Matching a Bipartite graph.

Course: Algorithms Analysis & Design II.

Course Code: CSC315.

Semester: Spring 2020.

Department: Computer Science.

Instructor: Assoc. Prof. Fatty M. Salem.

Link: https://github.com/BonyGeorge/Job_Vacancies_Project

Name: Abanoub George Lamie Allam.

ID: 2017 / 03340.

Faculty: Faculty of Computer Science MIU.

Table of Contents:

[1] Abstract:

In our recent years, we have been using the graph-based object representation more often till it became a more popular than ever. The graph edit distance emerged is as powerful and more flexibility graph which is used to assign different tasks such as: Pattern Recognition, Machine Learning and Data Mining.

[2] Keywords:

Graph based representation, Pattern, Pattern Recognition, Machine, Machine Learning, Graph edit distance, Bipartite graph matching, Image feature matching.

[3] Introduction:

The **bipartite graph (G)** or also called **bigraph** is a graph which its vertices are divided into parallel sets or we can also say two adjacent sets of independent vertices which are **U sets** and **V sets** that are connected with **edge E** and there is no connection between each set vertices.

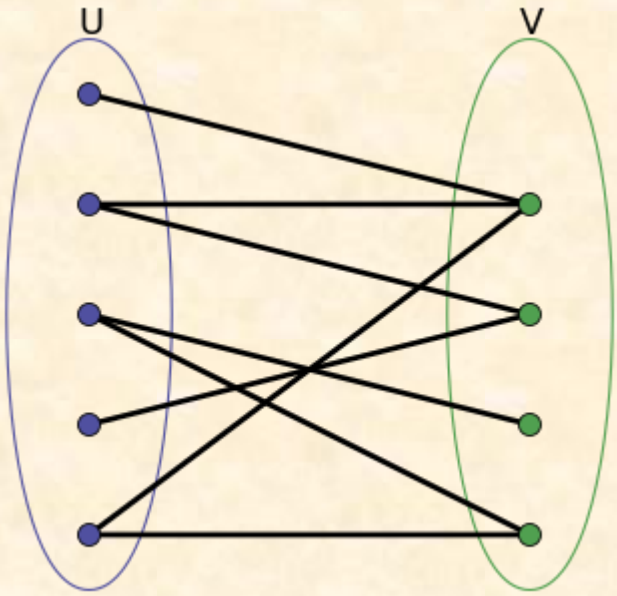

Fig (1): Simple bipartite example.

$$G = (U, \quad V, \quad E)$$

[4] Importance & Applications:

1 – Machine Learning.

2 – Pattern Recognition.

3 – Data Mining.

4 – Image Features matching.

