приведем матрицу полученного решения к верхнетреугольному виду. Это возможно потому что все строки в матрице будут линейно независимыми.

Будем делать поэтапно, как в алгоритме. Возьмем первую строку целиком, обнулим все остальные. В какой-то момент получим, что строка ответа имеет меньшую стоимость, нежели строка, которую мы вставляли в алгоритме. Однако, в таком случае в алгоритме мы почему-то взяли строку не с минимальной стоимостью (k-й коэффициент строки этого уравнения точно не нулевой).

Получаем противоречие $\Rightarrow$ предположение было неверно (оно было о том, что существует решение с меньшей стоимостью). Значит, наш алгоритм нашел правильное решение.

Оценим сложность алгоритма. Для начала определим сложность операции поиска строки с минимальной стоимостью. Пусть наша система уже имеет k строк. Тогда нам необходимо просмотреть $M-k$ строк, т.е. сложность составит $\mathcal{O}(M-k)$.

Следующая операция - обнуление строки. Для k-ой строки необходимо обнулять $N-k$ коэффициентов (в силу того, что предыдущие уже обнулены), т.е. сложность составляет $\mathcal{O}(N-k)$.

Теперь попытаемся оценить k-й шаг алгоритма: ищем k-ую строку, затем обнуляем $M-k$ строк. Сложность составит $\mathcal{O}(M-k) + \mathcal{O}((M-k) * (N-k))$.

Итоговая сложность получается суммированием по всем k:

$$\sum_{k=1}^{n} \mathcal{O}(M-k) + \mathcal{O}((M-k) \times (N-k)).$$

В итоге будет $\mathcal{O}(M \times N + N^2 \times M) = \mathcal{O}(N^2 \times M)$.

Исходный код

Реализацию алгоритма я разбил на несколько классов:

1. Класс уравнения: содержит в себе вектор коэффициентов. По сути экземпляр класса является строкой в матрице и имеет стоимость, что для нас самое ценное.

2. Класс системы: содержит в себе вектор уравнений. Имеет метод вставки уравнения в систему и нахождения решения системы, представляющий собой наш алгоритм.

```
#ifndef SRC__EQUATION_H
#define SRC__EQUATION_H

#include <vector>
class Equation {
public:
  Equation(std::vector<double> k, const int &cost, const int &ind);
  Equation(const Equation &other) = default;
  double &operator[](const int &i);
  int cost() const;
  int ind() const;
  friend bool operator<(const Equation &left, const Equation &right);
private:
  std::vector<double> coefficients_;
  int cost_;
  int ind_;
};

#endif //SRC__EQUATION_H
```

```
#include "Equation.h"

#include <utility>
#include <stdexcept>
```

```
Equation::Equation(std::vector<double> k, const int &cost, const int &ind)
        : coefficients_(std::move(k)), cost_(cost), ind_(ind) {}
double &Equation::operator[](const int &i) {
  if (coefficients_.size() <= i) {
        throw std::out_of_range("Выход за границы массива");
  }
  return coefficients_[i];
}
int Equation::cost() const {
  return cost_;
}
int Equation::ind() const {
  return ind_;
}
bool operator<(const Equation &left, const Equation &right) {
  return left.cost_ < right.cost_;
}
```

```
#ifndef SRC__SYSTEMOFEQUATIONS_H
#define SRC__SYSTEMOFEQUATIONS_H

#include "Equation.h"
class SystemOfEquations {
public:
  explicit SystemOfEquations(const int &maxCost, const int &n);
  void addEquation(const std::vector<double> &coeff, const int &cost);
  void findMinCost(const int &ind);
  void zero(const int &base, const int &ind);
  std::vector<int> solve();
private:
  std::vector<Equation> equations_;
  int curMinCostInd_;
  int curMinCost_;
  int size_;
  int numOfUnknowns_;
  int maxCost_;
};

#endif //SRC__SYSTEMOFEQUATIONS_H
```

```cpp
#include "SystemOfEquations.h"

#include <algorithm>

SystemOfEquations::SystemOfEquations(const int &maxCost, const int &n) :
        equations_(), maxCost_(maxCost), curMinCostInd_(-2), curMinCost_(0),
↪  size_(0), numOfUnknowns_(n) {}
void SystemOfEquations::addEquation(const std::vector<double> &coeff, const
↪  int &cost) {
  equations_.emplace_back(coeff, cost, size_);
  ++size_;
}
void SystemOfEquations::findMinCost(const int &ind) {
  curMinCost_ = maxCost_;
  curMinCostInd_ = -1;
  if (size_ == 0) {
        return;
  }
  for (int i = ind; i < size_; ++i) {
        if (equations_[i].cost() < curMinCost_ and equations_[i][ind] != 0)
↪  {
          curMinCost_ = equations_[i].cost();
          curMinCostInd_ = i;
        }
  }
}
std::vector<int> SystemOfEquations::solve() {
  if (size_ == 0) {
        return {-1};
  }
  std::sort(equations_.begin(), equations_.end());
  std::vector<int> ans;
  curMinCostInd_ = -2;
  for (int i = 0; i < numOfUnknowns_; ++i, curMinCostInd_ = -2) {
        if (curMinCostInd_ == -2) {
          findMinCost(i);
        }
        if (curMinCostInd_ == -1) {
          return std::move(std::vector<int>{-2});
        }
        // переносим вверх строку, которую не будем обнулять
        std::swap(equations_[i], equations_[curMinCostInd_]);
```

```
        // обнуляем остальные столбцы
        for (int j = i + 1; j < size_; ++j) {
          zero(i, j);
        }
        ans.emplace_back(equations_[i].ind());
  }
  std::sort(ans.begin(), ans.end());
  return std::move(ans);
}
void SystemOfEquations::zero(const int &base, const int &ind) {
  double k = equations_[ind][base] / equations_[base][base];
  for (int i = ind; i < numOfUnknowns_; ++i) {
        equations_[ind][i] -= k * equations_[base][i];
  }
}
```

```
#include <iostream>
#include "SystemOfEquations.h"

const int MAX_COST = 51;

int main() {
  int M, N;
  std::cin >> M >> N;
  SystemOfEquations system(MAX_COST, N);
  std::vector<double> coeff(N);
  int cost;
  for (int i = 0; i < M; ++i){
        for (int k = 0; k < N; ++k){
          std::cin >> coeff[k];
        }
        std::cin >> cost;
        system.addEquation(coeff, cost);
  }
  auto ans = system.solve();
  for (const auto &answer: ans){
        std::cout << answer + 1 << " ";
  }
  std::cout << std::endl;
  return 0;
}
```

