Цель: закрепление навыков структурного и объектно-ориентированного программирования, проектирования сложных ветвистых алгоритмов, а также закрепление ранее пройденного лекционного материала.

Оборудование: персональный компьютер, среда разработки Visual Studio 2019 или новее.

Задание:

Для выполнения данной практической работы студенту необходимо спроектировать и разработать программное средство (далее – ПС), позволяющее автоматизировать расчет прикладной математической модели, а именно: транспортной задачи.

Условие задачи:

1. Для расчетов задачи в ПС должен использоваться **метод северо-западного угла**, для проверки оптимальности плана – **метод потенциалов**.
2. Ввод с клавиатуры входных данных в ПС не предусматривается. Все данные, необходимые для корректной работы ПС, должны считываться из текстового файла. ПС должно предусматривать, что требуемый текстовый файл может отсутствовать по предполагаемому пути в каталоге, или быть пустым. Требуемый формат (пример) текстового файла для чтения входных данных представлен ниже:

text.txt:

3 3

2 5 2 4 1 5 3 6 8

90 400 160

140 300 160

ПС должно обрабатывать данные в следующие объекты:

* исходная матрица 3х3, где первое число в строке – кол-во строк, второе – кол-во столбцов;
* исходная матрица распределения = ;
* столбец запасов = ;
* строка потребностей = .

1. Текстовый файл должен располагаться по адресу «C:\PR3\», если каталога не существует, программа должна его создать автоматически, имя файла оставляется на усмотрение студента, расширение файла .txt;
2. Результат работы ПС должен выводиться в отдельный текстовый файл по адресу из п. 3, имя файла оставляется на усмотрение студента, расширение файла .txt;
3. Формат выходных данных в файле представлен ниже:

Исходная матрица:

140 300 160 50

90 2|90 5 2 0

400 4|50 1|300 5|50 0

160 3 6 8|40 0|50

Исходное значение F(x) равно 1810

Результативная матрица:

140 300 160 50

90 2 5 2|90 0

400 4 1|300 5|70 0|30

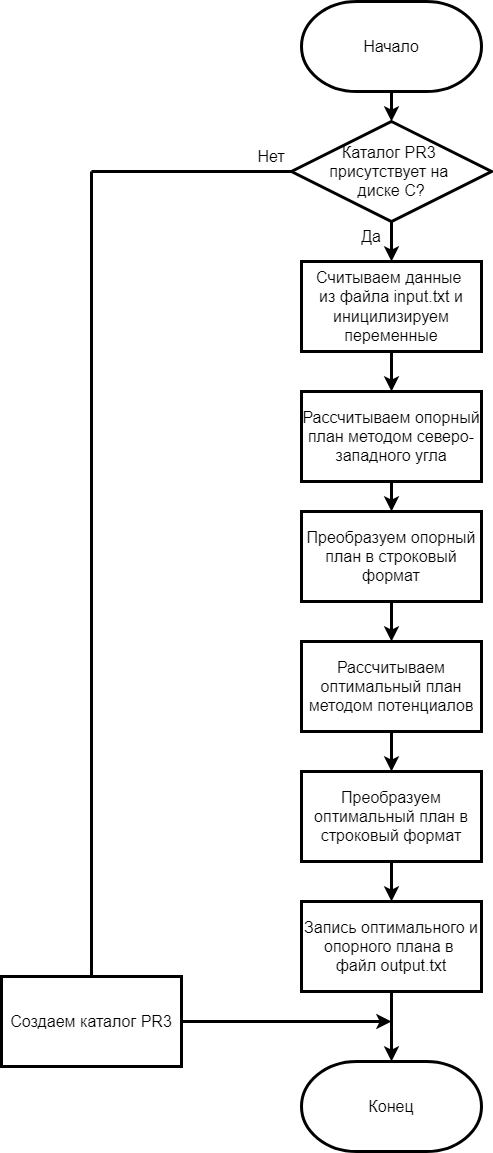
160 3|140 6 8 0|20

Оптимальное значение F(x) равно 1250

1. Если решение для заданных входных параметров не может быть найдено, то в файле должен быть следующий вывод: «С заданным условием задача неразрешима.»;
2. Поскольку ПС по условию не предусматривает явного взаимодействия пользователя с собой, то наличие какого-либо интерфейса ПС и его внешний вид оставляется на усмотрение студента;
3. Для проектирования алгоритма ПС не возбраняется (и даже поощряется) использование математических онлайн-ресурсов, которые приводят пошаговые объяснения решения поставленной задачи, например, таких как math.semestr.ru/transp/index.php.

