# An Algorithm for Finding a Perfect Matching

## Introduction

The goal of this paper is to understand and examine the algorithm of finding a perfect matching of a bipartite graph introduced by the textbook*, Discrete Mathematics, Elementary and Beyond, chapter 10.4*. A *perfect matching* is a set of edges such that every node is incident with exactly one of these edges. The computation of finding a perfect matching will become astronomically heavy with a large set of nodes. Therefore, we have to implement necessary algorithms to reduce the computation as much as possible. As discussed in the book, in order to reduce the computation to a reasonable level, we could apply the perfect matching algorithm, which is combination of *greedy matching algorithm* and backtracking algorithm by rigorously finding the *augmenting paths*. We will discuss in details how the perfect matching algorithm works in the next section.

## Algorithm Implementation

First and foremost, we have to check our bipartite graph to make sure the set of nodes on both sides are equal to each other ( | left set | = | right set | ).

After making sure that it is possible to construct a perfect matching with such pairs of nodes, we need to apply the *greedy matching algorithm.* Basically, we could start to construct a perfect matching in our bipartite graph with an empty set, and add in the matching one by one. We could select two nodes that are connected and mark the edge between them. This process could go on until you end up with nodes that do not have direct link to each other. This is called *greedy* is because you are only focusing on constructing the matching with the nodes you have on hands, without looking further into any future consequences.

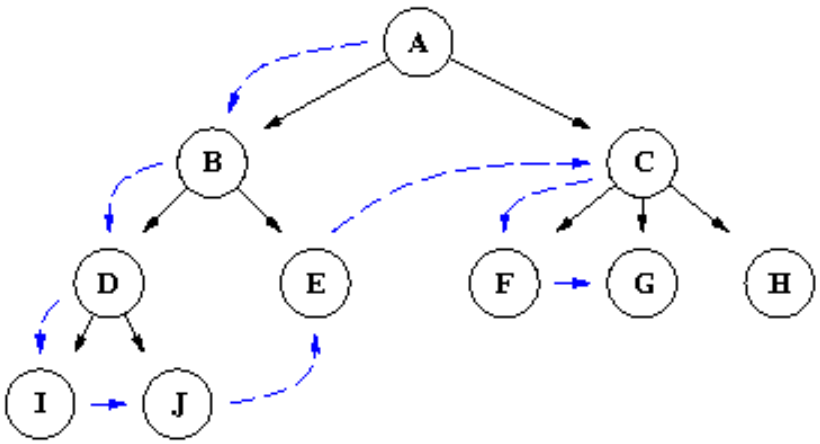
The programming language I apply for this task is Java. In GreedyMatch.java. you could see how the greedy matching algorithm gets implemented. Essentially, this class contains methods that will search for the matching greedily and thus allow us to come up with a list of unmatched nodes.

Although the greedy matching algorithm cannot guarantee a perfect matching (unless you are lucky), it provides enough benefits for us since from now on we only need to come up with ways to connect those unmatched nodes. One easy way suggested by

the book is that if the matching is not perfect, then we could start to increase the size of our matching set ***M*** by “backtracking”, which means we could delete some edges and replace them with some more edges that will connect more nodes. Basically, we are looking for a path with a particular pattern such that it contains an odd number of edges and every second of its edges belongs to ***M***. Such paths are called *augmenting paths.* Since we have noticed some interesting traits about this type of path, for instance, we could delete the edges that the augmenting path and the matching set ***M*** have in common and replace those deleted edges in ***M*** with the edges the augmenting path uniquely processes. By doing so each time, we could match a pair of unmatched nodes together without separating any previous pairs. This process of finding an augmenting path could keep on going until we have reached the nodes where no possible augmenting paths are between them.

As there is a proof in the book, once for all, we could be sure that if we have exhausted our ways of finding all possible augmenting paths and there are still unmatched nodes left, we could feel assured that the bipartite graph would never have a perfect matching. Since the main purpose of this paper is to investigate how to implement this algorithm, rigorous proof will not be discussed.

