**УФИМСКИЙ ГОСУДАРСТВЕННЫЙ АВИАЦИОННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**Кафедра информатики**

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовой работе

Дисциплина: Алгоритмы и структуры данных

Тема работы: Венгерский алгоритм решения задачи о назначениях

Выполнил: студент фак.ИРТ,

гр. ПРО 208сз

**Фаловский А. К.**

Проверил: профессор, к.т.н.,

**Сахабетдинов М. А.**

УФА 2015

Оглавление

1 [Постановка задачи 3](#_Toc409378160)

2 [Анализ возможных подходов к решению задач данного вида 4](#_Toc409378161)

3 [Разработка и описание алгоритма 5](#_Toc409378162)

4 [Разработка программного обеспечения 14](#_Toc409378163)

5 [Результаты тестирования программ 17](#_Toc409378164)

6 [Оценка сложности алгоритма 18](#_Toc409378165)

7 [Заключение 19](#_Toc409378166)

8 [Список использованных источников 20](#_Toc409378167)

1 Постановка задачи

Составить программу для решения задачи: есть N рабочих и N заданий. Для каждого рабочего известно, сколько денег он запросит за выполнение того или иного задания. Каждый рабочий может взять себе только одно задание. Требуется распределить задания по рабочим так, чтобы минимизировать суммарные расходы.

2 Анализ возможных подходов к решению задач данного вида

Алгоритм был разработан и опубликован Гарольдом Куном (Harold Kuhn) в 1955 г. Сам Кун дал алгоритму название "венгерский", потому что он был в значительной степени основан на более ранних работах двух венгерских математиков: Денеша Кёнига (Dénes Kőnig) и Эйгена Эгервари (Jenő Egerváry).

В 1957 г. Джеймс Манкрес (James Munkres) показал, что этот алгоритм работает за (строго) полиномиальное время (т.е. за время порядка полинома от n, не зависящего от величины стоимостей).

Поэтому в литературе данный алгоритм известен не только как "венгерский", но и как "алгоритм Куна-Манкреса" или "алгоритм Манкреса".

Впрочем, недавно (в 2006 г.) выяснилось, что точно такой же алгоритм был изобретён за век до Куна немецким математиком Карлом Густавом Якоби (Carl Gustav Jacobi). Дело в том, что его работа "About the research of the order of a system of arbitrary ordinary differential equations", напечатанная посмертно в 1890 г., содержавшая помимо прочих результатов и полиномиальный алгоритм решения задачи о назначениях, была написана на латыни, а её публикация прошла незамеченной среди математиков.

Также стоит отметить, что первоначальный алгоритм Куна имел асимптотику O(n^4), и лишь позже Джек Эдмондс (Jack Edmonds) и Ричард Карп (Richard Karp) (и независимо от них Томидзава(Tomizawa)) показали, каким образом улучшить его до асимптотики O(n^3).

3 Разработка и описание алгоритма

3.1 Переменные

Основные данные о текущем состоянии нашей матрицы будут храниться в следующих переменных:

* arr - исходная матрица m на m. На самом деле, алгоритм будет работать и для матриц, у которых высота меньше ширины. Значения матрицы arr никогда не будут изменяться в ходе работы алгоритма.
* rowCount, colCount - размеры исходной матрицы
* u, v - массивы размеров rowCount и colCount соответственно, при помощи которых будут "эмулироваться" операции модификации строк и столбцов. Элемент (i, j) модифицированной матрицы вычисляется как arr[i][j] - u[i] - v[j].
* solution - массив размера colCount, 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 solution[j] будет хранить номер строки i , — либо ни одного отмеченного элемента, и в этом случае в solution[j] будет записано undefined.

3.2 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, проделаем rowCount шагов, добавляя на каждом шаге очередную строку матрицы. Индекс добавляемой строки будем хранить в переменной i, i = 0 .. rowCount - 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 матрицы. Пример чередующейся цепи показан на рисунке:

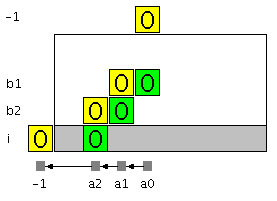
[](http://acm.mipt.ru/twiki/bin/oops/Algorithms/HungarianAlgorithmCPP?template=twikidraw&param1=no1)

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 (a0, a1, a2). Для каждого столбца мы можем узнать положение отмеченного элемента в этом столбце при помощи массива solution[].
* 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 way размера colCount. В примере, показанном на рисунке, way[a0] = a1, way[a1] = a2, наконец way[a2] = -1. Теперь всю чередующуюся цепочку можно восстановить, зная лишь только индекс первого столбца a0. Ссылки, хранящиеся в массиве way, показаны на рисунке стрелками.

