Министерство образования Республики Беларусь

Учреждение образования «Белорусский Государственный Университет Информатики и Радиоэлектроники»

Кафедра «Информатика»

Лабораторная работа № 4

«Метод потенциалов для решения матричной транспортной задачи»

**Выполнил:**

Волчецкий А. М., гр. 253504

**Проверила:**

Саттарова П.С.

Минск 2015

**Задание:**

Решить матричную транспортную задачу методом потенциалов.

**Листинг:**

public bool Solve(out Dictionary<Tuple<int, int>, double> sol)

{

basePlanValues = CalculateBasePlan();

basePlan = basePlanValues.Keys.ToList();

for (var i = 0; i < IterationsCount; i++)

{

Report += string.Format("{0} итерация\n", i+1);

Report += string.Format("\nБазовый план:\n{0}\n", basePlan.Aggregate("", (acc, x) => $"{acc} ({x.Item1}, {x.Item2};)"));

Step1CalculatePotencials();

Report += string.Format("\nПотенциалы A:\n {0}\n", aPotencials.Aggregate("", (ap, x) => ap + x + "; "));

Report += string.Format("\nПотенциалы B:\n {0}\n", bPotencials.Aggregate("", (ap, x) => ap + x + "; "));

Step2CalculateEstimations();

Report += string.Format("\nОценки:\n{0}\n", Helper.ToPrint(estimations));

WriteCostMatrix();

if (Step3CheckForOptimum())

{

sol = basePlanValues;

return true;

}

Step4GetNegativeEstimation();

double tet0;

Tuple<int, int> tet0Point;

Step5BuildCycle(out tet0, out tet0Point);

Step6BuildNewBasePlanValues(tet0);

Step7BuilNewBasePlan(tet0, tet0Point);

}

throw new Exception("iterations limit");

}

#endregion

#region Private methods

private void WriteCostMatrix()

{

var m = new DenseMatrix(a.Count, b.Count);

foreach (var el in basePlanValues)

{

m[el.Key.Item1, el.Key.Item2] = el.Value;

}

Report += string.Format("Матрица X:\n{0}\n", Helper.ToPrint(m));

}

private Dictionary<Tuple<int, int>, double> CalculateBasePlan()

{

var excludedRows = new List<int>();

var excludedCols = new List<int>();

var aNew = new List<double>(a);

var bNew = new List<double>(b);

var basePlanValues = new Dictionary<Tuple<int, int>, double>();

while (basePlanValues.Count != a.Count + b.Count - 1)

{

int iMin = -1, jMin = -1;

var cMin = Double.MaxValue;

for (var i = 0; i < c.RowCount; i++)

{

if (excludedRows.Contains(i))

{

continue;

}

for (var j = 0; j < c.ColumnCount; j++)

{

if (excludedCols.Contains(j))

{

continue;

}

if (c[i, j] < cMin)

{

cMin = c[i, j];

iMin = i;

jMin = j;

}

}

}

double xIjValue = 0;

if (iMin < 0 || jMin < 0)

{

for (var i = 0; i < c.RowCount; i++)

{

for (var j = 0; j < c.ColumnCount; j++)

{

if (!basePlanValues.Keys.Contains(new Tuple<int, int>(i, j)))

{

iMin = i;

jMin = j;

break;

}

}

if (iMin >= 0 && jMin >= 0)

{

break;

}

}

}

else

{

if (aNew[iMin] < bNew[jMin])

{

xIjValue = aNew[iMin];

excludedRows.Add(iMin);

bNew[jMin] -= aNew[iMin];

aNew[iMin] = 0;

}

else

{

xIjValue = bNew[jMin];

excludedCols.Add(jMin);

aNew[iMin] -= bNew[jMin];

bNew[jMin] = 0;

}

}

basePlanValues.Add(new Tuple<int, int>(iMin, jMin), xIjValue);

}

return basePlanValues;

}

private void Step1CalculatePotencials()

{

aPotencials = Enumerable.Repeat(double.NaN, a.Count).ToList();

bPotencials = Enumerable.Repeat(double.NaN, b.Count).ToList();

Queue<int> aKnownPotenc = new Queue<int>();

Queue<int> bKnownPotenc = new Queue<int>();

aPotencials[0] = 0;

aKnownPotenc.Enqueue(0);

while (aKnownPotenc.Count != 0 || bKnownPotenc.Count != 0)

{

if (aKnownPotenc.Count != 0)

{

var i = aKnownPotenc.Dequeue();

foreach (Tuple<int, int> x in basePlan.Where(y => y.Item1 == i))

{

int j = x.Item2;

if (Double.IsNaN(bPotencials[j]))

{

bPotencials[j] = c[i, j] - aPotencials[i];

bKnownPotenc.Enqueue(j);

}

}

}

if (bKnownPotenc.Count != 0)

{

var j = bKnownPotenc.Dequeue();

foreach (Tuple<int, int> x in basePlan.Where(y => y.Item2 == j))

{

int i = x.Item1;

if (Double.IsNaN(aPotencials[i]))

{

aPotencials[i] = c[i, j] - bPotencials[j];

aKnownPotenc.Enqueue(i);

}

}

}

}

if (aPotencials.Contains(Double.NaN) || bPotencials.Contains(Double.NaN))

{

throw new Exception("Error during potencials calculating");

}

}

private void Step2CalculateEstimations()

