Misr University for Science and Technology
Collage of Information Technology

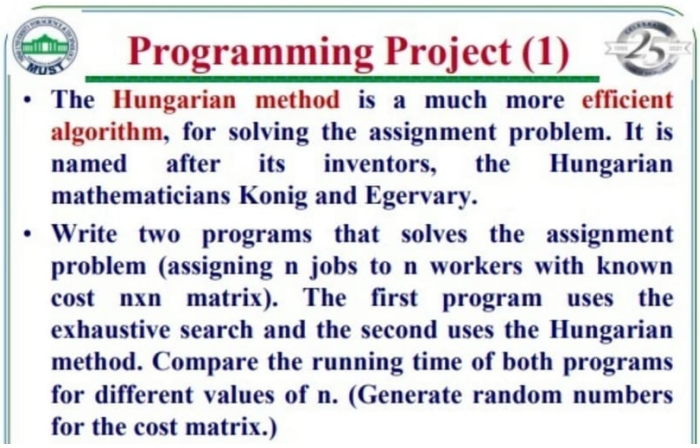
Course Code: CS 331
Title: Analysis & Design of Algorithms
Instructor: Prof. Dr. Mostafa Gad Al Haqq
Semester: Fall 2021/2022
Name: Nour El Din Gaser Yousri Algendi
Student ID: 89553

0 - Introduction

The purpose of this document is to give context to this project; what the project is, what my solution is, and a briefe explanation of the algorthms used.

Problem definition:

as per the project's text; assgn-ed by Prof. Dr. Mostafa Gad.



Programming Project (1)

- The Hungarian method is a much more efficient algorithm, for solving the assignment problem. It is named after its inventors, the Hungarian mathematicians Konig and Egervary.
- Write two programs that solves the assignment problem (assigning n jobs to n workers with known cost nxn matrix). The first program uses the exhaustive search and the second uses the Hungarian method. Compare the running time of both programs for different values of n. (Generate random numbers for the cost matrix.)

1 - The Assignment Problem

Assign n jobs to be executed be n people, each person is assigned to exactly one job and each job is assigned to exactly one person. The cost of assigning the i th person to the j th job is a known quantity, C[i, j]. The following is a cost matrix for n = 4.

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

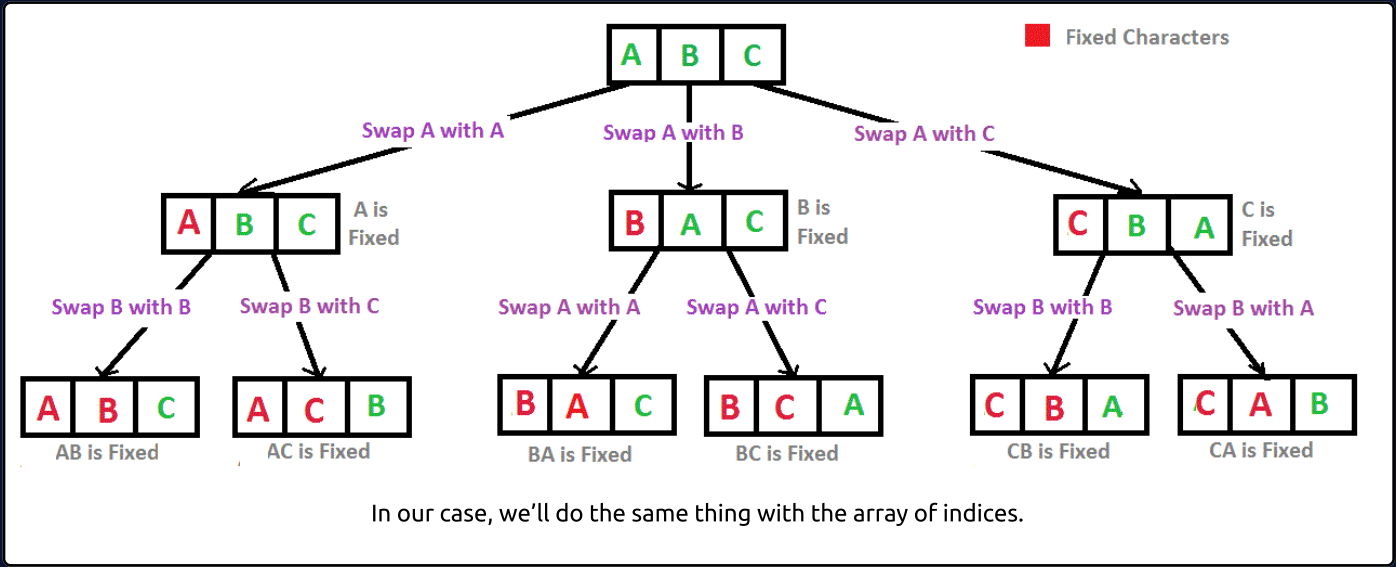
Solving this problem by a brute-force technique has complexity (n!). Fortunately, we have the Hungarian algorithm, which is a much more efficient algorithm for solving the assignment problem. It is named after its inventors, the Hungarian mathematicians Konig and Egervary.

