VIT®
UNIVERSITY
(Estd. u/s 3 of UGC Act 1956)
VELLORE ■ CHENNAI
www.vit.ac.in

CODE FOR THE ASSIGNMENT MODEL PROBLEM USING THE HUNGARIAN METHOD

Project in Operations Research (MEE437)

By

Mayank Jaggi (14 BCE 0848)
Kriti Gupta (14 BCE 0782)

Submitted to:

Dr. John Rajan
Professor
SMEC, VIT University

02nd November 2016

CODE FOR THE ASSIGNMENT MODEL PROBLEM USING THE HUNGARIAN METHOD

by Mayank Jaggi, Kriti Gupta

VIT University

2016

Abstract

Assignment problem comes into picture where we need to find an optimal way to assign n objects to m other objects in an injective (one-to-one) manner. The assignment problem is a well-studied topic in combinatorial optimization. It finds numerous applications in production planning, telecommunication VLSI design, economics etc. The assignment problem is a special case of transportation problem. Depending on the objective to be optimized, we obtain the typical assignment problem. Assignment problem is an important subject discussed in real physical world; we endeavour in this project to construct a simple code to solve the general case of any assignment model problem so as to reduce the time taken to solve the problem manually and avoid any chance of human error.

Introduction

The name ‘Assignment Problem’ seems to have first appeared in a 1952 paper by Votaw and Orden, what is generally recognized to be the beginning of the development of practical solution methods for and variations on the classic assignment problem. The beginning of the development of practical solution methods for and variations on the classic assignment problem was the publication in 1955 of Kuhn’s article on the Hungarian method for its solution.

Over the past 65 years, many variations on the classic AP have been proposed, a fact that becomes immediately obvious if the key words ‘assignment problem’ are entered into the search engine for the research database ABI/INFORM as different forms of the assignment model problem are discussed in daily life.

Solving the problem can be a tedious and cumbersome process, especially if we start dealing with real world problems where a large number of tasks and agents, and large values are involved. Writing a code for assignment problem, and giving real life problems based on it to a computer to solve, will reduce the time and effort to solve it every time and also reduce the chances of human error. Hence, by this project we aim to write a C++ code for the general assignment model problem to reduce time and effort required to manually solve the problem.

Literature Review

Although the name 'Assignment Problem' seems to have first appeared in a 1952 paper by Votaw and Orden, what is generally recognized to be the beginning of the development of practical solution methods for and variations on the classic assignment problem (hereafter referred to as the AP) was the publication in 1955 of Kuhn's article on the Hungarian method for its solution. In recognition of the significance of this article, Naval Research Logistics reprinted it in honour of its 50th anniversary, along with a statement by Kuhn about the development of the Hungarian method and a tribute by Frank explaining the relationship between Kuhn's technique and its antecedents in the papers by two Hungarian mathematicians.

Furthermore, in the summer of 1953, the National Bureau of Standards and other US government agencies had gathered an outstanding group of combinatorialists and algebraists at the Institute for Numerical Analysis (INA) located on the campus of University of California at Los Angeles. A rather feature of the INA was the presence of the Standards Western Automatic Computer (SWAC).

During the summer, CB Tompkins attempted to solve the 10 by 10 assignment problem by programming the SWAC to enumerate the 10! = 3,628,800 permutations of 10 objects. He never succeeded in the project.

Thus, the 10 by 10 assignment problem is a linear program with 100 non negative variable and 20 constraints. In 1953, there was no machine in the world that could solve a linear programming this large!

Kuhn, in 1955, developed a new combinatorial procedure for solving the assignment problem. The method is based on the work of two Hungarian mathematicians, J. Egervary and D. Konig [1931], and therefore Kuhn introduced the name Hungarian Method for it. The method was then sharpened by Munkres in 1957 where in he claimed the time complexity was reduced from $O(n^4)$ to $O(n^3)$.

