

Раздел «Алгоритмы» . HungarianAlgorithmCPP :

Реализация Венгерского алгоритма на C++

- Реализация Венгерского алгоритма на C++
 - Описание программы
 - Основной цикл: добавление строк
 - Поиск чередующейся цепи: разрешение коллизий
 - Итого
 - Код на C++

Имеется m заданий и столько же исполнителей. Каждый исполнитель способен выполнить каждое задание, но за каждое задание он возьмет с Вас определенную сумму денег. Вы торопитесь, поэтому хотите назначить каждому исполнителю по задаче (разные исполнители получают разные задачи), и заставить их всех работать одновременно. Ваша задача – сделать это так, чтобы минимизировать затраты.

Эта задача называется **задачей о назначениях**. Ниже приведена одна из возможных реализаций **Венгерского метода**, которая решает эту задачу за полиномиальное время ($O(m^3)$).

Описание программы

Следуя стилю C++, мы будем индексировать строки и столбцы нашей матрицы, а также элементы всех вспомогательных массивов, начиная с нуля. Основные данные о текущем состоянии нашей матрицы будут храниться в следующих переменных:

- `matrix` – исходная матрица m на m . На самом деле, алгоритм будет работать и для матриц, у которых высота меньше ширины. Значения матрицы `matrix` никогда не будут изменяться в ходе работы алгоритма.
- `height`, `width` – размеры исходной матрицы
- `u`, `v` – массивы размеров `height` и `width` соответственно, при помощи которых будут "эмулироваться" операции модификации строк и столбцов. Элемент (i, j) модифицированной матрицы вычисляется как `matrix[i][j] - u[i] - v[j]`.
- `markIndices` – массив размера `width`, хранящий информацию о текущем максимальном независимом множестве нулевых элементов. Поясним смысл значений в этом массиве. В процессе работы алгоритма мы постоянно будем поддерживать некоторый набор независимых нулевых элементов (здесь и далее имеется в виду модифицированная матрица). Элементы этого набора будем называть "отмеченными" (`marked`). Поскольку набор независимый, в каждом столбце матрицы (с номером j) будет находиться либо один отмеченный элемент (i, j) , – в этом случае значение `markIndices[j]` будет хранить номер строки i , – либо ни одного отмеченного элемента, и в этом случае в `markIndices[j]` будет записано число `-1`.

Основной цикл: добавление строк

Алгоритм начнет исполнение с некоторой условной подматрицы, состоящей из пустого набора строк исходной матрицы. В этой условной матрице максимальный независимый набор элементов уже найден: это пустой набор. Далее, сделаем `height` шагов, добавляя на каждом шаге очередную строку матрицы. Индекс добавляемой строки будем хранить в переменной `i`, `i = 0 .. height - 1`. Под "добавлением" i -й строки понимается решение следующей задачи: в строках с индексами $0..i-1$ имеется i отмеченных элементов (отмеченные элементы всегда нулевые и независимые). Требуется провести модификации строк и столбцов матрицы так, чтобы можно было увеличить на единицу количество отмеченных элементов и получить набор из $i+1$ отмеченного элемента, по одному в каждой строке с индексами $0..i$.

Увеличивать набор отмеченных элементов будем по аналогии с алгоритмом поиска **наибольшего паросочетания** при помощи *чередующейся цепи* нулевых элементов

[Поиск](#)[Раздел «Алгоритмы»](#)[Главная](#)[Форум](#)[Ссылки](#)[EJ Judge](#)[Инструменты:](#)[Поиск](#)[Изменения](#)[Index](#)[Статистика](#)[Разделы](#)[Информация](#)[Алгоритмы](#)[Язык Си](#)[Язык Ruby](#)[Язык](#)[Ассемблера](#)[EJ Judge](#)[Парадигмы](#)[Образование](#)[Сети](#)[Objective C](#)[Login>>](#)

матрицы. Пример чередующейся цепи показан на рисунке:

