UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Faculty of Engineering and Applied Science

SOFE 377U Design and Analysis of Algorithms

Assignment 2

Group Member 1

Name: Dhanushga Lionel

Student ID: 100616831

Group Member 2

Name: Thanushan Rameswaran

Student ID: 100462146

Date: 11/16/2018

Opening Comments:

For this assignment we used our UOIT issued laptops, running Windows 10. Our code is written in C++, compiled with Visual Studio.

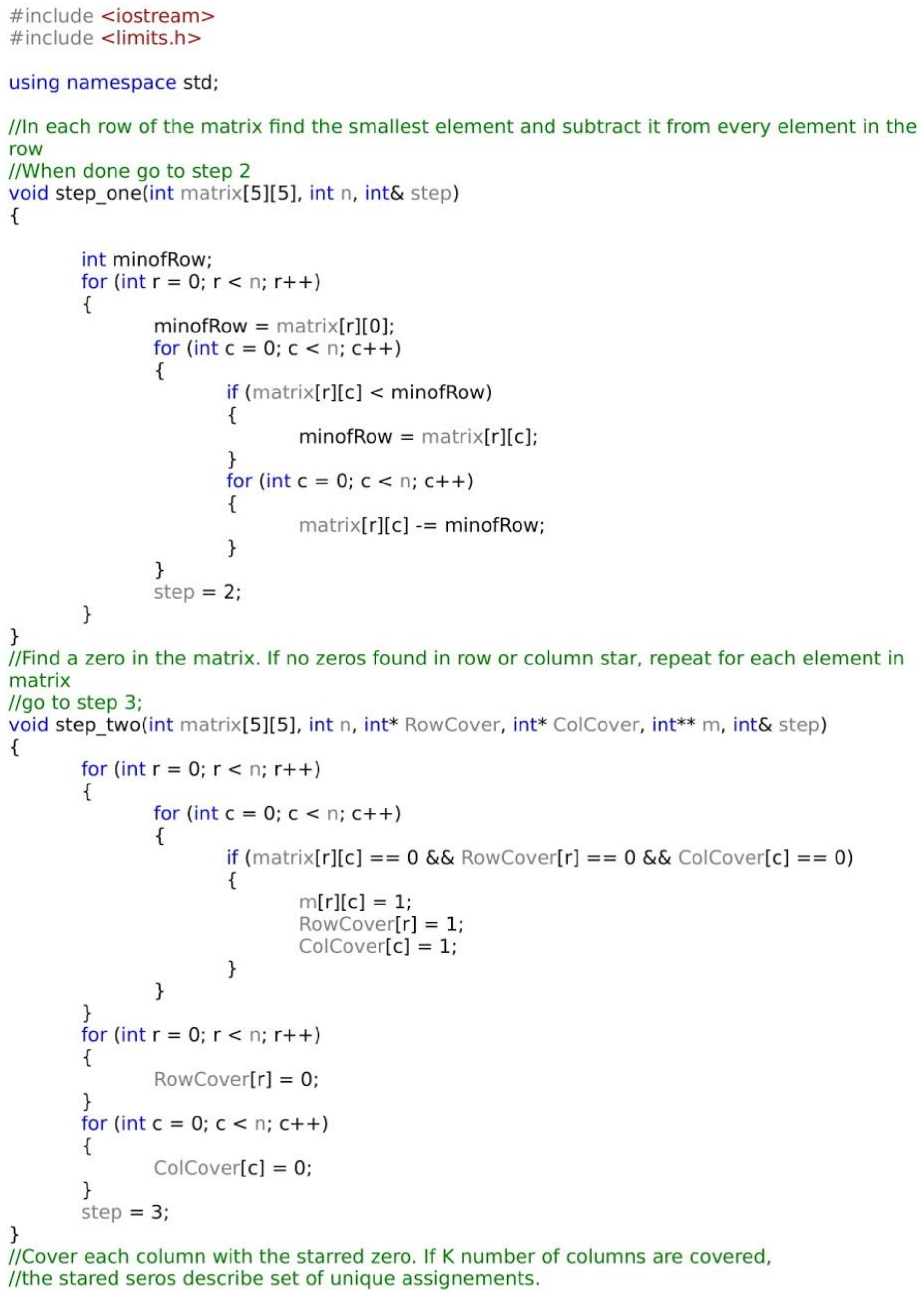
Pseudocode:

```
Algorithm hungarian
        Input: cost matrix of jobs and cost
        Output: optimal assignment of jobs to workers
For each row
        Subtract minimum row element from each row
For each column
        Subtract minimum column element from each column
Determine number of columns and rows that include all 0s in matrix (cover all zeros with
minimum number of lines)
        While lines != sizeof.matrix{
                For each element not covered{
                        Subtract minimum element from all elements}
                For each element covered by two lines{
                        Add minimum element}
                }
For each zero determine corresponding position in original cost matrix
For each column{
Job n = Worker (minimum element in column n)
}
```

