Experimental Comparison of Hungarian and Auction Algorithms to solve the Assignment Problem[*]

Arunachalam Narayanan
chalamy@tamu.edu

Bharani Babu Nagarathnam
bharani@tamu.edu

Murugappan Meyyappan
murgi@tamu.edu

Sorajack Mongkolsri
sorajack15@hotmail.com

Department of Industrial Engineering
Texas A&M University
College station, TX

Abstract:

Machine assignment problems are the central task in manufacturing planning. This project aims to show the difference between the use of centralized and distributed algorithm to solve the classical assignment problems. Centralized algorithms are based on the approach of a central agent trying to achieve optimization by considering all the elements in the system. In distributed algorithms several independent agents while trying to optimize within their bounds interact with each other to achieve a global optimization. Hungarian and Auction Algorithm are used as centralized and distributed algorithms respectively. The algorithms and solution concepts are explained. We test our algorithms by using 180 sample problems generated by a random generator with different ranges of cost, matrix density and random seed number. The solutions are compared in terms of quality of solution, time efficiency, and memory requirements. It is shown that the Hungarian algorithm performs better for assignment problems than Auction Algorithm. In cases, where we don't have data to perform a centralized computation we use Auction Algorithm.

Introduction

Industrial Engineers employ assignment techniques to position the man, machine or the jobs available with them, in an efficient and economic way. Assignment Problems are generally a type of Distribution Problems, for which a solution can be derived by using Transportation method, Linear Programming, Complete Enumeration, or by Branch & Bound methods. Allocation of single machine to one man, single machine to two

[*] This project is done as a part of INEN 654-Manufacturing Systems Modeling and analysis course during fall 2000.

men, various jobs to one man, indicating the minimization of energy and maximization of resource utilization etc. Depending on the result of the assignment problems. Even though there are many assignment procedures available, one would prefer a fastest, easiest and the best method to assign the resources. With the implementation of computers, the deciding factors mentioned above can be determined by the optimality of the final solution, time required to process the values fed and based on the memory requirements of the algorithm. Based on these ideas, our aim is to compare the Auction Algorithm and the Hungarian Algorithm for the Assignment purposes.

Albeit both the algorithms fall into the assignment techniques, Auction Algorithm is used to maximize the profit involved with the allocation, while Hungarian Algorithm deals with the minimization of the total cost involved in, say, assigning a man to a machine.

Hungarian Algorithm:

Hungarian Algorithm, a simple solution algorithm and a method that appears unrelated to the Transportation model, takes its natal from the Simplex method as of the Transportation problems. The main objective of Hungarian Algorithm is to minimize the assignment cost of the system. Given the cost matrix for the assignment problem taken into consideration, a simplex-based procedure of explanation for this algorithm is as follows:

Step1: From the original cost matrix, scan through each row and find out the minimum entry in every row. Subtract this minimum number from all the entries of the respective row.

Step2: For the matrix resulting in the above step, identify a minimum entry in each column. Subtract it from all the entries of the respective column.

Step3: Recognize the optimal assignment as where only a single zero appears in any row or column.

Step4: Make random assignments using the remaining zero cost cells and stop when all optimal assignments have been made.

Step5: Even now if the assignment is not made, check each row without an assignment and check each column that has a zero in a checked row.

Step6: From the resultant, check each row that has an assignment in a checked column.

Step7: Start to repeat the steps 5 & 6 until no more checks can be made further.

Step8: Draw a line through each unchecked row and checked column.

Step9: Updating the matrix: Subtract the minimum unchecked element from all the unchecked elements and add it to the elements at the intersection of the striking lines.

Step 9: Go to step 3

The Auction Algorithm

The Auction Algorithm is an intuitive method for solving the classical assignment problem developed by Dimitri P. Bertsekas in 1979 and subsequently revised in 1990. It outperforms substantially its main competitors for important types of problems, both theoretically and practically. This Algorithm is based on the process of auction, where each bidder raises the price of his her preferred object by the bidding increment. Just as in real auction, bidding increments and price increases spur competition by making the bidder's own preferred object less attractive to other potential bidders.

In the classical assignment problem there are *n* persons and *n* objects that have to be matched on a one on one basis. There is a benefit a_{ij} for matching person i with object *j*. The objective of the problem is to determine the most profitable assignments between persons and objects. In order to determine such assignments Auction Algorithm apply the economical concepts of prices and equilibria by viewing each person *i* as the economic agent acting in his own best interest. Suppose that object *j* has a price p_j and the person who wants this object must pay this price. Hence the total net value of object for each person is $a_{ij} - p_j$. Intuitively, each person should be assigned to the object that is most profitable for him or her, with $a_{ij} - p_j = max\ \{a_{ij} - p_j\};\ j=1\ldots n.$ If this condition holds for every person, then a set of prices is at equilibrium and all person are happy.