Problem description and formulation

The classic assignment problem

The original version of the assignment problem is discussed in almost every textbook for an introductory course in either management science/operations research or production and operations management. As usually described, the problem is to find a one-to-one matching between n tasks and n agents, the objective being to minimize the total cost of the assignments. Classic examples involve such situations as assigning jobs to machines, jobs to workers, or workers to machines.

Solving by Hungarian method

The Hungarian method (also known as Flood's Technique or the Reduced Matrix method) of assignment provides us with an efficient means of finding the optimal solutions without having to make a direct comparsion of every option. It operates on a principle of matrix reduction. This just means that by subtracting and adding appropriate numbers in the cost table or matrix, we can reduce the problem to a matrix of opportunity costs (opportunity costs

show the relative penalties associated with assigning any worker to a job as opposed to making the best or least cost assignment). If we can reduce the matrix to the point where there is one zero element in each row and column, it will then be possible to make optimal assignment, i.e. assignment in which all the opportunity costs are zero. Hungarian method of assignment problem (minimization case) can be summarized in the following steps:

Steps1. Find the opportunity cost table by:

a. Subtracting the smallest number in each row of the original cost table or matrix

from every number in that row and.

b. Then subtracting the smallest number in each column of the table obtained in part (a) from every number in that column.

2. Make assignments in the opportunity cost matrix in the following way:

a. Examine the rows successively until a row with exactly one unmarked zero is found. Enclose this zero in a box ( ) as an assignment will be made there and cross (x) all other zeros appearing in the corresponding column as they will not be considered for future assignment. Proceed in this way until all the rows have been examined.

b. After examining all the rows completely, examine the columns successively until a column with exactly one unmarked zero is found. Make an assignment to this single zero by putting square ( ) around it and cross out (x) all other zeros appearing in the corresponding row as they will not be used to make any other assignment in that row, Proceed in this manner until all columns have been examined.

c. Repeat the operations (a) and (b) successively until one of the following situations arises:

i. All the zeros in rows/columns are either marked ( ) or corssed ( ) and there is exactly one assignment in each row and in each column. In such a case optimal assignment policy for the given problem is obtained.

ii. there may be some row (or column) without assignment, i.e., the total number of marked zeros is less than the order of the matrix. In such a case, proceed to next Step 4.

3. Revise the opportunity cost table: Draw the minimum number of vertical and horizontal lines necessary to cover all the zeros in the reduced cost table obtained from Step 2 by adopting the following procedure:

i. Mark (√) all rows that do not have assignments.

ii. Mark (√) all columns (not already marked) which have zeros in the marked rows [step 4 (ii)].

iii. Mark (√) all rows (not already marked) that have assignments in marked columns [step 4(iii)].

iv. Repeat steps 4(ii) and (iii) until no more rows or columns can be marked.

v. Draw straight lines through each unmarked row and each marked column.

If the number of lines drawn (or total assignments) is equal to the number of rows (or columns) then the current solution is the optimal solution, otherwise go to Step 4.

4. Develop the new revised opportunity cost table:

An optimal solution is seldom obtained from the initial opportunity cost table. Often, we will need to revise the table in order to shift one (or more) of the zero costs from its present location (covered by lines) to a new uncovered location in the table. Intuitively, we would want this uncovered location to emerge with a new zero opportunity cost. This is accomplished by subtracting the smallest number not covered by a line from all numbers not covered by a straight line. This same smallest number is then added to every number (including zeroes) lying at the intersection of any two lines.

