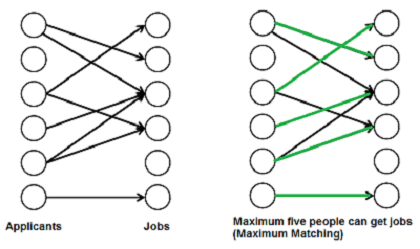
Maximum Bipartite Matching

Last Updated: 17-05-2019

A matching in a [Bipartite Graph](https://www.geeksforgeeks.org/bipartite-graph) is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matchings for a given Bipartite Graph.

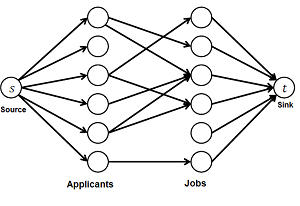
**Why do we care?**  
There are many real world problems that can be formed as Bipartite Matching. For example, consider the following problem:  
*There are M job applicants and N jobs. Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. Find an assignment of jobs to applicants in such that as many applicants as possible get jobs.*

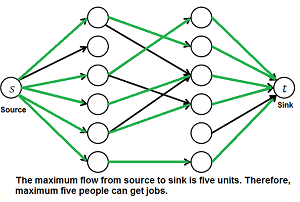
[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/maximum_matching1.png)

We strongly recommend to read the following post first.

[Ford-Fulkerson Algorithm for Maximum Flow Problem](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/)

**Maximum Bipartite Matching and Max Flow Problem**  
**M**aximum **B**ipartite **M**atching (**MBP**) problem can be solved by converting it into a flow network (See [this](http://www.youtube.com/watch?v=NlQqmEXuiC8)video to know how did we arrive this conclusion). Following are the steps.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/maximum_matching2.png)***1) Build a Flow Network***  
There must be a source and sink in a flow network. So we add a source and add edges from source to all applicants. Similarly, add edges from all jobs to sink. The capacity of every edge is marked as 1 unit.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/maximum_matching21.png) ***2) Find the maximum flow.***  
We use [Ford-Fulkerson algorithm](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/) to find the maximum flow in the flow network built in step 1. The maximum flow is actually the MBP we are looking for.

**How to implement the above approach?**  
Let us first define input and output forms. Input is in the form of [Edmonds matrix](http://en.wikipedia.org/wiki/Edmonds_matrix) which is a 2D array ‘bpGraph[M][N]’ with M rows (for M job applicants) and N columns (for N jobs). The value bpGraph[i][j] is 1 if i’th applicant is interested in j’th job, otherwise 0.  
Output is number maximum number of people that can get jobs.

A simple way to implement this is to create a matrix that represents [adjacency matrix representation](https://www.geeksforgeeks.org/graph-and-its-representations/)of a directed graph with M+N+2 vertices. Call the [fordFulkerson()](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/) for the matrix. This implementation requires O((M+N)\*(M+N)) extra space.

Extra space can be be reduced and code can be simplified using the fact that the graph is bipartite and capacity of every edge is either 0 or 1. The idea is to use DFS traversal to find a job for an applicant (similar to augmenting path in Ford-Fulkerson). We call bpm() for every applicant, bpm() is the DFS based function that tries all possibilities to assign a job to the applicant.

In bpm(), we one by one try all jobs that an applicant ‘u’ is interested in until we find a job, or all jobs are tried without luck. For every job we try, we do following.  
If a job is not assigned to anybody, we simply assign it to the applicant and return true. If a job is assigned to somebody else say x, then we recursively check whether x can be assigned some other job. To make sure that x doesn’t get the same job again, we mark the job ‘v’ as seen before we make recursive call for x. If x can get other job, we change the applicant for job ‘v’ and return true. We use an array maxR[0..N-1] that stores the applicants assigned to different jobs.

If bmp() returns true, then it means that there is an augmenting path in flow network and 1 unit of flow is added to the result in maxBPM().

Channel Assignment Problem

Last Updated: 26-08-2019

There are M transmitter and N receiver stations. Given a matrix that keeps track of the number of packets to be transmitted from a given transmitter to a receiver. If the (i; j)-th entry of the matrix is k, it means at that time the station i has k packets for transmission to station j.  
During a time slot, a transmitter can send only one packet and a receiver can receive only one packet. Find the channel assignments so that maximum number of packets are transferred from transmitters to receivers during the next time slot.  
**Example:**

0 2 0

3 0 1

2 4 0

The above is the input format. We call the above matrix M. Each value M[i; j] represents the number of packets Transmitter ‘i’ has to send to Receiver ‘j’. The output should be:

The number of maximum packets sent in the time slot is 3

T1 -> R2

T2 -> R3

T3 -> R1

Note that the maximum number of packets that can be transferred in any slot is min(M, N).

**Algorithm:**  
The channel assignment problem between sender and receiver can be easily transformed into Maximum Bipartite Matching(MBP) problem that can be solved by converting it into a flow network.

**Step 1: Build a Flow Network**  
There must be a source and sink in a flow network. So we add a dummy source and add edges from source to all senders. Similarly, add edges from all receivers to dummy sink. The capacity of all added edges is marked as 1 unit.

**Step 2: Find the maximum flow.**  
We use [Ford-Fulkerson algorithm](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/) to find the maximum flow in the flow network built in step 1. The maximum flow is actually the maximum number of packets that can be transmitted without bandwidth interference in a time slot.

**Implementation:**  
Let us first define input and output forms. Input is in the form of Edmonds matrix which is a 2D array ‘table[M][N]‘ with M rows (for M senders) and N columns (for N receivers). The value table[i][j] is the number of packets that has to be sent from transmitter ‘i’ to receiver ‘j’. Output is the maximum number of packets that can be transmitted without bandwidth interference in a time slot.  
A simple way to implement this is to create a matrix that represents adjacency matrix representation of a directed graph with M+N+2 vertices. Call the fordFulkerson() for the matrix. This implementation requires O((M+N)\*(M+N)) extra space.  
Extra space can be reduced and code can be simplified using the fact that the graph is bipartite. The idea is to use DFS traversal to find a receiver for a transmitter (similar to augmenting path in Ford-Fulkerson). We call bpm() for every applicant, bpm() is the DFS based function that tries all possibilities to assign a receiver to the sender. In bpm(), we one by one try all receivers that a sender ‘u’ is interested in until we find a receiver, or all receivers are tried without luck.  
For every receiver we try, we do following:  
If a receiver is not assigned to anybody, we simply assign it to the sender and return true. If a receiver is assigned to somebody else say x, then we recursively check whether x can be assigned some other receiver. To make sure that x doesn’t get the same receiver again, we mark the receiver ‘v’ as seen before we make recursive call for x. If x can get other receiver, we change the sender for receiver ‘v’ and return true. We use an array maxR[0..N-1] that stores the senders assigned to different receivers.  
If bmp() returns true, then it means that there is an augmenting path in flow network and 1 unit of flow is added to the result in maxBPM().

**Time and space complexity analysis:**  
In case of bipartite matching problem, F ? |V| since there can be only |V| possible edges coming out from source node. So the total running time is O(m’n) = O((m + n)n). The space complexity is also substantially reduces from O ((M+N)\*(M+N)) to just a single dimensional array of size M thus storing the mapping between M and N.