Design and Analysis of Algorithms

Assignment - IV

Members:

Lekhana Mitta - IIT2019204

Sanskar Patro - IIT2019205

Aamin Chaudhari - IIT2019206

Contents:

Content	**Slide Number**
Problem Statement	**3**
Algorithm	**4,5,6**
Pseudo Code	**7,8,9**
Algorithm Analysis	**10,11**
Conclusion	**12**

Optimal Assignment Solution

Let there be n agents and n tasks. Any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment. It is required to perform all tasks by assigning exactly one agent to each task and exactly one task to each agent in such a way that the total cost of the assignment is minimized

Algorithm:

Step - 1:The user inputs the number of test cases and it is stored in a variable t.

Step - 2: Next,the user has to input the number of jobs and persons.This is stored in a variable n.

Step - 3:For every test case we use random number generation function to fill an n*n cost matrix whose cells contains the efficiency of each person at each job.

Step - 4:Find the element with minimum cost in each row of the cost matrix. Then subtract all values in that row with this minimum value.

Step - 5:Find the element with minimum cost in each column of the cost matrix. Then subtract all values in that column with this minimum value.

Algorithm:

Step - 6:Examine rows successively until a row with exactly one unmarked zero is obtained. Make an assignment single zero by making a square around it.

Step - 7:For each zero value that becomes assigned, eliminate (Strike off) all other zeros in the same row and/ or column.

Step -8:Repeat steps 6 and 7 for each successive column also with exactly single zero value all that has not been assigned.

Step - 9:If some row/column has two or more unmarked zeros and one cannot be chosen by inspection, then choose the assigned zero cell arbitrarily.

Algorithm:

Step - 10:Continue this process until all zeros in row column are either assigned or struck off.

Step - 11:Now, if the number of assigned cells is equal to the number of rows/columns then it is the optimal solution.

Step - 12:The total cost associated with this solution is obtained by adding original cost figures in the occupied cells.

Step - 13:If the number of assigned cells is not equal to the number of rows/columns, further processing is required.

Pseudo Code:

```
Main Function :
    int size <- rand() (size of matrix)
    func1(int m)

func1 Function :
    int**matrix <- matrix on which operations are func2ne.
    for i <- 0 to size
        for j <- 0 to size
            matrix[i][j] <- rand()%20 + 1
    func2(int**matrix,int m)

func2 function :
    int** spare_matrix <- matrix
    func3(matrix,size)
    func4(matrix,size)
    func6(matrix,size)
    func3(matrix ,size)
    point** final_matrix <- func8(matrix, size)
    if(func14(final_matrix, size) ! <- 0)
        func15(final_matrix, size);
        func16(final_matrix, size);
    answer <- func17(spare_matrix, final_matrix, size);
    print answer
```

```
func3 function :
    for i <- 0 to size
        for j <- 0 to size
            amount <- length of every particular row.
            print matrix
func4 function :
    for i <- 0 to size
        int min <- func5(matrix[i], size);
        for l <- 0 to size
            matrix[i][l] <- matrix[i][l] - min
func5 function :
    min <- matrix[0][x]
    for i <- 0 to size
        if matrix[i][x] < min
            min <- matrix[i][x]
    return min
func6 function :
    for i <- 0 to size
        int min <- func7(int **matrix,int column_index,int size)
        for l <- 0 to size
            matrix[i][l] <- matrix[i][l] - min
func7 function :
    min <- matrix[x][0]
    for i <- 0 to size
        if matrix[x][i] < min
            min <- matrix[x][i]
    return min
```

Pseudo Code:

```
func8 function :
    point** new_matrix <- copy(matrix)
    for i <- 0 to size
        func9(new_matrix, size, i)
    return matrix
func9 function :
    bool flag <- false
    for l <- 0 to size
        new_matrix[l][i].data <- 0
            int row_index <- func10(new_matrix, size, l)
            if row_index <- -1
                flag <- true
                new_matrix[l][i].used <- 1
                break
    if flag !<- 1
        for l <- 0 to size
            if new_matrix[l][i].data <- 0
                int column_index <- func10(new_matrix, size, l)
                if func11(new_matrix, size, column_index, i) !<- 0
                    new_matrix[l][i].used = 1
                    new_matrix[l][column_index].used = 0
                    break
                else
                    new_matrix[l][i].used = 2
    func13(new_matrix, size)
func10 function :
    for i <- 0 to size
        if points[row_index][i].used == 1 & points[row_index][i].data == 0
            return i;
    return -1;
```

```
func11 function :
    if start_column == column
        return false
    int count = 0;
    for i <- 0 to size
        if new_matrix[i][column].data == 0
            count <- count + 1
            index <- func10(new_matrix, size, i)
            if index <- -1
                func12(new_matrix, size, column)
                new_matrix[i][column].used <- 1
                return true
    if count <- 1
        return false
    for j <- 0 to size
        if new_matrix[j][column].data == 0
            count <- count + 1
            index <- func10(new_matrix, size, l)
            if index <- -1
                func11(new_matrix, size, column_index, start_column)
                new_matrix[j][column].used <- 1
                return true
    return false
func12 function :
    for l <- 0 to size
        if matrix[l][column_index].data <- 0 && matrix[l][column_index] <- 1
            matrix[l][column_index].used <- 0
            break;
```

