Implementing the GRA Algorithm in MATLAB

Computational Physics (PHYS-3007)

Evan Wells

I. Introduction

Preface- At first glance, it may seem that there is not much to gain in implementing an existing algorithm in a somewhat unorthodox language. Especially in a world of massive linear optimization toolkits like IBM's CPLEX or Python's PuLP, I found myself wondering if this was a good idea for a project. However, after beginning the implementation of the GRA algorithm, I discovered that in doing so, there is so much possibility for growth and learning.

The GRA algorithm, also known as Group Role Assignment, is a process that takes part within the Role Based Collaboration framework. It follows the Agent Evaluation process and consists of assigning the optimal agents to a specific task. This algorithm can be used in a multitude of different circumstances such as a load balancing function, logical optimization and numerous real-world situations.

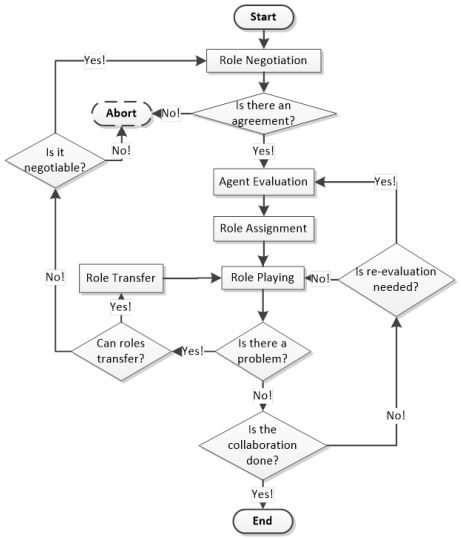

Figure1: Role Based Collaboration(RBC) framework

The goals that were to be achieved throughout this project are as follows:

1) Familiarize myself with the MATLAB environment
2) Learn and utilize the MATLAB built in library
3) Successfully implement the GRA algorithm in MATLAB code

The remainder of the paper is arranged as can be seen below. Section II overviews the methodology used throughout the project as well as how the original structure was derived. Section III covers the implementation, giving an in-depth analysis of the functions created. As for Section IV, it discusses results with a provided example. Section V considers the project as a whole and provides closure.

II. Methodology

The planning phase of this project was quick due to the fact that the GRA algorithm and Role Based Collaboration are the basis of Dr Zhu's research, a professor here at Nipissing University. There were numerous articles and a book I could refer to deduce the procedure necessary to achieve the goal of implementing the GRA algorithm [2], [3]. From these sources, it was evident that the process could be separated into three distinct sections: matrix preprocessing, K-M algorithm and deriving results. The K-M algorithm, or Kuhn-Munkres algorithm, is a well-known combinatorial optimization algorithm that solves the assignment problem [1]. The GRA algorithm utilizes this famous algorithm, while adding additional constraints to provide a broader more complex utility in the real world. The preprocessing steps are the essential addition that allows this. An in-depth explanation is provided in Section III.

III. Implementation

The implementation consists of 6 functions, all crucial to the overall functionality of the GRA algorithm. To aid in the explanation of the functions, an example of the arguments and the results of each function will be provided. For ease of reference, the initial arguments Q (mxn Qualification Matrix, where each row is an agent and each column is a role), L (n length Role Range Vector, where each value in the array is the amount of how much the specified role is needed) and τ (Qualification Threshold, lower bound) are provided below:

$$M = \begin{bmatrix} 0.2 & 0.6 \\ 0.4 & 0.9 \\ 0.8 & 0.3 \end{bmatrix} \quad L = [2 \quad 1] \quad \tau = 0.3$$

Figure 2: Reference initial values for example

Function 1. *transferQtoM*, the first function

required in the GRA algorithm takes a Qualification Matrix (Q), Role Range Vector (L) and threshold value (τ) and returns a mxm matrix M. As an initial condition to the K-M algorithm, the matrix it provides optimal assignment to, must be a square matrix. In order to ensure that the created matrix M is relative to the given inputs, this function adds column n, $L[n]$ times in the new M matrix. If the sum of the values in the Role Range Vector is less than m, than the remaining columns are zero column vectors. To finalize this function, the matrix needs to return the mirror value of itself. Essentially, in the GRA algorithm we try to find the maximal value, while the K-M algorithm evaluates for the minimal values. As such, the values in M must be reduced by 1 (1-M[i,j]). In addition, this function ensures that all values are above the threshold value. If they are not, this function ensures that they will not be picked as an optimal assignment. The following M matrix is the result of the *transferQtoM* function. As can be seen in Figure 3, the values of the Q matrix which are below the threshold value are now 7, making them much too large to ever be deemed optimal.

