Implementation of Hungarian Procedure Using C++ (Case study of Dangote flour mills)

Ekpenyong, A. D.
Madu, I. M.
Usman, S.
Computer Science Department
Federal Polytechnic, Bauchi, Nigeria

ABSTRACT

The Hungarian method is a combinatorial optimization algorithm which solves the assignment problem in polynomial time. It is also known as Kuhn–Munkres algorithm or Munkres assignment. The implementation of Hungarian procedure requires a nonnegative n×n matrix, where the element in the i-th row and j-th column represents the cost of assigning the j-th job to the i-th worker. We have to find an assignment of the jobs to the workers that have minimum cost. This project takes input from user and then performs the job of scheduling based on the best available options provided. It was observed that the algorithm works even when the minimum values in two or more rows are in the same column, when two or more of the rows contain the same values in the same order. Or even when all the values are the same (although the result is not very interesting). Munkres Assignment Algorithm is not exponential run time or intractable. Optimality is guaranteed in Munkres Assignment Algorithm. It takes fraction of seconds to perform it computation in most cases.

Keywords:

INTRODUCTION

Most companies and factories today, are faced with the problem of job assignment, that is, where the company tries to assign job to its personnel in the best way such that there is cost reduction and profit maximization. This problem is called an assignment problem. Assignment problem is one of the fundamental combinatorial optimization problems in the branch of optimization or operations research in mathematics. It consists of finding a maximum weight matching in a weighted bipartite graph. The assignment problem is a special case of the minimum cost flow problem. While it is possible to solve any of these problems using the simplex algorithm, each specialization has more efficient algorithm designed to take advantage of its special structure (Ozigbo, 2000). It is clear that many management decisions are essentially resource allocation and there exists several techniques that handle this problem such as transportation model and assignment technique. These techniques help management in dealing with allocation problems; a procedure concerned with the utilization of limited resources to its best advantage. The Hungarian procedure is one of the many algorithms that have been devised to solve the linear assignment

ISSN: 2276-8815

problem. This procedure deals with effective allocation of resources to a known activity with the objective of meeting designed sets of goals. In each case we may be maximizing profits or minimizing cost. Allocation problems are concerned with the utilization of limited resources to best advantage. In this research work, Hungarian procedure will be use to solve the company (Dangote flour mills) assignment problem. Assume that we have **N** workers and **N** jobs that should be done. For each pair (worker, job) we know salary that should be paid to worker for him to perform the job. Our goal is to complete all jobs minimizing total inputs, while assigning each worker to exactly one job and vice versa (and one job to one worker). The problem here is to assign the jobs to the machines such that the total cost of production is minimized, which in turn resolves the problem of delay in completion of task and the problem of job scheduling. The machine here may not necessarily be a physical machine it might be humans in some instance. C++ was chosen because of its flexibility etc. (John, 2000). The objectives of this research are:

i To have an efficient and more effective way of assigning jobs to machines using a computerized approach.

ii To promote job specialization in a firm and timely completion of the jobs.

iii To understand when an assignment technique should be used in companies allocation problems.

iv To know how to deal with unequal machines and job problems rising in management production processes.

METHOD

The primary means of data capture for this research work is the use of a matrix defined within the main program, which holds the cost of the individual assignment assigning one task to one distinct personnel. These costs of assignment are the values that are to be entered by the user (using the program) from the input screen at the start of the program. Output Design interface specifies the result of the computed algorithms. The output form will display the ordered pair of the assignment, the total cost of assignment, the dimension used in the computation and the time that elapsed in the course of carrying out the scheduling.

Input Design is the first screen shot the user (operator) using the program will see. The form displays information, requesting for the user to specify the size of the matrix followed by the number of rows and columns, after which the user is requested to enter the cost of the individual assignment supply in row order so that the program can then compute and display the final schedule format and the cost of assigning the total job. Mathematical Models help in simplifying the complexity of the assignment problem (Burkard, 2009). Given N workers and N tasks, an $n \times m$ matrix containing the cost of assigning each worker to a task. First the problem is written in the form of a matrix as given below

A1	A2	A3	A4
B1	B2	B3	B4
C1	C2	C3	C4
D1	D2	D3	D4

Where A, B, C and D are the workers who have to perform tasks 1, 2, 3, 4. A1, A2, A3, A4 denote the penalties incurred when worker "A" does task 1,2,3,4 respectively. The same holds true for the other symbols as well. The matrix is square so that each worker can perform only one task. The N × M matrix consist of non-negative element where *ith*-row and *jth*-column represent the cost of assigning the jth job to ith worker. The algorithm is easier to describe if we formulate the problem using a bipartite graph such that $G = (S, T, E)$ with n worker vertice(s) and n jobs vertices (T), each edge has a non-negative cost $C(i, j)$.