2 - Exhaustive Search Solution

As mentioned, a brute-force solution would obviously be of O(n!) complexity, but an attempt at it must be made as a basis for comparision with the Hungarian algorithm.

A brute force solution can roughly be implemented as follows:

//Inputs: M[][], an mxm cost matrix, for which we want to find the assignment with the minimum cost.
//Outputs: an array A[], contaning m+1 elements; where A[i] is the i'th assignment, and the last element is the cost for that assignment (the optimal cost)
1 - Calculate all the permutations possible for assignments:



In our case, we'll do the same thing with the array of indices.

2 - For each permutation, calculate the cost of the assignment.
3 - Compare the cost to the minimum cost seen, if it's smaller, store it along with the permutation. (In array A).
4 - Return array A.

For my implementation, check the project's GitHub repository @/src/utils.cpp:51
The number of recursive function calls of my implementation would be of order ~Θ((n+1))!

* https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/

Nour Gaser
/nourgaser
/nourgaser
You can view this project's repository on GitHub on:
https://github.com/nourgaser/DesignAndAnalysisOfAlgorithmsCourse_MUST
Email me at nour89553@student.must.edu.eg

3 - The Hungarian Algorithm

The Hungarian algorithm needs no introductions at this point; but here's very brief walkthrough of how to implement it:

The Hungarian algorithm consists of the four steps below. The first two steps are executed once, while Steps 3 and 4 are repeated until an optimal assignment is found. The input of the algorithm is an n by n square matrix with only nonnegative elements.

Step 1: Subtract row minima

For each row, find the lowest element and subtract it from each element in that row.

Step 2: Subtract column minima

Similarly, for each column, find the lowest element and subtract it from each element in that column.

Step 3: Cover all zeros with a minimum number of lines

Cover all zeros in the resulting matrix using a minimum number of horizontal and vertical lines. If n lines are required, an optimal assignment exists among the zeros. The algorithm stops.

If less than n lines are required, continue with Step 4.

Step 4: Create additional zeros

Find the smallest element (call it k) that is not covered by a line in Step 3. Subtract k from all uncovered elements, and add k to all elements that are covered twice.

* https://www.hungarianalgorithm.com/hungarianalgorithm.php

For my implementation, check the project's GitHub repository @/src/hungarian.cpp
This implementation of the Hungarian Method runs in the order of $O(n^3)$.

* https://www.cc.gatech.edu/~rpeng/18434_S15/hungarianAlgorithm.pdf **@3.2**

Nour Gaser

/nourgaser

/nourgaser

You can view this project's repository on GitHub on:

https://github.com/nourgaser/DesignAndAnalysisOfAlgorithmsCourse_MUST

Email me at nour89553@student.must.edu.eg

4 - Runtime Comparison

There's almost no need to mention that the Hungarian algorithm wins, but the key here is: by how much? Given that the brute-force implementations (not just mine) run in factorial runtime, whereas the Hungarian Method's implelentations run in polynomial runtimes, the difference is huge, here are my results:

All calculations were carried out on the same machine with the same cost matrix, each value of n was run 10 times and the average is included in the above table.

n	Hungarian	Brute Force
5	0 ms	1 ms
6	0 ms	3 ms
7	0 ms	20 ms
8	0 ms	114 ms
9	0 ms	532 ms
10	0 ms	5256 ms
11	0 ms	60920 ms
20	0 ms	-
50	1 ms	-
100	3 ms	-
1000	350 ms	-
2000	1684 ms	-
4000	6942 ms	-
8000	25684 ms	-
10000	57611 ms	-
20000	-	-

Nour Gaser
/nourgaser
/nourgaser

You can view this project's repository on GitHub on:

https://github.com/nourgaser/DesignAndAnalysisOfAlgorithmsCourse_MUST

Email me at nour89553@student.must.edu.eg