5. Repeat Steps 2 to 4 until an optimal solution is obtained.

Proposed algorithm for the code

1) Input number of agents -> row
2) Input number of jobs -> col
3) Input assignment matrix -> matrix[row][col]
4) If row=col, balanced
   Else, unbalanced
   i. row < col, add (col - row) number of rows, with all elements 0
   ii. row > col, add (row - col) number of columns, with all elements 0
5) matrix1<-matrix
6) Find highest number in each row
7) Subtract it from all elements in that row
8) Repeat 6, 7 for all rows
9) Repeat 6 - 8 for all columns
10) matrix2<-matrix
11) Find total number of zeroes in first row and store in first element of rowzero array
12) Find total number of zeroes in first column and store in first element of colzero array
13) Add all colzero values -> totalzero
14) Highest value from colzero and rowzero -> high
15) Check arrays rowzero and colzero one by one
16) If value=high, in row, add row number to rowelimintenum array, lines=lines+1 and make all 0s to -1 in that row
    If value=high, in column, add column number to colelimintenum, lines=lines+1 and make all 0s to -1 in that column
17) If totalzero != 0, repeat steps 6 - 16
    If totalzero=0, check optimal
    i. If lines!=order of matrix, not optimal
    ii. If lines=order of matrix, optimal
18) If not optimal, check for each element in matrix, flag0=0, flag1=0
    i. If row number present in roweliminatenum array, flag0=1
    ii. If col number present in coleliminatenum array, flag1=1
    iii. If flag0=1 and flag1=1, add 1 to the element

iv. If flag0=0 and flag1=1, subtract 1 from the element
v. Now repeat steps 6 - 17 for this matrix

19) If optimal, matrix<-matrix2
20) For each row, check all -1s; calculate total number of -1s in it row and column combined
21) The -1 with the least number of total -1s is selected and made 0, the value of -1s in its row and column is made -2
22) Check all elements one by one, totaltime=0
23) If element= -2, pick value of that element from matrix1 and add to totaltime
24) totaltime is the minimum time that can be obtained.