If $y(i) + y(j)$ d" $C(i, j)$ for each i£ S, j£T

The value of potential y is

$$\sum y(v)$$

v£suT.

It can be seen that the cost of each perfect matching is at least the value of each potentials. The Hungarian method finds a perfect matching and a potential with equal cost value which proves the optimality of both the assignment. An edge *ij* is called a tight for a potential y if $y(i) + y(j) = C(i, j)$. Resulting to

$\{Cij\}$ $N \times M$ where Cij is the cost of worker i to perform job j.

$\{Xij\}$ $N \times M$ binary matrix where $Xij = 1$ if and only if *ith* worker is assigned to *jth* job

"*xij* = one worker to one job

"*xij* = one job to one worker assign

" *Cij Xij* min = total cost of assignment

Algorithm

Step 0: Create an n× m matrix such that each element represents the cost of assigning one of n worker to one of m jobs. Rotate the matrix so that there are at least as many columns as rows and let $k = \min(n, m)$.

Step 1: For each row of the matrix find the smallest element and subtract it from every element in its row go to step 2.

Step 2: Find a zero in the resulting matrix. If there is no starred zero in its row or column, star *Z*. Repeat for each element in the matrix go to step 3.

Step 3: Cover each column containing a starred zero. If k columns are covered, the starred zeros describe a complete set of unique assignments. In this case go to **DONE**, otherwise go to step 4.

Step 4: Find a non-covered zero and prime it. If there is no starred zero in the row containing this prime go to step 5. Otherwise, cover this row and uncover the column containing the starred zero. Continue in this manner until there are no uncovered zero left, save the smallest uncovered value and go to step 6.

Step 5: Start at the primed zero found in step 4. Star this zero, erase its prime, and check for a starred zero in current column. If there is one, erase its star and go to the primed zero in its row. Continue until we arrive at a primed zero with no starred zero in its column.

Uncover every row and column in the matrix and return to step 3.

Step 6: Subtract the smallest uncovered value found in step 4 from all uncovered values and add it to all values whose row AND column are covered.

Return to step 4.

Done

Assignment pairs are indicated by the position of the starred zeros in the cost matrix. If C (i j) is a starred zero, then the element associated with row i is assigned to the element associated with column j.

Description of the Program

This research work basically performs the function of scheduling jobs to its best available options based on the cost of assignment entered by the user (operator). In the course of supply of the assignment it should be put into considerations that the number of jobs must equal the number of personnel. The program works based on the cost of assignments entered to compute the total cost of assignment as well as a display of the final job scheduling on the output form for the user to see. The major steps in the operation of this system are:

i Size of work (i.e. the row and columns must be specified).

ii Cost of assignment entered in a row wise order.

iii Cost of assignment computed.

iv The ordered pair of assignment displayed.

v The time that elapsed in the course of carrying out of the computation.

For example taking these set of data as sample for the cost of production, such that, there are 7 jobs to be performed by 7 personnel and the table below is the cost of assigning a distinct job to a personnel.

Jobs / Personnel	Cleaning	Conditioning	Breaking	Scalping	Middling	Purifying	Packaging
A	5	10	11	45	21	19	7
B	32	30	25	14	15	45	6
C	12	20	18	45	14	33	5
D	22	11	18	16	16	21	4
E	12	22	5	20	20	22	3
F	18	21	3	5	1	5	2
G	14	11	6	4	3	6	1

When these values was captured by the program the following result was gotten as the ordered pair of the assignment (i.e. the personnel and the assigned job to be perform by this personnel), the total cost of assignment, dimension of work (size) and the time that elapsed as the duration in carrying out of the computations by the program.

Dimension (7, 7)

Time elapsed: 0.028ns.