Auction Process

Step 1: The algorithm starts when every person is randomly assigned to the object. and the price of each object p_j is set to zero.

Step 2: Compute the amount of minimum bid $e = 1/(n-1)$ where *n* is the number of persons or number of objects that participate in the Auction process. We take the value of e to be $1/(n-1)$, because it has been proved that for values

$e = 1/(n-1)$, the program doesn't converge and we should take a slightly lesser value.

Step 3: A random person who is the unhappy is selected.

Step 4: Exchange object of the unhappy person with the person assigned to object k at the beginning of the round. The new price of the bidding object k will be computed by $newp_k = oldp_k + g_i + e$ where $g_i = v_i - w_i$ which is the largest increment by which the best object price p_k can be increased so that object k still be the best object for person i. v_i is the best object value for most unhappy agent $i = max_j \{a_{ij} - p_j\}$ and w_i is second best object value $i = max_j \{a_{ij} - p_j\}; j^1 j_i$

Step 5: Update the profit of object k by subtracting new bidding price p_k from the profit of assigning object k to each agent i (a_{ik}).

Step6: Repeat the process by picking up a new random unhappy person and proceed until the optimal condition $a_{ij} - p_j = max \{a_{ij} - p_j\}; j=1\ldots n$ holds (every person is happy with his current object and the price equilibrium has been reached).

If we want to make this auction algorithm centralized, instead of taking a random unhappy person, we take the most unhappy person from the matrix by $max [max \{a_{ij} - p_j\} - \{a_{ij} - p_j\}]; j=1\ldots n$. This makes the algorithm converge faster.

Coding procedure and explanation

Hungarian Algorithm

Step1: The algorithm proceeds to the traditional method of row and column reduction. This step requires $2n^2$ iterations for each reduction (row and column).

Step 2: After this step, we count the number of zeros in each row and column. We make first assignments to the rows and columns, which have unique zeros dynamically, meaning, during the process of assignments if any row or column comes up with a unique zero will also be assigned.

Step 3: Then there may be rows and columns, which has more than one zero. Here we make the arbitrary assignments (for the reaming cells). It is a heuristic process; we take a method, which had solved all the test problems we generated (apart from the 180 problems, we tested 10,000 5 x 5 matrix problems and some higher order matrix problems like 200,500,1000). In all the cases we were able to get optimal assignments. But still this is a heuristic step in the process and it may fail in certain cases. This can be avoided by using the alternate method available at that point of heuristic decision.

A sample matrix in which our heuristic works is shown below

0	5	1	0	9	1	7
6	9	0	0	0	5	0
2	0	0	9	2	0	5
0	3	2	7	6	1	4
6	0	0	9	2	0	5
0	4	0	2	0	10	0
3	0	1	5	9	9	8

1st Assignment	row4,col1
2nd Assignment	row7,col2
3rd Assignment	Row1,col4
4th Assignment	Row5,col3

5th Assignment	row2,col5
6th Assignment	row3,col6
7thAssignment	row6,col7

If we proceed through other way , like the one shown below for the same matrix, it shows that an optimal assignment is not feasible,

0	5	1	🟥 0	9	1	🟩 7
6	9	🟥 0	0	0	5	🟩 0
2	0	0	9	2	🟥 0	🟩 5
🟥 0	3	2	7	6	1	🟩 4
🟩 6	🟩 0	🟩 0	🟩 9	🟩 2	🟩 0	🟩 5
0	4	0	2	🟥 0	10	🟩 0
3	🟥 0	1	5	9	9	🟩 8
🟩	Row and Column Without an Assignment					

1st Assignment	row4,col1
2nd Assignment	row7,col2
3rd Assignment	Row1,col3
4th Assignment	Row2,col3
5th Assignment	Row3,col6
6th Assignment	Row6,col5
7thAssignment	Not possible

This problem has aroused because at the point of selecting an arbitrary assignment to the zero cell, we need to have a definite heuristic method, but here we selected the 4th assignment as the element in row 2 and column 3, which is first zero element that occurs in the selection process.

But our heuristic works, since we select the row or column that has the minimum number of zeros (more than one) and the assign arbitrarily.

Step 4: After all the assignments have been made, we check the rows that were not assigned. Then we check each column that has a zero in the checked column. From the resultant we check each row that has an assignment in the checked column. The number of strikes is given by the number of unchecked rows and unchecked columns.

Step 5: If the number of strikes is not equal to the number of rows, we proceed to step 2 and repeat the process, until an optimal assignment is made.

We make heuristic assignment in each and every step were we may encounter more than one row or column having the same number of zeros(greater than 1). When all the optimal assignments have been made we calculate the cost based on the assigned cells.

The syntax for execution of Hungarian algorithm is, Hungarian <filename>

It stores the result in two files

1) answer.txt -- Has the optimal cost, time required for execution , and the number of iterations (here we term iteration as one matrix operation involved after the row and column reduction).
2) Assignments.txt – The assignments are stored in this file.

