The Maximum Bipartite Matching Problem

Olivia Anastassov and Nikki Kirk
CSC235 - Applied Algorithms

Problem Description:

The Maximum Bipartite Matching (MBM) problem is used to find the maximum number of matches between a set of "applicants" and "jobs" such that no two applicants are assigned the same job. Since each applicant has a preference of jobs, there can be multiple matchings available. However, a *maximum matching* occurs when the largest number of applicants are matched to their preferred jobs.

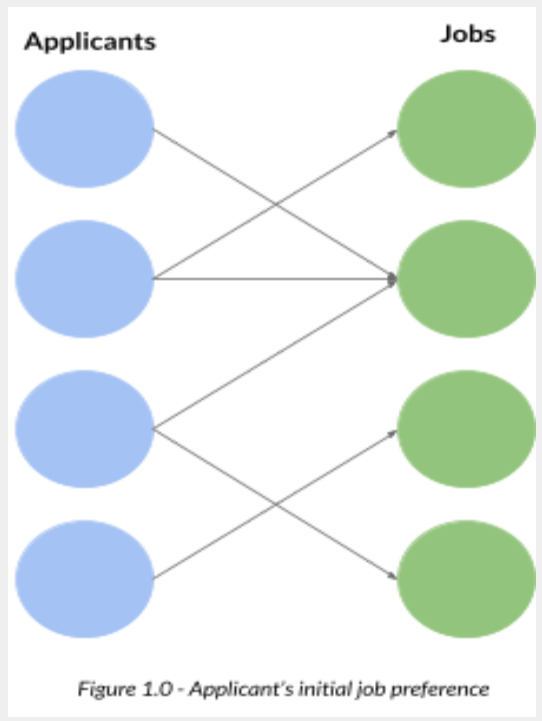

Figure 1.0 - Applicant's initial job preference

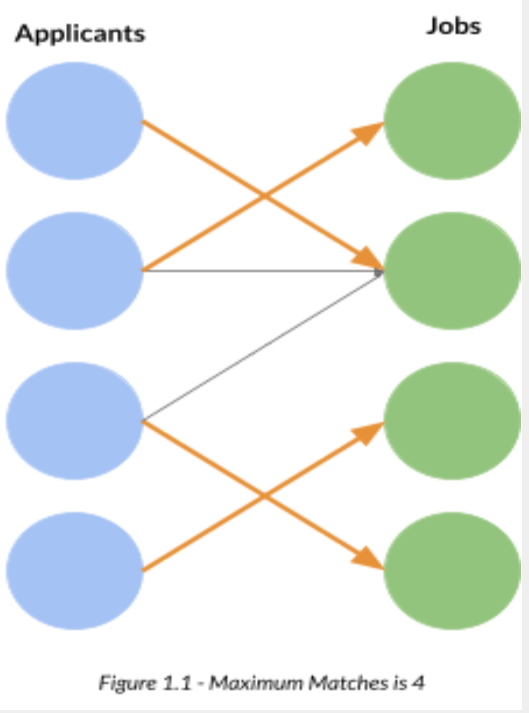

Figure 1.1 - Maximum Matches is 4

In Figure 1.0, we see the initial Bipartite graph representing a set of applicants and their preference of jobs. Each applicant can have more than one job or even no job at all. The point of the problem is to get as many applicants as possible paired up with jobs. Figure 1.1 demonstrates the correct pattern of matches in order to achieve a maximum matching. Here, the maximum number of matches is four, as shown by the orange edges. We can also see that no two applicants are assigned to the same job.

Working Solution:

While there are many ways to find the maximum bipartite matching, we chose to use the Ford-Fulkerson algorithm. The Ford-Fulkerson algorithm uses a version of the breadth-first search (BFS) algorithm to see if there is a possible path between two vertices. After it has identified if there is a path or not it goes on to identify what the maximum matching of the graph would be. To be able to run the Ford-Fulkerson algorithm on the bipartite graph we needed to connect all the vertices on the left to a “source” vertex and all the vertices on the right to a “sink” vertex.

We used BFS to return if there is a possible path from our applicant node to our job node. This path represents an edge between the two.

Ford-Fulkerson is used to find the max-flow, or in this case, the maximum number of matches in the graph. This algorithm recursively calls the boolean function BFS for each applicant to confirm if there is a possible path.