The Ordered pair displayed by the program:

(0 , 0), (1 , 3), (2 , 6), (3 , 1), (4 , 2), (5 , 4), (6 , 5), meaning:

Personnel **A** have been scheduled to **clean** the wheat to remove foreign materials.
Personnel **B** have been scheduled to **scalp** the wheat i.e. in charge of shifting away the loose wheat.
Personnel **C** have been scheduled to **package** the flour into bags ready for market.
Personnel **D** have been scheduled to **condition** the wheat to facilitate separation of the bran from the endosperm.
Personnel **E** have been scheduled to **break** the wheat into smaller particles.
Personnel **F** have been scheduled to **midds** of the wheat handles the medium granular particles of endosperm.
Personnel **G** have been scheduled to **purify** the wheat into flour. Resulted to 47units as the total cost of production

CONCLUSION

In summary, Hungarian procedure is a form of transportation model which in turn is an assignment problem that deals with assigning of jobs to workers so as to minimize (or maximize) the total cost of production. Since each worker can perform only one job and each job can be assigned to only one worker the assignments constitute an independent set of a given matrix (Terry, 2002). In the course of this research work in relation to this algorithm we found out that the Hungarian procedure can be solved using different algorithms such as brute force algorithm, koning graph theory, bipartite graph models etc. We recommend that in the course of subsequent studies these algorithms should be understood to give a wider range of diversity in the course of implementation of the Hungarian procedure and to get familiarized with different solution so as to further confirm the result computed. The Munkres assignment algorithm is not limited to square matrix alone it can be implemented as a sparse matrix, but one need to ensure that the correct (optimal) assignment pairs (Munkres 1957) are in correlation, and (of course) any pairwise assignment application. Munkres can be extended to rectangular arrays (i.e. more jobs than workers, or more workers than jobs).

REFERENCES

Burkard R., Dell'Amico M. and **Martello S. (2009),** *Assignment Problem.* Philadelphia: SIAM (PA)

John R. (2000). *Theory and problem of programming with C++* (second edition). The McGraw hill companies, USA.

Munkres J. (1957). Algorithm for the assignment and transportation model. Journal of the society for industrial and applied mathematics, 5(1):32-38.

Ozigbo N. (2000). *Quantitative analysis for management decisions* (first edition). Enugu: Precision Printers and Publishers, Nigeria.

Terry L. (2002), *quantitative techniques* (sixth edition). Padstow Cornwall: T. J. International, United Kingdom.

Source code listing

```
/*
 *  main.cpp
 *  Hungaraian Assignment Algorithm
 */
#include <vector>
#include <iostream>
#include <iomanip>
#include “time.h”
#include “munkres.h”
#include <algorithm>
#include <stdio.h>
#include <stdlib.h>
//max and min sizes of the sample matrix
const int min_size = 500;
const int max_size = 500;
//the maximum and minimum weights that can be assigned to a single edge
const int min_weight = 20;
const int max_weight = 50;
//to show the sample set
const bool display_matrix = true;
//for building pseudo-random test cases
//input custom minimum and maximum values for both the size of the matrix
//and the weights contained in it
void test_generator(std::vector< std::vector<int> > &x)
        int num_rows;
        std::cout << “Enter the size of the Matrix” << std::endl;
  std::cin >> num_rows;
        int num_columns = num_rows;
        //cout << num_rows <<
        //resize x to match size
        //all values are initialized to -1
        x.resize(num_columns, std::vector <int> (num_rows, -1));
        //load weights with random(ish) values between min_weight and wax_weight inclusive
        for (int i = 0; i < num_columns; i++)
        {
                for (int j = 0; j < num_rows; j++)
                {
                        printf(“enter row %d column %d\n”,i,j);
                        std::cin>>x[i][j];
                }
        }

        if (display_matrix)
        {
                //output the starting vector
                for (int i = 0; i < num_columns; i++)
                {
                        for (int j = 0; j < num_rows; j++)
                        {
                                std::cerr << std::setw(3);
                                std::cerr << x[i][j] << “ “;
                        }
                        std::cerr << std::endl;
                }
        }

        //for performance testing
        std::cerr << “Dimensions: (“ << num_rows << “, “ << num_columns << “)” << std::endl;
         std::cout << “Go Time!” << std::endl;
```


