The Flow Deviation Algorithm for SDN-SATCOMM

1. Input:

`n_ship`: number of ships

`n_sat`: number of satellites

`connectivity`: adjacency matrix of size n_ship by n_sat

- `connectivity[i][j]=1` means ship i and satellite j are connected.
- `connectivity[i][j]=0` means ship i and satellite j are disconnected.

`Req`: request matrix of size `n_ship` by `n_ship`:

- `Req[i][j]` is the requested traffic from source ship i to destination ship j
- `Req[i][j]=0` means there is no traffic from ship i to ship j

`sat_capacities`: Satellite capacity vector of size `1` by `n_sat`

- `sat_capacities[i] is the capacity of satellite i`

`downlink_capacities`: downlink capacity matrix of size `n_sat` by `n_ship`

- `downlink_capacities[i][j] is the capacity of downlink from satellite i to ship j`
- `downlink_capacities[i][j]=0 means satellite i and ship j are disconnected.`

2. Topology builder:

Based on the input parameters and graph structure, we build a new topology that will be later fed to the FDM algorithm. Specifically, the new topology consists of `n_ship` ship nodes and `n_sat` satellite nodes. We also create a dummy node for each satellite node. Therefore, the total number of nodes is `n_ship+2*n_sat`. Since each ship-satellite link specified in the matrix `connectivity` represents a full duplex channel, we create a unidirectional uplink and downlink for each ship-satellite connection in the new topology. The uplink starts from the ship and ends at the satellite. The downlink starts from the dummy node of each satellite and ends at the ship. In addition, we connect each satellite to its dummy node. Therefore, if the total number of ship-satellite links (or the number of ones) in `connectivity` is `count_link`, the total number of links in our new topology would be `2* count_link +n_sat`.

To store the new topology, we define the following variables:

`NN=n_ship+2*n_sat`: The total number of nodes

`count_link`: The total number of ship-satellite channels (or total number of ones in `connectivity`)

`NL=2* count_link +n_sat` The total number of links

In our program, we implicitly label the nodes from 0 to `NN-1`, and the links from 0 to `NL-1`. The concrete labels are as follows:

Ship nodes	Satellite nodes	Dummy nodes
`0 ~ n_ship-1`	`n_ship ~ n_ship+n_sat-1`	`n_ship+n_sat ~` `n_ship+2*n_sat-1`

Ship to satellite uplinks	Satellite-dummy node links	Dummy node-ship downlinks
`0 ~ count_link-1`	`count_link ~` `count_link+n_sat-1`	`count_link+n_sat ~` `2*count_link+n_sat-1`

`LINK`: The maximum number of links attached to a node:

- In the program we assign `LINK=max(n_ship, n_sat)`, since any node (ships, satellites or the dummy nodes) cannot have more links than the maximum possible number of uplinks or downlinks.

`End1[NL]`: vector of size `NL` that stores the start point of each link

- Index of `End1` is the label of a particular link.
- `End1[i]` is the start point of link i.

`End2[NL]`: vector of size `NL` that stores the end point of each link

`Adj[NN][LINK]`: Link adjacency matrix of size `NN` by `LINK`.

- `Adj[i]` is a vector of links attached to node i.
- The start node of link `Adj[i][j]` is i, i.e. `End1[Adj[i][j]]=i`, and `End2[Adj[i][j]]` gives the end node following that link.
- If the number of links at node i `nl[i] < LINK, Adj[i][j]=-1 for nl[i]<= j < LINK`.

`Cap[NL]`: vector of size `NL` that stores the capacity for each link

- `Cap[i]=INFINITY` for `0 <= i <= count_link-1`
- `Cap[count_link+i]=sat_capacities[i]`, for `0 <= I <= n_sat-1`
- `Cap[i]=downlink_capacities[End1[i]][End2[i]]`, for `count_link+n_sat <= I < NL`

`Req[n_ship][n_ship]`: the same request matrix as in Input.

3. FDM Algorithm:

In the FDM algorithm, we define the following variables to keep intermediate flow assignment, link length, and shortest paths based on the link length, etc.

3.1 Important variables

`FDlen[NL]`: Current link length of each links calculated based on link capacity and flows.

- Initialize to zero.

`SPdist[NN][NN]`: Shortest path distance matrix.

- `SPdist[i][j]` is the shortest path distance from node i to node j.

`SPpred[NN][NN]`: Shortest path predecessor matrix.

- `SPpred[i][j]` is the last link in the shortest path from i to j.

`Eflow[NL]`: Current extremal flow, i.e. the flow assignment based on shortest paths formed from the current link lengths.

- `Eflow[i]` is the total amount of flow on link i.
- Initialize to zero.

`Gflow[NL]`: Global flow on each link, i.e. the superposition of all previous `Eflows`.

- Initialize to zero.