This file can execute multiple problems that are in the text file. And the results are stored in the files with their case numbers.

The Auction process:

This is a distributed algorithm. The coding complexity is less compared to Hungarian. In this algorithm our initial assignment is diagonal in nature, that is person 1 is assigned to n^{th} object, person 2 is assigned to $n\text{-}1^{th}$ object and so on.
The epsilon we use is restricted to three decimals. The epsilon is calculated as $1/n$, where n is the size of the matrix. Here the Limitation for the size of matrix is 1000. Beyond which the algorithm cannot proceed. We can overcome it by eliminating the limitation on the number of decimal places in the epsilon value.

Step 1: We scan through each row and find out the unhappy persons, they are stored in an array called unhappy.

Step 2: If all persons are happy we exit, else we generate a random number with a limit as the size of the unhappy array.

Step 3: We select the unhappy person (selected by the random number) and assign him the new object using the Bertsekas principle (The new price of the bidding object k will be computed by $newp_k = oldp_k + g_i + e$ where $g_i = v_i - w_i$ which is the largest increment by which the best object price p_k can be increased so that object k still be the best object for person i. v_i is the best object value for most unhappy agent i = $\max_j \{a_{ij} - p_j\}$ and w_i is second best object value = $\max_{j \neq ji} \{a_{ij} - p_j\}$.)

Step 4: We switch the assignments between the two persons considered, the one chosen and the person who was assigned to the object, which is been bid by the latter.

Step 5: We proceed to step 1. and the process is continued until all are happy.

The syntax for execution of Auction algorithm is, Auction<filename>

It stores the result in two files

3) answer.txt -- Has the optimal cost, time required for execution, and the number of iterations (here we term iteration as one matrix operation involving updating every unhappy person to a happy person).
4) Assignments.txt – The assignments are stored in this file.

This file can execute multiple problems that are in the text file. And the results are stored in the files with their case numbers.

This algorithm has different execution time and iterates through different number of iterations each item, due to the random selection of the unhappy row. This algorithm is slower compared to Hungarian, which is centralized, but if the situation demands this type of algorithm, like a distributed system, we will use Auction algorithm.

Analysis of the results

Quality of solution

The quality of solution is measured as a function of cost of assignment. In our case both the Hungarian and Auction yields the same assignment cost for all the 180 test problems. This shows that in terms of quality of solution there is no difference between both the algorithms. To make sure that the quality of solution is consistent for any problem, we test run the algorithms for problem sizes of matrices varying from 200 –1000. From the results shown below we can conclude that the quality of solution is consistent for both the Algorithms.

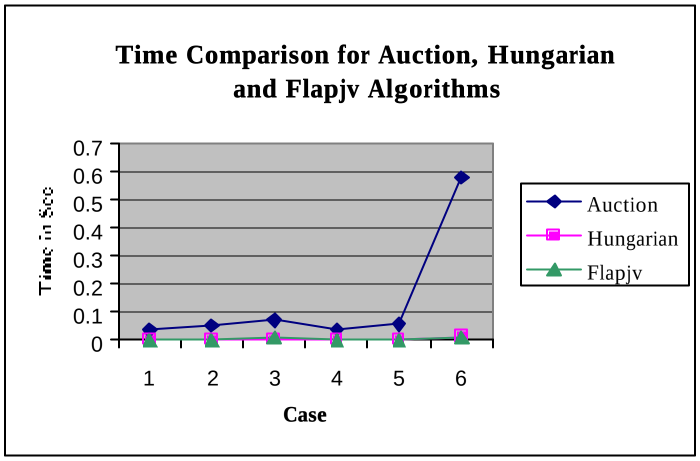
Auction Algorithm					
Matrix Size	**200**	**300**	**400**	**500**	**1000**
Density					
0.5	0.38 /0	1.222/ 0	2.854 / 0	7.03 /0	42.5 / 0
0.75	0.641 / 0	1.923 / 0	5.258 / 0	10.735 / 0	77.5 / 0
1	0.771 / 0	2.664 / 0	6.92 / 0	12.207 / 0	92.6 / 0
Hungarian Algorithm					
Matrix Size	**200**	**300**	**400**	**500**	**1000**
Density					
0.5	0.02 / 0	0.03 /0	0.07 / 0	0.1 /0	0.440/ 0
0.75	0.02 /0	0.04/ 0	0.06 /0	0.1 / 0	0.451 / 0
1	0.02 / 0	0.04 / 0	0.5 / 0	0.9 / 0	0.471 / 0

Note: The computational time and cost are given as Time / Cost in the matrix cells

Time efficiency

We use time as a measure of performance of the algorithm. In addition to comparing the time required to execute the code of both the algorithms, the time efficiency is also compared with the *flapjv* program provided by the instructor. It's evident that the Hungarian algorithm out performs Auction as the density of the matrix increases. The Auction algorithm is almost 50 times slower for large matrices with density one.