**Ход работы**

**Общий алгоритм работы ПС**

****

**Листинг класса “Перевозка”**

Program.cs

// класс перевозка

public class Shipment

{

// Кол-во вывезенного груза

public double Quantity { get; set; }

// Стоимость перевозки

public double CostPerUnit { get; }

// номер строки в матрице распределения

public int R { get; }

// номер столбца в матрице распределения

public int C { get; }

// конструктор класса перевозка, в нем мы иницилизируем переданные значения

public Shipment(double quantity, double costPerUnit, int r, int c)

{

Quantity = quantity;

CostPerUnit = costPerUnit;

R = r;

C = c;

}

}

**Листинг полей класса Program**

Program.cs

// строка потребностей

private static int[] \_demand;

// столбец запасов

private static int[] \_supply;

// исходная матрица распределения

private static double[,] \_costs;

private static string errorMessage = "С заданным условием задача неразрешима.";

**Листинг функции получения данных из файла**

Program.cs

// Считываем данные из файла и инициализируем переменные

// Считывается - строка потребностей, столбец запасов, матрица распределения

// Помимо считывания, вводим дополнительную (фиктивную) потребность или запас

private static void Init(string filename)

{

// объявляем считываемую строку

string line;

using (StreamReader file = new StreamReader(filename))

{

// Считываем из файла первую строку (кол-во строк и столбцов исходной матрицы)

line = file.ReadLine();

// Заполняем массив значениями из строки

var numArr = line.Split();

// Иницилизируем переменную первым элементом массива (кол-во строк)

int countSupply = int.Parse(numArr[0]);

// Иницилизируем переменную вторым элементом массива (кол-во столбцов)

int countDemand = int.Parse(numArr[1]);

// проверяем правильность введенных данных (если количество строк или столбцов равно нулю

// то выводим ошибку)

if (countSupply == 0 || countDemand == 0)

{

throw new Exception();

}

// Считываем из файла вторую строку (значения исходной матрицы распределения)

line = file.ReadLine();

// Заполняем массив (одномерный) значениями из строки

var costsArray = line.Split();

// проверяем правильность введенных данных (если количество элементов

// массива не соответствует количеству строк и столбцов то выводим ошибку)

if (costsArray.Length != countDemand \* countSupply)

{

throw new Exception();

}

// Иницилизируем списки, которые будут принимать значения запасов и потребностей

List<int> supplyList = new List<int>();

List<int> demandList = new List<int>();

// Считываем из файла третью строку (столбец запасов)

line = file.ReadLine();

// Заполняем массив (одномерный) значениями из строки

numArr = line.Split();

// Заполняем список значениями запасов

for (int i = 0; i < countSupply; i++)

{

supplyList.Add(int.Parse(numArr[i]));

}

// проверяем правильность введенных данных (если количество поставщиков

// не равно количеству строк то выводим ошибку)

if (supplyList.Count != countSupply)

{

throw new Exception();

}

// Считываем из файла четвертую строку (строка потребностей)

line = file.ReadLine();

// Заполняем массив (одномерный) значениями из строки

numArr = line.Split();

// Заполняем список значениями потребностей

for (int i = 0; i < countDemand; i++)

{

demandList.Add(int.Parse(numArr[i]));

}

// проверяем правильность введенных данных (если количество потребителей

// не равно количеству столбцов то выводим ошибку)

if (demandList.Count != countDemand)

{

throw new Exception();

}

// Вводим дополнительную потребность или запас.

// Данное действие необходимо если модель исходной задачи

// является открытой (нужно привести к закрытой модели).

// В закрытой модели сумма потребностей равна сумме запасов.

// Рассчитываем суммарное кол-во запасов.

int totalSupply = supplyList.Sum();

// Рассчитываем суммарную потребность

int totalDemand = demandList.Sum();

// Если суммарная потребность больше суммарного кол-во запасов

if (totalSupply > totalDemand)

{

// то добавляем вводим дополнительную (фиктивную) потребность.

demandList.Add(totalSupply - totalDemand);

} // то же самое только с запасом

else if (totalDemand > totalSupply)

{

supplyList.Add(totalDemand - totalSupply);

}

// присваиваем массивам запасов и потребностей новые значения

\_supply = supplyList.ToArray();

\_demand = demandList.ToArray();

// иницилизируем матрицу распределения

// (дополнительный запас или потребность учитываются, так как мы уже изменили переменные)

\_costs = new double[\_supply.Length, \_demand.Length];

// заполняем матрицу распределения значениями из файла

for (int i = 0; i < countSupply; i++)

{

for (int j = 0; j < countDemand; j++)

{

// индекс считаем при этом перенося строку (i\*количество столбцов + j)

\_costs[i, j] = int.Parse(costsArray[i \* (countDemand) + j]);