`NewCap[NL]`: vector of size `NL` that stores intermediate falsely increased link capacities.

3.2 Helper functions

We also define the following helper functions for FDM algorithm. We give a brief description of these helper functions. The detailed pseudo code is attached to the end.

```
void SetLinkLens(NL, Gflow[], Cap[], MsgLen, FDlen[])
```

- This function sets the link lengths according to the message length, link capacity and current flow on the link. The exact link length is calculated by function `DerivDelay`.

```
double DerivDelay(flow, cap, MsgLen)
```

- Recall that the objective function FDM minimizes is $\sum_{link} \frac{MsgLen \cdot flow_{link}}{cap_{link} - flow_{link}}$. The link length is defined as the marginal cost of the objective function at this link, i.e. link length is $\frac{\partial}{\partial flow} \sum_{link} \frac{MsgLen \cdot flow_{link}}{cap_{link} - flow_{link}} = \frac{MsgLen \cdot cap}{(cap - flow)^2}$. This function calculates and returns link length based on this formula.

```
void SetSP( NN, LINK, End2[], FDlen[], Adj[][], SPdist[][], SPpred[][])
```

- This function sets the shortest path between every node pair using Bellman's algorithm.

```
void Bellman(NN, LINK, root, End2[], LinkLength[], Adj[][], Pred[], Dist[])
```

- This function implements the Bellman's algorithm.

```
void LoadLinks(NN, NL, Req[][], SPPred[][], End1[], Flow[])
```

- This function load `Req[s][d]` amount of traffic on the links along the shortest path between s and d.

```
bool AdjustCaps(NL, Gflow[], Cap[], NewCap[])
```

- This function falsely increases the capacity to hold the traffic specified by `Gflow`. It returns 1 if link capacities are large enough to hold `Gflow`. Otherwise it returns some value larger than 1.

```
double CalcDelay(NL, Gflow[], Cap[], MsgLen, TotReq)
```

- This function calculates and returns the weighted sum of packet delay (or average packet delay) on all the links , defined as $\frac{1}{TotReq}\sum_{link}\frac{MsgLen\cdot flow_{link}}{cap_{link}-flow_{link}}$.

```
double LinkDelay(flow, cap, MsgLen)
```

- This function calculates and returns the packet delay on a particular link defined as $\frac{MsgLen}{cap-flow}$.

```
void Superpose(NL, Eflow[], Gflow[], Cap[], TotReq, MsgLen)
```

- This function superposes the newly discovered Eflow with previous aggregated Gflow so as to maximally reduce the current average packet delay. Specifically, Gflow is updated as $Gflow = x\cdot Eflow + (1-x)\cdot Gflow$. This function will call `FindX` to discover the optimal value of x such that average delay is minimized.

```
double FindX(NL, Gflow[], Eflow[], Cap[], TotReq, MsgLen)
```

- This function returns the optimal x using binary search, due to the fact that the objective function is convex.

```
double DelayF(x, nl, Eflow[], Gflow[], Cap[], MsgLen, TotReq)
```

- This function is called by `FindX` to calculate the average packet delay of a possible superposition $x\cdot Eflow + (1-x)\cdot Gflow$.

3.3 FDM Algorithm

Next, we present the detailed FDM algorithm.

- Line 1-13 are declaration and initialization of variables. Line 15- 32 are the main FDM algorithm. The guard condition in line 15 has two components. `Aflag` is false if current `Cap` is enough for `Gflow`, otherwise true. `CurrentDelay < PreviousDelay*(1-EPSILON)` is to check if the average delay has converged. Therefore, the loop will naturally exit if the problem is feasible (`Cap` can hold `Gflow`) and the delay has converged.

- Line 16-21 computes the link length, shortest paths between node pairs, and load the flows to links by using the defined helper functions. The newly loaded flows are kept in Eflow which represents extremal flows on the greedily discovered shortest paths with minimal marginal delay. However, Eflow might not be feasible with respect to Cap. Therefore, a superposition of Eflow and Gflow is necessary to produce feasible flows with minimal average delay. The superposition is kept in Gflow.

The new `Aflag` is computed if necessary. If `Aflag` is true and delay has converged, then it indicates the problem is infeasible, as in line 27.

If the problem is infeasible, we develop an algorithm that searches for Max-Min requests using binary search. We declare `max_request` to be the maximum value of Req, and `min_request` to be 0. Every iteration we set the requests to be `min(Req[NN][NN],mid)` as in line 44, where `mid` is the middle point of `max_request` and `min_request.` FDM algorithm is run with this new request until `max_request` and `min_request` are close enough.