The C++ code developed

```
#include<iostream>
#include<stdlib.h>
#include<stdio.h>
#include<windows.h>
using namespace std;
class AssignmentModel
{
      int matrix[10][10], matrix1[10][10], matrix2[10][10],
rowzero[10], colzero[10], totalzero, row, col,
roweliminate[10],coleliminate[10];
      int roweliminatenum, coleliminatenum, lines, n;
      public:
      AssignmentModel()
      {
            totalzero=0; row=0; col=0; roweliminatenum=0;
coleliminatenum=0; lines=0; n=0;
      }
      void input();
      void print();
      void reduce();
      void zeroes();
      void cut();
      void eliminate();
      void optimal();
      void iterate();
      void assign();
      int calculate(int, int);
      void balance();
      void final();
};
int main()
{
      AssignmentModel a1;
      a1.input();
      getchar();
      return 0;
}

//input
void AssignmentModel::input()
{
      cout<<"\nAssignment Problem\n";
      cout<<"\nEnter no. of people/machines(rows): ";
      cin>>row;
```

```
	cout<<"Enter no. of jobs(cols): ";
	cin>>col;

	//take input in the 2D array
	for(int i=0; i<row; i++)
	{
		cout<<"\nEnter for person/machine "<<char(65+i)<<endl;
		for(int j=0; j<col; j++)
		{
			cout<<"job "<<j+1<<": ";
			cin>>matrix[i][j];
			matrix1[i][j]=matrix[i][j];
		}
	}
	//check balanced or unbalanced
	if(row==col)
	{
		cout<<"\n\nBalanced problem\n";
		n=col;
	}
	else
	{
		cout<<"\n\nUnbalanced problem\n";
		this->balance();
	}
	//print input
	this->print();
	//call reduce function
	this->reduce();
}

//print function
void AssignmentModel::print()
{
	cout<<"\nElements of the matrix are-\n\n\t";
	for(int i=0; i<n; i++)
	{
		cout<<i+1<<"\t";
	}
	for(int i=0; i<n; i++)
	{
		cout<<endl<<char(65+i)<<"\t";
		for(int j=0; j<n; j++)
		{
			cout<<matrix[i][j]<<"\t";
		}
```

```
      }
}

//balance
void AssignmentModel::balance()
{
    if(row>col)
    {
        int d=row-col;
        n=row;
        for(int i=col; i<n; i++)
        {
            for(int j=0; j<n; j++)
            {
                matrix[j][i]=0;
            }
        }
    }
    else if(col>row)
    {
        int d=col-row;
        n=col;
        for(int i=row; i<n; i++)
        {
            for(int j=0; j<n; j++)
            {
                matrix[i][j]=0;
            }
        }
    }
    for(int i=0; i<n; i++)
      {
            for(int j=0; j<n; j++)
            {
                 matrix1[i][j]=matrix[i][j];
            }
      }
      this->print();
      this->reduce();
}

//reduction
void AssignmentModel::reduce()
{
      //row reduction
      cout<<"\n\nPerforming row reduction-\n";
```

```
	for(int i=0; i<n; i++)
	{
		int least=matrix[i][0];
		for (int j=0; j<n; j++)
		{
			if(matrix[i][j]<least)
			least=matrix[i][j];
		}
		for (int j=0; j<n; j++)
		{
			matrix[i][j]-=least;
		}
	}
	//print reduced row values
	this->print();
	//col reduction
	cout<<"\n\nPerforming column reduction-\n";
	for(int i=0; i<n; i++)
	{
		int least=matrix[0][i];
		for (int j=0; j<n; j++)
		{
			if(matrix[j][i]<least)
			least=matrix[j][i];
		}
		for (int j=0; j<n; j++)
		{
			matrix[j][i]-=least;
		}
	}
	//transfer reduced values to matrix2
	for(int i=0; i<n; i++)
	{
		for(int j=0; j<n; j++)
		{
			matrix2[i][j]=matrix[i][j];
		}
	}
	//print reduced col values
	this->print();
	//call zeroes function to calculate no. of zeroes in each row
and column
	this->zeroes();
}

//zeroes function
```

```
void AssignmentModel::zeroes()
{
	//calculating zeroes in rows
	for(int i=0; i<n; i++)
	{
		int temp=0;
		for(int j=0; j<n; j++)
		{
			if(matrix[i][j]==0)
			temp++;
		}
		rowzero[i]=temp;
	}
	//calculating zeroes in cols
	for(int i=0; i<n; i++)
	{
		int temp=0;
		for(int j=0; j<n; j++)
		{
			if(matrix[j][i]==0)
			temp++;
		}
		colzero[i]=temp;
	}
	//printing matrix with no. of zeroes
	cout<<"\n\nValue of the matrix with no. of zeroes in each
column and row\n\n\t";
	for(int i=0; i<n; i++)
	{
		cout<<i+1<<"\t";
	}
	cout<<"0";
	for(int i=0; i<n;i++)
	{
		cout<<endl<<char(65+i)<<"\t";
		for(int j=0; j<n; j++)
		{
			cout<<matrix[i][j]<<"\t";
		}
		cout<<rowzero[i];
	}
	cout<<"\n0";
	//calculating total number of zeroes in matrix
	totalzero=0;
	for(int i=0; i<n; i++)
	{
```

```
            cout<<"\t"<<colzero[i];
            totalzero+=colzero[i];
      }
      cout<<"\t"<<totalzero;
      //cut function call to start cutting lines
      this->cut();
}

//cut function
void AssignmentModel::cut()
{
      while(totalzero!=0)
      {
            this->eliminate();
      }
      if(totalzero==0)
      {
            this->optimal();
      }
}

//eliminate function
void AssignmentModel::eliminate()
{
      //calculating maximum number of zeroes in columns and rows
      int high=rowzero[0], flag=0, row=0, col=0;
      for(int i=0; i<n; i++)
      {
            if(rowzero[i]>high)
            {
                  high=rowzero[i];flag=0;
                  row=i;
            }
            if(colzero[i]>high)
            {
                  high=colzero[i];flag=1;
                  col=i;
            }
      }
      if(row>=col&&flag!=1)
      {