$$M = \begin{bmatrix} 7 & 7 & 0.4 \\ 0.6 & 0.6 & 0.1 \\ 0.2 & 0.2 & 7 \end{bmatrix}$$

Figure 2: Resulting M matrix from transferQtoM

Function 2. *prepKM*, a function that completes the preprocessing steps, ensures the intial conditions for the K-M algorithm are obeyed. The two conditions that remain are: there must be a zero in every row and there must be a column in every row. This is done through simple row and column reduction. With this function and its according results in Figure 4, the matrix M is ready for the K-M algorithm.

$$M = \begin{bmatrix} 6.6 & 6.6 & 0 \\ 0.5 & 0.5 & 0 \\ 0 & 0 & 6.8 \end{bmatrix}$$

Figure 1: Resulting M matrix of prepKM

Function 3. *KM_algorithm*, with the resulting matrix M from the *prepKM* function, we can begin this long and iterative process lovingly named the K-M algorithm. It begins by "starring" one zero in each row and column. A starred zero represents the current optimal state for the specific row (agent). The final output of this function is the T matrix, which contains the optimal assignments – which is the final state of starred zeros. The key to this function resides in the tracking of the state of numerous flags. Most importantly, checking if all the columns have a starred zero at each iteration of the function.This function itself follows the following pattern. It first calls the *primeZero* function to ensure there are non-starred zeros. If there aren't any, the matrix is reduced in a similar way to the *prepKM* function. This process continues until a prime is found. Once the prime is found, you use it's index to find the starred zero in the prime zero's row. If there is no starred zero, the sequence of the zeros in the matrix M must be switch. This subprocess is done by finding all the stars and primes in the matrix and storing their indexes. Once this is complete, the algorithm iterates through the set of indexes and swaps their positions when necessary, shifting the order of starred zeros within the matrix M.

After reordering, the algorithm verifies whether an optimal assignment has been achieved. If not, it continues the process until all conditions are met. Ultimately, the function finishes with a mxm T matrix of optimal assignments. However, this output does not always relate to the original input. This is especially the case in the example provided so far in this section. As can be seen in Figure 5, it implies that there are 3 roles, while in the intial Q and L, it is evident that there are only two roles. Thus comes into existence Function 5, the *deriveT* function.

$$T = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Figure 5: Resulting T matrix from KM_algorithm

Function 4. *primeZero*, this function aids in the *KM_algorithm* by promoting modularity and consistency while avoiding repeative code, as it is used more than once. Although simple in implementation, it reduces all the minimal rows and columns from the matrix, similar to the *prepKM* function.

Function 5. *deriveT*, this final function in the GRA algorithm fixes the issue addressed in Function 3. It transforms the results of Function 3 into a T matrix relative to the original Role Range Vector. This function proceeds by summating columns together for L[n] columns from T. This combines the columns to be completely relative to the role range vector. As can be seen in Figure 6, the first and second columns of the results T in Function 3 are combined to create the first column of the results below.

$$T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}$$

Figure 6: Resulting T matrix from deriveT

Function 6. *GRA*, this funtion does not necessarily count as a final function due to the fact that it does not add any new implementation. This function simply encapsulates functions one through five, as to only have

the original inputs of the *tranferQtoM* and return the optimal assignment matrix T.

With those 6 functions, the implementation is complete. Space complexity for this implementation is $O(n^2)$ and it's time complexity is $O(n^3)$. Essentially, this implementation works in a mxm by matrix in which it must iterate through the entire matrix to solve optimal assignment. The next section quickly explains the results found in the example seen above (Figure 6).

