Université d'Ottawa
Faculté de génie

École de science
d'informatique
et de génie électrique

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Electrical
Engineering
and Computer Science

Assignment 0

CSI2120 Programming Paradigms

Sommer 2020

Part 1 and 2 due on June 22nd, 2020 before 11:00 pm in Virtual Campus

Part 3 and 4 due on July 26th, 2020 before 11:00 pm in Virtual Campus

[12+12 marks]

Problem Description

This assignment asks you to implement solutions to the optimal assignment problem. We look at in the context of visual face tracking. We assume that we have implemented some neural network-based face detector which in each video frame detects all faces in the frame. Below is an example from the face detector running in a House-of-Commons recording.

We get therefore a list of detected faces in one frame and another list of faces in the next frame. We would like to know which face in the list of the first frame matches which face in the second frame. Our matching algorithm is to find the optimal match of faces based on some criteria. A simple criterion is the

Euclidian distance between the centre of the first detection. Consider the following simple example with only 3 detected faces in each frame.

Frame 1:

Face Label	Upper-Corner		Width in X	Height in Y
	X	Y		
A	100	80	41	52
B	300	392	32	45
C	405	160	28	31

Frame 2:

Face Label	Upper-Corner		Width in X	Height in Y
	X	Y		
I	300	120	43	51
II	312	236	28	40
III	395	241	25	30

Based on these detections and the Euclidean cost function between the box centers, we can construct a 3x3 table. We round the cost to integer values which prevents issues with floating point calculations.

	I	II	III
A	205	254	324
B	269	159	183
C	102	123	81

We could now simply try all possible assignments:

$$\{(A,I),(B,II),(C,III)\},\{(A,I),(B,III),(C,II)\},$$
$$\{(A,II),(B,I),(C,III)\},\{(A,II),(B,III),(C,I)\},$$
$$\{(A,III),(B,I),(C,II)\},\{(A,III),(B,II),(C,I)\}$$

In our example, we have an optimal solution $\{(A,I),(B,II),(C,III)\}$ with a total cost of 445.

There are $3 * 2 * 1$ possible assignments, or in general $n * (n-1) * ... * 1 = n!$ In other words, trying all possibilities with a large number of detection becomes, very quickly prohibitively expensive to calculate.

Fortunately, there is a much more efficient algorithm which run $O(n\text{^}3)$. It proceeds in five steps:

1. Row reduction
2. Colum reduction
3. Test for an optimal assignment, if an optimal assignment is found, go to step 5
4. Shift zeros, go to step 3
5. Making the final assignment

This is the so-called Hungarian algorithm due to Kuhn-Munkres. The reduction steps are simple: In each row or column we need to find the minimum and subtract it from all values in that row or column, respectively. The test for an optimal assignment involves selecting a minimum number of horizontal and vertical lines crossing out rows or columns which cover all entries equal 0. If we can cover the zeros with n lines, then we can go to Step 5, otherwise we have to go to Step 4. In Step 5, we need to pick exactly one 0 entry in each row and column. If there is more than one way to do so, than there is more than one optimal assignment. Step 4 is only needed if we did not find a solution to cover all 0 with n lines but rather the zeros can be covered with less than n lines. In Step 4, the zeros are shifted by a two step process: First, we find the smallest value not covered by a line and subtract it from all uncovered values. This will become our new 0. Secondly, we add this value to all values covered by a row and column line. Than we can go back to Step 3. Note that this loop from step 3 to step 4 and back to step 3 may be needed at most $O(n)$ times. You can find an excellent explanation video on youtube https://www.youtube.com/watch?v=cQ5MsiGaDY8

The most difficult part of the algorithm is finding the minimum lines to cover all 0 in step 3. Below is pseudocode for this step with its 5 sub steps.

```
// 3.a
// Mark the first unmarked 0 in each row as assigned
// and cross out others in the row and in the newly assigned col
// This is the same as step 5 assuming we always pick the first 0
for r in rows
    first = true
    for c in cols
        if A(r,c) = 0
            if first
                A(r,c) is assigned; first = false
                for rr in rows
                if A(rr,c) = 0 and r != rr
                    A(rr,c) is crossed out
            else
                A(r,c) is crossed out
```

```
// 3.b
// Ticking rows
for r in rows
    tick( r ) = true
    for c in cols
        if A(r,c) is assigned
            tick( r ) = false

// 3.b
// Ticking cols
for r in rows
    if not(tick( r )) continue
    for c in cols
        if A(r,c) is crossed out
            tick( c ) = true

// 3.c
// Ticking rows again
for c in cols
    if not(tick( c )) continue
    for r in rows
        if A(r,c) is assigned
            tick( r ) = true

// 3.d
// Go back to step 3.b unless no new ticks were made

// 3.e
// Draw lines for all ticked columns and all unticked rows.
for r in rows
    if not(tick(r))
        line(r) = true
    else
        line(r) = false
for c in cols
    if tick(c)
        line(c) = true
    else
        line(c) = false
```