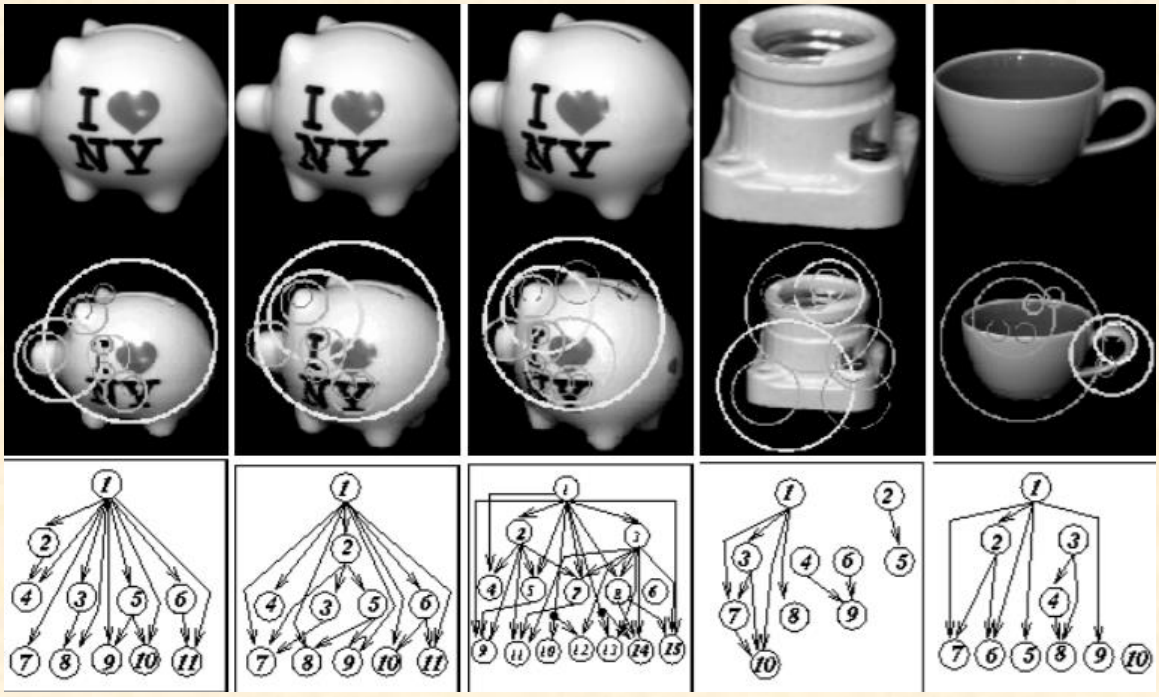

Fig (2): Example of Pattern Recognition using Maximum Bipartite Matching.

[5] Inputs & Outputs:

At the beginning of the program it will ask you to insert how many applicants you have and how many jobs are available. Then it will ask you to insert foreach applicant how many jobs did he submitted at. After inserting them in number values the program will calculate automatically the maximum flow (How many applicants are suitable for a job) by using the **Ford Fulkerson** Maximum flow algorithm. The size of the inputs depends on how many jobs the user is going to insert and how many applicants and also it depends on how many jobs the user has submitted at. The input type for the whole project is **Integer** which is equal to **4 Bits/entry**.

```
        int job = 0;
        int Count_Applicant = 0;
        int Count_Job = 0;
        int i = 0;
        int j;

        // The Bipartite Graph.
        boolean [][]Graph;

        System.out.println("Enter the Number of Applicants : ");
        Count_Applicant = number.nextInt();
        obj.insert_Count_Applicants(Count_Applicant);

        System.out.println("Enter the Number of Jobs : ");
        Count_Job = number.nextInt();
        obj.insert_Count_Jobs(Count_Job);

        Graph = new boolean [Count_Applicant][Count_Job];

        // Here we make the user enter how many jobs did the applicant submittied in.
        while (i < Count_Applicant)
        {
            System.err.println("Enter the jobs that Applicant : "+ i + " entered or enter -1 to go to next applicant");
            job = number.nextInt();

            if (job != -1)
            {
                Graph[i][job] = true;
            }
            else
            {
                for (j = 0; i < j; j++)
                {
                    if (Graph[i][j] != true)
                    {
                        Graph[i][j] = false;
                    }
                }
                i ++;
            }

        }
        System.out.println("Maximum flow of Applicants that are suitble for the job : " + obj.Maximum_Bipartite_Matching(Graph));

    }
}
```

Fig (3): Inserting inputs in the program.

[6] Techniques:

The maximum bipartite matching problem has many techniques to be solved by but most of the people nowadays, prefer using the Ford Fulkerson maximum flow algorithm.

From these techniques are:

1 – Greedy Algorithm.

2 – Augmenting Path Algorithm.

3 – Ford Fulkerson Algorithm.

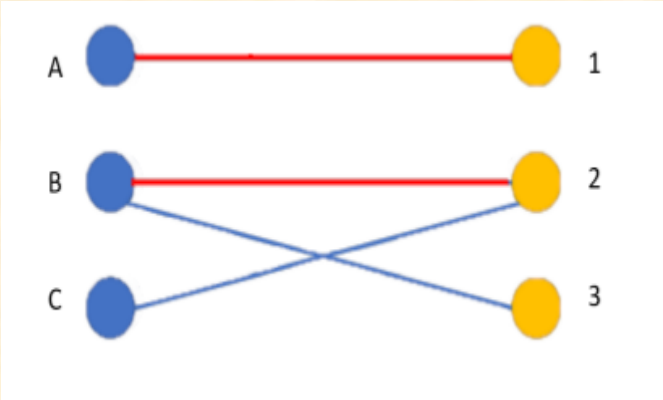

Fig (4): Example of Augmenting Path Algorithm.

