Project: Hungarian algorithm

STMO

2021-2022

project by Triana Forment

Introduction

Optimal transportation leads with two probability distributions and there is a cost function for moving elements from one distribution to the other distribution. And you have to find a transportation scheme to map one distribution into the other one. The original version of this problem of optimal transport dates back to Gaspard Monge in 1781. The Monge's problem has a discrete n number iron mines and n factories and, the cost is the distance between the mine and the factory. The solution is to find which mine suplies which factory, finding the minimum average distance transported.

The Hungarian algorithm solves an assignment problem which can be considered as a special type of transportation problem in which the number of sources and sinks are equal. The capacity of each source as well as the requirement of each sink is taken as 1. In this project, the Hungarian algorithm is implemented using an adjacency matrix. This algorithm can be used to find the map that minimizes the transport cost, given two distributions and a cost function. For two discrete probability vectors, $\mathbf{a} \in \Sigma_n$ and $\mathbf{b} \in \Sigma_m$, we have a $n \times m$ cost matrix C. It can be represented as a adjacency matrix, with elements in a as rows and elements in b as columns, and their weights as entries in the matrix.

An example of an assignment problem, can be, given n agents and the money they ask for performing each task, finding which agent performs which task in order to minimize the cost of performing all m tasks. When there are the same number of agents than tasks, the the problem is called balanced assignment, otherwise is called unbalanced assignment. And if the total cost is the sum for each agent performing its task, the problem is called linear assignment, because the cost function to be optimized as well as all the constraints contain only linear terms. This implementation can be used to solve linear balanced assignment problems.

Implementation

Outline of the approach

1. Subtract the smallest entry in each row from all the entries in the row, thus making the smallest entry in the row now equal to `0`. The same for each column.
2. Mark the rows and columns which have 0 entries, such that the fewest lines possible are drawn.
3. If there are m marked rows and columns, an optimal assignment of zeros is possible and the algorithm is finished. The cost value and the optimal agent-task combination can be calculated. If the number of lines is less than m, then the optimal number of zeroes is not yet reached and the next step needs to be performed.
4. Find the smallest entry not marked by any marked row and by any column. Subtract this entry from each row which is not marked, and then add it to each column which is marked. Then, go back to step 2.

`Hungarian_algorithm` is the funtion that uses as an input the cost matrix and gives as an output the cost value and the matrix solution. This function performs steps 1 to 4, by calling the previous functions. Once the marked rows and columns sum the number of tasks to perform, it calculates the cost value by adding each cost value of the different agent-task selected. The matrix solution has all zeros, except for each agent-task pair selected, where it has its original value.

Main.workspace2.Hungarian_algorithm

```
"""
    Hungarian_algorithm(mat)

Solves a linear balanced assignment problem.

Input:
    - 'mat`: the cost matrix

Outputs:
    - `cost`: the resulting cost value
    - `matrix solution`: the matrix with the selected agent-task as its original cost value and the rest zeros.
"""
function Hungarian_algorithm(mat)
    num_rows = size(mat)[1]
    num_cols = size(mat)[1]
         @assert num_rows == num_cols throw(DimensionMismatch("It is not balanced. Number of sources and sinks need to match"))

    matrix = copy(mat)
    #Step 1: subtract its internal minimum from every column and every row
    matrix .-= minimum(matrix, dims=2) # from rows
    matrix .-= minimum(matrix, dims=1) # from columns


    num_zeros = 0

    # Step 2: mark rows and columns that have the 0 entries
    pos = []
    while num_zeros < num_rows
        pos, rows, cols = mark_matrix(matrix)
        rows = reverse(rows)
        num_zeros = length(rows) + length(cols)
        # Step 3: if marked rows & columns is less than num_rows, go to step 4.
        if num_zeros < num_rows
            # Step 4: adjust the matrix
            matrix = adjust_matrix(matrix, rows, cols)
        end
    end
    # Calculate total cost and matrix solution
    cost = 0
    matrix_solution = zeros(Int64, size(mat)[1], size(mat)[2])
    for i in 1:num_rows
        cost += mat[pos[i][1], pos[i][2]]
        matrix_solution[pos[i][1], pos[i][2]] = mat[pos[i][1], pos[i][2]]
    end
    return cost, matrix_solution
end
```

find_min_row is a function that gets the row with the fewest zeros. For that purpose, it needs as an input a boolean matrix, having `true` if the original value after subtracting the smallest entry in each row from all the other entries in the row was `0`, and a `false` if it was different from `0`. It also needs a vector `zero_list` that will store the coordinates of the `0` in the row with fewest zeros. After the first finding, it reajusts the boolean matrix so the whole row and column from the last coordinate stored is set to `false`. This function is used in the next function `mark_matrix`.

```
md"""
**`find_min_row`** is a function that gets the row with the fewest zeros. For that purpose, it needs as an input a boolean matrix, having `true` if the original value after subtracting the smallest entry in each row from all the other entries in the row was `0`, and a `false` if it was different from `0`. It also needs a vector `zero_list` that will store the coordinates of the `0` in the row with fewest zeros. After the first finding, it reajusts the boolean matrix so the whole row and column from the last coordinate stored is set to `false`. This function is used in the next function `mark_matrix`.
"""
```