Documentation:

We chose to document our program using Javadocs. For a detailed description of the program and all of its contents, please refer to the following path located inside of the project folder:

> *FinalProject/javadocs/index.html*

Input/Output:
Our program was developed to take three arguments through the command line:

Argument 0 is the file directory of the input file.
Argument 1 is the file directory of the graph containing coordinate information
Argument 2 is the file directory where the program will output the Mathematica file

The graph input files (*FinalProject/Data/MBMTxt/)* are formatted in such a way that allows us to first read the number of vertices that represent the “applicants'' and then read the number of vertices that represent the “jobs”. This way, we can always ensure that the input graph will be a bipartite graph, and allows for easier formatting within certain methods of the program.

The graphic files (*FinalProject/Data/MBMGraphicsTxt/)* that contain the graphical information are configured in a similar format, except they also contain an integer representing the exact number of edges in the graph, as well as a list of coordinates. The list of coordinates are important because they give the program exact formatting details for the vertices. With this, we can separate the applicant vertices from the job vertices, and give the appearance of them being two distinct sets. For the output, we chose to have an adjacency matrix as this made everything look a bit cleaner and it was a little bit simpler to implement than if we had used an adjacency list. The program also

is designed to output the graph with the maximum matched edges. We accomplished this by utilizing the Java Graphics library. Below are some examples of the graphics produced by our program, where the orange edges represent a match between the two sets of vertices. The maximum number of matches is the sum of all of the orange edges in the graph. While there are a few combinations of edges that would achieve a maximum matching, our program will always return the last possible matching after searching through the adjacency matrix for possible paths.

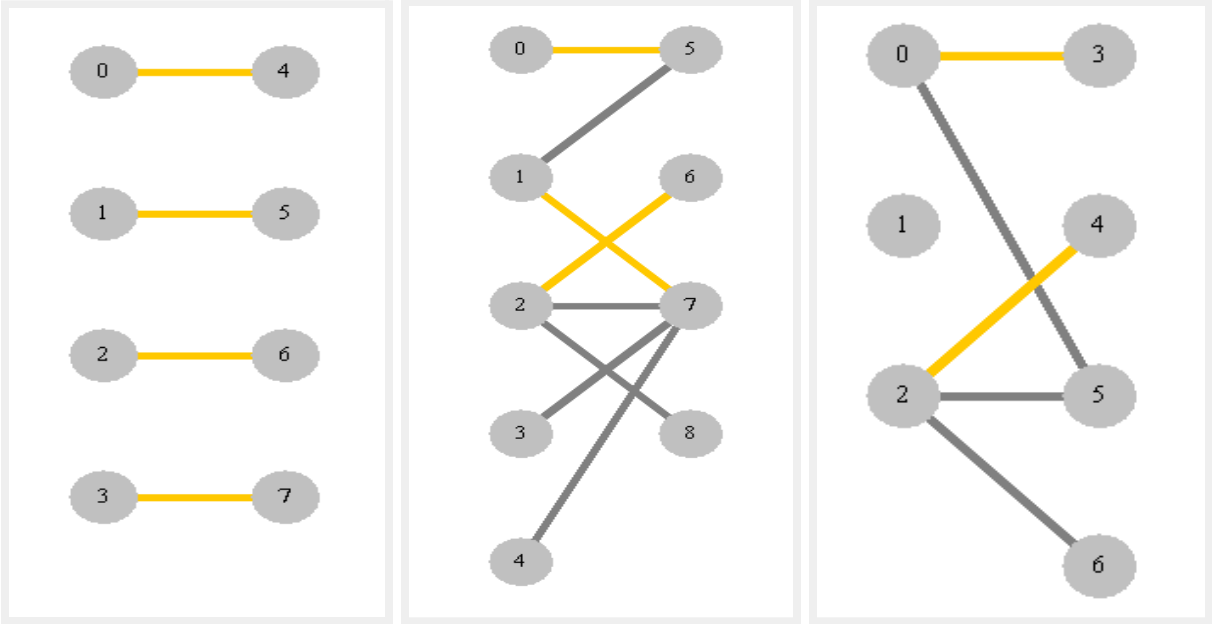

Figure 2.0 - Program output using Java Graphics

References:

GeeksForGeeks:
https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/
https://www.geeksforgeeks.org/maximum-bipartite-matching/

CLRS:
Cormen, Thomas H., et al. *Introduction to Algorithms*. Third ed., Mit Press, 2009.

Additional Authors:
Code contributions from Professor Ileana Streinu