Pseudo Code

```
func15 function :
    rows <- new bool[size]
    columns <- new bool[size]
    for i <- 0 to size
        rows[i] <- false
        columns[i] <- false
    flag <- false
    for i <- 0 to size
        for j <- 0 to size
            if matrix[i][j].used <- 1
                for l <- 0 to size
                    if matrix[i][j].used <- 2
                        rows[j] <- true
                        flag <- true
                        break
            if flag <- true
                break
        if flag <- false
            for j <- 0 to size
                if matrix[j][i].used <- 1
                    columns[i] <- true
                    break
        flag <- false
    for i <- 0 to size
        for j <- 0 to size
            if rows[i] <- false && columns[j] <- false
                if min > matrix[i][j].data
                    matrix[i][j].data <- min
    for i <- 0 to size
        for j <- 0 to size
            if rows[i] <- false && columns[j] <- false
                if min > matrix[i][j].data
                    matrix[i][j].data <- matrix[i][j].data - min
            if rows[i] <- true && columns[j] <- true
                if min > matrix[i][j].data
                    matrix[i][j].data <- matrix[i][j].data + min
            matrix[i][j].used <- 0
```

```
func14 function :
    for i <- 0 to size
        bool flag <- false
        for j <- 0 to size
            if matrix[i][j].used <- 1
                flag <- truebreak
    return true

func16 function :
    for i <- 0 to size
        func9(point** new_matrix,int size,int i)
    return matrix

func17 function :
    int sum <- 0
    for i <- 0 to size
        for j <- 0 to size
            if point_matrix[i][j].use <- 1
                sum <- sum + matrix[i][j]
    return sum
```

Algorithm Analysis

A. Time Complexity Analysis:

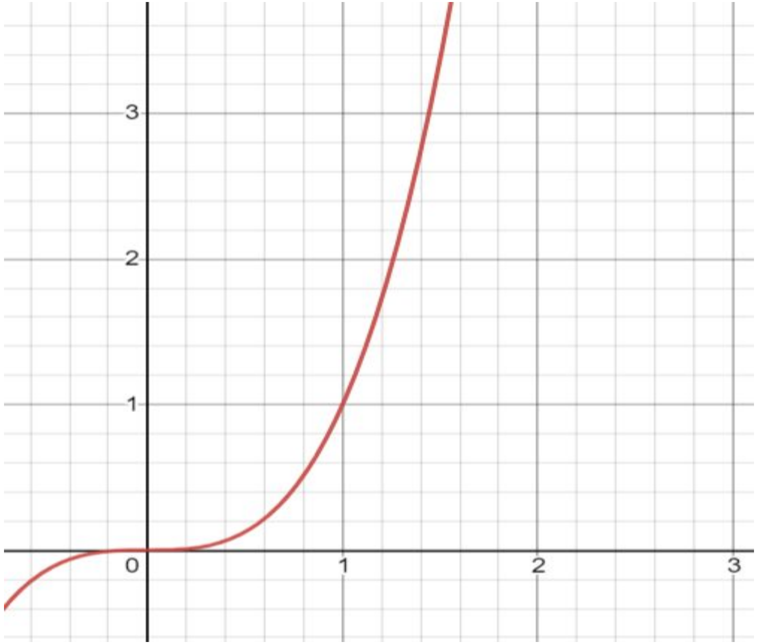
We can observe that, we are using nested for loops in the functions to compute whether the element is assigned or not.Hence , Complexity can be calculated as O(n3) , where n is the size of Input matrix size.

B. Space Complexity :

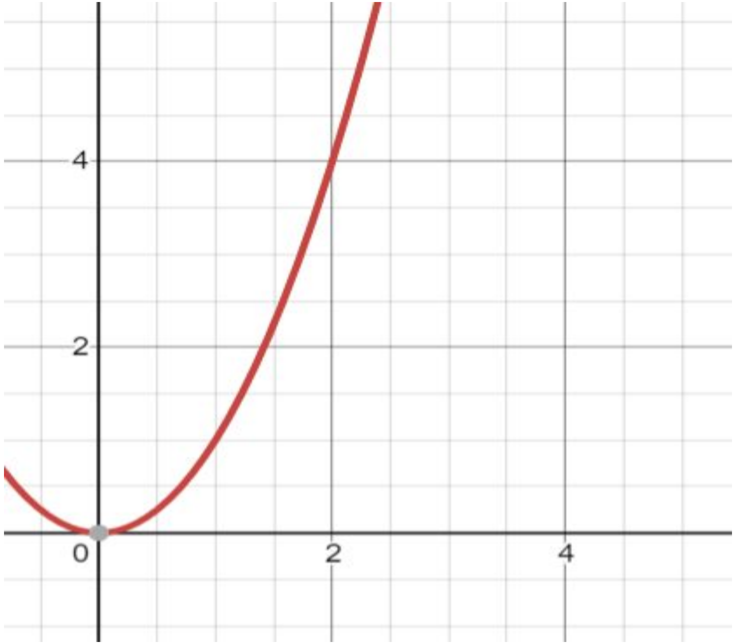
The space complexity of the Program is O(n2) because we at maximum use four 2d arrays.

Algorithm Analysis:

Time Complexity:



Space Complexity:



Conclusion:

Here,the technique gives the viable outcome in regards to optimality of a Balanced Assignment issue.Brute force solution is to consider every possible assignment implies a complexity of $\Omega(n!)$.The Hungarian algorithm, aka Munkres assignment algorithm, utilizes the following theorem for polynomial runtime complexity (worst case $O(n^3)$ and guaranteed optimality.