Finally, the feasible flows are stored in Pflow (line 34 or 74). We print the flows on the ship-satellite links in line 81-90. These flows dictate the bandwidth allocation suggested by FDM.

```
1   //Initializing link lenghth and Gflow
2   SetLinkLens(NL, Gflow, Cap, MsgLen, FDlen)
3   SetSP(NN, LINK, End2, FDlen, Adj, SPdist, SPpred)
4   LoadLinks(NN, NL, Req, SPpred, End1, Gflow)
5
6   //Declare intermediate variables
7   declare Aflag <- AdjustCaps(NL, Gflow, Cap, NewCap),
8           CurrentDelay <- CalcDelay(NL, Gflow, NewCap, MsgLen, TotReq),
9           PreviousDelay <- INFINITY,
10          feasible <- true,
11          TotReq <- sum of all Req[i][j],
12          //Pflow is used to store final result
13          Pflow[NL]
14
15  while( Aflag or (CurrentDelay < PreviousDelay*(1-EPSILON))) {
16      SetLinkLens(NL, Gflow, NewCap, MsgLen, FDlen)
17      SetSP(NN, LINK, End2, FDlen, Adj, SPdist, SPpred)
18      LoadLinks(NN, NL, Req, SPpred, End1, Eflow)
19      PreviousDelay = CalcDelay(NL, Gflow, NewCap, MsgLen, TotReq)
20      Superpose(NL, Eflow, Gflow, NewCap, TotReq, MsgLen)
21      CurrentDelay = CalcDelay(NL, Gflow, NewCap, MsgLen, TotReq)
22
23      if(Aflag) {
24          Aflag = AdjustCaps(NL, Gflow, Cap, NewCap)
25      }
26
27      if(Aflag and (CurrentDelay >= PreviousDelay*(1-EPSILON))) {
28          print("The problem is infeasible. Now reduce the request.\n")
29          feasible = false
30          break
31      }
32  }
33  if(feasible) {
34      Pflow=Gflow
```

```
}
else {
    //Run Max-Min algorithm, binary search for feasible Req

    declare max_request <- max of Req[NN][NN], min_request <- 0, mid

    while (max_request - min_request > EPSILON) {
        declare feasible = true,
                mid = min_request + (max_request - min_request) / 2,
                MM_Req[NN][NN] <- min(Req[NN][NN],mid),
                TotReq <- sum of MM_Req,
                PreviousDelay <- INFINITY
                Gflow[NL] <- 0

        SetLinkLens(NL, Gflow, Cap, MsgLen, FDlen)
        SetSP(NN, LINK, End2, FDlen, Adj, SPdist, SPpred)
        LoadLinks(NN, NL, MM_Req, SPpred, End1, Gflow)
        Aflag= AdjustCaps(NL, Gflow, Cap, NewCap)
        CurrentDelay = CalcDelay(NL, Gflow, NewCap, MsgLen, TotReq)

        while (Aflag || (CurrentDelay < PreviousDelay*(1 - EPSILON))) {
            SetLinkLens(NL, Gflow, NewCap, MsgLen, FDlen)
            SetSP(NN, LINK, End2, FDlen, Adj, SPdist, SPpred)
            LoadLinks(NN, NL, MM_Req, SPpred, End1, Eflow)
            PreviousDelay = CalcDelay(NL, Gflow, NewCap, MsgLen, TotReq)
            Superpose(NL, Eflow, Gflow, NewCap, TotReq, MsgLen)
            CurrentDelay = CalcDelay(NL, Gflow, NewCap, MsgLen, TotReq)

            if (Aflag) {
                Aflag = AdjustCaps(NL, Gflow, Cap, NewCap)
            }

            if (Aflag == 1 && (CurrentDelay >= PreviousDelay*(1 - EPSILON))) {
                feasible = false
                break
            }
        }
        if (feasible) {
            min_request = mid
            Pflow=Gflow
        }
        else
            max_request = mid
    }

    //count  traffic at each ship
    for each ship node u{
        sum = 0
        for each link l at u {
            if (Pflow[l] > 0) {
                print ("Usage at sat ", End2[l]-n_ship, " is ", Pflow[l])
                sum += Pflow[l]
            }
        }
        print("Total out going flow at ship ", u, " is ", sum)
    }

    //count traffic at each satellite
    for each satellite node s {
        for each link l at s {
            print("Total load at sat ", s, " is ", Pflow[l])
        }
    }
```