}

}

}

}

**Листинг метода северо-западного угла**

Program.cs

// Получение начального опорного плана методом северо западного угла

private static Shipment[,] NorthWestCornerRule(int[] demand, int[] supply, double[,] costs)

{

// иницилизируем опорный план

Shipment[,] plan = new Shipment[supply.Length, demand.Length];

// копируем значения в локальные переменные

// Массивы это ссылочные типы, поэтому чтобы переданные значения не изменялись вне метода

// необходимо скопировать их в локальные переменные и изменять уже их.

int[] currentDemand = (int[])demand.Clone();

int[] currentSupply = (int[])supply.Clone();

double[,] currentCosts = (double[,])costs.Clone();

// Заполняем план с верхнего левого угла

// Обходим матрицу распределения (элемент матрицы это перевозка)

for (int r = 0, northwest = 0; r < currentSupply.Length; r++)

{

for (int c = northwest; c < currentDemand.Length; c++)

{

// Смотрим по перевозке в матрице распределения

// Для перевозки есть число его запасов - currentSupply[r]

// и число его потребностей currentDemand[c]

// находим минимальное из них. Оно будет являться количеством вывезенного груза

// во время перевозки

int quantity = Math.Min(currentSupply[r], currentDemand[c]);

// Если мин. число = 0 (запас или потребность данного значения является фиктивной)

// то пропускаем данную перевозку (не вносим в план)

if (quantity > 0)

{

// Иницилизируем объект класса "перевозка"

plan[r, c] = new Shipment(quantity, currentCosts[r, c], r, c);

// вычитаем минимальный элемент (кол-во вывезененого груза) из значения запаса и потребности

currentSupply[r] -= quantity;

currentDemand[c] -= quantity;

// если значение запаса равно нулю

if (currentSupply[r] == 0)

{

// то переходим на следующую строку

northwest = c;

break;

}

}

}

}

// возвращаем план

return plan;

}

**Листинг функции получения плана в строковом виде**

Program.cs

// Получение плана в строковом виде

private static string PlanToString(Shipment[,] planMatrix, int[] demand, int[] supply, double[,] costs, string titleMessage, string resultMessage)

{

string planString = titleMessage + "\n";

// значение целевой функции опорного плана

double totalCosts = 0;

// запись первой строки плана (потребности)

for (int i = 0; i < demand.Length; i++)

{

planString += demand[i] + " ";

}

planString += "\n";

// обходим весь план

for (int r = 0; r < supply.Length; r++)

{

for (int c = 0; c < demand.Length; c++)

{

// запись первого столбца плана (запасы)

if (c == 0)

{

planString += supply[r] + " ";

}

// иницилизируем объект перевозки, для текущих индексов

Shipment s = planMatrix[r, c];

// если перевозка внесена в план

if (s != null && s.R == r && s.C == c)

{

if (s.Quantity == double.Epsilon)

{

planString += costs[r, c] + " ";

}

else

{

// то записываем стоимость и количество вывезенного груза данной перевозки

planString += costs[r, c] + "|" + s.Quantity + " ";

}

// прибавляем стоимость перевозки в целевую функцию

totalCosts += (s.Quantity \* s.CostPerUnit);

} //если перевозка не внесена в план

else

{

// то записываем ее стоимость

planString += costs[r, c] + " ";

}

}

planString += "\n";

}

// вывод значения целевой функции опорного плана

planString += "\n" + resultMessage + " F(x) равно " + totalCosts + "\n";

return planString;

}

**Листинг функции записи результата в файл**

Program.cs

// метод записи ответа в файл

private static void Out(string fileName, string result)

{

// проверка на наличие файла в директории

if (File.Exists(fileName) == false)

{

// если нету то создаем

File.Create(fileName);

}

// записываем результаты в файл (параметр false означает что если в файле

// уже записаны какие либо данные то они будут удалены)

using (StreamWriter file = new StreamWriter(fileName, false))

{

file.Write(result);

}

}

**Листинг функций реализующих метод потенциалов**

Program.cs

// Метод представляющий собой алгоритм потенциалов (метод потенциалов)

// вовзращает оптимальный план

private static Shipment[,] SteppingStone(Shipment[,] planMatrix, int[] demand, int[] supply, double[,] costs)

{

// наибольшое значение по абсолютной величине

double maxReduction = 0;

// перевозки

Shipment[] move = null;

// перевозка с наибольшим значением по абсолютной величине

Shipment leaving = null;

// объект оптимального плана

Shipment[,] resultPlanMatrix = (Shipment[,])planMatrix.Clone();

// устраняем вырожденность опорного плана

FixDegenerateCase(resultPlanMatrix);