The reason for this slow performance of auction algorithm is that the increment in Auction is very small and thus results in a large number of iterations consuming time.



Time to execute in Seconds

Auction	Hungarian	Flapjv
0.036	0.002	0.000
0.052	0.003	0.000
0.073	0.002	0.010
0.038	0.003	0.000
0.058	0.003	0.000
0.578	0.011	0.010

Time Comparison for Large Size Matrices

Auction Algorithm					
Matrix Size	**200**	**300**	**400**	**500**	**1000**
Density					
0.5	0.38	1.222	2.854	7.03	42.5
0.75	0.641	1.923	5.258	10.735	77.5
1	0.771	2.664	6.92	12.207	92.6
Hungarian Algorithm					
Matrix Size	**200**	**300**	**400**	**500**	**1000**
Density					
0.5	0.02	0.03	0.07	0.1	0.44
0.75	0.02	0.04	0.06	0.1	0.451
1	0.02	0.04	0.5	0.9	0.471

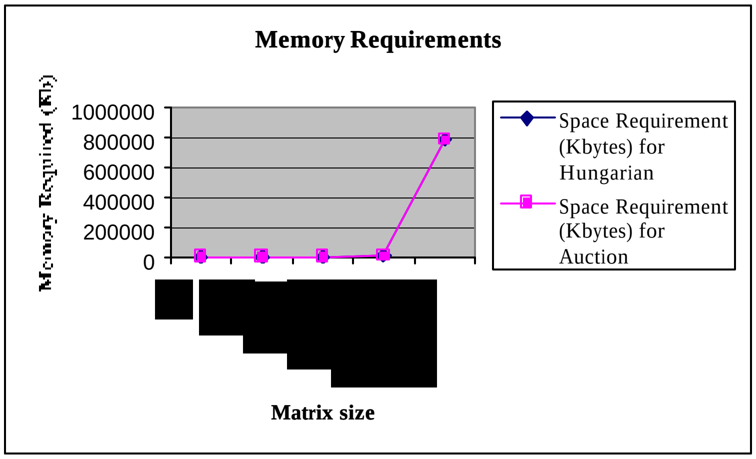
Computational Complexity

Computational complexity is a measure of the amount of complex operations that are necessary to perform the algorithm. It can be measured in terms of memory required to perform these operations.

For Hungarian algorithm our program needs $8n^2 + 12n$ memory locations

The orgsource, source are double pointers which requires 4 bytes of n^2 memory locations which adds up to 8 n^2 memory locations. The row score, colscore, colcheck, rowcheck are Boolean values that occupy 4n bytes. The rowzerocount, colzerocount are integers which occupy 4bytes each which accounts for 8n bytes. In total the Hungarian program will require $8n^2 + 12n$ memory locations.

For Auction algorithm our program needs $8n^2 + 20n$ memory locations. The orgsource, source are double pointers which requires 4 bytes of n^2 memory locations which adds up to 8 n^2 memory locations. The rowassign, rowfirstmatrix, rowsecmatrix are int values which occupy 3x4n bytes. The firstmaxcol and assignment are floats which occupy 2x4n bytes. Thus in total the Auction algorithm requires $8n^2 + 20n$ memory locations.



Note: Due to the minor difference in the values, both the lines in the plot merge.

Space Requirement Comparison Between Hungarian and Auction Algorithm		
Matrix Size	**Space Requirement (Kbytes) for Hungarian**	**Space Requirement (Kbytes) for Auction**
5x5	0.25390625	0.29296875
10x10	0.8984375	0.9765625
100x100	79.296875	80.078125
1000x1000	7824.21875	7832.03125
10000x10000	781367.1875	781445.3125
Note: Space Requirement for Hungarian: $8(n^2)+12n$		
Note: Space Requirement for Auction: $8(n^2)+20n$		

Hence we can say that there is no major difference between memory requirements for both algorithms.

Conclusion

The Hungarian and Auction algorithms are compared numerically in terms of quality of solution, computational time, number of iterations and memory requirements. We can see that except in the computational time, both algorithms perform equally well. We can conclude that the Hungarian algorithm performs better than Auction algorithm for assignment problems of any size. The reason being the centralized approach where in the agent has a total view of the problem. In contrast the Auction algorithm performs better for problems where we cannot obtain data to perform a centralized computation.

References

Bertsekas P. Dimitri, 1990, **The Auction Algorithm for Assignment and Other Network Flow Problems: A Tutorial**, Interfaces 20:4, pp 133-149.