3.3 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 a2. При разрешении этой коллизии был найден новый нулевой элемент (b2, a1). В этот момент возникла вторая коллизия в столбце a1. При разрешении этой коллизии был найден элемент (b1, a0), 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.

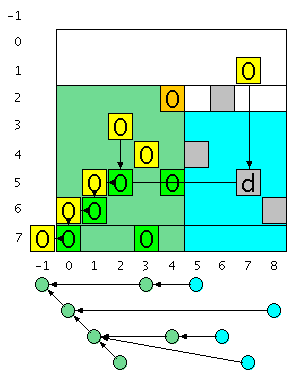
[](http://acm.mipt.ru/twiki/bin/oops/Algorithms/HungarianAlgorithmCPP?template=twikidraw&param1=no2)

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 хранится следующим образом:

* Массив флагов used[] из colCount элементов. Единичками в этом массиве отмечены посещенные столбцы. Посещенные строки нигде хранить не нужно, так как по номерам посещенных столбцов при помощи массива solution[] 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 minimums[] из colCount элементов. Для посещенных столбцов значения minimums[] никак не используются. Для всякого непосещенного столбца t элемент minimums[t] содержит минимальное значение в этом столбце в поисковом блоке. Например, на рисунке 2 minimums[7] = delta.
* Массив ссылок way[] из colCount 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 way[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 delta. В одной строке с этим элементом находится уже посещенный элемент (5, 1), поэтому ссылка 7-го столбца указывает на 1-й столбец: way[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 (curRow, colNumOfMinimum). 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 minimums[] и way[]. 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 delta 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 delta. При этом элементы посещенного блока никак не изменятся (к ним прибавится сначала -delta, а затем +delta). Все элементы поискового блока уменьшатся на delta (и в силу минимальности delta 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 delta 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.

Весь описанный алгоритм можно кратко записать так:

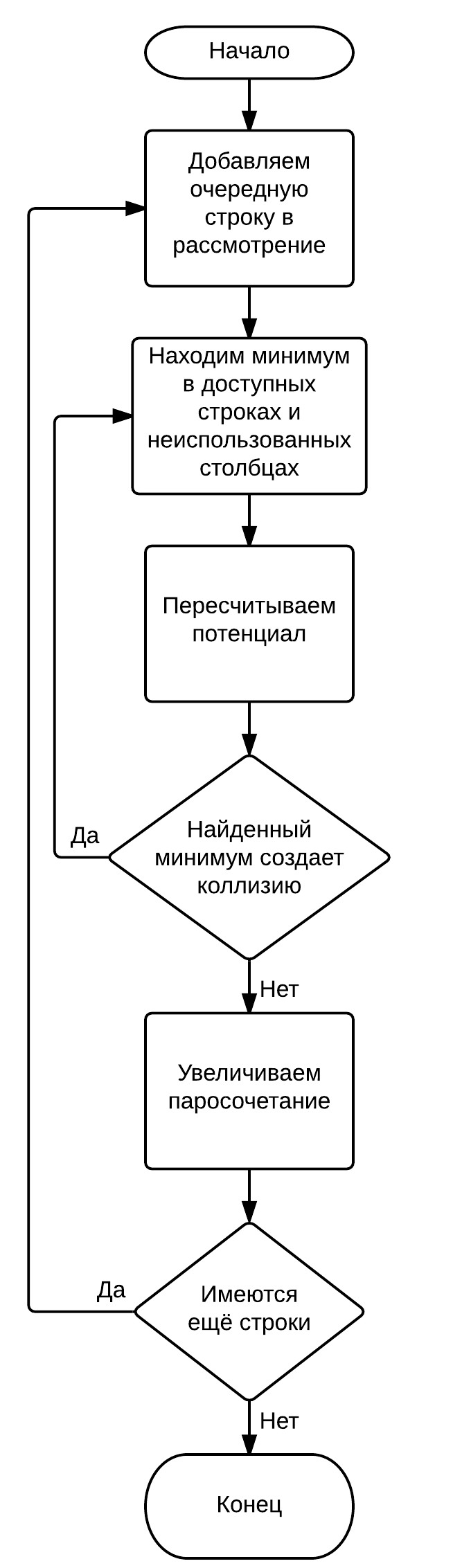


Рисунок 3 – Блок-схема алгоритма

4 Разработка программного обеспечения

Исходный код программы:

function solveAssignmentProblem(arr){

var u, v, minimums, i, j, solution, curRow, markedCol, delta, rowCount, colCount;

var used, way, colNumOfMinimum;

solution = [];

used = [];

minimums = [];

way = [];

v = [];

u = [];

rowCount = arr.length;

colCount = arr[0].length;

for(j = 0; j < colCount; ++j){

u[j] = 0;

v[j] = 0;

