```
import numpy as np
from operator import itemgetter


# Forward star representation [T, H, C, U]
# [0, 0, 0] is a dummy node.
FStar = np.array([[0, 0, 0, 0],
                  [1, 2, 1, 3],
                  [1, 3, 1, 3],
                  [1, 4, 1, 2],
                  [2, 5, 1, 4],
                  [3, 4, 1, 1],
                  [3, 6, 1, 2],
                  [4, 2, 1, 1],
                  [4, 6, 1, 2],
                  [5, 4, 1, 1],
                  [5, 6, 1, 1]]).reshape(-1, 4)

def adjacency_list(FStar):
    '''
    Outarc adjacency list
    '''
    A = dict()
    # Excludes the dummy node 0.
    N = np.sort(np.unique(FStar[:,0:2].reshape(-1,)))[1:]
    for i in N:
        A[i] = np.sort(FStar[:,1][np.argwhere(FStar[:,0]==i)].reshape(-1,)).tolist()
    return A

def out_adjacency_list(FStar):
    '''
    Outarc adjacency list along with flow. The initial flow is zero.
    '''
    A = dict()
    # Excludes the dummy node 0.
    N = np.sort(np.unique(FStar[:,0:2].reshape(-1,)))[1:]
    for i in N:
        nodes = np.sort(FStar[:,1][np.argwhere(FStar[:,0]==i)].reshape(-1,)).tolist()
        # [Head node, x_ij]
        A[i] = [[j, 0] for j in nodes]
    return A
def in_adjacency_list(FStar):
    '''
    Inarc adjacency list along with flow. The initial flow is zero.
    '''
    AI = dict()
    # Excludes the dummy node 0.
    N = np.sort(np.unique(FStar[:,0:2].reshape(-1,)))[1:]
    for j in N:
        nodes = np.sort(FStar[:,0][np.argwhere(FStar[:,1]==j)].reshape(-1,)).tolist()
        #[Tail node, x_ij]
        AI[j] = [[i, 0] for i in nodes]
    return AI

def complete_adjacency_list(AO, AI):
    '''
    Generates complete adjacency list assuming if x_ij exists, x_ji exists.
    '''
    for key, values in AO.items():
        for node, x in AI[key]:
            # Reverse arc is not there
            flag = False
            for j, x in values:
                if j == node:
                    #Reverse arc is there
                    flag = True
                    break
            if not flag:
                AO[key].append([node, 0])
```

```
        AO[key].sort()
    for key, values in AI.items():
        for node, x in AO[key]:
            # Reverse arc is not there
            flag = False
            for j, x in values:
                if j == node:
                    #Reverse arc is there
                    flag = True
                    break
            if not flag:
                AI[key].append([node, 0])
        AI[key].sort()
    return AO, AI

def bfs(FStar, s):
    '''
    Breadth First Search using Forward Star representation.
    FStar - Forward Star representation
    s - source node
    '''
    A = adjacency_list(FStar)
    # Number of nodes
    n = len(A)
    # Number of arcs
    m = FStar.shape[0] - 1
    # Tails of arcs
    T = FStar[:,0]
    H = FStar[:,1]
    # Heads of arcs
    nodesToTraverse = list([s])
    mark = np.zeros(n+1)
    next_= 1
    mark[s] = 1
    pred = np.zeros(n+1)
    pred[0] = np.inf
    pred[s] = 0
    order = np.zeros(n+1)
    order[s] = next_
    while len(nodesToTraverse) != 0:
        i = nodesToTraverse[0]
        # Unmarked j nodes of (i,j) arcs
        jj = [j for j in A[i] if mark[j]==0]
        # Lexicographical ordering
        jj.sort()
        if len(jj) !=0:
            for j in jj:
                mark[j] = 1
                pred[j] = i
                next_ += 1
                order[j] = next_
                nodesToTraverse.append(j)
        else:
            nodesToTraverse.pop(0)
    '''
    index = np.arange(1, n+1, 1).reshape(-1,1)
    print "Order\n[Index, order]"
    print np.hstack((index, np.array(order[1:]).reshape(-1,1))).astype(int)
    print "Predecessor\n[Index, pred]"
    print np.hstack((index, np.array(pred[1:]).reshape(-1,1))).astype(int)
    print "Used lexicographical ordering to choose a sibling node to traverse first."