ISSN: 2276-8815

```
}
int main (int argc, char * const argv[]) {
	//The vector of vectors of integers that we need to pass in
	std::vector< std::vector<int> > x ;
	//Build a test case
	test_generator(x);
	//start clock
	clock_t start = clock();
	//actual class instantiation and function calls
	munkres test;
	test.set_diag(false);
	test.load_weights(x);
	int cost = 0;
   int num_rows = std::min(x.size(), x[0].size());
   printf("min result=%d\n",num_rows);
   int num_columns = std::max(x.size(), x[0].size());
   printf("max result=%d\n",num_rows);
	ordered_pair *p = new ordered_pair[num_rows];
	cost = test.assign(p);
	//output size of the matrix and list of matched vertices
	std::cerr << "The ordered pairs are \n";
	for (int i = 0; i < num_rows; i++)
	{
		std::cerr << "(" << p[i].row << ", " << p[i].col << ")" << std::endl;
	}
	std::cerr << "The total cost of this assignment is " << cost << std::endl;
	std::cerr << "Dimensions: (" << num_rows << ", " << num_columns << ")" << std::endl;
	std::cerr << "Time elapsed:" << ((double)clock() - start) / CLOCKS_PER_SEC << std::endl;
	delete p;
	char c;
	std::cin >> c;
	return 0;
}
/*
 *  munkres.cpp
 *  Hungaraian Assignment Algorithm
 */
#include "munkres.h"
/*! Constructor */
munkres::munkres(void)
{
	//default to not showing steps
	diag_on = false;
}
/*! Destructor */
munkres::~munkres(void)
{
}
/*! Load wieght_array from a vector of vectors of integers. */
void munkres::load_weights(std::vector< std::vector<int> > x)
{
	//get the row and column sizes of the vector passed in
	int a = x.size(), b = x[0].size();
	//default vectors for pupoulating the matrices
	//these are needed because you need a default object when calling vector::resize()
	std::vector<int> ivector;
	cell default_cell;
	std::vector<cell> default_vector;
	//We need at least as many columns as there are rows
	//If we currently have more rows than columns
	if (a > b)
	{
```

```
                //set matrix sizes
                num_rows = b;
                num_columns = a;
                ivector.resize(num_columns, -1);
                weight_array.resize(num_rows, ivector);
                default_vector.resize(num_columns, default_cell);
                cell_array.resize(num_rows, default_vector);
                //populate weight_array and cell_array with the weights from x
                for (int i = 0; i < num_rows; i++)
                {
                        for (int j = 0; j < num_columns; j++)
                        {
                                weight_array[i][j] = x[j][i];
                                cell_array[i][j].weight = x[j][i];
                        }
                }
        }

        //if the dimensions are correct
        else
        {
                //set matrix sizes
                num_rows = a;
                num_columns = b;
                ivector.resize(num_columns, -1);
                weight_array.resize(num_rows, ivector);
                default_vector.resize(num_columns, default_cell);
                cell_array.resize(num_rows, default_vector);
                //populate weight_array and cell_array with the weights from x
                for (int i = 0; i < num_rows; i++)
                {
                        for (int j = 0; j < num_columns; j++)
                        {
                                weight_array[i][j] = x[i][j];
                                cell_array[i][j].weight = x[i][j];
                        }
                }
        }

        //resize our covered and starred vectors
        row_starred.resize(num_rows, false);
        row_cov.resize(num_rows, false);
        column_starred.resize(num_columns, false);
        column_cov.resize(num_columns, false);
        if (diag_on)
        {
                diagnostic(1);
        }
}
//function to copy weight values from cell_array to weight_array
int munkres::assign(ordered_pair *matching)
{
        //total cost of the matching
        int total_cost = 0;
        //did we find a matching?
        bool matching_found = false;
        //For Checking
        if (diag_on)
        {
                diagnostic(1);
        }
        //try to find a matching
```

```
		matching_found = find_a_matching();
		//For Checking
		if (diag_on)
		{
			diagnostic(1);
		}
		//total up the weights from matched vertices
		for (int i = 0; i < num_rows; i++)
		{
			for (int j = 0; j < num_columns; j++)
			{
				if (cell_array[i][j].starred)
				{
					matching[i].col	= j;
					matching[i].row = i;
					total_cost += weight_array[i][j];
				}
			}
		}
		return total_cost;
}
//functions to check if there is a star or prime in the
//current row or column
int munkres::find_star_row(int r)
{
		//check row
		for (int i = 0; i < num_columns; i++)
		{
			//If a starred value is found in current row return true
			if (cell_array[r][i].starred == true)
			{
				row_starred[r] = true;
				column_starred[i] = true;
				return i;
			}
		}
		//If no stars are found return -1
		return -1;
}
int munkres::find_star_column(int c)
{
		//check column
		for (int i = 0; i < num_rows; i++)
		{
			//If a starred value is found in current column return true
			if (cell_array[i][c].starred == true)
			{
				column_starred[c] = true;
				row_starred[i] = true;
				return i;
			}
		}
		//If no stars are found return -1
		return -1;
}
int munkres::find_prime_row(int r)
{
		//check row
		for (int i = 0; i < num_columns; i++)
		{
			//If a primed value is found in current row return
			//its psoition
```

```
                if (cell_array[r][i].primed == true)
                {
                        return i;
                }
        }
        //If no primes are found return -1
        return -1;
}
int munkres::find_prime_column(int c)
{
        //check column
        for (int i = 0; i < num_rows; i++)
        {
                //If a primed value is found in current column return
                //its position
                if (cell_array[i][c].primed == true)
                {
                        return i;
                }
        }
        //If no primes are found return -1
        return -1;
}
//The function that will call each of step of Munkres' algorithm in turn
//We're using this since multiple functions use the algorithm
bool munkres::find_a_matching(void)
{
        step1();
        step2();
        return step3();
}
//Function definitions for the steps of Munkres' algorithm
/*!
```