[7] Results & Discussions:

At the beginning the user enters how many applicants and how many jobs that are available. Then the program starts to ask the user foreach applicant to enter the id of

the job that the applicant with id has submitted to as long as the user doesn't exceed the number of available jobs or doesn't enter (-1) the program will resume to take jobs. After entering all the submitted applications the program automatically constructs a **Boolean Bipartite Graph**.

Then we make an instance of class Job_Vacancies and call to its member function which is the **Maximum_Bipartite_Matching** and pass to it the graph that we constructed earlier. The function takes the graph and make an array with the number of the jobs available. Then it begins to initialize this array by making all the jobs available for any applicant to submit in it. Then we try to count how many jobs did the applicant submit in then we use the **Bipartite_Matching** member function to check if the user is suitable for the job that he

did submit in. If he is suitable, he would be assigned to this job.

```
// Maximum matching using ford fulkerson.
public int Maximum_Bipartite_Matching(boolean[][] Graph)
{
    int[] Matched_Applicant = new int[Jobs_Number];

    // It initialize all the jobs by available for any applicant to submit in it.
    for (int i = 0; i < Jobs_Number; ++i)
    {
        Matched_Applicant[i] = -1;
    }

    int result = 0;

    // We count here how many jobs the applicant submitted in.
    for (int Vertex_U = 0; Vertex_U < Applicants_Number; Vertex_U++)
    {

        boolean[] viewed = new boolean[Jobs_Number];

        for (int i = 0; i < Jobs_Number; ++i)
        {
            viewed[i] = false;
        }

        // It checks if the applicant is suitable for the job.
        if (Bipartite_Matching(Graph, Vertex_U, viewed, Matched_Applicant))
        {
            result++;
        }
    }

    return result;
}
```

Fig (5): The Maximum Bipartite Matching function.

When using the Bipartite_Matching member function this function checks if the user is suitable for the job that he did submit in the it will automatically assign this job for this submitted applicant but first it checks if that the applicant hasn't been assigned to any other job from the jobs that are available.

```java
// Function to check matching between verticies.
public boolean Bipartite_Matching(boolean[][] Bipartite_Graph, int Applicants_Vertex, boolean[] viewed, int[] Matched)
{
    for (int Jobs_Vertex = 0; Jobs_Vertex < Jobs_Number; Jobs_Vertex++)
    {
        // Checks if the applicant submitted fo this job or not he must not be in another job.
        if (Bipartite_Graph[Applicants_Vertex][Jobs_Vertex] && !viewed[Jobs_Vertex])
        {
            viewed[Jobs_Vertex] = true;

            /*
                If the job isn't assigned to any applicant
                then the applicant will be assigned for the
                job.
            */

            if (Matched[Jobs_Vertex] < 0 || Bipartite_Matching(Bipartite_Graph, Matched[Jobs_Vertex], viewed, Matched))
            {
                Matched[Jobs_Vertex] = Applicants_Vertex;
                return true;
            }
        }
    }
    return false;
}
```

Fig (6): The Bipartite Matching function.

[8] Flowchart:

- The Bipartite Matching Function:

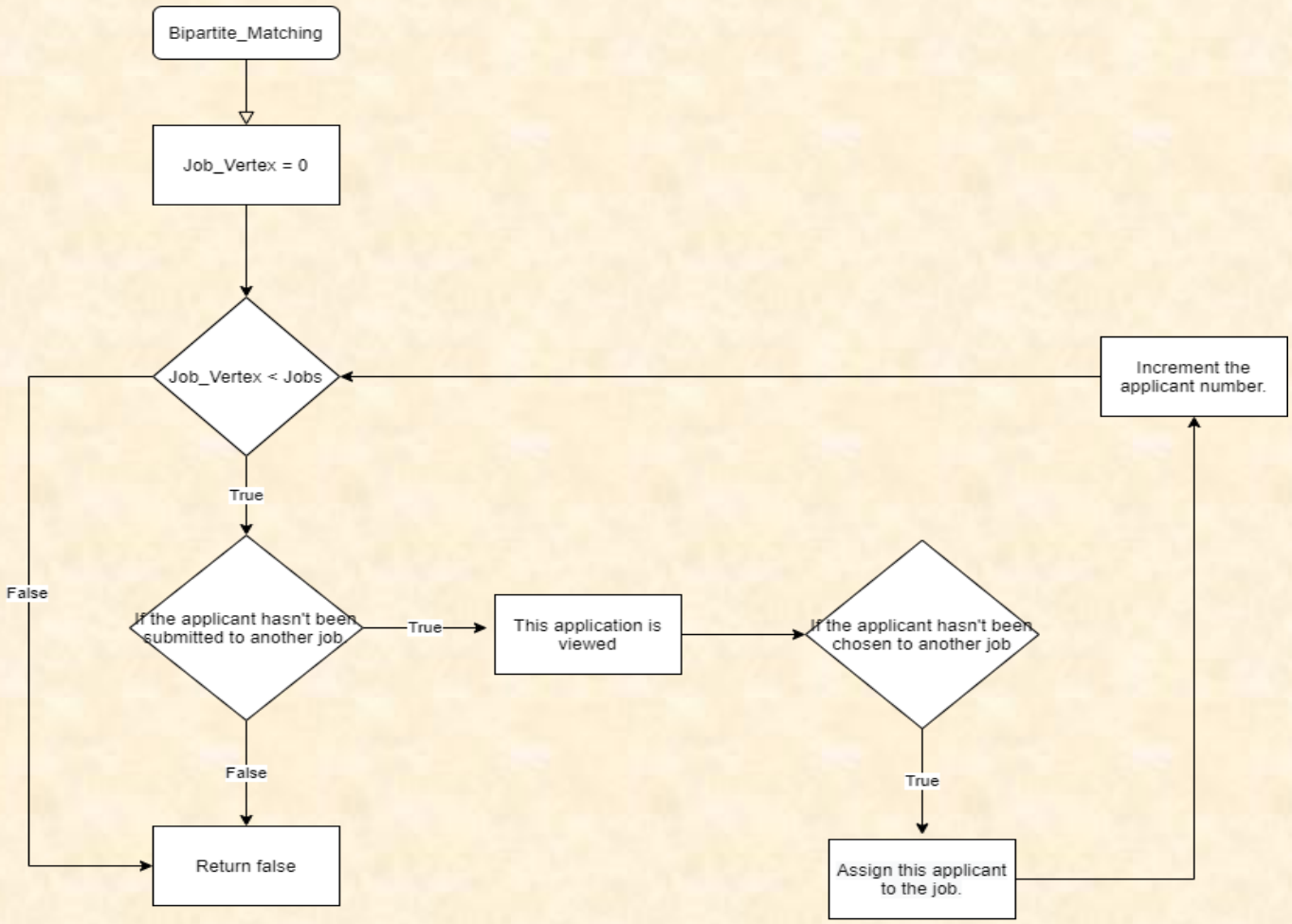

Fig (7): Bipartite Matching function flowchart.