// проверяем план на оптимальность

for (int r = 0; r < \_supply.Length; r++)

{

for (int c = 0; c < \_demand.Length; c++)

{

// если перевозка есть в плане

if (resultPlanMatrix[r, c] != null)

{

// то завершаем шаг итерации

continue;

}

// создадим перевозку с грузом равным нулю

Shipment trial = new Shipment(0, \_costs[r, c], r, c);

// вовзращаем цикл пополненый нулевой перевозкой (если элементы есть

// то добавление сделало план циклическим, если нету то план ациклический)

// циклический план (является неоптимальным)

Shipment[] path = GetClosedPath(resultPlanMatrix, trial);

// сумма потенциалов

double reduction = 0;

// задаем максимальное значение, для того чтобы найти

// минимальное значения груза

double lowestQuantity = int.MaxValue;

// перевозка с минимальным значением груза

Shipment leavingCandidate = null;

// отмечаем перевозку знаком плюс

bool plus = true;

// ищем максимальную значение по абсолютной величине

foreach (var s in path)

{

// если плюс то нам надо найти наибольшую перевозку

if (plus)

{

// суммируем потенициал для этой перевозки

reduction += s.CostPerUnit;

} // если минус то находим наименьную

else

{

// вычитаем значение груза клетки

reduction -= s.CostPerUnit;

// ищем перевозку с минимальным значением груза

if (s.Quantity < lowestQuantity)

{

// запоминаем минимальную перевозку

leavingCandidate = s;

// и запоминаем минимальное значение груза

lowestQuantity = s.Quantity;

}

}

// поочередно меняем знак

plus = !plus;

}

// если сумма потенциалов (reduction) меньше значения

// максимального абсолютного значения

// то план не оптимальный

if (reduction < maxReduction)

{

// запоминаем цикл (план) при наибольшой абсолютной величине

move = path;

// запоминаем элемент с наибольшим значением по абсолютной величене

leaving = leavingCandidate;

// наибольшая абсолютная величина

maxReduction = reduction;

}

}

}

// если план неоптимальный то перестроим его

// если массив move имеет значения значит план неоптимальный

// (массив заполнен перевозками разности которых меньше нуля)

if (move != null)

{

// все что ниже это этап перераспределения поставок

// запоминаем кол-во груза (минимальное значение груза)

double q = leaving.Quantity;

// отмечаем ячейку знаком плюс

bool plus = true;

// обходим все перевозки разности которых меньше нуля

foreach (var s in move)

{

// если true то вычитаем

// если false то прибавляем

// прибавляем или вычитаем минимальное значение

s.Quantity += plus ? q : -q;

// записываем в план новое значение

// если новое кол-во груза равно нулю то эта перевозка null

// в плане ее не учитываем

// если не равно нулю то добавляем ее в план

resultPlanMatrix[s.R, s.C] = s.Quantity == 0 ? null : s;

// меняем знак (будет меняться поочередно т.к. мы в цикле)

plus = !plus;

}

// пересчитываем оптимальный план

resultPlanMatrix = SteppingStone(resultPlanMatrix, demand, supply, costs);

}

// возвращаем оптимальный план

return resultPlanMatrix;

}

// метод преобразования плана в список (из двумерного массива в список)

static List<Shipment> PlanToList(Shipment[,] plan)

{

// создаем объект списка

List<Shipment> newList = new List<Shipment>();

// заполняем список ненулевыми элементами массива

foreach (var item in plan)

{

if (null != item)

{

newList.Add(item);

}

}

// возвращаем список

return newList;

}

// проверяем план на ацикличность при его пополнении нулевой перевозкой

// возвращаем массив представляющий собой цикл перевозок (если массив без элементов

// т.е. цикла нету т.е. план ациклический)

// s - перевозка которой пополняем план

// plan - план для пополнения

static Shipment[] GetClosedPath(Shipment[,] plan, Shipment s)

{

// преобразуем план в список (из двумерного массива)

List<Shipment> path = PlanToList(plan);

// и добавляем в его конец перевозку

path.Add(s);

// удаляем (и продолжаем удалять) элементы

// у которых нет соседа по горизонтали и по вертикали

int before;

do

{

// before - количество элементов до удаления

before = path.Count;

// удаляем все элементы удовлетворяющие условию

path.RemoveAll(ship =>

{

// nbrs - соседи элемента по вертикали и горизонтали (их 2)

var nbrs = GetNeighbors(ship, path);

// условие удаления

// если хотя бы один сосед не является перевозкой

// то удаляем элемент

return nbrs[0] == null || nbrs[1] == null;

});

}

while (before != path.Count);