Step 1.

```
 We skip step 0 as the matrix is already formed
 Here we subtract the smallest element in each row from the other values in that row
*/
void munkres::step1(void)
{
        //variable to keep track of the smallest value in each row
        int smallest = 0;
        //iterate through rows
        for (int i = 0; i < num_rows; i++)
        {
                //set smallest = first element in the current row
                while (smallest == 0){
                        smallest = cell_array[i][0].weight;
                        //if the first element is 0 then increase the row
                        if (smallest == 0)
                        {
                                if (i < num_rows-1)
                                        i++;
                                else
                                        break;
                        }
                }
                //iterate through each value in current row and find the smallest value
                for (int j = 1; j < num_columns; j++)
                {
                        //if the current value is a zero, then set smallest to zero
                        //and stop searching for a smaller value
```


ISSN: 2276-8815

```
                        if (cell_array[i][j].weight == 0)
                        {
                                smallest = 0;
                                //break out of the for loop
                                j = num_columns;
                        }
                        //if the current value is smaller than smallest, then
                        //set smallest == to current value
                        else if (cell_array[i][j].weight < smallest)
                        {
                                smallest = cell_array[i][j].weight;
                        }
                }

                //if the smallest == 0 then we don't need to subtract anything
                //otherwise we need to subtract smallest from everything in the current row
                if (smallest != 0)
                {
                        //iterate through the values of current row and subtract
                        //smallest from evrything
                        for (int j = 0; j < num_columns; j++)
                        {
                                cell_array[i][j].weight -= smallest;
                        }
                }

                //reset smallest for next iteration
                smallest = 0;
        }

        if (diag_on)
        {
                std::cerr << "Step 1" << std::endl;
                diagnostic(2);
        }
}
/*!
```

Step 2.

```
 Star zeroes that don't have stars (upon thars :P) in the
 same row or column
 */
void munkres::step2(void)
{
        //iterate through rows
        for (int i = 0; i < num_rows; i++)
        {
                //iterate through columns
                for (int j = 0; j < num_columns; j++)
                {
                        //if the current index is equal to 0
                        if (cell_array[i][j].weight == 0)
                        {
                                //check for stars in current row
                                if (!row_starred[i])
                                {
                                        //if not try to find one
                                        find_star_row(i);
                                }
                                //check for stars in current column
                                if (!column_starred[j])
                                {
```

```
                                        //if not try to find one
                                        find_star_column(j);
                                }
                                //if no stars in column or row then star current index
                                if (!row_starred[i] && !column_starred[j])
                                {
                                        //star index
                                        cell_array[i][j].starred = true;
                                        //mark row as having a star
                                        row_starred[i] = true;
                                        //mark column as having a star
                                        column_starred[j] =  true;
                                }
                        }
                }
        }
        if (diag_on)
        {
                std::cerr << "Step 2" << std::endl;
                diagnostic(3);
        }
}
/*!
```