- **The Maximum Bipartite Matching Function:**

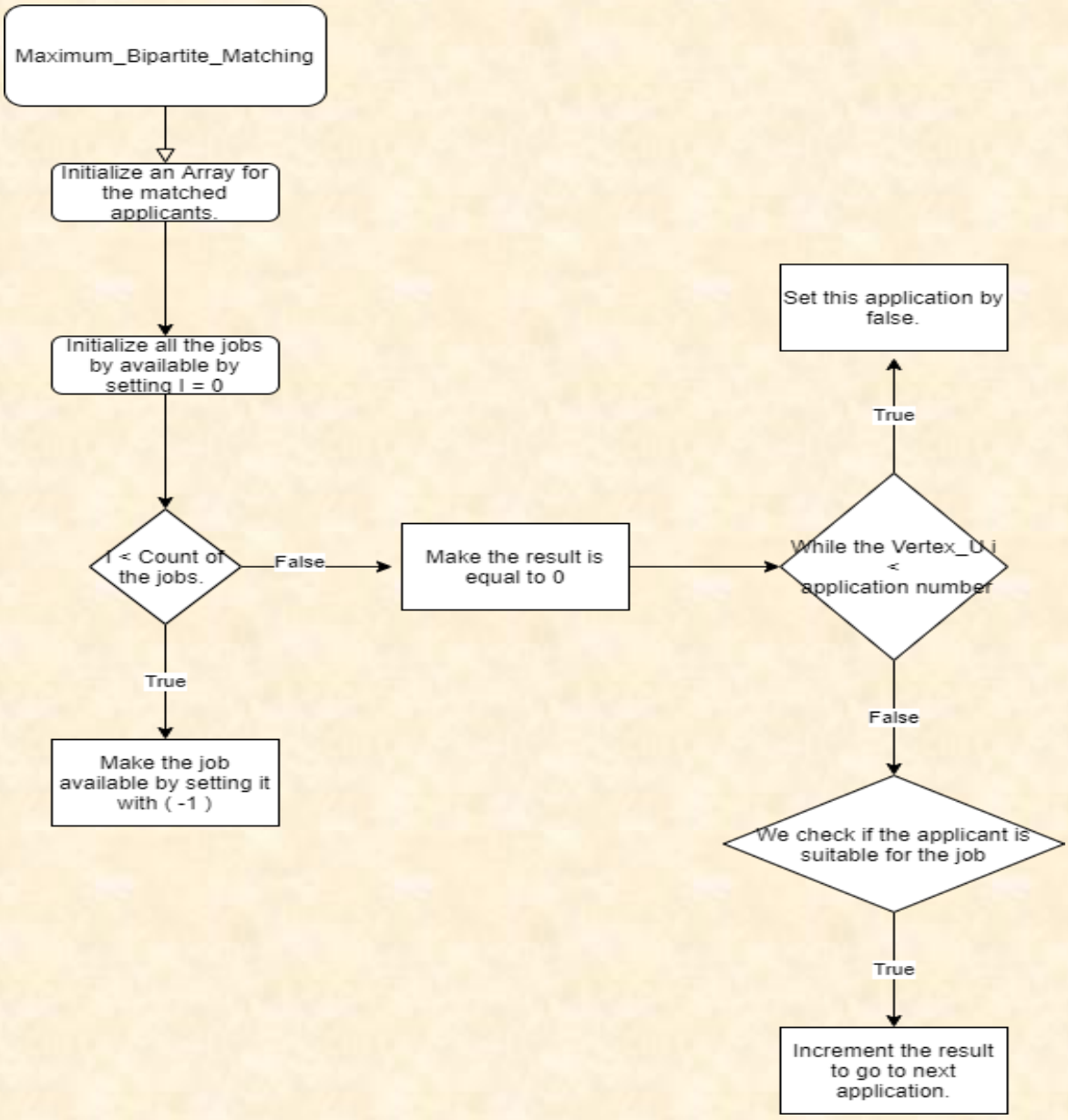

Fig (8): The Maximum Bipartite Matching Function.

[9] Time Complexity:

Here our variables are the number of Jobs and the number of Applicants which refers to the nodes at each side of the network flow in the two adjacent sets of nodes.

The complexity of the Depth First Search algorithm in the run time is which is **O (E + V)** where that the E is the number of edges in the bipartite graph and the V is the number of vertices the bipartite graph has.

In the Bipartite graph the **edge** will be equal to **(Jobs * Applicants)** and the vertex will be equal to **(Jobs + Applicants)**. After running DFS the **Time Complexity** will be equal to **O ((Jobs * Applicants) ^2).**

[10] Conclusion:

To sum up all this document importance we can say that the Maximum Bipartite Matching is one of the important algorithms that has a lot of uses in many fields such as: Machine Learning, Image objects matching, Pattern Recognition and Data Mining. This algorithm uses many other algorithms to complete its duty such as Augmenting path algorithm, Greedy Algorithm and Ford Fulkerson Maximum flow of the network.

Here we used the **Ford Fulkerson** maximum flow of a network to get the most suitable applicant for the jobs that he submitted at.

[11] References:

1 – Geeks for Geeks:

https://www.geeksforgeeks.org/maximum-bipartite-matching/

2 – Brian's paper:

https://www3.nd.edu/~kogge/courses/cse60742-Fall2018/Public/StudentWork/KernelPaperFinal/Page-Final-Kernel.pdf

3 – Princeton.edu:

https://www.cs.princeton.edu/courses/archive/fall04/cos226/lectures/maxflow.4up.pdf

Thank you,