IV. Results

The results returned from the GRA function are as seen from Figure 6. The question now is, how does this matrix T relate to a world scenario. To show it's relation, we can set a real-world circumstance to the values presented in the previous section. In this scenario, we are trying to organize a batch job on the servers at company X. Company X has three servers, server0, server1 and server2. These servers have been evaluated to give a "server-health" value in two distinct categories. These categories are reading and writing respectively. From this information, we can present the Qualification Matrix (Q) seen in Figure 7, where "server-health" is a value $x \in [0, 1]$. The role range vector (L) showcases the requirements necessary for the next batch job. As L = [2, 1], the patch job will need 2 servers to read and 1 to write. The administration decides that if a server-health is lower than 30% (0.3) for a specific role, the server should not be considered for this role. This real-world scenario provides construct for the values given from the example in Section III. With these arguments for the algorithm, we can get the following results, where bolded values are the optimal assignments dictated by the GRA algorithm (Figure 6):

	read	write
server0	0.2	**0.6**
server1	**0.4**	0.9
server2	**0.8**	0.3

Figure 7: Screenshot of GRA results in real-world scenario

As can be seen, the GRA algorithm dictates that server1 and server2 should read, while server0 should write. Figure 7 seems to be wrong, as it is evident that the server-health for server2's writing is a impressive 30% more healthy than the server that was considered optimal. This is due to the constraint set by the administration, as server0's reading ability was below the threshold value. Overall, the interpreted results can be derived from the intial input of the Qualification Matrix. These results can be computed for an average qualification value, in order to evaluate the algorithm. In this specific case, the group score (σ) is 0.6, which for this real-world scenario represents a 60% averge server health across all servers.

$$\sigma = \frac{\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} Q\,[i,j] \times T[i,j]}{\sum_{i=0}^{m-1} T[i,j]}$$

Figure 8: Equation to calculate group score

V. Conclusion

Evidently, a real-world scenario for these circumstances would be within a much more complex environment, with many more considerations than reading and writing. However, this example does provide a general formulation for the use of the GRA algorithm. As the results in Section IV demonstrate, the algorithm can help optimize a modeled system of interactions, considering different factors such as skillset for different roles, minimal requirements and the need for some to act in the same role. In sum, this implementation is complete and can have real-world value.

For future work in this field, there are many directions to explore. When it comes to modelling a system, as is done in the GRA algorithm, there is no limit to what can be added. There is potential for adding mechanisms to model inter-agent cooperation or conflict, adding additional constraints to the system, like limiting the number of roles an agent can undertake, or even implementing an agent evaluation algorithm, which studies an agents' work, and updates the Q matrix according to their performance. There are many options when considering the potential future work in this field. Specifically, the implementation of a GMRA algorithm. This algorithm is similar to GRA, but each agent can play multiple roles and are not limited to one. This algorithm opens the door to many more real-world scenarios. In addition, its implementation is much more involved and would be a very fun challenge to undertake.

Before looking to the future, we must conclude on this project. Looking back on the goals set in Section I, I can say with absolute certainty that I have achieved my goals. I am very comfortable in a MATLAB environment. So much so – that it is now one of my favourite languages to write one off tools with. Its large library of useful functions – that I now know how to use and access, are extremely useful and intuitive to me now. Also, I am very confident working within a matrix in the MATLAB environment, thanks to the numerous hours spent doing so. Lastly, I achieved the ultimate goal, successfully implementing the GRA algorithm in

MATLAB. Overall, the project was quite enjoyable, as it allowed me to get into the very roots of an algorithm I've worked with quite a bit with, giving a much deeper understanding of how it works. I've come out of this project with more knowledge, increased coding ability and a new coding language in my arsenal.

VI. References

[1] CP-Algorithms, "Hungarian Algorithm for Assignment Problem,". https://cp-algorithms.com/graph/hungarian-algorithm.html.

[2] H. Zhu, *E-CARGO and Role-Based Collaboration: Modeling and Solving Problems in the Complex World*, Wiley-IEEE Press, NJ, USA, Dec. 2021.

[3] H. Zhu, M.C. Zhou, and R. Alkins, "Group Role Assignment via a Kuhn-Munkres Algorithm-based Solution," *IEEE Trans. on Systems, Man, and Cybernetics, Part A: Systems and Humans,* vol. 42, no. 3, May 2012, pp. 739-750.

[4] MathWorks. *Find array elements that meet a condition.* https://www.mathworks.com/help/matlab/matlab_prog/find-array-elements-that-meet-a-condition.html

[5] MathWorks. *min - Smallest elements in an array.* https://www.mathworks.com/help/matlab/ref/double.min.html