Step 3.

```
 cover all columns with starred zeros
 if (num_rows) columns are covered then return true
 to signify that we're done
*/
bool munkres::step3(void)
{

        //an iterator for our while loop
        int iter = 0;
        //loop through columns
        for (int i = 0; i < num_columns; i++)
        {
                //if the column is starred
                if (column_starred[i])
                {
                        //cover it
                        column_cov[i] = true;
                }
        }
        //while every column so far is covered
        for (int i = 0; i < num_columns; i++)
        {
                if (column_cov[i])
                {
                        iter++;
                }
        }

        if (diag_on)
        {
                std::cerr << "Step 3" << std::endl;
                diagnostic(6);
        }
        //if all the rows were covered
        if (iter == num_rows)
        {
                //exit algorithm
```

```
                return true;
        }
        //else goto step 4
        else
                return step4();
}
/*!
 Step 4.
Find a noncovered zero and prime it
if there isn't a starred zero in its row then goto step 5
if there is then cover the current row and uncover the column with the starred zero
then look for more uncovered zeros
if there are no uncovered zeros we go to step 6
 */
bool munkres::step4(void)
{
        //To find the smallest uncovered value
        int smallest = 0;
        //iterate through rows
        for (int i = 0; i < num_rows; i++)
        {
                //if the current row isn't covered
                if (!row_cov[i])
                {

                        //set smallest = first element in the current row
                        while (smallest == 0){
                                smallest = cell_array[i][0].weight;
                                //if the first element is 0 then increase the row
                                if (smallest == 0)
                                {
                                        if (i < num_rows-1)
                                                i++;
                                        else
                                                break;
                                }
                        }
//iterate through columns
                        for (int j = 0; j < num_columns; j++)
                        {
                                //if the column and row aren't covered, the current index is zero,
                                //and there isn't a star in the current row,
                                //then prime the current index and go to step 5
                                if (!column_cov[j] && !row_cov[i] && cell_array[i][j].weight == 0
&& !row_starred[i])
                                {
                                        //prime current index
                                        cell_array[i][j].primed = true;
                                        //if a primed zero with no star in the row exists
                                        //goto step 5
                                        if (diag_on)
                                        {
                                        std::cerr << "Step 4: " << i << ",  " << j <<std::endl;
                                        diagnostic(6);
                                        }
                                return step5(i, j);

                                }

                                //if the column and row aren't covered, the current index is zero,
                                //and there is a star in the current row,
                                //then prime the current index, cover the current row,
```

```
                                        //and uncover the column with the starred zero
                                        //also reset indeces to 0 to look for zeros that may have been
uncovered
                                        else if (!column_cov[j] && !row_cov[i] && cell_array[i][j].weight
==0)
                                        {
                                                //prime current index
                                                cell_array[i][j].primed = true;
                                                //cover current row
                                                row_cov[i] = true;
                                                //uncover column with starred zero
                                                column_cov[find_star_row(i)] = false;
                                                i = 0;
                                                j = 0;
                                        }

                                        //if the column isn't covered, the current index isn't zero,
                                        //and the current index is smaller than smallest so far,
                                        //then set smallest == current index
                                        else if (!column_cov[j] && cell_array[i][j].weight != 0 &&
cell_array[i][j].weight < smallest)
                                        {
                                                //set smallest == current index
                                                smallest = cell_array[i][j].weight;
                                        }
                                }
                        }
                }

                if (diag_on)
                {
                        std::cerr << "Step 4" << std::endl;
                        diagnostic(6);
                }

                //if we don't go to step 5 then go to step 6
                return step6(smallest);
}

/*!

 Step 5.
Start at the primed zero found in step 4.
Star this zero, erase its prime, and check for a starred zero in current column.
 If there is one, erase its star and go to the primed zero in its row.
Continue until we arrive at a primed zero with no starred zero in its column.
Uncover every row and column in the matrix and return to step 3.

 This step checks to see if there is a matching in the current
 matrix without altering any values
 */
bool munkres::step5(int r, int c)
{
        //to determine if we're done creating the sequence
        //of starred and primed zeros
        bool done = false;

        //are we looking for a star or prime
        bool looking_for_star = true;

        //for stupid special case in which we would look for a star in
        //the current column after starring the current index
```


ISSN: 2276-8815

```
//this returns the current index without looking further down
//the column, resulting in an error
int a;

//create our sequence
while (!done)
{
	switch (looking_for_star)
	{

		//if we're looking for a star
		case true:

			//special case protection
			a = r;

			//if there isn't a starred zero in the current column
			if (!column_starred[c])
			{
				//unprime current index
				cell_array[r][c].primed = false;
				//star current index
				cell_array[r][c].starred = true;
				//mark current row starred
				row_starred[r] = true;
				//set done to true
				done = true;
			}
			else
			{
				//set the next row to search to the location of the
				//starred zero in current column
				r = find_star_column(c);
				//unprime current index
				cell_array[a][c].primed = false;
				//star current index
				cell_array[a][c].starred = true;
				//mark current row starred
				row_starred[a] = true;
				//set case to look for prime next
				looking_for_star = false;
			}

			//we can't do this earlier due to needing to check for stars in the
column
			//mark the column as starred
			column_starred[c] = true;
			break;

		//if we're looking for a prime
		case false:

			//prime current index
			cell_array[r][c].primed = false;
			//unstar current index
			cell_array[r][c].starred = false;

			//unmark current row as starred
			row_starred[r] = false;

			//set the next column to search to the location of the
			//primed zero in current row
```

```
                                        c = find_prime_row(r);

                                        //set case to look for star next
                                        looking_for_star = true;
                                        break;
                        }
                }

                //erase all primes and uncover all rows
                for (int i = 0; i < num_rows; i++)
                {
                        for (int j = 0; j < num_columns; j++)
                        {
                                cell_array[i][j].primed = false;
                        }
                        row_cov[i] = false;
                }

                //uncover all columns
                for (int i = 0; i < num_columns; i++)
                {
                        column_cov[i] = false;
                }

                if (diag_on)
                {
                        std::cerr << "Step 5" << std::endl;
                        diagnostic(6);
                }
                //go back to step 3
                return step3();
}

/*!
```

Step 6.
Subtract the smallest uncovered value found in step 4 from all uncovered values and add it to all values whose row AND column are covered.