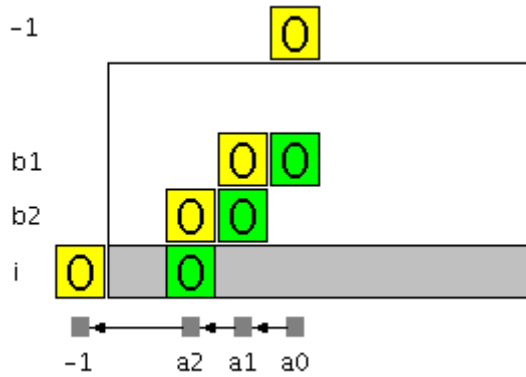


Рисунок 1: чередующаяся цепь

Чередующаяся цепь состоит из нечетного количества нулевых элементов матрицы. При этом

- Все элементы с нечетным номером (отмечены на рисунке зеленым цветом) не отмечены
- Все элементы с четным номером отмечены (отмеченные элементы на всех рисунках показаны желтым цветом).
- В столбце с первым элементом отмеченных элементов нет (аналог свободной вершины в двудольном графе)
- Последний элемент находится в "добавляемой" строке (в этой строке отмеченных элементов тоже нет, так как эта строка еще только обрабатывается).
- Следующий за всяким "зеленым" элементом "желтый" элемент находится с ним в одной строке
- Следующий за всяким "желтым" элементом "зеленый" элемент находится с ним в одном столбце

Если чередующаяся цепь найдена, то увеличение количества отмеченных элементов не представляет труда: достаточно в этой цепи снять отметки со всех четных элементов и отметить все нечетные (их на один больше). Представленная здесь реализация хранит информацию о цепи следующим образом:

- Во-первых, удобно считать, что в начале и в конце цепи есть еще два фиктивных отмеченных элемента, которые также изображены на рисунке. Первый элемент находится в "строке" с индексом -1 , а последний – в столбце с индексом -1 .
- Чтобы восстановить всю цепочку, достаточно знать лишь номера столбцов, которые в ней участвуют, по порядку. На рисунке это последовательность столбцов (a_0, a_1, a_2) . Для каждого столбца мы можем узнать положение отмеченного элемента в этом столбце при помощи массива `markIndices[]`.
- Учитывая это соображение, логично организовать столбцы цепочки в виде односвязного списка. Для каждого столбца будем хранить ссылку на последующий столбец. Все эти ссылки будем хранить в массиве `links` размера `width`. В примере, показанном на рисунке, `links[a0] = a1`, `links[a1] = a2`, наконец `links[a2] = -1`. Теперь всю чередующуюся цепочку можно восстановить, зная лишь только индекс первого столбца a_0 . Ссылки, хранящиеся в массиве `links`, показаны на рисунке стрелками.

Поиск чередующейся цепи: разрешение коллизий

Первый шаг, который хочется сделать для поиска чередующейся цепи – это найти в i -той строке (показанной на рисунке серым цветом) минимальный элемент и вычесть его из всей строки. Тогда на месте минимального элемента получим 0 . Если в столбце с этим нулем нет отмеченных элементов, то чередующаяся цепь уже найдена: она состоит из одного этого нулевого элемента. Если же в этом столбце уже есть отмеченный элемент, то будем говорить, что произошла коллизия. В процессе *разрешения коллизий* мы и построим чередующуюся цепь. Например, на рисунке выше сначала возникла коллизия в столбце a_2 . При разрешении этой коллизии был найден новый нулевой элемент (b_2, a_1) . В этот момент возникла вторая коллизия в столбце a_1 . При разрешении этой коллизии был найден элемент (b_1, a_0) , и этот элемент к коллизии не привел, в результате чего и закончилось построение чередующейся цепи.

Опишем процесс разрешения коллизий в общем случае. Процесс будет состоять из не более чем $i + 1$ шагов. Опишем ситуацию после исполнения k шагов. На следующем рисунке $i = 7$ и $k = 5$.