// рассполагаем оставшиеся элементы в правильном порядке

Shipment[] stones = path.ToArray();

// объект перевозки которой пополняем план

Shipment prev = s;

// обходим список после удаления элементов

// если элементы останутся то план циклический

for (int i = 0; i < stones.Length; i++)

{

// записываем в массив цикла перевозку

stones[i] = prev;

// записываем в значение prev ее соседа

// для того чтобы потом записать его в цикл

// возвращаем только соседей кратных двум

prev = GetNeighbors(prev, path)[i % 2];

}

// возвращаем цикл

return stones;

}

// метод поиска соседей перевозки в плане

static Shipment[] GetNeighbors(Shipment s, List<Shipment> lst)

{

// массив соседей перевозки

Shipment[] nbrs = new Shipment[2];

// ищем соседей по всему списку

foreach (var o in lst)

{

// если перевозка которую обходим по списку

// не является перевозкой для которую ищем соседей

if (o != s)

{

// то, если индекс строки перевозки в матрице распределения

// равен такому же индексу но для перевозки для которой ищем соседей

// и при этом мы не нашли соседа для перевозки по строке

if (o.R == s.R && nbrs[0] == null)

{

// то записываем соседа в массив

nbrs[0] = o;

} // если индекс столбца перевозки в матрице распределения

// равен такому же индексу но для перевозки для которой ищем соседей

// и при этом мы не нашли соседа для перевозки по столбцу

else if (o.C == s.C && nbrs[1] == null)

{

// то записываем соседа в массив

nbrs[1] = o;

}

// если нашли двух соседей, то выходим из цикла foreach

if (nbrs[0] != null && nbrs[1] != null)

{

//выход из цикла foreach

break;

}

}

}

// возвращаем соседей

return nbrs;

}

// Метод возвращает ациклический план (базисные клетки не содержат циклов)

// Метод сработает если план является невырожденым

// кол-во поставщиков + кол-во потребителей - 1.

// если данное значение не равно количеству перевозок в плане, то план невырожденный

static Shipment[,] FixDegenerateCase(Shipment[,] plan)

{

const double eps = double.Epsilon;

// объект пополненного плана

Shipment[,] fixedPlan = (Shipment[,])plan.Clone();

// условие невырожденности

if (\_supply.Length + \_demand.Length - 1 != PlanToList(plan).Count)

{

// проходим по элементам плана

for (int r = 0; r < \_supply.Length; r++)

{

for (int c = 0; c < \_demand.Length; c++)

{

// если перевозка отсутствует

if (plan[r, c] == null)

{

// то ставим туда заглушку

// (перевозку с значением кол-во груза равным эпсилону)

// т.е. ставим нулевую перевозку в недостающие клетки

// и превращаем эти клетки в базисные

Shipment dummy = new Shipment(eps, \_costs[r, c], r, c);

// Выбираем клетки для пополнения плана

// (GetClosedPath - возвращает массив представляющий собой цикл перевозок)

// если цикл перевозок равен нулю, то план является ациклическим при добавлении перевозки dummy

// значит можем добавлять

if (GetClosedPath(plan, dummy).Length == 0)

{

// ставим нулевую перевозку в недостающие клетки

plan[r, c] = dummy;

// возвращаем пополненный план

return fixedPlan;

}

}

}

}

}

// возвращаем план

return fixedPlan;

}

**Листинг основной функции программы**

Program.cs

// основная функция программы (запускается первой)

static void Main()

{

// название каталога

var catalogName = @"C:\PR3\";

// название входного файла

var inputFileName = catalogName + "input.txt";

// название файла с результатами

var outputFileName = catalogName + "output.txt";

try

{

// проверка на наличие каталога

if (Directory.Exists(catalogName) == false)

{

// если каталог отсутствует, то создаем его

Directory.CreateDirectory(catalogName);

// и создаем там файл

File.Create(inputFileName);

// выходим из программы, так как мы создали файл, но заполнить его должен пользователь

return;

}

// вызываем метод считывания данных из файла

Init(inputFileName);

// Вызываем метод реализующий алгоритм (Северо-западный угол)

// получая тем самым объект опорного плана

Shipment[,] firstPlanMatrix = NorthWestCornerRule(\_demand, \_supply, \_costs);

// преобразуем план в строковый формат

string firstPlanString = PlanToString(firstPlanMatrix, \_demand, \_supply, \_costs, "Исходная матрица:", "Исходное значение");

// Вызываем метод реализующий алгоритм потенциалов (метод потенциалов)

// получая тем самым объект оптимального плана

Shipment[,] secondPlanMatrix = SteppingStone(firstPlanMatrix, \_demand, \_supply, \_costs);