Big-O Analysis:

The original Hungarian/Munkres Algorithm worst-case running time was O(n^4). The running time in recent years has decreased to O(n^3).

Code:



```cpp
#include <iostream>
#include <limits.h>

using namespace std;

//In each row of the matrix find the smallest element and subtract it from every element in the
row
//When done go to step 2
void step_one(int matrix[5][5], int n, int& step)
{

        int minofRow;
        for (int r = 0; r < n; r++)
        {
                minofRow = matrix[r][0];
                for (int c = 0; c < n; c++)
                {
                        if (matrix[r][c] < minofRow)
                        {
                                minofRow = matrix[r][c];
                        }
                        for (int c = 0; c < n; c++)
                        {
                                matrix[r][c] -= minofRow;
                        }
                }
                step = 2;
        }
}
//Find a zero in the matrix. If no zeros found in row or column star, repeat for each element in
matrix
//go to step 3;
void step_two(int matrix[5][5], int n, int* RowCover, int* ColCover, int** m, int& step)
{
        for (int r = 0; r < n; r++)
        {
                for (int c = 0; c < n; c++)
                {
                        if (matrix[r][c] == 0 && RowCover[r] == 0 && ColCover[c] == 0)
                        {
                                m[r][c] = 1;
                                RowCover[r] = 1;
                                ColCover[c] = 1;
                        }
                }
        }
        for (int r = 0; r < n; r++)
        {
                RowCover[r] = 0;
        }
        for (int c = 0; c < n; c++)
        {
                ColCover[c] = 0;
        }
        step = 3;
}
//Cover each column with the starred zero. If K number of columns are covered,
//the stared seros describe set of unique assignements.
```

```
//If this is the case, go to step 7, else go to step 4
void step_three(int matrix[5][5], int n, int* ColCover, int** m, int& step)
{
	int ColCount;
	for (int r = 0; r < n; r++)
	{
		for (int c = 0; c < n; c++)
		{
			if (matrix[r][c] == 1)
			{
				ColCover[c] = 1;
			}
		}
	}
	ColCount = 0;
	for (int c = 0; c < n; c++)
	{
		if (ColCover[c] == 1)
		{
			ColCount += 1;
		}
	}
	if (ColCount >= n || ColCount >= n)
	{
		step = 7;
	}
	else
	{
		step = 4;
	}
}
//method to support step 4
void Finding_a_Zero(int matrix[5][5], int n, int* RowCover, int* ColCover, int row, int col)
{
	int r = 0;
	int c;
	bool done;
	row = -1;
	col = -1;
	done = false;
	while (!done)
	{
		c = 0;
		while (true)
		{
			if (matrix[r][c] == 0 && RowCover[r] == 0 && ColCover[c] == 0)
			{
				row = r;
				col = c;
				done = true;
			}
			c += 1;
			if (c > n || done)
			{
				break;
			}
		}
		r += 1;
		if (r >= n)
		{
			done = true;
		}
	}
}
bool Star_in_Row(int** m, int n, int row)
{
	bool tmp = false;
	for (int c = 0; c < n; c++)
	{
		if (m[row][c] == 1)
		{
			tmp = true;
		}
	}
	return tmp;
}
void Find_Star_in_Row(int** m, int n, int row, int col)
{
	col = 1;
	for (int c = 0; c < n; c++)
	{
		if (m[row][c] == 1)
		{
			col = c;
		}
	}
}
//FInd non-covered zero and prime it. If there is no starred zero in row containing primed zero
//Go to step 5.
//Else conitnue till no uncovered zeros remain.
//Save smallest value and go to step 6.
void step_four(int matrix[5][5], int n, int* RowCover, int* ColCover, int** m, int& path_row_0, int&
path_col_0, int& step)
{
	int row = -1;
	int col = -1;
	bool done;
	done = false;
	while (!done)
	{
		Finding_a_Zero(matrix, n, RowCover, ColCover, row, col);
		if (row == -1)
		{
			done = true;
			step = 6;
		}
		else
		{
			m[row][col] = 2;
			if (Star_in_Row(m, n, row))
			{
				Find_Star_in_Row(m, n, row, col);
				RowCover[row] = 1;
				ColCover[col] = 0;
			}
			else {
```

```
                                done = true;
                                step = 5;
                                path_row_0 = row;
                                path_col_0 = col;
                        }
                }
        }
}

//methods that will support step 5;
void Find_Star_in_Col(int n, int** m, int c, int& r)
{
        r = -1;
        for (int i = 0; i < n; i++)
        {
                if (m[i][c] == 1)
                {
                        r = i;
                }
        }
}
void Find_Prime_in_Row(int n, int** m, int r, int& c)
{
        for (int j = 0; j < n; j++)
        {
                if (m[r][j] == 2)
                {
                        c = j;
                }
        }
}
void Augment_Path(int** m, int path_count, int** path) {
        for (int p = 0; p < path_count; p++)