Taha A. Hamandy, 1997, **Operations Research – An Introduction,** Prentice – Hall © 1997

Appendix

```
/*******************************************************************************
********
 Auction algorithm
created by Arunachalam N, Bharani Babu, Murugappan Meyappan, Sorajack Mongkolsri
INEN 654 Manufacturing planning and Analysis
*******************************************************************************
*******/

#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <malloc.h>
#include <memory.h>
#include <math.h>
#include <time.h>

FILE *stream;
FILE *stream1;
FILE *stream2;

float **orgsource;
float **source;
int i,j,n;
double x,m;
double eata;
int *rowassign;
int *firstmaxcol;
bool *col;
float *rowfirstmax;
float *rowsecmax;
int *unhappy;
double boolean;
int boolean1;

int rowsel,randsel,booleanrowassign,prevrowassign;
float increment;
float *assignment;

	int cost=0;



int main(int argc, char* argv[])
{

	if (argc==1)
		printf ("The syntax is auction filename");
	else
	{
		if (argc==2)
		{
			char *filename= argv[1];
			clock_t start,finish;
			//file from which data is retrieved
			stream = fopen(filename,"r");

			//file were the answers are stored are stored
			stream1 = fopen("answer.txt","w");
			stream2= fopen("assignments.txt","w");

	if (stream == NULL)
		printf("not able");
	else
	{
		fseek(stream,0L, SEEK_SET);
		int number=1;
		printf("Generating the answer files answer.txt and
assignments.txt");
		// proceed till the end of file occurs
```

```
                printf("Generating the answer files answer.txt and
assignments.txt");
                 while(!feof(stream))
          {
                n=0;
                //Get the size of matrix
                fscanf(stream,"%d",&n);
                fprintf(stream1,"matrix size %d\n",n);

                if(n==0)
                break;
                x=n;
                source = new float*[n];
                orgsource = new float*[n];
                rowassign = new int[n];
                firstmaxcol = new int[n];
                rowfirstmax = new float[n];
                rowsecmax= new float[n];

                unhappy= new int[n];
                assignment= new float[n];

                        for (i=1;i<=n;i++)
                        {

                        source[i]= new float[n];
                        orgsource[i] = new float[n];
                        rowassign[i]=0;
                        firstmaxcol[i]=0;
                        rowfirstmax[i]=-9999999;
                        rowsecmax[i]=-999999;

                        for (j=1;j<=n;j++)
                        {
                                        //Get the matrix elements
                                fscanf(stream,"%f",&source[i][j]);
                                        orgsource[i][j]=source[i][j];
                                        source[i][j]=-source[i][j];

                        }

                        }
                        //Start of algorithm
                        start=clock();
                                //Calculation of epsilon
                                eata = 1/x;
                                boolean=eata*1000;

                                boolean1=abs(boolean);
                                eata=double(boolean1)/1000;

                                srand( (unsigned )time(NULL) );


                        j=1;
                        //making an initial random assignment
                        for (i=n;i>=1;i--)
                        {
                                rowassign[j]=i;
                                assignment[j]=source[j][i];
                                j++;

                        }

                        //Finding the First maximum in each row
                        for (i=1;i<=n;i++)
                        {
                                for (j=1;j<=n;j++)
                                {
                                        if (source[i][j]>rowfirstmax[i])
                                        {
                                                rowfirstmax[i]=source[i][j];
                                                firstmaxcol[i]=j;
                                        }
                                }
                        }
                        //Finding the Second maximum in each row
                        for (i=1;i<=n;i++)
                        {
                                for (j=1;j<=n;j++)

                                {
                                    if (firstmaxcol[i] != rowassign[i])
                                    if (source[i][j]>rowsecmax[i] && firstmaxcol[i]
!= j)
                                        {
                                                rowsecmax[i]=source[i][j];

                                        };
                                }
                        }

                        j=1;
                        //Find if the person is unhappy or not
                        for (i=1;i<=n;i++)
                        {


                                if ((fabs(assignment[i]-rowfirstmax[i])) > fabs(eata
) )
                                        {

                                        unhappy[j]=i;
                                        j++;
                                        }

                        }

                        int iteration=0;
                        //If all are happy then exit else proceed
                        while (j!=1)
                        {

                        j--;

                        bool test;
                        if(j!=1)
                        {
                                while (test != true)
                                {
//random generation of the a number with the limits of number of unhappy perons

                        randsel=(float(rand())/float(RAND_MAX))*j;

                        if ( randsel!=0)
                                break;
                                }
                        }
                        else
                        {

                                randsel=1;
                        }
                                rowsel=unhappy[randsel];
                                //Calculate the bidding increment
                        increment=(rowfirstmax[rowsel]-rowsecmax[rowsel])+ (eata);

                                //Decrement the profit by that value
                                for (i=1;i<=n;i++)
                                {

        source[i][firstmaxcol[rowsel]]=source[i][firstmaxcol[rowsel]]-increment;
```

```
                              }

                              prevrowassign=rowassign[rowsel];
                              rowfirstmax[rowsel]=0;