{

estimations = new DenseMatrix(a.Count, b.Count);

for (var i = 0; i < estimations.RowCount; i++)

{

for (var j = 0; j < estimations.ColumnCount; j++)

{

estimations[i, j] = basePlan.Contains(new Tuple<int, int>(i, j)) ? 0 : c[i, j] - (aPotencials[i] + bPotencials[j]) ;

}

}

}

private bool Step3CheckForOptimum()

{

for (var i = 0; i < estimations.RowCount; i++)

{

for (var j = 0; j < estimations.ColumnCount; j++)

{

if (!basePlan.Contains(new Tuple<int, int>(i, j)) && estimations[i, j] < 0)

{

return false;

}

}

}

return true;

}

//return (i0, j0)

private void Step4GetNegativeEstimation()

{

for (var i = 0; i < estimations.RowCount; i++)

{

for (var j = 0; j < estimations.ColumnCount; j++)

{

if (estimations[i, j] < 0)

{

negativeEstimationPoint = new Tuple<int, int>(i, j);

return;

}

}

}

throw new Exception("Negative estimation nof founded");

}

private static bool IsMoveDirectionCorrect(EMoveDirection direction, Tuple<int, int> startPoint, Tuple<int, int> nextPoint)

{

return direction == EMoveDirection.Horizontal

? startPoint.Item1 == nextPoint.Item1

: startPoint.Item2 == nextPoint.Item2;

}

private static EMoveDirection InverseMoveDirection(EMoveDirection direction)

{

return direction == EMoveDirection.Horizontal

? EMoveDirection.Vertical

: EMoveDirection.Horizontal;

}

private bool BuildCycleRecursively(Tuple<int, int> startPoint, EMoveDirection moveDirection)

{

if (!Equals(startPoint, negativeEstimationPoint) && IsMoveDirectionCorrect(moveDirection, startPoint, negativeEstimationPoint))//change to equals

{

return true;

}

for (var i = 0; i < basePlan.Count; i++)

{

if (!basePlanUsingFlags[i])

{

var nextPoint = basePlan[i];

if (IsMoveDirectionCorrect(moveDirection, startPoint, nextPoint))

{

basePlanUsingFlags[i] = true;

cyclePointsOrdered.Push(nextPoint);

if (BuildCycleRecursively(nextPoint, InverseMoveDirection(moveDirection)))

{

return true;

}

basePlanUsingFlags[i] = false;

cyclePointsOrdered.Pop();

}

}

}

return false;

}

//returns tet0, (i\*, j\*)

private void Step5BuildCycle(out double tet0, out Tuple<int, int> tet0Point)

{

basePlanUsingFlags = Enumerable.Repeat(false, basePlan.Count).ToList();

cyclePointsOrdered = new Stack<Tuple<int, int>>();

tet0 = Double.MaxValue;

tet0Point = new Tuple<int, int>(-1, -1);

if (BuildCycleRecursively(negativeEstimationPoint, EMoveDirection.Horizontal))

{

var flag = true;

foreach (var point in cyclePointsOrdered)

{

if (flag)

{

double tet = basePlanValues[point];

if (tet < tet0)

{

tet0 = tet;

tet0Point = point;

}

}

flag = !flag;

}

}

if (tet0 == Double.MaxValue)

{

throw new Exception("Tet0 not founded");

}

}

private void Step6BuildNewBasePlanValues(double tet0)

{

var flag = false;

foreach (var point in cyclePointsOrdered)

{

basePlanValues[point] += (flag ? 1 : -1) \* tet0;

flag = !flag;

}

}

private void Step7BuilNewBasePlan(double tet0, Tuple<int, int> tet0Point)

{

if (!basePlanValues.Remove(tet0Point))

{

throw new Exception("Point not founded in base plan");

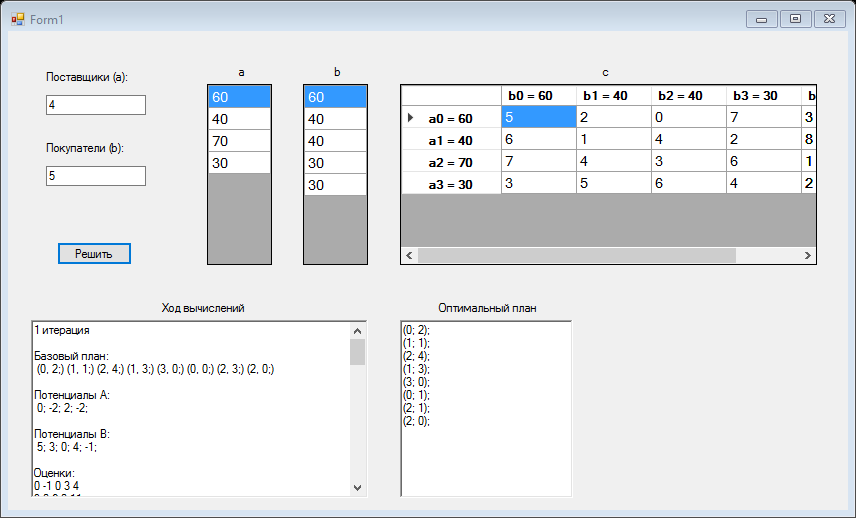
}

basePlanValues.Add(negativeEstimationPoint, tet0);

basePlan = basePlanValues.Keys.ToList();

}

**Пример:**

****