1 Graphs of conflic depth two

The star topology has a conflict depth one. We now look at the topologies with a conflict depth two.

In order to generate some random digraphs $G = (V,A)$ of conflict depth two, we first generate some random bipartite graphs $G' = (V',E')$, which represent the core of the network. Indeed, it seems to be a good idea to model the internet topology as mentioned in [1]. This papers also gives us some background to generate some good random bipartite graphs. V' is composed of two sets: The first set S_1, of size a, models the last switch before the data-centers, and the second set S_2, of size b, models some distant switches, reparted on the filed. We fix a and b and we uniformly draw the edges of E' with a given probability. Note that $|E'| = a \times b$. If the generated graph G' is not connected, we generate another bipartite graph. Figure 1 is an example of a random bipartite graph generated, with $a = 2$, $b = 3$ and $|E'| = 4$.

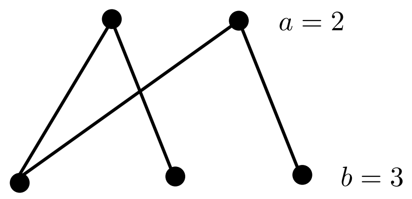

Figure 1: A random bipartite graph G'.

We now use G' to build our routed network G In order to create some arcs with conflict, the vertices of G' are extended to become some arcs in G. Let us call **conflict arcs** those arcs. The egdes of E' becomes some arcs of A, that we will call **core arcs**.

Figure 2 and figure 3 show how the first part of G is build from G'.

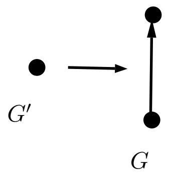

Figure 2: A node of G' represent an arc in G.

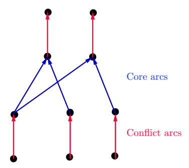

Figure 3: A node of G' represent an arc in G.

We now want to create some **final arcs**, that will model the last link before the antennas. We generate $1 \leq k \leq K$ final arcs before each contention arc (u,v). We add k new vertices $\{w_1, \ldots, w_k\}$ of indegree 0 and outdegree 1 to the set of vertices V. The new arcs $\{(w_1,u), \ldots, (w_k,u)\}$ are then added to A.

Each route of the set of routes $\mathcal{R}$ of the routed network (G,$\mathcal{R}$) is composed either of one final arc and one contention arc (obtained from the set S_1 of arcs in G'), or one final arc, two contention arcs (one from S_1, one frome S_2)) and the core arc between those two contention arcs. Note that one final arc belongs to one and only one route, while the core and contention arcs can be shared by several routes.

Figure 4 and figure 5 shows how the final arcs are generated and define the routes of the routed network (G,$\mathcal{R}$). The routes of the same color share the same arcs after the final arc.

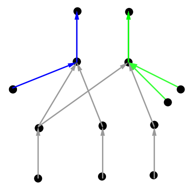
Figure 4: Final arcs before contention arcs obtained from S_1.

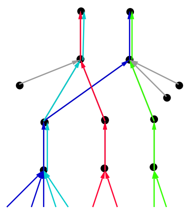
Figure 5: Final arcs before contention arcs obtained from S_2.

The obtained routed network (G,$\mathcal{R}$) models a meshed network. The vertices of indegree 0 of final arcs represent the antennas, the contention arcs from S_1 represent the last link before the datacenters, and the other nodes and arcs of the graph represent some links and switch of the network. It appears that this topology is realitic in core networks. Figure 6 shows the shape of the network modelized.

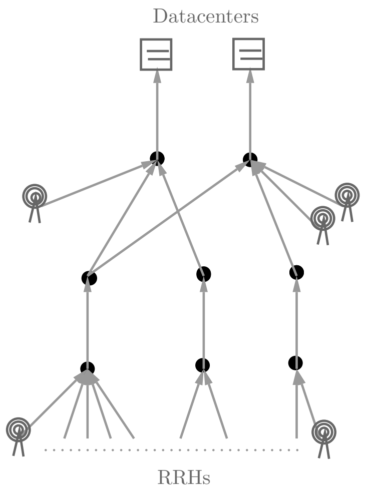

Figure 6: A network represented by a graph G.

