Lecture 7: Network Flow

subject	TDT4121
topics	Ford-Fulkerson Algorithm, Maximum-Flow Problem, Minimum Cut, The Bipartite Matching Problem
created	@November 14, 2022 11:35 AM

Chapter 7 - Network Flow

> In graph theory, a flow network is a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge. While the initial motivation for network flow problems comes from the issue of traffic in a network, we will see that they have applications in a surprisingly diverse set of areas and lead to efficient algorithms not just for Bipartite Matching, but for a host of other problems as well.

Goals 🏁

The student should be able to:

- Understand the *Maximum-Flow Problem* ✅
- Understand the *Ford-Fulkerson Algorithm* ✅
- Understand and solve the problem of *Minimum Cut* in a graph. ✅
- Use *Network Flow* to solve *The Bipartite Matching Problem* ✅
- Identify problems that can be solved using network flow. ✅

Maximum-Flow Problem

Flow can be visualised by, for example, a pipe system to deliver water in a city, or as a network with different capacities on the various cables. Maximum flow is how much actually flows through the network. There may be edges with very small capacity that prevent flow, so regardless of whether all the other edges have large capacity, the maximum flow will depend on the smallest edge if there is no way around it. Maximum flow is reached if and only if the residual network has no more flow-increasing paths.

Network Flow

A flow network is a directed graph, where all edges have a non-negative capacity. In addition, there is a requirement that if there is an edge between u and v, there is no edge opposite from v to u. A flow network has a source, s, and a drain, t. The source can be seen as the start node, and the drain as end node. The graph is not split, so for all v there is a path $s \rightsquigarrow v \rightsquigarrow t$. All edges except s have an edge in. A node, apart from the source and sink, has as much flow in as it has flow out.

A flow network can have many sources and sinks. To eliminate the problem, we create a super-source and/or a super-sink. The super source has an edge to each of the sources, and we set the capacity on those edges as infinite. In the same way, we make the super-sink. An edge from each of the gullies, setting the capacity to infinity. Then there is a new network, with only one source and one drain, and we can solve the problem as usual

Residual Network

The residual network is what is left of capacity. That is

$$c_f(u,v) = c(u,v) - f(u,v)$$

Keeping an eye on the residual network is useful. If one sends 1000 litres of water from u to v, and 300 litres from v to u, it is enough to send 700 litres from u to v to have the same result.

Augmenting Path

An augmenting path is a path from the start to a node, which increases total flow in the network. An augmenting path is a simple path from s to t in the residual network. By definition of the residual network, we can increase $f(u,v)$ in an augmenting path with $c_f(u,v)$ without going over the limitations.

Ford-Fulkerson’s Algorithm

In each iteration of Ford-Fulkerson, we find a flow-increasing path p, and use p to modify f. Note that Ford-Fulkerson does not specify how this is implemented.

```
#FORD-FULKERSON-METHOD(G, s, t)
initialise flow to 0
while there exists an augmenting path p in the residual network G_f
  augment flow f alog p
return f
```

Best case	Average case	Worst case
$O(VE^2)$		$O(E_f)$

Edmonds-Karp

Edmonds-Karp is a Ford-Fulkerson method where BFS is used to find the augmenting path. This gives a runtime of $O(VE^2)$

Minimum Cut

A cut in a flow network is to divide the graph into two, S and T, and look at the flow through the cut. That is

$$f(S,T) = \sum_{s \in S} \sum_{t \in T} f(u,v) - \sum_{s \in S} \sum_{t \in T} f(v,u)$$

The number of total possible cuts in a network with n nodes are:

$$|C| = 2^{n-2}$$

Of all the possible cuts, we want to look at the cut that has the least flow, as this is the "bottleneck" in the network.