        assignment[rowsel]=source[rowsel][firstmaxcol[rowsel]];
                              rowassign[rowsel]=firstmaxcol[rowsel];
                              // Swap the persons who were assigned to the
previous object
                              for(i=1;i<=n;i++)
                              {
                                     if (rowassign[rowsel]==rowassign[i] && i !=
rowsel)
                                     {
                                            rowassign[i]=prevrowassign;

        assignment[i]=source[i][prevrowassign];
                                     }
                              }


                              //reinitialize the max and unhappy arrays

                                     for (i=1;i<=n;i++)
                              {
                              firstmaxcol[i]=0;
                              rowfirstmax[i]=-999999;
                              rowsecmax[i]=-9999999;


                              unhappy[i]=0;

                              }

                              //Finding the First maximum in each row
                              for (i=1;i<=n;i++)
                              {
                              for (j=1;j<=n;j++)
                              {
                                     if (source[i][j]>rowfirstmax[i])
                                     {
                                            rowfirstmax[i]=source[i][j];
                                            firstmaxcol[i]=j;
                                     }
                              }
                              }
                              //Finding the Second maximum in each row
                       for (i=1;i<=n;i++)
                       {
                              for (j=1;j<=n;j++)

                              {
                                 if (firstmaxcol[i] != rowassign[i])
                                 if (source[i][j]>rowsecmax[i] && firstmaxcol[i]
!= j)
                                     {
                                            rowsecmax[i]=source[i][j];

                                     };
                              }
                       }

                       j=1;
                       //Find if the person is unhappy or not
                       for (i=1;i<=n;i++)
                       {
                              if ((fabs(assignment[i]-rowfirstmax[i])) >fabs(eata
) )

                                      {
                                      unhappy[j]=i;
                                      j++;
                              }
                      }


                      }
              fprintf(stream2,"Assignment for case number%d\n",number);
              //Printing the assignments to the file
              for (i=1;i<=n;i++)
              {
              for (j=1;j<=n;j++)
              {
                      if (rowassign[i]==j)
                      {
              cost= orgsource[i][j]+cost;
              fprintf(stream2,"Row %d\t",i);
              fprintf(stream2,"Column %d\n",j);
                      }
              }
              }
                      //End of algorithm
                              finish=clock();
                       float test11=(finish-start)/float(CLOCKS_PER_SEC);
                              fprintf(stream1,"case %d\n",number);
                              fprintf(stream1," cost %d\n",cost);
                              fprintf(stream1,"execution time %f\n",test11);

                              number++;

        }
                              fclose(stream1);
                              fclose(stream);
                              fclose(stream2);
        }
        }
        }
        return 0;
}
```

```
/*******************************************************************************
**
Hungarian algorithm
created by Arunachalam N, Bharani Babu, Murugappan Meyappan, Sorajack Mongkolsri
INEN 654 Manufacturing planning and Analysis
*******************************************************************************
**/

#include "stdafx.h"


#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <malloc.h>
#include <memory.h>
#include <time.h>

FILE *stream;
FILE *stream1;
FILE *stream2;
void function (int);
void functioncol (int);
void colscoreset();
int **source;
int i,n,j,m,a,b;
bool *rowscore;
bool *colscore;
bool *colcheck;
bool *rowcheck;
bool test;
int *colbool;
int test1;
int *rowzerocount;
int	*colzerocount;
int strikes;
int cost;
int **orgsource;
void strikingmethod();
int iteration;
int number=0;
int main(int argc, char* argv[])
{


	if(argc==1)
		printf("The syntax is Hungarian filename");
	else
	{
		if (argc==2)
		{
			char *filename = argv[1];
			printf("%c\n",&argv[1]);
			int maxzero=0;
		//open the file which has the data
		stream = fopen(filename,"r");
		//files to which the output is assigned
		stream1 = fopen("answer.txt","w");
		stream2 = fopen("answerassignments.txt","w");

	if (stream == NULL)
		printf("not able");
	else
	{

		fseek(stream,0L, SEEK_SET);
		while (!feof(stream))
		{
			n=0;
		//Get the matrix size
		fscanf(stream,"%d",&n);
```

```
                printf("%d\n",n);
                rowzerocount=new int[n];
                colzerocount=new int[n];
                rowscore= new bool[n];
                rowcheck= new bool [n];
                colscore=new bool[n];
                colcheck= new bool [n];
                colbool=new int[n];
                source = new int*[n];
                orgsource = new int*[n];

                for (i=1;i<=n;i++)
                {
                        source[i]= new int[n];
                        orgsource[i] = new int[n];
                        //Get the elemnts in the matrix
                        for (j=1;j<=n;j++)
                        {
                        fscanf(stream,"%d",&source[i][j]);
                        orgsource[i][j]=source[i][j];
                        }
                        rowzerocount[i]=0;
                        colzerocount[i]=0;


                }

                clock_t start,finish;
                //Start of algorithm
                start=clock();
                //Row reduction
                for (i=1;i<=n;i++)
                {
                        m=source[i][1];
                        for (j=1;j<=n;j++)
                        {
                        if(source[i][j]<=m)
                                m=source[i][j];
                        }
                        if (m!=0)
                                for (j=1;j<=n;j++)
                                        source[i][j]-=m;
                }
                //Column reduction
                for (i=1;i<=n;i++)
                {
                        m=source[1][i];