2 Implemented algorithms

2.1 Simplification of the model

The steams are composed of one datagram. For simplicity in the notations, the datagram of the stream on the route r is then denoted d_r. All links have the same speed, so $l(d,u,r) = 0$, for all vertex u and route r of the network. All the datagrams have the same size $|d_r| = \tau$. Also, we do not allow the messages to be buffered in the nodes of the network so $l(d,u,r) = 0$, exept the last node of the routes, which correspond to the datacenter. The **waiting time** of a datagram d_r on a route r is defined by $w_r = \{\theta_{\rho_r}(d_r) - t(d_r,v,r)$. This is the time it waits in v, the last node of the route before being sent back. We also consider that all routes i have the same deadline, that is, $d(i) = Tmax$.

The presented algorithm are designed to solve the problem PAZL and PALL. A **partial assignment**, is a set of departure times and waiting times $\{(\theta_0,w_0),\ldots,(\theta_k,w_k)\}|k < n$, where n is the number of routes in the network.

In most of the presented algorithm, the following subroutine is used. Given a route r, a departure time θ_r, a partial assignment of the routed network, and a way of the message, it returns 1 if the message sent at time θ_r (if the way is forward) or θ_{ρ_r} (if the way is backward) does not collide with the other messages.

Algorithm 1 MessageNoCollisions

```
Input: A route r, a departure time, and a way of the message (FORWARD/BACKWARD).
Output: 1 if the messages can use the route with the given departure time and without collisions, 0 otherwise.
  for all Arcs in the route do
    if There is a collision with the previous scheduled messages then
      return 0
    end if
  end for
  return 1
```

2.2 Algorithm to solve PAZL

2.2.1 Greedy Prime

The basic idea is to try to schedule the routes one by one. The routes are selected by id. Considering the generation of our graph described ahead, this means we select first the routes close from the datacenters. Then, given a route, we set the departure time for the datagram at the beginning of the route to 0. If there is no collisions with the other routes, we give this departure time to the route. Otherwise, we increase the departure time of 1 and call the algorithm 1

MessageCollisions again, until we get a departure date that allows the datagram to pass the arcs without collisions with the routes already scheduled.

Algorithm 2 Greedy Prime

```
Input: A graph, a set of routes, a period $P$
Output: A P-periodic assignment in p $\leq P$, or FAILURE
  for all routes $i$ do
    date = 0
    while !MessageCollisions($i$,date,FORWARD) || !MessageCollisions($i$,date+routeLength($i$),BACKWARD) do
      date++;
      if date> $P$ then
        return FAILURE
      end if
    end while
    DepartureDate($i$) = date;
  end for
  return departureDate
```

The complexity of this algorithm is $\mathcal{O}(n \times P)$, with n the number of routes and P the period. It is very bad, since it depends of the period.

2.2.2 Greedy Min

We now try a smarter greedy algorithm. We stil take the routes one by one, sorted by id. Then, we try all departure times in the period. To each departure time is associated a "tic win" score, computed by the following : Considering a departure time that allow the datagram to pass the network without collisions, for each arc, we count evaluate how much the new message is close to the other ones. We look τ tics before (and after) the message, and if there is another message scheduled at tic i in the period, we increase the tic win score to $\tau - i$. The sum of the tic won score on each arc correspond to the tic won score of the departure time. We then assign to the route the departure time with the maximal tic won score. This algorithm helps us to pack the message in the arcs, indeed, if two message are spaced of a size $s < \tau$, those t tics will be definitely lost since we can not use them to send a message. Thus, the idea is to pack as much as possible the messages on every arcs.