// преобразуем план в строковый формат

string secondPlanString = PlanToString(secondPlanMatrix, \_demand, \_supply, \_costs, "Результативная матрица:", "Оптимальное значение");

// вызываем метод для записи опорного и оптимального плана

Out(outputFileName, firstPlanString + "\n" + secondPlanString);

}

catch (Exception)

{

// записываем сообщение об ошибке в файл

Out(outputFileName, errorMessage);

}

}

**Демонстрация работы программной логики**

1. Проверка на наличие каталога на диске C

// название каталога

var catalogName = @"C:\PR3\";

// название входного файла

var inputFileName = catalogName + "input.txt";

// название файла с результатами

var outputFileName = catalogName + "output.txt";

// проверка на наличие каталога

if (Directory.Exists(catalogName) == false)

{

// если каталог отсутствует, то создаем его

Directory.CreateDirectory(catalogName);

// и создаем там файл

File.Create(inputFileName);

// выходим из программы, так как мы создали файл, но заполнить его должен пользователь

return;

}

Если каталог отсутствует на диске C, то мы создадим его вместе с входным файлом. Созданный входной файл будет создан пустым и в формате .txt.

1. Считывание данных из файла.

// Считываем из файла первую строку (кол-во строк и столбцов исходной матрицы)

line = file.ReadLine();

// Заполняем массив значениями из строки

var numArr = line.Split();

Считываем данные из файла по строчке.

// Иницилизируем переменную первым элементом массива (кол-во строк)

int countSupply = int.Parse(numArr[0]);

// Иницилизируем переменную вторым элементом массива (кол-во столбцов)

int countDemand = int.Parse(numArr[1]);

Получаем количество строк и столбцов.

// проверяем правильность введенных данных (если количество строк или столбцов равно нулю

// то выводим ошибку)

if (countSupply == 0 || countDemand == 0)

{

throw new Exception();

}

Если данные являются неккоректными, например количество строк или столбцов равно нулю, то вызываем исключение.

catch (Exception)

{

// записываем сообщение об ошибке в файл

Out(outputFileName, errorMessage);

}

Здесь исключение будет обработано и в файл будет выведено значение переменной errorMessage.

private static string errorMessage = "С заданным условием задача неразрешима.";

По аналогии получаем все данные из файла, построчно считывая их и проверяю на корректность.

1. Рассчитываем опорный план методом северо-западного угла.  
    // Заполняем план с верхнего левого угла

// Обходим матрицу распределения (элемент матрицы это перевозка)

for (int r = 0, northwest = 0; r < currentSupply.Length; r++)

{

for (int c = northwest; c < currentDemand.Length; c++)

{

// Смотрим по перевозке в матрице распределения

// Для перевозки есть число его запасов - currentSupply[r]

// и число его потребностей currentDemand[c]

// находим минимальное из них. Оно будет являться количеством вывезенного груза

// во время перевозки

int quantity = Math.Min(currentSupply[r], currentDemand[c]);

// Если мин. число = 0 (запас или потребность данного значения является фиктивной)

// то пропускаем данную перевозку (не вносим в план)

if (quantity > 0)

{

// Иницилизируем объект класса "перевозка"

plan[r, c] = new Shipment(quantity, currentCosts[r, c], r, c);

// вычитаем минимальный элемент (кол-во вывезененого груза) из значения запаса и потребности

currentSupply[r] -= quantity;

currentDemand[c] -= quantity;

// если значение запаса равно нулю

if (currentSupply[r] == 0)

{

// то переходим на следующую строку

northwest = c;

break;

}

}

}

Обходим матрицу распределения с верхнего левого угла, рассчитываем минимальное число вывезенного груза для перевозки. Если данное значение больше нуля (не является фиктивным), то добавляем перевозку в план и вычитаем количество вывезенного груза из соответствующих значений запаса и потребностей. Если значение запаса равно нулю после вычитания, то переходим на следующую строку матрицы распределения.

1. Запись плана в строковый формат

string planString = titleMessage + "\n";

// значение целевой функции опорного плана

double totalCosts = 0;

// запись первой строки плана (потребности)

for (int i = 0; i < demand.Length; i++)

{

planString += demand[i] + " ";

}

planString += "\n";

Записываем в первую строку потребности плана.