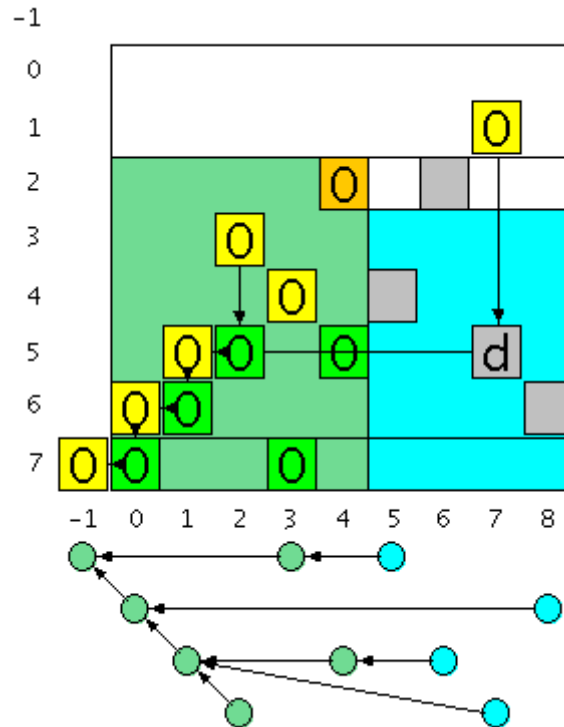


Рисунок 2: разрешение коллизий

Мы имеем k столбцов, в которых уже произошли коллизии. Будем называть эти столбцы "посещенными". На рисунке 2 посещенными являются столбцы с 0 по 4. В каждом посещенном столбце имеется ровно один отмеченный элемент, который мы назовем посещенным элементом. Строки, в которых находятся эти отмеченные элементы, а также строку с номером i , будем называть посещенными строками. Пересечение посещенных строк и столбцов образует "посещенный блок", который на рисунке показан зеленым цветом. Потребуем, чтобы на каждом шаге каждый посещенный элемент удовлетворял следующему свойству: должна существовать цепочка элементов, начинающаяся в данном элементе, заканчивающаяся в фиктивном элементе из столбца -1 , не выходящая за пределы посещенного блока. Эта цепочка должна удовлетворять определению чередующейся цепочки, за тем исключением, что ее начало – наш отмеченный элемент (т.е. это чередующаяся цепочка без начала). К примеру, на рисунке 2 элемент (3, 2) служит началом цепочке, которая показана стрелками.

После каждого шага нам также известен последний посещенный элемент. На рисунке этот элемент показан оранжевым цветом.

Далее, рассмотрим блок нашей матрицы, образованный пересечением всех *непосещенных* столбцов и всех посещенных строк, кроме последней. Назовем его блоком поиска. На рисунке блок поиска показан голубым цветом. После l шагов этот блок имеет l строк и $m - l$ столбцов. Будем на каждом шаге поддерживать информацию о минимальном элементе в каждом столбце этого блока.

Вся описанная информация, доступная после очередного шага разрешения коллизий, в предложенном коде на C++ хранится следующим образом:

- Массив флагов `visited[]` из `width` элементов. Единичками в этом массиве отмечены посещенные столбцы. Посещенные строки нигде хранить не нужно, так как по номерам посещенных столбцов при помощи массива `markIndices[]` можно узнать положение всех посещенных отмеченных элементов (на рисунке это оранжевый и желтые элементы в посещенном блоке), а значит и индексы посещенных строк.
- Массив значений `mins[]` из `width` элементов. Для посещенных столбцов значения `mins[]` никак не используются. Для всякого непосещенного столбца t элемент `mins[t]` содержит минимальное значение в этом столбце в поисковом блоке. Например на рисунке 2 `mins[7] = d`.