}

// основной цикл по строкам

for(i = 0; i < rowCount; ++i){

for(j = 0; j < colCount; ++j){

minimums[j] = Infinity;

used[j] = false;

}

curRow = i;

markedCol = undefined;

colNumOfMinimum = 0;

// коллизии (чередующася цепь)

do {

delta = Infinity;

// находим минимум

for(j = 0; j < colCount; ++j){

if (!used[j]) {

if (arr[curRow][j] - u[curRow] - v[j] < minimums[j]) {

minimums[j] = arr[curRow][j] - u[curRow] - v[j];

way[j] = markedCol;

}

if(minimums[j] < delta) {

delta = minimums[j];

colNumOfMinimum = j;

}

}

}

// пересчитываем минимумы и потенциал

for(j = 0; j < colCount; ++j) {

if (used[j]) {

u[solution[j]] += delta;

v[j] -= delta;

}

else minimums[j] -= delta;

}

u[i] += delta;

used[colNumOfMinimum] = true;

markedCol = colNumOfMinimum;

curRow = solution[colNumOfMinimum];

} while(curRow != undefined);

while (way[colNumOfMinimum] != undefined){

solution[colNumOfMinimum] = solution[way[colNumOfMinimum]];

colNumOfMinimum = way[colNumOfMinimum];

}

solution[colNumOfMinimum] = i;

}

return solution;

}

5 Результаты тестирования программ

Проведем эксперименты с N = 10, 100, 500, 1000 и взглянем на таблицу 1 и рисунок 4.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размер массива | 10 | 100 | 500 | 1000 |
| Time, мс | 2 | 9 | 945 | 10297 |

Таблица 1 - Время работы алгоритма

Рисунок 4 - График времени работы алгоритма

6 Оценка сложности алгоритма

Во внешнем цикле мы добавляем в рассмотрение строки матрицы одну за другой. Каждая строка обрабатывается за время O(n^2), поскольку при этом могло происходить лишь O(n) пересчётов потенциала (каждый — за время O(n)), для чего за время O(n^2) поддерживается массив minv[]; алгоритм Куна суммарно отработает за время O(n^2) (поскольку он представлен в форме O(n) итераций, на каждой из которых посещается новый столбец). Итоговая асимптотика составляет O(n^3) — или, если задача прямоугольна, O(n^2 m).

7 Заключение

Венгерский алгоритм позволяет решать задачу за полиномиальное время не превышающее O(n^3). Что гораздо быстрее полного перебора, который занимает время *O(n!)*. Применения жадного алгоритма быстрее, но он, очевидно, может выдать не оптимальное решение.

Кроме задачи о назначениях венгерский алгоритм можно применять для решения схожих задач, например:

* Дан двудольный граф, требуется найти в нём паросочетание максимальное паросочетание минимального веса (т.е. в первую очередь максимизируется размер паросочетания, во вторую — минимизируется его стоимость).
* Задача детектирования движущихся объектов по снимкам: было произведено два снимка, по итогам которых было получено два набор координат. Требуется соотнести объекты на первом и втором снимке, т.е. определить для каждой точки второго снимка, какой точке первого снимка она соответствовала. При этом требуется минимизировать сумму расстояний между сопоставленными точками (т.е. мы ищем решение, в котором объекты суммарно прошли наименьший путь).

8 Список использованных источников

1. М.О.Асанов, В.А.Баранский, В.В.Расин - Дискретная математика: графы, матроиды алгоритмы Ижевск: НИЦ "РХД", 2001, 288 стр.
2. Христос X. Пападимитриу, Кеннет Стайглиц – Комбинаторная оптимизация, алгоритмы и сложность. — М.:«Мир», 1984. — С. 510.
3. Венгерский алгоритм решения задачи о назначениях [Электронный ресурс] // MAXimal – описание различных алгоритмов – Режим доступа: http://e-maxx.ru/algo/assignment\_hungary, свободный
4. Реализация Венгерского алгоритма на C++ [Электронный ресурс] // Олимпиады по программированию на Физтехе. – Режим доступа: http://acm.mipt.ru/twiki/bin/view/Algorithms/HungarianAlgorithmCPP, свободный