    '''
    return order, pred

def depth(pred):
    """
    Given a pred vector, returns depths of nodes in BFS tree as distance labels
    """
    n = len(pred) - 1
```

```
    d = np.ndarray((n+1,), dtype=int)
    root = np.argwhere(pred==0)[0][0]
    d[0] = n+1
    for i in np.arange(n):
        #level
        l = 0
        node = i+1
        while node != root:
            p = int(pred[node])
            node = p
            l += 1
        d[i+1] = l
    return d

def push_or_relabel(RFStar, A, AO, AI, e, d, i):
    '''
    Pushes flow on an admissible arc or relabels i.
    '''
    j = None
    relabeled = False
    # Finding an admissible arc
    for k in A[i]:
        if d[i] == d[k] + 1:
            j=k
            break
    if j is not None:
        delta = e[i]
        saturating = False
        r_ij = RFStar[np.all(RFStar[:,0:2]==[i,j], axis=1),3]
        if delta > r_ij:
            delta = r_ij
        #Updating the flow
        for idx, data in enumerate(AO[i]):
           if data[0] == j:
               AO[i][idx][1] = int(data[1] + delta)
               break
        for idx, data in enumerate(AI[j]):
           if data[0] == i:
               AI[j][idx][1] = int(data[1] + delta)

        #Updating the residual network
        r_ij = RFStar[np.all(RFStar[:,0:2]==[i,j], axis=1),3]
        if r_ij >= delta:
            RFStar[np.all(RFStar[:,0:2]==[i,j], axis=1),3] = r_ij-delta
            if r_ij == delta:
                A[i].remove(j)
                A[i].sort()
                saturating = True
        if np.any(np.all(RFStar[:,0:2]==[j, i], axis=1)):
            r_ji = RFStar[np.all(RFStar[:,0:2]==[j,i], axis=1),3]
            RFStar[np.all(RFStar[:,0:2]==[j,i], axis=1),3] = r_ji + delta
        else:
            RFStar = np.vstack((RFStar, [j, i, 1, delta]))
            A[j].append(i)
            A[j].sort()
    else:
        #Relabeling
        relabeled = True
        min_d = np.inf
        for j in A[i]:
            r_ij = RFStar[np.all(RFStar[:,0:2]==[i, j], axis=1),3]
            if r_ij > 0 and (d[j] + 1) < min_d:
                min_d = d[j] + 1
        d[i] = min_d
        saturating = None
    return RFStar, A, AO, AI, d, saturating, relabeled

def excess(AO, AI):
    '''
```

```
    Returns an excess vector.
    Excess = Inflow - Outflow
    '''
    e = np.zeros(len(AO)+1, dtype=int)
    for i, nodes in AO.items():
        outflow = 0
        inflow = 0
        if nodes !=[]:
            tmp = np.array(AO[i]).reshape(-1,2)
            outflow = tmp[:,1].sum()
        if AI[i] != []:
            tmp = np.array(AI[i]).reshape(-1,2)
            inflow = tmp[:,1].sum()
        e[i] = inflow - outflow
    return e

def FIFOPreflow_push(FStar, s, t):
    """
    FIFO preflow-push algorithm
    ---------------------------
    """
    #Preprocess
    #=========================================================================
    order, bfs_pred = bfs(FStar[:,[1,0]], t)
    #Distance labels
    d = depth(bfs_pred)
    RFStar = FStar.copy()
    #Excludes the dummy node 0.