Return to step 4.
This gives us a new set of zeros to work with since a matching wasn't available with the previous set

```
*/
bool munkres::step6(int sub)
{
        //iterate through rows
        for (int i = 0; i < num_rows; i++)
        {
                //iterate through columns
                for (int j = 0; j < num_columns; j++)
                {
                        //if the current index's row and column are uncovered
                        if (!row_cov[i] && !column_cov[j])
                        {
                                //substract sub from its weight
                                cell_array[i][j].weight -= sub;
                        }

                        //else if the current index's row and column are covered
                        else if (row_cov[i] && column_cov[j])
                        {
```


ISSN: 2276-8815

```
                                        //add sub to its weight
                                        cell_array[i][j].weight += sub;
                                }
                        }
                }

                if (diag_on)
                {
                        std::cerr << "Step 6" << std::endl;
                        diagnostic(6);
                }
                //go back to step 4
                return step4();
}
//Diagnostics only
//Does not affect any results
void munkres::diagnostic(int a) const
{
                switch (a)
                {
                        //Show base weights in weight_array
                        case 1:
                                std::cerr << std::endl << "Base Weights" << std::endl;
                                for (int i = 0; i < num_rows; i++)
                                {
                                        for (int j = 0; j < num_columns; j++)
                                        {
                                                std::cerr << weight_array[i][j] << " | ";
                                        }
                                        std::cerr  << std::endl;
                                }
                                std::cerr << std::endl;
                                break;

                                //show current weight values of cell_array
                                case 2:
                                std::cerr << std::endl << "Current Weights" << std::endl;
                                for (int i = 0; i < num_rows; i++)
                                {
                                        for (int j = 0; j < num_columns; j++)
                                        {
                                                std::cerr << cell_array[i][j].weight << " | ";
                                        }
                                        std::cerr << std::endl;
                                }
                                std::cerr << std::endl;
                                break;

                                //Show current star placement
                                case 3:
                                std::cerr << std::endl << "Starred values" << std::endl;
                                for (int i = 0; i < num_rows; i++)
                                {
                                        for (int j = 0; j < num_columns; j++)
                                        {
                                                if (cell_array[i][j].starred == true)
                                                {
                                                        std::cerr << cell_array[i][j].weight << "* |  ";
                                                }
                                                else
                                                {
                                                        std::cerr << cell_array[i][j].weight << "  |  ";
```

```
                    }
          }
          std::cerr  << std::endl;
}
std::cerr << std::endl;
break;

//Show current star placement, covered rows, and covered columns
case 4:
std::cerr << std::endl << "Starred values and Lines" << std::endl;
for (int i = 0; i < num_rows; i++)
{
          for (int j = 0; j < num_columns; j++)
          {
                    if (cell_array[i][j].starred == true)
                    {
                              std::cerr << cell_array[i][j].weight << "* |  ";
                    }
                    else
                    {
                              std::cerr << cell_array[i][j].weight << "  |  ";
                    }
          }

          if (row_cov[i])
          {
                    std::cerr << "  X";
          }
          std::cerr << std::endl;
}

for (int i = 0; i < num_columns; i++)
{
          if (column_cov[i]){

                    std::cerr << "X  |  ";
          }
          else
          {
                    std::cerr << "   |  ";
          }
}
std::cerr << std::endl;
break;

//Show current prime placement, covered lines and covered columns
case 5:
std::cerr << std::endl << "Primed values and Lines" << std::endl;
for (int i = 0; i < num_rows; i++)
{
          for (int j = 0; j < num_columns; j++)
          {
                    if (cell_array[i][j].primed == true)
                    {
                              std::cerr << cell_array[i][j].weight << "' |  ";
                    }
                    else
                    {
                              std::cerr << cell_array[i][j].weight << "  |  ";
                    }
          }
```

```
                if (row_cov[i])
                {
                        std::cerr << "  X";
                }
                std::cerr << std::endl;
        }

        for (int i = 0; i < num_columns; i++)
        {
                if (column_cov[i]){

                        std::cerr << "X  |  ";
                }
                else
                {
                        std::cerr << "   |  ";
                }
        }
        std::cerr << std::endl;
        break;

        //Show current star and prime placement, covered rows, and covered
columns
        case 6:
        std::cerr << std::endl << "Starred values and Lines" << std::endl;
        for (int i = 0; i < num_rows; i++)
        {
                for (int j = 0; j < num_columns; j++)
                {
                        if (cell_array[i][j].starred == true)
                        {
                                std::cerr << cell_array[i][j].weight << "* |  ";
                        }
                        else if (cell_array[i][j].primed == true)
                        {
                                std::cerr << cell_array[i][j].weight << "' |  ";
                        }
                        else
                        {
                                std::cerr << cell_array[i][j].weight << "  |  ";
                        }
                }

                if (row_cov[i])
                {
                        std::cerr << "  X";
                }

                else
                {
                        std::cerr << "   ";
                }
                if (row_starred[i])
                {
                        std::cerr << "  *";
                }
                std::cerr << std::endl;
        }

        for (int i = 0; i < num_columns; i++)
        {
                if (column_cov[i]){
```

```
                                        std::cerr << "X  |  ";
                                }
                                else
                                {
                                        std::cerr << "   |  ";
                                }
                        }
                        std::cerr << std::endl;

                        for (int i = 0; i < num_columns; i++)
                        {
                                if (column_starred[i]){

                                        std::cerr << "*  |  ";;
                                }
                                else
                                {
                                        std::cerr << "   |  ";;
                                }
                        }
                        std::cerr << std::endl;
                        break;


                        default:
                        break;
        }
}

/*
 *  munkres.h
 *  Hungaraian Assignment Algorithm
 */
#ifndef MUNKRES_H
#define    MUNKRES_H
#include <iostream>
include <vector>
//! This is a struct for the cells of the matrix.
struct cell
{
        //The weight value in the cell
        int weight;

        //Values for whether the cell is primed or starred
        bool starred, primed;
        //initialize starred and primed values to false
        cell()
        {
                starred = false;
                primed = false;
                weight = -1;
        }
};
//! It's an ordered pair, not much else to say
struct ordered_pair
{
        int row, col;
};

//! This munkres class.
/*!
```