3.4 Pseudo code for Helper functions

```
void SetLinkLens(NL, Gflow[], Cap[], MsgLen, FDlen[]) {
    for each link from 0 to NL {
        FDlen[link] = DerivDelay(Gflow[link], Cap[link], MsgLen)
    }
}

double DerivDelay( flow, cap, MsgLen) {
    f = 1 - flow / cap;
    return((MsgLen / cap) / (f*f)
}


void SetSP( NN, LINK, End2[], FDlen[], Adj[][], SPdist[][], SPpred[][]) {
    for each node from 0 to NN{
        Bellman(NN, LINK, node, End2, FDlen, Adj, SPpred[node], SPdist[node])
    }
}

void Bellman(NN, LINK, root, End2[], LinkLength[], Adj[][], Pred[], Dist[]) {
    declare Hop[NN] <- INFINITY, Queue queue

    Dist[root] = 0
    Pred[root] = root

    queue.push(root)
    while (!queue.empty()) {
        node = queue.top()
        queue.pop();
        for i from 0 to LINK-1 {
            curlink = Adj[node][i]
            if (curlink == -1)
                break
            node2 = End2[curlink]
            d = Dist[node] + LinkLength[curlink]
            if (Dist[node2] > d) {
                Dist[node2] = d
                Pred[node2] = curlink
                Hop[node2] = Hop[node] + 1
                //setting hop limit to disable ship relays
                if (Hop[node2]<3)
                    queue.push(node2)
            }
        }

    }
}
```

```
void LoadLinks(NN, NL, Req[][], SPPred[][], End1[], Flow[]) {
    Flow <- 0

    for each src from 0 to NN-1 {
        for each dest from 0 to NN-1 {
            if (Req[src][dest] > 0) {
                m = dest
                while (m != s) {
                    link = SPpred[src][m]
                    p = End1[link]
                    Flow[link] += Req[src][dest]
                    m = p
                }
            }
        }
    }
}


bool AdjustCaps(NL, Gflow[], Cap[], NewCap[]) {
    declare factor <- 1

    for each link from 0 to NL-1 {
        factor = max(factor, (1 + DELTA)*GFlow[link] / Cap[link])
    }
    for each link from 0 to NL-1 {
        NewCap[link] = factor*Cap[link]
    }
    if (factor==1)
        return false
    else
        return true
}
double CalcDelay(NL, Gflow[], Cap[], MsgLen, TotReq) {
    declare sum <- 0

    for each link from 0 to NL-1 {
        sum = sum + GFlow[link] * LinkDelay(GFlow[link], Cap[link], MsgLen)
    }
    return sum / TotReq
}

double LinkDelay(flow, cap, MsgLen) {
    return (MsgLen / cap) / (1 - flow / cap)
}

void Superpose(NL, Eflow[], Gflow[], Cap[], TotReq, MsgLen) {
    declare x <- FindX(NL, Gflow, Eflow, Cap, TotReq, MsgLen)

    for each link from 0 to NL-1 {
        Gflow[l] = x*Eflow[l] + (1 - x)*Gflow[l]
    }
}


double FindX(NL, Gflow[], Eflow[], Cap[], TotReq, MsgLen) {
    declare xLimit, st <- 0, end <- 1
    declare Flow[NL] <- 0

    //binarily search for a feasible end point xLimit
    while (end >st+EPSILON) {
        mid = st + (end - st) / 2
        exceed = false
        for each link from 0 to NL-1 {
            Flow[link] = mid*Eflow[link] + (1 - mid)*Gflow[link]
            if (Flow[link] > Cap[link]) {
                exceed = true
                break
            }
        }
        if (exceed) {
            end = mid
        }
        else
            st = mid
    }

    xLimit = st

    declare x0 <- 0.0,           f0 <- DelayF(x0, NL, Eflow, Gflow, Cap, MsgLen, TotReq),
            x4 = xLimit,         f4 <- DelayF(x4, NL, Eflow, Gflow, Cap, MsgLen, TotReq),
            x2 = (x0 + x4) / 2, f2 <- DelayF(x2, NL, Eflow, Gflow, Cap, MsgLen, TotReq)

    while (x4 - x0 > EPSILON) {
        x1 = (x0 + x2) / 2, f1 = DelayF(x1, NL, Eflow, Gflow, Cap, MsgLen, TotReq)
        x3 = (x2 + x4) / 2, f3 = DelayF(x3, NL, Eflow, Gflow, Cap, MsgLen, TotReq)
        if ((f0 <= f1) || (f1 <= f2)) {
            x4 = x2; x2 = x1
            f4 = f2; f2 = f1
        }
        else if (f2 <= f3) {
            x0 = x1; x4 = x3
            f0 = f1; f4 = f3
        }
        else {
            x0 = x2; x2 = x3
            f0 = f2; f2 = f3
        }
    }
    if ((f0 <= f2) && (f0 <= f4)) {
        return(x0)
    }
    else if (f2 <= f4) {
        return(x2)
    }
    else {
        return (x4)
    }
}

double DelayF(x, nl, Eflow[], Gflow[], Cap[], MsgLen, TotReq) {
    declare Flow[NL]
    for each link from 0 to NL-1 {
        Flow[link] = x*Eflow[link] + (1 - x)*Gflow[link]
    }
    return  CalcDelay(NL, Flow, Cap, MsgLen, TotReq)
}
```