I have split up this task into two halves. In the first half, AllPaths.java. I applied Depth First Search (DFS) to find all possible paths between two unmatched nodes with “brute force”, by which I mean that you search all possible edges between the source node and its adjacent nodes until you have reached your sink node. Here is a graphic

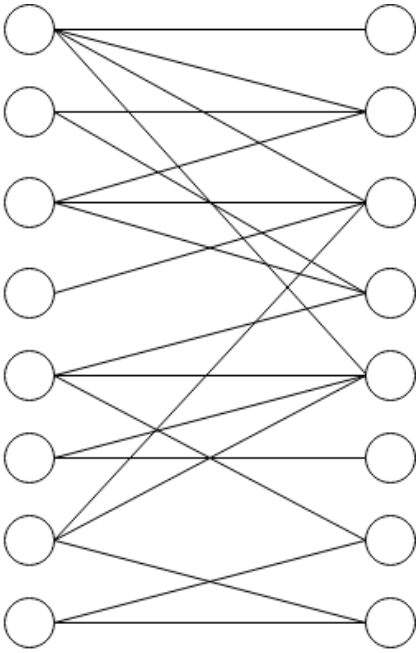



representation of DFS:
After we have looked through all possible paths between two unmatched nodes, we could start to “select and discard”. As introduced previously, every augmenting path has an odd number of edges and every second of its edges could be also found in the

matching set ***M***, found by the greedy algorithm. Based on these rules, we could easily eliminate invalid paths. In PerfectMatching.java, you could see that the list which I used to store the matching pairs gets constantly updated whenever an augmenting path is found.

## Testing and Results

To check whether or not the algorithm works, I have included a test file, TestPerfectMatching.java. In the example graph of the test code, I am basically testing a graph like this,

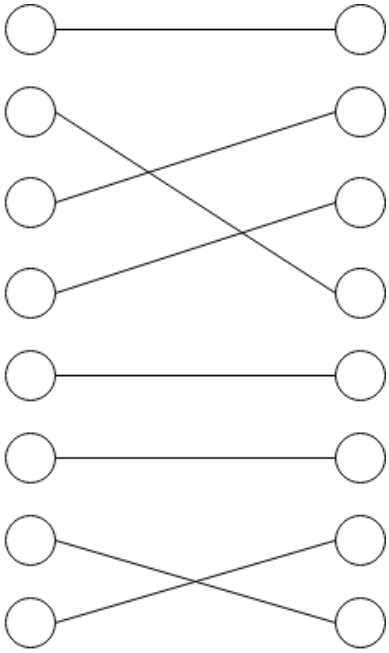


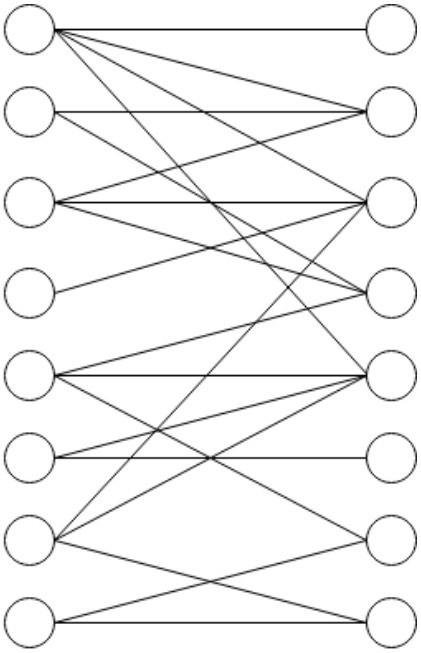
After running the test, I was informed that a perfect matching exists:

Perfectly Matched Nodes:
|0| -> |8|
|6| -> |15|

|7| -> |14|
|3| -> |10|
|2| -> |9|
|1| -> |11|
|4| -> |12|
|5| -> |13|

we could present this relationship as a graph as well, and observe that the matching is indeed perfect.





## Reference


Lovász, L., Pelikán, J., & Vesztergombi, K. L. (2006). Discrete mathematics: elementary and beyond. Beijing: Tsinghua University Press.

Problem Solving and Search. (n.d.). Retrieved March 15, 2017, from http://casimpkinsjr.radiantdolphinpress.com/pages/cogsci109/pages/search_reading/search.html