`Main.workspace2.find_min_row`

```
"""
    find_min_row(matrix_zero, zero_list)

Inputs:
    - `matrix_zero`: boolean matrix
    - `zero_list`: array to store the coordinates of the zeros in the row with fewest zeros.

Outputs:
    - `min_row`: tuple with the first element the number of zeros in the row with fewest zeros and the index of that row as second element
    - `zero_list`: list of coordinates tuples of the zeros found
"""
function find_min_row(matrix_zero, zero_list)
    min_row = [Inf, -1]

    for row in 1:size(matrix_zero)[1]
        #if true in matrix_zero[row, :] # Needs a 0 in the row
            # If the number of zeros < than the last min stored
            if sum(matrix_zero[row,:]) > 0 && min_row[1] > sum(matrix_zero[row,:])
                #stores the number of zeros and the row index
                min_row = [sum(matrix_zero[row,:]), row]
            end
    end
    # Get the column index
    zero_index = findall(x->x==true, matrix_zero[min_row[2],:])[1]
    # Store tuple in zero_list the tuple of coordinates
    append!(zero_list, [(min_row[2], zero_index)])
    # Mark the specific row and column as false.
    matrix_zero[min_row[2], :] .= false
    matrix_zero[:, zero_index] .= false
    return min_row, zero_list
end
```

`mark_matrix` needs as an input the matrix with the already subtracted the smallest entry in each row from all the other entries and it returns the `zero_list` and the marked row and column indexes. For that, it creates the boolean matrix and calls the function `find_min_row`. It performs steps 2 and 3.

```
md"""
**`mark_matrix`** needs as an input the matrix with the already subtracted the smallest entry in each row from all the other entries and it returns the `zero_list` and the marked row and column indexes. For that, it creates the boolean matrix and calls the function `find_min_row`. It performs steps 2 and 3.
"""
```

Main.workspace2.mark_matrix

```
"""
    mark_matrix(mat)

Perform steps 2 and 3: It marks the rows and columns that have the 0 entries such that the fewest lines possible are drawn. If there are "m" marked rows and columns, an optimal assignment solution can be found.

Input:
    - `mat` : cost matrix already modified by step 1 or an adjusted matrix by `adjust_matrix`

Outputs:
    - `zero_list` : coordinates of the zeros found by `find_min_row`
    - `marked_rows`: array of marked rows
    - `marked_cols`: array of marked columns
"""
function mark_matrix(mat)

    #Transform the matrix to boolean matrix(0 = true, others = false)
    cur_mat = mat
    boolean_mat = iszero.(mat)
    boolean_mat_copy = copy(boolean_mat)
    #Recording possible answer positions by zero_list
    zero_list = []
    while true in boolean_mat_copy
        find_min_row(boolean_mat_copy, zero_list)
    end
    #Recording the row indexes
    zero_list_row = []
    for i in 1:length(zero_list)
        append!(zero_list_row, zero_list[i][1])
    end

    # Get non marked rows
    non_marked_row = (x->collect(x))(setdiff(Set(1:size(cur_mat)[1]), Set(zero_list_row)))

    marked_cols = []
    flag = 0
    while flag == 0 # Enter in the loop until there's no more unmarked cols and rows
        flag = 1
        for i in 1:length(non_marked_row)
            row_array = boolean_mat[non_marked_row[i], :]
            for j in 1:length(row_array)
                # Find unmarked 0 elements in the corresponding column
                if row_array[j] == true && j ∉ marked_cols
                    # Store column index in "marked_cols"
                    append!(marked_cols, j)
                    flag = 0
                end
            end
        end
        zero_rows = [t[1] for t in zero_list]
        zero_cols = [t[2] for t in zero_list]
        for i in 1:length(zero_list)
            # If in "zero_list" coordinates there was a 0 marked in cols,
            # but not in marked_rows, add its row index to "non_marked_row"
            if zero_rows[i] ∉ non_marked_row && zero_cols[i] in marked_cols
                append!(non_marked_row, zero_rows[i])
                flag = 0
            end
        end
    end
    # Add those the indexes not stored in "non_marked_row" to "marked_rows"
```

```
    marked_rows = (x->collect(x))(setdiff(Set(1:size(cur_mat)[1]),
Set(non_marked_row)))
    return zero_list, marked_rows, marked_cols
end
```

adjust_matrix is a function needed when, after performing steps 1, 2 and 3, the number of marked rows plus the number of marked columns is not equal to the number of tasks/activities to perform by the agents. Then, it adjustes and returns the matrix as described in step 4.