Дневник отладки

Во время реализации я столкнулся с небольшой проблемой: предварительно отсортировав уравнения по цене, я наивно предположил, что первый по порядку элемент можно взять в качестве опорного для построения матрицы.

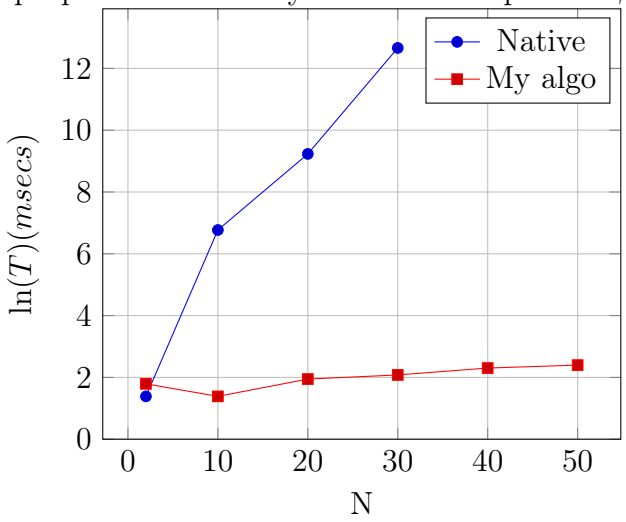
К моему несчастью, случай, когда первый коэффициент этого уравнения будет равен нулю, разбивает напрочь мое решение.

Тест производительности

Возьмем самый наивный алгоритм, работающий предположительно за $\mathcal{O}(m \times 2^n)$ и сравним его с решением задачи.

Сравнивать будем на заранее сгенерированных мною тестах, где гарантируется наличие решения.

Будем работать с входными данными $N = 2, 10, 20, 30, 40, 50$ и $M \in [N, 2 \times N]$. Для наивного алгоритма при $N = 30$ я так и не смог дождаться завершения работы программы. Поэтому в качестве времени для $N = 30$ взят предел моего терпения.



Выводы

В результате выполнения лабораторной работы я изучил основные алгоритмы, использующие идею жадных алгоритмов, составил и отладил программу для своего варианта задания.

В отличие от динамического программирования жадные алгоритмы предполагают, что задача имеет оптимальное решение, которое строится из оптимальных решений для подзадач с заранее определённым выбором, а не перебором всех вариантов перехода. Такой подход уменьшает временные и пространственные ресурсы, нужные для решения задачи.

На практике почти все наши решения подчиняются логике жадных алгоритмов: пообедать вкуснее, купить телефон получше, заплатить меньше. Полезно понимать,

что такой подход не всегда работает и что в половине случаев нужно продумывать свои действия наперёд, перебирая возможные варианты.