Part 1: Object-oriented solution (Java) **[6 marks]**

Create the classes needed to solve the visual face tracking problem based on the Hungarian algorithm as exactly as described above. Your program must be a Java application called `FaceTracker` that takes as input the names of two files containing all detections for two frames as csv files. Your program must print the optimal assignment to a csv file called `tracker_java_`**`n`**`.csv` where **`n`** is the size of in the problem. The file should have n rows and 2 columns corresponding to one match per row and the id of the detection in the two columns (see the attached example). The file is to be saved in the current directory. If there is more than one optimal assignment, your implementation must simply return one of them. If the number of detections is not the same in each frame (i.e., your cost matrix would not be square), your implementation can simply throw an exception.

Your implementation must follow an **object-oriented design** for full marks. You must at least use separate classes for reading and holding the face detection, for calculating the matching cost, for the Hungarian algorithm and for the optimal assignment. Your implementation must print the cost matrix at the beginning and after each step of the main algorithm. Make sure to print out the row and column numbers covered by lines after step 3. Your implementation must also **save the initial cost matrix to a csv-file**.

The above cost function is the Euclidean distance. You must design and use at least one alternative cost function which the user can select, e.g., matching the area of the detected boxes or the aspect ratio of the box or any combinations of criteria. Give an equation how you calculate the cost along with your UML diagram.

In addition to the source code, you must also submit a UML class diagram showing all classes, their attributes, methods, and associations. Hand-drawings are not acceptable (if you are looking for a drawing program, you can use draw.io). You can not use static methods (except `main`).

Part 2: Concurrent solution (Go) **[6 marks]**

Create a Go application that implements the Hungarian algorithm. Your Go application is to read a cost matrix from a csv file. Your program must produce a Go executable called `face_tracker.exe` that takes as input the name of the csv-file for the cost matrix. Your program must save the optimal assignment in a file `tracker_go_`**`n`**`.csv` where **`n`** is the size of in the problem with the same format as in Part 1. As for Part 1, your implementation must print the cost matrix at the beginning and after each step of the main algorithm. Make sure to print out the row and column numbers covered by lines after step 3.

Your program must execute the row reduction in step 1 concurrently for each row and in step 2 the column reduction concurrently for each column. Note that you must make sure that all go routines for step 1 have finished before starting step 2. You will not receive full marks if your solution does not run concurrently, even if you use the keywords go and/or channel.

You must follow proper imperative and concurrent design for full marks including the creation and use of packages.

Part 3: Logic solution (Prolog) **[6 marks]**

Create the following Prolog predicate

```
hungarianMatch( CostMatrix, OptimalAssign, OptimalCost )
```

that is true if the Hungarian matching problem with the given cost matrix by the optimal assignment with an optimal cost.

Your solution for the predicate `hungarianMatch/3` must function as a predicate with all three parameters instantiated but also find **all** optimal assignments for full marks. Your predicate is to produce one optimal assignment at a time and using the `;` in the interpreter, the next optimal assignment is found. Your implementation should follow the generate-and-test approach which is `O(n!)`. Don't use test cases that are too large as they would take a long time.

Create also the following two Prolog helper predicates, to read the cost matrix (as in the attached example) and to save one optimal assignment, respectively.

```
readCostMatrixCSV( "face_cost3.csv", CostMatrix )

saveOptimalAssignment( OptimalAssign, "tracker_prolog_3.csv")
```

Part 4: Functional solution (Scheme) **[6 marks]**

Create a Scheme function that implements the Hungarian algorithm. Your Scheme application is to read a cost matrix from a csv file. Your function prototype must work as follows:

```
(hungarianMatch (readCostMatrixCSV "face_cost3.csv")
                          "tracker_scheme_3.csv")
 ⇨  (("A" "I") ("B" "II") ("C" "III"))
```

Your function takes as input the cost matrix and a file name to save the optimal assignment to as a side effect. The format of the csv file should be the same as in Part 1. It is suggested that you use `tracker_scheme_`**`n`**`.csv` where **`n`** is the size of in the problem as a file name. As for Part 1, your implementation must print the cost matrix at the beginning and after each step of the main algorithm. Make sure to print out the row and column numbers covered by lines after step 3.

Your solution must follow proper functional design for full marks. If you are using and manipulating variables with `set!` and/or use `define` for variables, you are likely to use an imperative style and you will lose marks.