// обходим весь план

for (int r = 0; r < supply.Length; r++)

{

for (int c = 0; c < demand.Length; c++)

{

// запись первого столбца плана (запасы)

if (c == 0)

{

planString += supply[r] + " ";

}

// иницилизируем объект перевозки, для текущих индексов

Shipment s = planMatrix[r, c];

// если перевозка внесена в план

if (s != null && s.R == r && s.C == c)

{

if (s.Quantity == double.Epsilon)

{

planString += costs[r, c] + " ";

}

else

{

// то записываем стоимость и количество вывезенного груза данной перевозки

planString += costs[r, c] + "|" + s.Quantity + " ";

}

// прибавляем стоимость перевозки в целевую функцию

totalCosts += (s.Quantity \* s.CostPerUnit);

} //если перевозка не внесена в план

else

{

// то записываем ее стоимость

planString += costs[r, c] + " ";

}

}

planString += "\n";

}

Обходим весь план, учитывая то что первый столбец плана это запасы. Если перевозка внесена в план, то записываем ее стоимость, количество вывезенного груза и увеличиваем стоимость перевозки, если перевозки нету в плане записывает ее стоимость.

// вывод значения целевой функции опорного плана

planString += "\n" + resultMessage + " F(x) равно " + totalCosts + "\n";

Последней строкой будет являться значение целевой функции плана.

1. Метод потенциалов.

// условие невырожденности

if (\_supply.Length + \_demand.Length - 1 != PlanToList(plan).Count)

{

// проходим по элементам плана

for (int r = 0; r < \_supply.Length; r++)

{

for (int c = 0; c < \_demand.Length; c++)

{

// если перевозка отсутствует

if (plan[r, c] == null)

{

// то ставим туда заглушку

// (перевозку с значением кол-во груза равным эпсилону)

// т.е. ставим нулевую перевозку в недостающие клетки

// и превращаем эти клетки в базисные

Shipment dummy = new Shipment(eps, \_costs[r, c], r, c);

// Выбираем клетки для пополнения плана

// (GetClosedPath - возвращает массив представляющий собой цикл перевозок)

// если цикл перевозок равен нулю, то план является ациклическим при добавлении перевозки dummy

// значит можем добавлять

if (GetClosedPath(plan, dummy).Length == 0)

{

// ставим нулевую перевозку в недостающие клетки

plan[r, c] = dummy;

// возвращаем пополненный план

return fixedPlan;

}

}

}

}

}

Проверяем план на вырожденность и устраняем ее при наличии. Если сумма количества потребителей и количества источников запасов не равна количеству перевозок в плане, то план является невырожденным. Проходим по всем элементам матрицы стоимости, если перевозка с соответствующими индексами отсутствует, то создаем нулевую перевозку для пополнения плана. Проверяем план с добавленной перевозкой на ацикличность, если план ацикличный, то добавляем перевозку в измененный план.

// удаляем (и продолжаем удалять) элементы

// у которых нет соседа по горизонтали и по вертикали

int before;

do

{

// before - количество элементов до удаления

before = path.Count;

// удаляем все элементы удовлетворяющие условию

path.RemoveAll(ship =>

{

// nbrs - соседи элемента по вертикали и горизонтали (их 2)

var nbrs = GetNeighbors(ship, path);

// условие удаления

// если хотя бы один сосед не является перевозкой

// то удаляем элемент

return nbrs[0] == null || nbrs[1] == null;

});

}

while (before != path.Count);

Проверяем план на ацикличность. Для этого удаляем все перевозки, у которых нету соседа по горизонтали и вертикали.

// рассполагаем оставшиеся элементы в правильном порядке

Shipment[] stones = path.ToArray();

// объект перевозки которой пополняем план

Shipment prev = s;

// обходим список после удаления элементов

// все оставшиеся элементы составляют ациклический план

for (int i = 0; i < stones.Length; i++)

{

// записываем в массив цикла перевозку

stones[i] = prev;

// записываем в значение prev ее соседа

// для того чтобы потом записать его в цикл

// возвращаем только соседей кратных двум

prev = GetNeighbors(prev, path)[i % 2];

}

Записываем оставшиеся элементы в массив. Данный массив представляет собой ациклический план. Метод GetNeighbors – возвращает двух соседей перевозки (если соседа нету, то возвращает null).

// возвращаем ациклический план

Shipment[] path = GetClosedPath(resultPlanMatrix, trial);

Во время обхода каждого значения матрицы распределения возвращаем ацикличный опорный или любой другой план (метод SteppingStone является рекурсивным), к плану добавляется нулевая перевозка.

// отмечаем перевозку знаком плюс

bool plus = true;

// ищем максимальную значение по абсолютной величине

foreach (var s in path)

{

// если плюс то нам надо найти наибольшую перевозку

if (plus)

{

// суммируем потенициал для этой перевозки

reduction += s.CostPerUnit;

} // если минус то находим наименьную

else

{

// вычитаем значение груза клетки

reduction -= s.CostPerUnit;

// ищем перевозку с минимальным значением груза

if (s.Quantity < lowestQuantity)

{

// запоминаем минимальную перевозку

leavingCandidate = s;

// и запоминаем минимальное значение груза

lowestQuantity = s.Quantity;

}

}

// поочередно меняем знак

plus = !plus;

}

Считаем потенциалы, их разности и минимальное значение груза для каждой ячейки матрицы распределения.