            roweliminate[roweliminatenum]=row;
            roweliminatenum++;
            goto a;
      }
      else
```

```
	{
		coleliminate[coleliminatenum]=col;
		coleliminatenum++;
		goto b;
	}
	a://if row is to eliminated
		for(int i=0; i<n; i++)
		{
			if(matrix[row][i]==0)
			matrix[row][i]--;
		}
		cout<<"\n\nrow "<<row+1<<" eliminated\n";
		lines++;
		goto c;
	b://if column is to be eliminated
		for(int i=0; i<n; i++)
		{
			if(matrix[i][col]==0)
			matrix[i][col]--;
		}
		cout<<"\n\ncol "<<col+1<<" eliminated\n";
		lines++;
	c:
		this->print();
		//calculate zeroes again after a line is eliminated
		this->zeroes();
}

//optimal function

void AssignmentModel::optimal()
{
	if(lines==n)
	{
		cout<<endl<<"lines= "<<lines<<endl;
		cout<<"\nOPTIMALITY ACHIEVED!\n";
		this->assign();
		this->final();
		exit(0);
	}
	else
	{
		cout<<"\nNOT AN OPTIMAL SOLUTION!\n";
		lines=0;
		//iterate if optimality not achieved
		this->iterate();
```

```
	}
}

//iterate function
void AssignmentModel::iterate()
{
	int lines=0;
	//transfer matrix1 to matrix
	for(int i=0; i<n; i++)
	{
		for(int j=0; j<n; j++)
		{
			matrix[i][j]=matrix2[i][j];
		}
	}

	int least=100;
	//obtaining least value from uncovered cells
	for(int i=0; i<n; i++)
	{
		for(int j=0; j<n; j++)
		{
			int flag=0;
			for(int k=0; k<roweliminatenum; k++)
			{
				if(i==roweliminate[k])
				{
					flag=1;
					break;
				}
			}
			for(int k=0; k<coleliminatenum; k++)
			{
				if(j==coleliminate[k])
				{
					flag=1;
					break;
				}
			}
			if(flag==0)
			{
				if(matrix[i][j]<least)
				{
					least=matrix[i][j];
				}
			}
```

```
            }
      }
      //least obtained out of uncovered cells
      //subtracting and adding least values
      for(int i=0; i<n; i++)
      {
            for(int j=0; j<n; j++)
            {
                  int flag1=0, flag2=0;
                  for(int k=0; k<roweliminatenum; k++)
                  {
                        if(i==roweliminate[k])
                        {
                              flag1=1;
                              break;
                        }
                  }
                  for(int k=0; k<coleliminatenum; k++)
                  {
                        if(j==coleliminate[k])
                        {
                              flag2=1;
                              break;
                        }
                  }
                  if(flag1==1&&flag2==1)
                  {
                        matrix[i][j]+=least;
                  }
                  else if(flag1==0&&flag2==0)
                  {
                        matrix[i][j]-=least;
                  }
            }
      }
      //calculating zeroes in the new matrix formed
      this->zeroes();
}

//assign function
void AssignmentModel::assign()
{
      for(int i=0; i<n; i++)
      {
            int temp[10], count=0;
            for(int j=0; j<n; j++)
```

```
            {
                  if(matrix[i][j]==-1)
                  {
                        temp[count]=this->calculate(i,j);
                        count++;
                  }
            }
            int least=100,k=0;
            while(k<count)
            {
                  if(temp[k]<least)
                  {
                        least=temp[k];
                  }
                  k++;
            }
            int counter1=0;
            for(int l=0; l<n; l++)
            {
                  if(matrix[i][l]==-1)
                  {
                        if(temp[counter1]==least)
                        {
                              matrix[i][l]++;
                              for(int x=i;x<n;x++)
                              {
                                    if(matrix[x][l]==-1)
                                    matrix[x][l]=-2;
                              }
                              for(int y=l;y<n;y++)
                              {
                                    if(matrix[i][y]==-1)
                                    matrix[i][y]=-2;
                              }
                        }
                        else
                        {
                              matrix[i][l]=-2;
                        }
                        counter1++;
                  }
            }
      }
      this->print();
      getchar();
}
```

```
//calculate function
int AssignmentModel::calculate(int l, int m)
{
      int temp=0;
      for(int i=l+1; i<n; i++)
      {
            if(matrix[i][m]==-1)
                  temp++;
      }
      for(int j=0; j<n; j++)
      {
            if(matrix[l][j]==-1)
                  temp++;
      }
      return temp;
}

//final function
void AssignmentModel::final()
{
    cout<<"\n\nFinal assignments are-\n";
    int totaltime=0;
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<n; j++)
        {
            if(matrix[i][j]==0)
            {
                cout<<char(i+65)<<" -> "<<j+1<<" time =
"<<matrix1[i][j]<<endl;
                totaltime=totaltime+matrix1[i][j];
            }
        }
    }
    cout<<"\nTotaltime is "<<totaltime<<endl;
    getchar();
}
```