                        for (j=1;j<=n;j++)
                        {
                        if(source[j][i]<=m)
                                m=source[j][i];
                        }
                        if (m!=0)
                                for (j=1;j<=n;j++)
                                        source[j][i]-=m;
                }
                //call the function striking method
                strikingmethod();
                //if the optimal assignment is not reached enter the loop
                if (strikes!=n)
                {
                        strikes=0;
//CHECK THE ROWS WHICH HAVE NO ASSIGNMENTS   AND CHECK EACH COLUMN THAT HAS A ZERO
IN A CHECKED ROW
                        for (i=1;i<=n;i++)
                        {
                                for (j=1;j<=n;j++)
                                {
                                if (rowscore[i] != true &&  source [i][j] == 0)
                                {
                                        colcheck[j] = true ;

                                        rowcheck[i] = true;

                                        break;
                                }
                        }

                }


                                int arb = -1;
                while (arb<0)
                {
                                        arb= 1;
                ////CHECK THE EACH ROWS WHICH HAVE AN ASSIGNMENT IN THE CHECKED
COLUMN
                for (j=1;j<=n;j++)
                {
                        if (colcheck[j]==true)
                        for (i=1;i<=n ; i++)
                        {
                                if (source[i][j]==0 && rowscore[i] == true &&
colscore[j]==true && rowcheck[i] != true && colbool[j]== i)
                                {
                                        rowcheck[i] =true;

                                        arb = -1;
                                }

                        };

                }


                for (i=1;i<=n;i++)
                {
                        if (rowcheck[i]==true)
                                for(j=1;j<=n;j++)
                                {
                                        if (source[i][j]==0 && colcheck[j] != true )
                                        {
                                                colcheck[j] = true;

                                                arb = -1;
                                        }
                                };
                }

                }
                }
//if still optimal solution is not reached strike each unchecked row and unchecked
column
                if (strikes!=n)
                {
                        strikes=0;
                for (i=1;i<=n;i++)
                {
                        if (rowcheck[i]!=true )

                                strikes++;

                        if( colcheck[i] ==true)
                                strikes++;
                }
                }
                iteration=1;

                while (strikes != n)
                {
                        iteration++;
```

```
                        m=2000000;
                        for (i=1;i<=n;i++)
                        {
                        for (j=1;j<=n;j++)
                        {
                                // Find the minimum among the remaining elements
                        if (rowcheck[i]==true && colcheck[j]!=true)
                        {
                                if (source[i][j]<m)
                                        m=source[i][j];
                        }

                        }
                        }
                        for (i=1;i<=n;i++)
                        {
                        for (j=1;j<=n;j++)
                        {
                // Subtract the minimum value from the other unchecked elements
                                if (rowcheck[i]==true && colcheck[j]!=true)
                                {
                                        source[i][j]-=m;
                                }
                                // Add the minimum value to the elements at the
intersection
                                if (rowcheck[i]!=true && colcheck[j]==true)
                                {
                                        source[i][j]+=m;
                                }
                        }
                        }
                        //proceed to find the assignments
                        strikingmethod();

                if (strikes == n)
                break;
                else
                strikes=0;
                //if not feasible proceed

                        for (i=1;i<=n;i++)
                        {
                                for (j=1;j<=n;j++)
                                {
                                if (rowscore[i] != true &&  source [i][j] == 0)
                                {
                                        colcheck[j] = true ;

                                        rowcheck[i] = true;

                                        break;
                                }
                        }

                }

                int     arb=-1;
                        while (arb<0)
                {
                                        arb= 1;
                for (j=1;j<=n;j++)
                {
                        if (colcheck[j]==true)
                        for (i=1;i<=n ; i++)
                        {
                                if (source[i][j]==0 && rowscore[i] == true &&
colscore[j]==true && rowcheck[i] != true && colbool[j]==i)
                                {
                                        rowcheck[i] =true;
                                        arb = -1;
                                }
                        };
                }
                for (i=1;i<=n;i++)
                {
                        if (rowcheck[i]==true)
                                for(j=1;j<=n;j++)
                                {
                                        if (source[i][j]==0 && colcheck[j] != true)
                                        {
                                                colcheck[j] = true;

                                                arb = -1;
                                        }
                                };
                }
                }
                        if (strikes!=n)
                {
                        strikes=0;
                for (i=1;i<=n;i++)
                {
                        if (rowcheck[i]!=true )


                                strikes++;


                        if( colcheck[i] ==true)
                                strikes++;
                }
                }