// если сумма потенциалов (reduction) меньше значения

// максимального абсолютного значения

// то план не оптимальный

if (reduction < maxReduction)

{

// запоминаем цикл (план) при наибольшой абсолютной величине

move = path;

// запоминаем элемент с наибольшим значением по абсолютной величене

leaving = leavingCandidate;

// наибольшая абсолютная величина

maxReduction = reduction;

}

Проверяем ячейку на оптимальность. Если ячейка является неоптимальной для нашего плана, то запоминаем ее значения для того, чтобы перераспределить поставки в следующем цикле.

// если план неоптимальный то перестроим его

// если массив move имеет значения значит план неоптимальный

// (массив заполнен перевозками разности которых меньше нуля)

if (move != null)

{

// все что ниже это этап перераспределения поставок

// запоминаем кол-во груза (минимальное значение груза)

double q = leaving.Quantity;

// отмечаем ячейку знаком плюс

bool plus = true;

// обходим все перевозки разности которых меньше нуля

foreach (var s in move)

{

// если true то вычитаем

// если false то прибавляем

// прибавляем или вычитаем минимальное значение

s.Quantity += plus ? q : -q;

// записываем в план новое значение

// если новое кол-во груза равно нулю то эта перевозка null

// в плане ее не учитываем

// если не равно нулю то добавляем ее в план

resultPlanMatrix[s.R, s.C] = s.Quantity == 0 ? null : s;

// меняем знак (будет меняться поочередно т.к. мы в цикле)

plus = !plus;

}

// пересчитываем оптимальный план

resultPlanMatrix = SteppingStone(resultPlanMatrix, demand, supply, costs);

}

Если план неоптимальный, то перераспределяем поставки и рекурсивно вызываем метод SteppingStone (метод реализующий алгоритм потенциалов) и проходим те же шаги пока не найдем оптимальный план.

1. Запись результатов в файл

// метод записи ответа в файл

private static void Out(string fileName, string result)

{

// проверка на наличие файла в директории

if (File.Exists(fileName) == false)

{

// если нету то создаем

File.Create(fileName);

}

// записываем результаты в файл (параметр false означает что если в файле

// уже записаны какие либо данные то они будут удалены)

using (StreamWriter file = new StreamWriter(fileName, false))

{

file.Write(result);

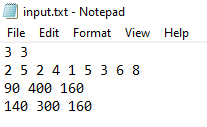
}

}

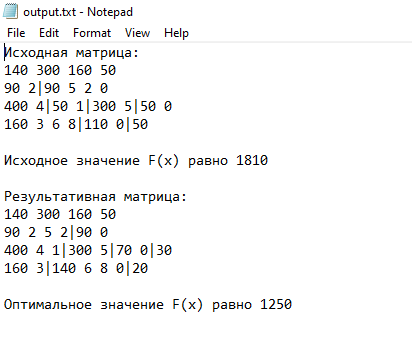
Проверяем наличие файла в директории, если файла нету, то создаем его. Записываем результаты в файл, если в файле были какие-либо данные они удаляются.

**Демонстрация входного и выходного файла**

Входной файл:



Выходной файл:



**Выводы по проделанной работе**

При разработке программного средства для решения транспортной задачи были сделаны следующие выводы по поводу метода северо-западного угла и метода потенциалов.

Метод северо-западного угла является простым в реализации, но за время работы он ни разу не вывел оптимальный план, что обусловлено особенностью его реализации, в данном методе транспортная таблица заполняется по диагонали, из-за чего пропускает большое количество возможных улучшений плана. Данный метод является неэффективным для нахождения оптимального плана, но если необходимо найти допустимый опорный план, то реализация данного алгоритма будет самым оптимальным решением.

Метод потенциалов позволяет упростить наиболее трудоемкую часть вычислений, связанную с нахождением оценки свободной клетки. В качестве достоинств данного метода можно выделить скорость определения оптимального плана, она повышается за счет того, что в алгоритме не ищутся циклы с отрицательной ценой. Недостатком является то, что при использовании метода необходимо иметь допустимый опорный план, полученный каким-либо способом, сам алгоритм не рассчитывает такой план.

Если рассматривать связку этих двух методов для совместного поиска оптимального плана, то данные алгоритмы позволяют наиболее быстро найти оптимальный план, при этом относительно быстро реализовав программное средство.