Main.workspace2.adjust_matrix

```
"""
    adjust_matrix(mat, marked_rows, marked_cols)

Perform step 4: Adjust the matrix by finding the smallest entry not marked by any
marked row and by any column. Subtract this entry from each row that isn't marked,
and then add it to each column that is marked.

Inputs:
    - `mat` : boolean matrix created and modified by `mark_matrix`
    - `marked_rows`: array of marked rows produced by `mark_matrix`
    - `marked_cols`: array of marked columns produced by `mark_matrix`

Output:
    - `adjusted`: the adjusted matrix
"""
function adjust_matrix(mat, marked_rows, marked_cols)
    adjusted = mat
    min_values = []
    n = size(adjusted)[1]

    # Find the minimum value for that is not in marked_rows and not in marked_cols
    for row in 1:n
        if row ∉ marked_rows
            for i in 1:n
                if i ∉ marked_cols
                    append!(min_values, adjusted[row,i])
                end
            end
        end
    end
    min_value = minimum(min_values)
    # Substract that min value to all elements not in marked rows and columns
    for row in 1:n
        if row ∉ marked_rows
            for i in 1:n
                if i ∉ marked_cols
                    adjusted[row, i] = adjusted[row, i] - min_value
                end
            end
        end
    end
    # Add the min value to elements that are in both marked row and columns
    for row in marked_rows
        for col in marked_cols
            adjusted[row, col] = adjusted[row,col] + min_value
        end
    end
    return adjusted
end
```

Example and solution of assignment problem

Suppose we have 5 companies having different costs to realize 5 different tasks. You want to select a different company for each task in the way that we spend the minimum money. In order to find that combination, you just have to create a matrix with the cost each company asks for performing each activity. Here we have the following cost matrix with the companies as rows, the tasks as columns, and the values in thousand euros unit.

```
cost_matrix = 5×5 Matrix{Int64}:
               7  6  2  9  2
               1  2  5  3  9
               5  3  9  6  5
               9  2  5  8  7
               2  5  3  6  1
```

```
cost_matrix = [7 6 2 9 2;
               1 2 5 3 9;
               5 3 9 6 5;
               9 2 5 8 7;
               2 5 3 6 1]
```

After applying the function `Hungarian_algorithm`, you get the minimum cost (12000 €), and the solution matrix, which shows the company-task combination to get that minimum cost. The first company will perform the third task for 2000 €, the second company the first task for 1000 €, etc.

```
(12, 5×5 Matrix{Int64}:)
      0  0  2  0  0
      1  0  0  0  0
      0  0  0  6  0
      0  2  0  0  0
      0  0  0  0  1
```

```
Hungarian_algorithm(cost_matrix)
```

In case you want to maximize the values (e.g. instead of cost values you have benefit values), you just have to create the benefit matrix with negative values. This would solve an assigment problem maximizing the solution.

```
5×5 Matrix{Int64}:
 -7  -6  -2  -9  -2
 -1  -2  -5  -3  -9
 -5  -3  -9  -6  -5
 -9  -2  -5  -8  -7
 -2  -5  -3  -6  -1
```

```
begin
    benefit_matrix = copy(cost_matrix)
    benefit_matrix .= - cost_matrix
end
```

After applying `Hungarian_agorithm` to the benefit matrix, you get a maximum benefit of 41000 € by selecting the first company performing the fourth task, the second company the fifth, etc.

```
(-41, 5×5 Matrix{Int64}: )
        0   0   0  -9   0
        0   0   0   0  -9
        0   0  -9   0   0
       -9   0   0   0   0
        0  -5   0   0   0
```

```
Hungarian_algorithm(benefit_matrix)
```

References

- H. W. Kuhn. The Hungarian Method for the Assignment Problem. (2010), from http://bioinfo.ict.ac.cn/~dbu/AlgorithmCourses/Lectures/Lec10-HungarianMethod-Kuhn.pdf
- M. Thorpe. Introduction to Optimal Transport. (2018), from https://www.math.cmu.edu/~mthorpe/OTNotes
- Chapter 6 - Selected Topic in Mathematical Optimization course by Michiel Stock
- https://brilliant.org/wiki/hungarian-matching/
- https://python.plainenglish.io/hungarian-algorithm-introduction-python-implementation-93e7c0890e15