Results and discussions

In this project, the solving of the assignment model by hand is made into a code to reduce the time and possibility of human error. The space require for Hungarian to solve the problem is $O(n^2)$.The applicability of the Hungarian model presented in this paper is most satisfying, since in addition to it being based on sound behavioural assumptions, which lead to a clear mathematical programming formulation, it is solved by a polynomial time algorithm that has time and memory requirements comparable to those of a shortest path algorithm.
The benefits of having a computer program that calculates the assignments over doing the same manually are-

1. Time taken to solve reduces drastically, especially if we consider large order problems.
2. No or negligible chance of error, especially if we consider large order problems.
3. Very cheap, as zero or negligible human labour required to execute the code.

Conclusions and future scope

The assignment model problem has a large number of applications in various fields – from factories to component placing on chips to image processing. The following are some places where the assignment model is used in today's date-

1. Optimal assignment of factories.
2. Hospital Layout
3. Airport gate assignment
4. Minimize total passenger movement
5. Minimize total baggage movement
6. Steinberg wiring problem
7. component placing on circuit boards
8. Minimizing the number of transistors needed on integrated circuits
9. Optimal placing of letters on keyboards
10. Optimal placing of letters on touchscreen devices
11. To balance the turbine runner in electricity generation
12. Bandwidth minimization of a graph
13. Image processing
14. Economics
15. Molecular conformations in chemistry
16. Scheduling
17. Supply Chains
18. Manufacturing lines

References

[1] Konig, D., “Uber Graphen und ihre Anwendung auf determinantentheorie und Mengenlehre.” Math. Ann, 77 (1916) 453-465.

[2] Egervary, J., “Matrixok kombinatorius tuljdonsagairol.” Mat. Fiz. Lapok (1913) 16-28 (translated as “Combinatorial properties of Matrices” by H. W. Kuhn, ONR Logistics Project, Princeton (1953), mimeographed).

[3] H.W. Kuhn, The Hungarian method for the assignment problem, Naval Research Logistics Quarterly 2 (1&2) (1955) 83–97 (original publication).

[4] J. Munkres, “Algorithms for the assignment and transportation problems,” J. Society for Industrial and Applied Mathematics, vol. 5, pp. 32–38, 1957.

[5] Shweta Singh, G.C. Dubey, Rajesh Shrivastava, A Comparative Analysis of Assignment Problem, IOSR Journal of Engineering (IOSRJEN), Volume 2, Issue 8 (August 2012), PP 01-15

[6] David W. Pentico, Assignment problems: A golden anniversary survey, European Journal of Operational Research 176 (2007) 774–793

[7] Jameer. G. Kotwal, Tanuja S. Dhope, Unbalanced Assignment Problem by Using Modified Approach, International Journal of Advanced Research in Computer Science and Software Engineering, Volume 5, Issue 7, July 2015

[8] Anna M Gil-Lafuente, Luciano Barcellos de-Paula, José M Merigó-Lindahl, Fernando Augusto Silva-Marins, Antonio Carlos de Azevedo, Decision Making Systems in Business Administration, Ritto World Scientific, 29-Nov-2012