Algorithm 3 ticsWin

```
Input: A route $r$, a departure time $t$
Output: $-1$ if the message can not pass without collisions, the number of tics
  win otherwise
  tmp $\leftarrow 0$
  for all Arcs $j$ in the route (forward AND backward) do
    if There is a collision with the previous scheduled messages then
      return $-1$
    else
      tmp += numberOfTicsWon(t,j)
    end if
  end for
  return tmp
```

Algorithm 4 Greedy tics won

```
Input: A graph, a set of routes of size $n$, a period $P$
Output: A P-periodic assignment in p $\leq P$, or FAILURE
  for all routes $i \in [0; n[$ do
    maxTicsWin $\leftarrow -1$
    bestDepartureTime $= -1$
    departureTime[$n$]
    for all offset $j \in [0; P[$ do
      tmp = ticsWin($r$,$j \times \tau$)
      if tmp ¿ maxTicsWin then
        maxTicsWin $\leftarrow$ tmp
        bestDepartureTime $\leftarrow j$
      end if
    end for
    if bestDepartureTime $= -1$ then
      return FAILURE
    end if
    departureTime($i$) $= j$
  end for
  return departureTime
```

The complexity of this algorithm is $\mathcal{O}(n \times P)$, with n the number of routes and P the period. Unlike the previous algorithm, the worst and medium complexity is the same, since we look at every departure times in the period.

2.3 Algorithms to solve PALL

We now allow the messages to be buffered in the BBUs. Thus have to choose the departure times and the waiting times of the datagrams.

2.3.1 Greedy Deadline

We choose to manage the routes in two steps. First, we send the datagrams in the way forward (from RRH to BBU). This part of the algorithm is basic and uses the same routine than the Algorithm: 2, the Greedy Prime algorithm. We take the route one by one, sorted by id, and we give for each route the first departure time that allow the datagram to reach the BBU without collisions. Note that we only check the way forward.

Once we have scheduled the messages in the way forward, we want to send the answers. Consider that u is the last vertex of the route forward (and so, the first of the route backward), the first step of the algorithm has fixed $t(d,u,r)$, that is, the time at which the datagram sent from the RRH has reach the node u. For each route, we compute $deadline(i) = Tmax - 2 \times \lambda(i) - DepartureTime(i)$. This deadline can be seen as a budget: the lowest the deadline will be, the less waiting time is allowed on the route.
We the send the routes with the lowest deadline first in the way backward, trying all waiting times, starting to 0, until the message can reach the RRH without collisions.

2.3.2 Greedy Loaded

In this algorithm, we choose to first take care of the most critical contention points first. Thus, we sort the arcs of the graph for the one in which there is the greater number of routes to the one in which there is the lower number of routes. Then we take first the more loaded link and we schedule the routes on it in two steps. First, we search the lowest departure time such that there is no collisions in the way forward, then we do it again on the way backward.

Algorithm 5 Greedy Loaded

```
Input: A graph, a set of routes, a period $P$
Output: A P-periodic assignment
  for all arcs $i$ sorted by decreasing number of routes using $i$ do
    for all route $j$ on $i$ do
      if $j$ has not been scheduled yet then
        depTime = 0
        while !MessageCollisions($j$,depTime,FORWARD) do
          depTime++;
          if depTime> $P$ then
            return FAILURE
          end if
        end while
        DepartureTime($j$) = depTime;
        beginBackTime = depTime + routeLength($j$);
        backTime = beginBackTime;
        while !MessageCollisions($j$,backTime,BACKWARD) do
          tmp++;
          if backTime> $P + beginBackTime$ then
            return FAILURE
          end if
        end while
        waitingTime($j$) = backTime-beginBackTime;
      end if
    end for
  end for
  return departureTime, waitingTime
```

We also propose 3 different orders to treat the route on the arcs.

- By id.
- From the longest to the shortest route.
- By the number of collisions with the other routes on the entire network.

References

[1] F. Tarissan, B. Quoitin, P. Mérindol, B. Donnet, J.-J. Pansiot, and M. Latapy, "Towards a bipartite graph modeling of the internet topology," *Computer Networks*, vol. 57, pp. 2331–2347, Aug. 2013.

[2] D. Barth, M. Guiraud, B. Leclerc, O. Marce, and Y. Strozecki, "Deterministic scheduling of periodic messages for cloud RAN," in *2018 25th Interna-*

tional Conference on Telecommunications (ICT) (ICT 2018), (Saint Malo, France), June 2018.