ISSN: 2276-8815

```
 This class handles the implementation of the Kuhn-Munkres
 Assignment algorithm.
 The description of the algorithm can befound at
 http://csclab.murraystate.edu/bob.pilgrim/445/munkres.html
*/
class munkres
{
public:
        munkres(void);
        ~munkres(void);

        /*! This function is used to turn diagnostics on or off */
        void set_diag(bool a)
        {
                diag_on = a;
        }

 /*!
   Load weight array from a vector of vectors of integers
   @param x An object of type vector< vector<int> >, which is
   a matrix of any dimensions with integer values > -1
 */
        void load_weights(std::vector< std::vector<int> > x);

 /*!
   This function will assign a matching and
          return the total weight of the matching.
   @param *matching takes a pointer to integer as a parameter
   and this will be the matching in the form of an array
 */
        int assign(ordered_pair *matching);

private:

        int num_rows; /*!< An integer to show the total number of rows */
        int num_columns; /*!< An integer to show the total number of columns */

        std::vector<bool> row_starred, column_starred;
  /*!< Arrays to track which columns and rows are starred */

        std::vector<bool> row_cov, column_cov;
  /*!< Arrays to track which columns and rows are covered */

        bool diag_on; /*!< A boolean value to turn daignostics on or off */

        //Initialize all variables for operations
        void init(void);

        //The matrix operated on by Munkres' algorithm
        //(could be better than an array in the future)
        std::vector< std::vector<cell> > cell_array;

        //array to store the weights for calculating total weight
        std::vector< std::vector<int> > weight_array;

        //functions to check if there is a starred zero in the current row or column
        int find_star_column(int c);
        int find_star_row(int r);

        //functions to check if there is a primed zero in the current row or column
        int find_prime_column(int c);
        int find_prime_row(int r);
```


ISSN: 2276-8815

```
        //These are the declarations for Munkres' algorithm steps
        void step1(void);
        void step2(void);
        bool step3(void);
        bool step4(void);
        bool step5(int, int);
        bool step6(int sub);

        //The function that will call each of step of Munkres' algorithm in turn
        //We're using this since multiple functions use the algorithm
        bool find_a_matching(void);

        //A function simply for diagnostic purposes
        //Useful for testing new code and to help both myself and anyone who
        //wants to modify this in the future
        void diagnostic(int a)          const;

#endif MUNKRES_H
```


ISSN: 2276-8815