- Массив ссылок `links[]` из `width` элементов. Именно по этому массиву можно построить все цепочки, начинающиеся в посещенных отмеченных элементах. На рисунке значения ссылок показаны под матрицей. Например, цепочка, которая начинается в элементе (3, 2) и показана на рисунке стрелками, может быть получена проходом по ссылкам: 2 -> 1 -> 0 -> -1. Для непосещенных столбцов значения ссылок тоже имеют смысл. Ссылка `links[t]` указывает на столбец, содержащий посещенный отмеченный элемент, который находится в той же строке, что и минимальный элемент столбца `t` в блоке поиска. К примеру, минимальным элементом в 7-м столбце является элемент (5, 7) со значением `d`. В одной строке с этим элементом находится уже посещенный элемент (5, 1), поэтому ссылка 7-го столбца указывает на 1-й столбец: `links[7] = 1`. Можно сказать, что в дереве столбцов, показанном на рисунке под матрицей, столбец 1 является родителем столбца 7.
- Наконец, последний посещенный элемент, с которым произошла коллизия, задается переменными (`markedI`, `markedJ`). На рисунке этот элемент показан оранжевым цветом.

Имея эту информацию, мы должны либо найти чередующуюся цепочку (в этом случае процесс разрешения коллизий будет закончен), либо увеличить количество посещенных столбцов на 1 (в этом случае мы должны обновить всю перечисленную информацию). Все это делается за два шага.

1. Добавим последнюю посещенную строку к поисковому блоку (чтобы в него входили все посещенные строки). При этом обновим значения `mins[]` и `links[]`. Затем найдем минимальный элемент в поисковом блоке (на рисунке это элемент со значением `d`). В программе номер столбца с этим элементом оказывается в переменной `j`.
2. Теперь вычтем значение `d` из всех посещенных строк и добавим его ко всем посещенным столбцам. Заметим, что посещенных строк на 1 больше, чем столбцов, то есть сумма элементов массивов `u[]` и `v[]` возрастет на `d`. При этом элементы посещенного блока никак не изменятся (к ним прибавится сначала `-d`, а затем `+d`). Все элементы поискового блока уменьшатся на `d` (и в силу минимальности `d` останутся неотрицательными). Наконец, все отмеченные но непосещенные элементы никак не изменятся (они лежат в непосещенных строках и непосещенных столбцах). Итого, все отмеченные элементы по-прежнему нулевые, и из всех посещенных элементов по-прежнему идут цепочки в фиктивный элемент. Однако теперь вместо `d` мы имеем новый нулевой элемент.
3. Теперь имеется две возможности. Если в столбце с этим новым нулевым элементом нет отмеченных элементов, то мы построили чередующуюся цепочку. Действительно, начнем с нового нулевого элемента, затем пройдем по ссылке в посещенный элемент в его строке, а затем из этого посещенного элемента мы можем дойти по цепочке до фиктивного элемента. Если же в столбце с новым нулевым элементом есть отмеченный элемент (как на рисунке), то мы обнаружили новую коллизию. Просто объявим столбец посещенным и перейдем к следующей итерации разрешения коллизий.

Ясно, что процесс не может продолжаться бесконечно. Если предположить противное, то после `i` шагов мы будем иметь `i+1` посещенную строку (включая `i`-тую) и `i < m` посещенных столбцов. Это значит, что все отмеченные элементы будут уже посещены, поэтому в столбце с минимальным элементом из поискового блока не может произойти коллизии.

Итого

Весь описанный алгоритм можно кратко записать так:

```

1: function hungarian
2:   for i от 0 до height-1
3:     do создать в матрице новый нулевой элемент
4:     while (пока) новый элемент приводит к коллизии
5:       Изменить метки элементов вдоль найденной цепочки
6:   return все height отмеченных элементов

```

Здесь мы имеем два вложенных цикла размера `height` (`m`, если мы решаем задачу для квадратной матрицы), причем операция "создать новый нулевой элемент" требует обновления массивов `mins[]`, `links[]`, `u[]`, `v[]`, и требует времени `m`. Итого время работы алгоритма - $O(m^3)$.