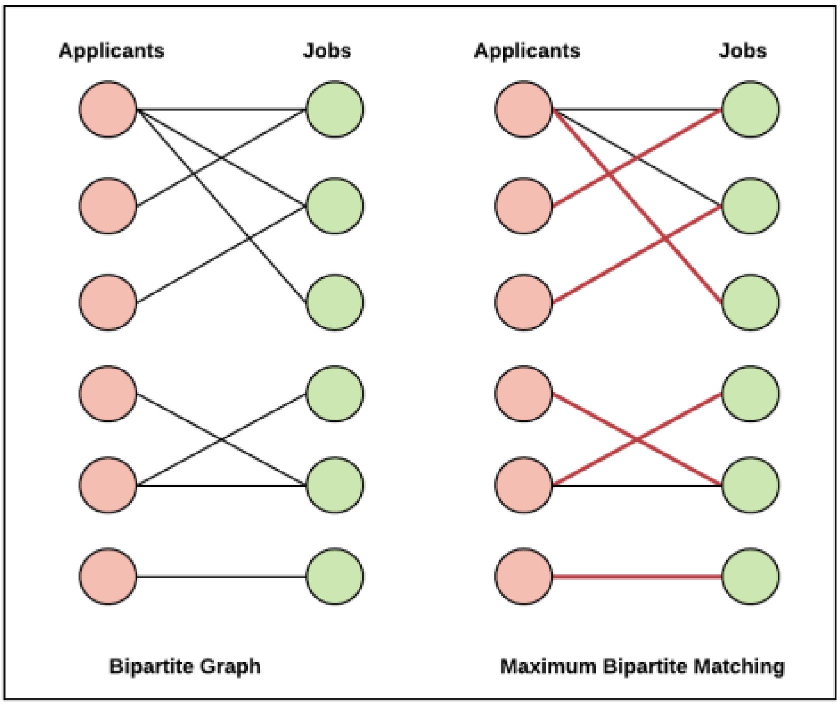
The Bipartite Matching Problem

> An undirected graph $G = (V, E)$ is bipartite if the nodes can be coloured red or blue such that every edge has one red and one blue end.

A matching in a **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.

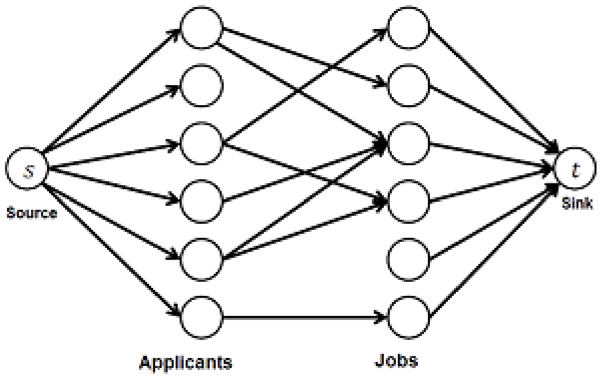
There are many real world problems that can be formed as Bipartite Matching. For example, consider the following problem:

"*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.*"

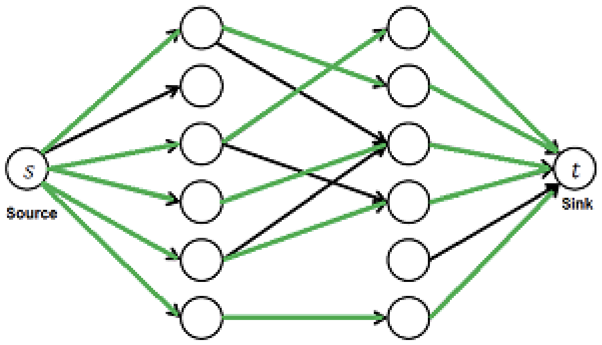


Maximum Bipartite Matching and Max Flow Problem

Maximum **B**ipartite **M**atching (**MBP**) problem can be solved by converting it into a flow network (See **this** video to know how did we arrive this conclusion). Following are the steps.



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.



The maximum flow from source to sink is five units. Therefore, maximum five people can get jobs.

2. ***Find the maximum flow:*** We can use **Ford-Fulkerson algorithm** 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 this approach

First, we define input and output forms. Input is in the form of **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.

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()`.

Identifying problems that can be solved using network flow

Network Flow is part of the graph theory toolbox and it is used **to model problems such as transportation networks, scheduling/planning, and matching of resources** to name a few.

There exist a number of flavors of network flows that differ by their objective. One of the most used ones is **maximum flow**. This algorithm finds the maximum feasible flow through a network respecting the capacities of each edge. This can be useful in a number of scenarios as the water system in our homes, traffic in a city, and as we will see, to seemingly unrelated problems.