        {
                if (m[path[p][0]][path[p][1]] == 1)
                {
                        m[path[p][0]][path[p][1]] = 0;
                }
                else
                {
                        m[path[p][0]][path[p][1]] = 1;
                }
        }
}
void Covers_Clear(int n, int* RowCover, int* ColCover) {
        for (int r = 0; r < n; r++)
        {
                RowCover[r] = 0;
        }
        for (int c = 0; c < n; c++)
        {
                ColCover[c] = 0;
        }
}
void Primes_Erase(int n, int** m)
{
        for (int r = 0; r < n; r++)
        {
                for (int c = 0; c < n; c++)
                {
                        if (m[r][c] == 2)
                        {
                                m[r][c] = 0;
                        }
                }
        }
}
//Constructing series of alternating primed and starred zeros
//Z0 represents uncovered primed zero found in step 4
//Z1 denoes starred sero in column of Z0 (if any).
//Z2 denotes primed zero in row of Z1. Continue unitl the series terminates at a primed zero that
has no starred
//zero in its column. Unstar each of starred zeros and star each primed zeros of series
//erase all primes and uncover every line in matrix,
//when finished return to step 3;
void step_five(int n, int* RowCover, int* ColCover, int** m, int& path_row_0,
        int& path_col_0, int& path_count, int** path, int& step)
{
        bool done;
        int r = -1;
        int c = -1;
        path_count = 1;
        path[path_count - 1][0] = path_row_0;
        path[path_count - 1][1] = path_col_0;
        done = false;
        while (!done) {
                Find_Star_in_Col(n, m, path[path_count - 1][1], r);
                if (r > -1) {
                        path_count += 1;
                        path[path_count - 1][0] = r;
                        path[path_count - 1][1] = path[path_count - 2][1];
                }
                else
                        done = true;
                if (!done) {
                        Find_Prime_in_Row(n, m, path[path_count - 1][0], c);
                        path_count += 1;
                        path[path_count - 1][0] = path[path_count - 2][0];
                        path[path_count - 1][1] = c;
                }
        }
        Augment_Path(m, path_count, path);
        Covers_Clear(n, RowCover, ColCover);
        Primes_Erase(n, m);
        step = 3;
}

//method to support step 6
void Find_Smallest(int matrix[5][5], int n, int* RowCover, int* ColCover,
        int& minVal)
{
        for (int r = 0; r < n; r++)
        {
                for (int c = 0; c < n; c++)
                {
                        if (RowCover[r] == 0 && ColCover[c] == 0)
                        {
                                if (minVal > matrix[r][c])
                                {
                                        minVal = matrix[r][c];
                                }
                        }
                }
        }
}
//Add value found in step 4 to every column in covered row and subtract from every element
//in uncovered column.
//Return to step 4 without altering stars, primes or covered lines
void step_six(int matrix[5][5], int n, int* RowCover, int* ColCover, int& step)
{
        int minVal = INT_MAX;
        Find_Smallest(matrix, n, RowCover, ColCover, minVal);
        for (int r = 0; r < n; r++)
                for (int c = 0; c < n; c++) {
                        if (RowCover[r] == 1)
                                matrix[r][c] += minVal;
                        if (ColCover[c] == 0)
                                matrix[r][c] -= minVal;
                }
        step = 4;

}

void runHungarian(int matrix[5][5], int n)
{
        int* RowCover = new int[n];
        int* ColCover = new int[n];
        int** m = new int*[n];
        int path_row_0 = 0;
        int path_col_0 = 0;
        int path_count = 0;
        for (int i = 0; i < n; i++) {
                RowCover[i] = ColCover[i] = 0;
                m[i] = new int[n];
                for (int j = 0; j < n; j++) {
                        m[i][j] = 0;
                }
        }
        int** path = new int*[n * 2];
        for (int i = 0; i < n * 2; i++) {
                path[i] = new int[2];
        }
        bool done = false;
        int step = 1;
        while (!done)
        {
                switch (step)
                {
                case 1:
                        step_one(matrix, n, step);
                        break;
                case 2:
                        step_two(matrix, n, RowCover, ColCover, m, step);
                        break;
```