    A = adjacency_list(RFStar)
    AO = out_adjacency_list(RFStar)
    AI = in_adjacency_list(RFStar)
    AO, AI = complete_adjacency_list(AO, AI)

    #Saturates (s,j) arcs with preflow
    for row in np.argwhere(RFStar[:,0]==s):
        x_sj = int(RFStar[row, 3])
        j = RFStar[row,1][0]
        pos = [p for p, node in enumerate(AO[s]) if node[0]==j][0]
        AO[s][pos][1] = x_sj
        pos = [p for p, node in enumerate(AI[j]) if node[0]==s][0]
        AI[j][pos][1] = x_sj
        if np.any(np.all(RFStar[:,0:2]==[j, s], axis=1)):
            r_js = RFStar[np.all(RFStar[:,0:2]==[j,s], axis=1),3]
            RFStar[np.all(RFStar[:,0:2]==[j,s], axis=1),3] = r_js + x_sj
        else:
            RFStar = np.vstack((RFStar, [j, s, 1, x_sj]))
            A[j].append(s)
            A[j].sort()
        RFStar[row, 3] = 0
        A[s].remove(j)
    N = np.sort(np.unique(FStar[:,0:2].reshape(-1,)))[1:]
    # Number of nodes
    n = len(N)
    d[s] = n
    #=========================================================================
    e = excess(AO, AI)
    LIST = np.argwhere(e>0).reshape(-1,).tolist()
    saturating_cnt = 0
    nonsaturating_cnt = 0
    while np.any(e[1:-1]>0):
        i = LIST[0]
        RFStar, A, AO, AI, d, saturating, relabeled = push_or_relabel(RFStar, A, AO, A
I, e, d, i)
        if saturating is not None:
            if saturating:
                saturating_cnt += 1
            else:
                nonsaturating_cnt +=1
        e = excess(AO, AI)
```

```
        #Keep the active node till it is inactive
        if e[i] == 0:
           LIST.pop(0)
        new_active = [node for node in np.argwhere(e>0).reshape(-1,) if node not in LI
ST]
        for node in new_active:
            if not (node == s or node == t):
                LIST.append(node)
        '''
        #It removes relabeled node and appedns it to the end of the LIST
        #Uncomment if you want to examine the relabeled node later.
        #The numbers of saturating pushes and nonsaturating pushes is the same for thi
s example.
        #But the number of total iteration is 30 which is 2 more iterations than
        #the current node examination
        if relabeled:
            LIST.pop(0)
            LIST.append(i)
        '''

    #Lexicographical ordering by T, H
    #The last key is the primary key
    idx = np.lexsort((RFStar[:,1].reshape(-1,), RFStar[:,0].reshape(-1,)))
    RFStar = RFStar[idx, :]
    print "Original network-[T H C u]:"
    print FStar
    print "Residual network-[T H C r]:"
    print RFStar
    max_flow = 0
    x = np.ndarray(shape=(FStar.shape[0],3), dtype=int)
    x[:,[0,1]] = FStar[:, [0,1]]
    for idx, row in enumerate(FStar):
        r_ij = RFStar[np.all(RFStar[:,0:2]==[row[0], row[1]], axis=1),3]
        if row[3] >= r_ij:
            x[idx,2] = row[3] - r_ij
        else:
            x[idx,2] = 0
        if row[1] == t:
            max_flow += x[idx,2]
    print "The max flow, x-vector - [T H x]:"
    print x
    print "The max flow value:"
    print max_flow
    print "saturating push count:", saturating_cnt
    print "Nonsaturating push count:", nonsaturating_cnt
    return x, max_flow
if __name__ == "__main__":
    s=1
    t=6
    x , max_flow = FIFOPreflow_push(FStar, 1, 6)
```

```
Original network-[T H C u]:
[[0 0 0 0]
 [1 2 1 3]
 [1 3 1 3]
 [1 4 1 2]
 [2 5 1 4]
 [3 4 1 1]
 [3 6 1 2]
 [4 2 1 1]
 [4 6 1 2]
 [5 4 1 1]
 [5 6 1 1]]
Residual network-[T H C r]:
[[0 0 0 0]
 [1 2 1 2]
 [1 3 1 1]
 [1 4 1 0]
 [2 1 1 1]
 [2 4 1 0]
 [2 5 1 3]
 [3 1 1 2]
 [3 4 1 1]
 [3 6 1 0]
 [4 1 1 2]
 [4 2 1 1]
 [4 3 1 0]
 [4 5 1 0]
 [4 6 1 0]
 [5 2 1 1]
 [5 4 1 1]
 [5 6 1 0]
 [6 3 1 2]
 [6 4 1 2]
 [6 5 1 1]]
The max flow, x-vector - [T H x]:
[[0 0 0]
 [1 2 1]
 [1 3 2]
 [1 4 2]
 [2 5 1]
 [3 4 0]
 [3 6 2]
 [4 2 0]
 [4 6 2]
 [5 4 0]
 [5 6 1]]
The max flow value:
5
saturating push count: 9
Nonsaturating push count: 8
```