HW3 report of max flow

Zhuoyang Chen 12/07/2019

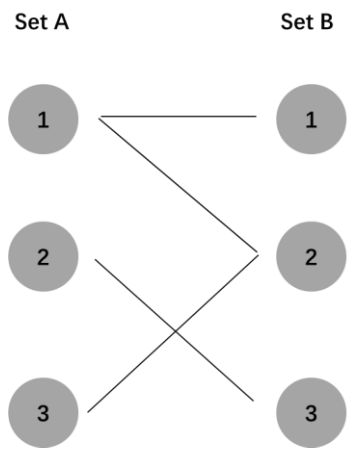
There are three files involved: DFS.py, Ford_Fulkerson.py and HW3.py

DFS.py is simply the Dmitry's code but for convenience to import, I comment the example session and the *dfs_path* function need to be redefined later to implement the algorithm.
Fold_Fulkerson.py is for problem1 to implement the Fold-Fulkerson algorithm. Here I make a little change to Dmitry's code and define addition functions such as those to find the miniflow and update the residuals network. An example on the lecture slides are given to test the algorithm. And also for easier import, I comment the example session but it can be visualized by uncomment and rerun it.
HW3.py is for problem2 to utilize Ford-Fulkerson algorithm to solve the equilibrium interaction problem. The input is included in the code.

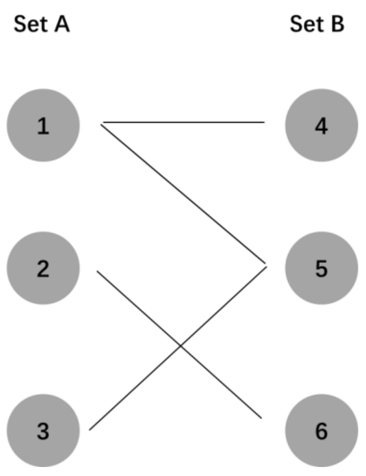
Some necessary comments and explanation have been included in the code files.

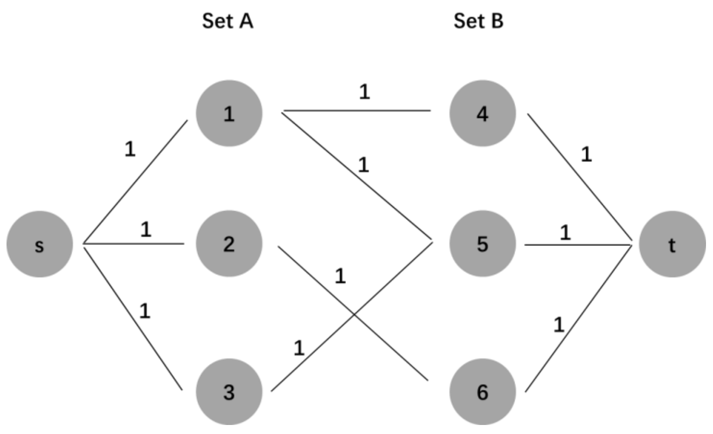
How I logically develop the program
Q1:
Dmitry's code can just return a path from source to sink, because each time reach **t**, **t** could be added to the set **visitied**, prevent the *dfs_path()* to "yield" a new path to t, so the function could just return one path and that why I change "yield" to "return".
Next, since the *dfs_path()* doesn't keep track of the residuals at all and although the class **Graph** uses class **Edge** but actually it doesn't contain any "start, end and capacity" information about edges, an additional residual network should be defined as well. That's why I create a list of class **Edge** to store the direction and capacity of the network.
With the network, each time use *dfs_path()* to find a path, we find the smallest available capacity within that path, and subtract it in every edge involved in that path. Every time a miniflow is found, it is added to a variable called **maxflow**. Then repeat the find path and update network until no paths are found anymore. The output is the **maxflow**.

Q2:
While the example given is a little bit confusing, it could be written as:

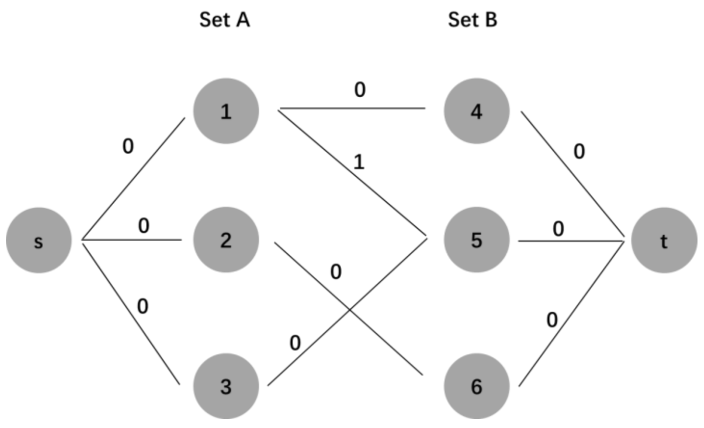


To be convenient to implement the algorithm, I change the notations in set B which are identical to set A with an increment of n (in this case, 3). And then add a sink s and a destination **t**, and capacity 1 in every edge, with direction from **s** to **t**, from set A to set B.





To find the connect, implement Ford-Fulkerson algorithm the the newly defined network until it reaches the maxflow. An output of “3” indicates that all the nodes have paired to each other. Next we track the ultimate status of residuals and see that the path that involves connection should be all 0 capacity of every edge.



A natural idea is to traverse the network and find all the path with 0 capacity. The output should be something like [s, 1, 4, t], [s, 2, 6, t], [s, 3, 5, t]. Extract the node involved and it should give the connect of 1-4, 2-6 and 3-5, which is

```
Output:
1 1
3 2
2 3
```

as the final answer. The traverse algorithm are included in the HW3.py and it yields all the paths but final output format is not shown.

But there is also a more convenient way to find the connection. Since all the capacity is 1 because there are only one to one pairing connections in two sets, no matter how many nodes in set B that a node in set A can possibly paired to, at equilibrium it can just pair one in set B. To use this property, use *get_edges()* function in Dmitry's code to return possible connection of node 1, node 2 and node 3 in set A, and check the edge corresponding to each connection if its capacity is 0.

get_edges("1"): [4,5]
check capacity of edge(1, 4), the capacity is 0, the 1-4 is the correct connection.
check capacity of edge(1, 5), the capacity is 1, so 1-5 is not the final connection.
get_edges("2")
get_edges("3")

At last, the notation of 4, 5, 6 will be changed back to 1, 2, 3, respectively.