}


                number++;
                printf("No. of strikes %d\n", strikes);
                cost=0;
                int assignment=0;
                for (i=1;i<=n;i++)
                {
                        for (j=1;j<=n;j++)
                        {
                                if ( colbool[j]==i )
                                {
                                        assignment++;
                                        //Calculate the assignment cost and print
the assignments to the file
                                        cost=orgsource[i][j] + cost;
                                                fprintf(stream2,"Row %d\t",i);
                                        fprintf(stream2,"Column %d\n",j);

                                }
                        }
                }

                printf("Cost%d\n",cost);
                finish=clock();
                float test=(finish-start)/float(CLOCKS_PER_SEC);
                fprintf(stream1,"%f\n",test);
                fprintf(stream1,"  %d\n",cost);
                fprintf(stream1,"Assignments %d\n",assignment);
                fprintf(stream1,"Iterations %d\n",iteration);
                }
                fclose(stream1);
                fclose(stream);
        }
        }
        }
        return 0;
```

```
}
//Function which looks after row striking after an assignment is made in the row
 void function (int i)
{
               for (int j =1 ;j <= n; j++)
               {
               if (source[i][j] == 0 && colscore[j] != true && colzerocount[j] !=
0 && test != true)
                      {
                      colscore[j]=true;
                      colbool[j]=i;

                      colzerocount[j] = 0;
                      rowscore[i]=true;

                      rowzerocount[i]=0;
                      for (a=1;a<=n;a++)
                      {
                             if (source[a][j] == 0 && rowzerocount[a]!=0)
                                    rowzerocount[a]--;
                      }
                      for (a=1;a<=n;a++)
                      {
                             if (source[i][a] == 0 && colzerocount[a]!=0)
                                    colzerocount[a]--;
                      }

                      test = true;
                      break;
                      }

               }
}
 //Function which looks after column striking after an assignment is made in the
column
void functioncol (int j)
{
               for (int i =1 ;i <= n; i++)
               {
               if (source[i][j] == 0 && rowscore[i] != true && rowzerocount[i] !=
0 && test !=true)
                      {
                      rowscore[i]=true;
                      colbool[j]=i;
                      colscore[j]=true;


                      colzerocount[j]=0;
                      rowzerocount[i]= 0;
                      for (a=1;a<=n;a++)
                      {
                             if (source[i][a] == 0 && colzerocount[a]!=0)
                                    colzerocount[a]--;
                      }
                      for (a=1;a<=n;a++)
                      {
                             if (source[a][j] == 0 && rowzerocount[a]!= 0)
                                    rowzerocount[a]--;
                      }


                      test=true;
                      break;
                      }
               }

}
//reinitializing the value after evry iteration
void colscoreset()
{
               rowscore= new bool[n];
               colscore= new bool[n];
               rowcheck= new bool[n];
               colcheck=new bool[n];

               colbool=new int[n];
}

//the function which looks after which row to strike or column based on the zero
count
void strikingmethod()

{
               colscoreset();
               for (i=1;i<=n;i++)
               {
                       for (j=1;j<=n;j++)
                       {
                               if (source[i][j]==0)
                               {
                                       rowzerocount[i]++;
                                       colzerocount[j]++;
                               }
                       }

               }
               strikes =0;
                       int test1=100;
               while (test1!=0)
               {
                       test1=0;
               for (i=1;i<=n;i++)
               {
                       if (rowzerocount[i] == 1)
                       {
                               test = false;
                               function(i);
                               test1=1;
                       }
               }
                       for (j=1;j<=n;j++)
               {
                       if (colzerocount[j] == 1)
                       {
                               test = false;
                               functioncol(j);
                               test1=1;
                       }
               }
               }
                       m=n;
               int identity;

               test1= 100;
               int rowtester,coltester;
               //This loop makes arbitary assignments based on our procedure to
the rows and coulmns with more than one zerocounts
               while (test1!=0)
               {
               test1=0;
               m=n;
               rowtester=0;
               coltester=0;
               for (j=1;j<=n;j++)
               {
                       if (rowzerocount[j]<=m  && rowscore[j] != true &&
rowzerocount[j]!=0)
                       {

                               m=rowzerocount[j];
                               test1=1;
                               identity=j;
```

```
                              rowtester=1;
                              coltester=0;

                       }
                       if (colzerocount[j]<=m  && colscore[j] != true &&
colzerocount[j]!=0)
                       {
                              m=colzerocount[j];
                              test1=1;
                              identity=j;
                              coltester=1;
                              rowtester=0;

                       }
               }
               if (m>0 && test1==1)
               {

                              test= false;
                              if (rowtester==1)
                              function(identity);
                              if (coltester==1)
                              functioncol(identity);

               }

               }
               //Find if the solution obtained is optimal or not
               for (j=1;j<=n;j++)
               {
                       if (rowscore[j]==true && colscore[j]==true)
                              strikes++;


               }
}
```