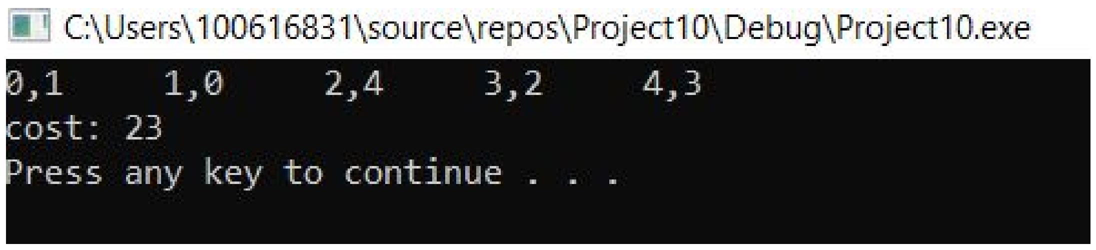
```
			case 3:
				step_three(matrix, n, ColCover, m, step);
				break;
			case 4:
				step_four(matrix, n, RowCover, ColCover, m, path_row_0, path_col_0, step);
				break;
			case 5:
				step_five(n, RowCover, ColCover, m, path_row_0, path_col_0,
					path_count, path, step);
				break;
			case 6:
				step_six(matrix, n, RowCover, ColCover, step);
				break;
			case 7:
				done = true;
				break;
			}

		}

}
int main()
{
	int matrix[5][5] = {
				{ 10, 5, 13, 15, 16 },
				{  3, 9, 18, 13,  6 },
				{ 10, 7,  2,  2,  2 },
				{  7, 11, 9,  7,  12},
				{  7,  9, 10, 4,  12 }
	};

	runHungarian(matrix, 5);
	for (int x = 0; x < 5; x++)
	{
		cout << x << "," << assignment[x] << "\t";
		cout << "\ncost: " << cost << endl;
	}
	system("pause");
	return 0;
}
```

Output:



C:\Users\100616831\source\repos\Project10\Debug\Project10.exe

```
0,1     1,0     2,4     3,2     4,3
cost: 23
Press any key to continue . . .
```