Основная сложность реализации заключается в строках 3 и 4: создавать новые нулевые элементы нужно так, чтобы не "испортить" уже полученный независимый набор, и так, чтобы в итоге получить чередующуюся цепочку. В предыдущем разделе было рассказано, как это сделать. Далее следует полная реализация на C++, которая оказывается существенно короче, чем ее описание.

Код на C++

```

/* Венгерский алгоритм.
 * Даниил Швед, 2008. danshved [no-spam] gmail.com
 * Реализация навеяна псевдокодом А.С.Лопатина из книги
 * "Оптимизация на графах (алгоритмы и реализация)".
 */
#include <vector>
#include <limits>
using namespace std;

typedef pair<int, int> PInt;
typedef vector<int> VInt;
typedef vector<VInt> VVInt;
typedef vector<PInt> VPInt;

const int inf = numeric_limits<int>::max();

/*
 * Решает задачу о назначениях Венгерским методом.
 * matrix: прямоугольная матрица из целых чисел (не обязательно положительных).
 * Высота матрицы должна быть не больше ширины.
 * Возвращает: Список выбранных элементов, по одному из каждой строки матрицы.
 */
VPInt hungarian(const VVInt &matrix) {

    // Размеры матрицы
    int height = matrix.size(), width = matrix[0].size();

    // Значения, вычитаемые из строк (u) и столбцов (v)
    VInt u(height, 0), v(width, 0);

    // Индекс помеченной клетки в каждом столбце
    VInt markIndices(width, -1);

    // Будем добавлять строки матрицы одну за другой
    for(int i = 0; i < height; i++) {
        VInt links(width, -1);
        VInt mins(width, inf);
        VInt visited(width, 0);

        // Разрешение коллизий (создание "чередующейся цепочки" из нулевых элементов)
        int markedI = i, markedJ = -1, j;
        while(markedI != -1) {
            // Обновим информацию о минимумах в посещенных строках непосещенных столбцов
            // Заодно поместим в j индекс непосещенного столбца с самым маленьким из них
            j = -1;
            for(int j1 = 0; j1 < width; j1++)
                if(!visited[j1]) {
                    if(matrix[markedI][j1] - u[markedI] - v[j1] < mins[j1]) {
                        mins[j1] = matrix[markedI][j1] - u[markedI] - v[j1];
                        links[j1] = markedJ;
                    }
                    if(j == -1 || mins[j1] < mins[j])
                        j = j1;
                }

            // Теперь нас интересует элемент с индексами (markIndices[links[j]], j)
            // Произведем манипуляции со строками и столбцами так, чтобы он обнулится
            int delta = mins[j];
            for(int j1 = 0; j1 < width; j1++)
                if(visited[j1]) {

```

```

        u[markIndices[j1]] += delta;
        v[j1] -= delta;
    } else {
        mins[j1] -= delta;
    }
    u[i] += delta;






    // Если коллизия не разрешена - перейдем к следующей итерации
    visited[j] = 1;
    markedJ = j;
    markedI = markIndices[j];
}

// Пройдем по найденной чередующейся цепочке клеток, снимем отметки с
// отмеченных клеток и поставим отметки на неотмеченные
for(; links[j] != -1; j = links[j])
    markIndices[j] = markIndices[links[j]];
markIndices[j] = i;
}

// Вернем результат в естественной форме
VPInt result;
for(int j = 0; j < width; j++)
    if(markIndices[j] != -1)
        result.push_back(PInt(markIndices[j], j));
return result;
}

```

-- DanielShved - 22 Mar 2008

Attachment 	Action	Size	Date	Who	Comment
 no1.draw	manage	5.0 K	22 Mar 2008 - 22:40	DanielShved	TWikiDraw? draw file
 no1.gif	manage	1.9 K	22 Mar 2008 - 22:40	DanielShved	TWikiDraw? GIF file
 no2.draw	manage	12.6 K	22 Mar 2008 - 22:48	DanielShved	TWikiDraw? draw file
 no2.gif	manage	4.3 K	22 Mar 2008 - 22:49	DanielShved	TWikiDraw? GIF file

Copyright © 2003-2022 by the contributing authors.