In [1]:
```
%load_ext autoreload
%autoreload 2

import tme10
```

TME Échantillonage

Diffusion dans les graphes

Au cours des vingt dernières années, les réseaux sociaux sont devenus un média d'information incontournable, mettant en jeu des dynamiques complexes de communication entre utilisateurs. La modélisation de la diffusion d'information sur les réseaux constitue depuis lors un enjeu majeur, pour diverses tâches telles que l'identification de leaders d'opinions, la prédiction ou la maximisation de l'impact d'un contenu diffusé, la détection de communautés d'opinions, ou plus généralement l'analyse des dynamiques du réseau considéré.

Le modèle proposé par (Saito et al, 2009) considère une diffusion en cascade dans laquelle l'information transite de noeuds en noeuds du réseau en suivant des relations d'influence entre les utilisateurs. Lorsqu'un utilisateur est ``infecté'' par une information, il possède une chance unique de la retransmettre à chacun de ses successeurs dans le graphe, selon une probabilité définie sur le lien correspondant. Le modèle définit en fait deux paramètres sur chaque lien (u,v) du graphe:

- $k_{u,v}$: la probabilité que l'utilisateur u transmette une information diffusée à v
- $r_{u,v}$: si la transmission s'effectue, l'utilisateur v la reçoit au temps $t_v=t_u+\delta$, avec $\delta \sim Exp(r_{u,v})$

Question 1

Pour utiliser ce modèle, on devra donc échantillonner selon la distribution exponentielle. Pour commencer, on cherche alors à écrire une méthode $exp(rate)$ qui échantillonne des variables d'une loi exponentielle selon le tableau d'intensités $rate$ passé en paramètre. Cet échantillonnage se fera par **Inverse Transform Sampling**. Pour éviter les divisions par 0, on ajoutera $1e-200$ aux intensités qui valent 0.

In [2]:
```
import numpy as np
np.random.seed(0)

#Test
a=tme10.exp(np.array([[1,2,3],[4,5,6]]))
print("tirages selon 6 lois exponentielles de paramètres [[1,2,3],[4,5,6]]")
print(a)

for i in range(10000):
  a+=tme10.exp(np.array([[1,2,3],[4,5,6]]))
print(a/10000)
```

```
tirages selon 6 lois exponentielles de paramètres [[1,2,3],[4,5,6]]
[[0.79587451 0.62796538 0.30774105]
 [0.19680029 0.1102097  0.17302655]]
[[0.98796784 0.49198855 0.33501196]
 [0.25022762 0.19644862 0.16723749]]
```

Question 2

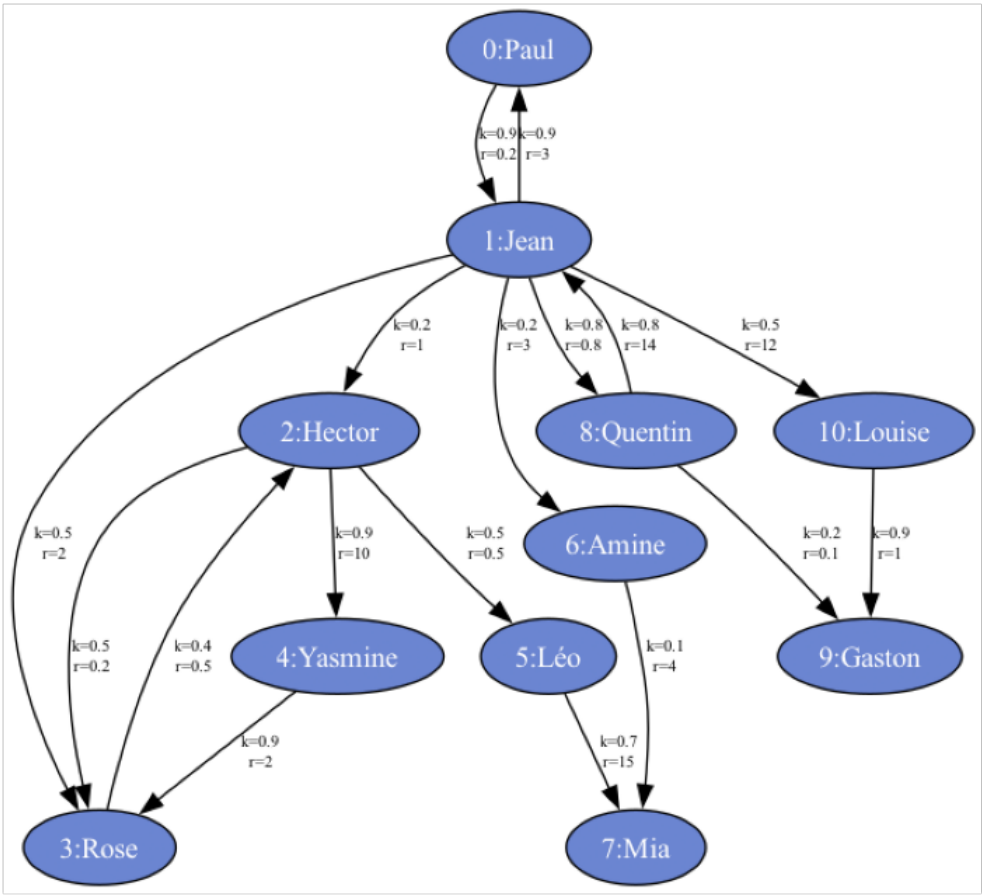
Soit le graphe de diffusion donné ci dessous:

In [3]:
```
names={0:"Paul",
       1:"Jean",
       2:"Hector",
       3:"Rose",
       4:"Yasmine",
       5:"Léo",
       6:"Amine",
       7:"Mia",
       8:"Quentin",
       9:"Gaston",
       10:"Louise"}

k={(0,1):0.9,
   (1,0):0.9,
   (1,2):0.2,
   (2,3):0.5,
   (3,2):0.4,
   (2,4):0.9,
   (4,3):0.9,
   (1,3):0.5,
   (2,5):0.5,
   (5,7):0.7,
   (1,6):0.2,
```

```
   (6,7):0.1,
   (1,8):0.8,
   (8,9):0.2,
   (1,10):0.5,
   (10,9):0.9,
   (8,1):0.8}

r={(0,1):0.2,
   (1,0):3,
   (1,2):1,
   (2,3):0.2,
   (3,2):0.5,
   (2,4):10,
   (4,3):2,
   (1,3):2,
   (2,5):0.5,
   (5,7):15,
   (1,6):3,
   (6,7):4,
   (1,8):0.8,
   (8,9):0.1,
   (1,10):12,
   (10,9):1,
   (8,1):14}

graph=(names,k,r)
```

La fonction display_graph ci dessous permet de visualiser le graphe de diffusion correspondant:

In [4]:
```
import pydot
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

style = { "bgcolor" : "#6b85d1", "fgcolor" : "#FFFFFF" }

def display_graph ( graph_data, style, graph_name="diffusion_graph" ):
    graph = pydot.Dot( graph_name , graph_type='digraph')
    names,k,r=graph_data
    # création des noeuds du réseau
    for (i,name) in names.items():
        new_node = pydot.Node( i,
                               label=f'"{i}:{name}"',
                               style="filled",
                               fillcolor=style["bgcolor"],
                               fontcolor=style["fgcolor"] )
        graph.add_node( new_node )

    # création des arcs
    for edge,valk in k.items():
        valr=r[edge]
        n1=edge[0]
        n2=edge[1]
        new_edge = pydot.Edge ( n1, n2, label=f"k={valk}\nr={valr}",fontsize="8")
        graph.add_edge ( new_edge )

    # sauvegarde et affichage
    outfile = "res/" + graph_name + '.png'
    graph.write_png( outfile )
    img = mpimg.imread ( outfile )

    fig, ax = plt.subplots(figsize=(9,9))
    fig.patch.set_visible(False)
    ax.axis('off')
    plt.imshow( img )
display_graph(graph,style)
```



Question 2.1

On souhaite être capable d'estimer les probabilités marginales d'infection des différents utilisateurs du réseau par une information pour laquelle on connaît les sources (i.e., les utilisateurs infectés au temps 0).

Etant donnés les cycles possibles dans le graphe de diffusion, considérer un calcul exact des probabilités d'infection des différents utilisateurs sachant le début de la diffusion est inenvisageable : il faudrait considérer toutes les combinaisons possibles (infinies) de temps d'infection pour tous les utilisateurs non sources.

Une possibilité pour calculer ces probabilités d'infections est de travailler par échantillonnage de Monte Carlo: on réalise n tirages d'infections connaissant les sources et on recense le ratio des simulations dans lesquelles chacun des utilisateurs est infecté avant un temps $maxT$.

L'idée est alors dans un premier temps d'écrire une méthode $simulation(graph,sources,maxT)$ qui, à partir d'une liste de sources, retourne les temps d'infection de l'ensemble des noeuds en fin de diffusion, sous la forme d'un tableau où chaque case i contient le temps d'infection du noeud i. Si le noeud i n'a pas été infecté ou bien si il l'a été après un temps maximal $maxT$, la case i contient alors la valeur $maxT$.

Le pseudo-code de la méthode de simulation est donné ci dessous, avec t_i le temps d'infection courant du noeud i:

```
ti=maxT pour tout i non source
Tant qu'il reste des infectieux dont le temps est < maxT:
  i=Infectieux de temps d'infection minimal
  Pour tout noeud j tel que tj>ti:
    sampler x selon Bernouilli(kij)
    si x==1:
       sampler delta selon Exp(rij)
       t=ti+delta
       si t<tj: tj=t
  Retrait de i de la liste des infectieux
```

Complétez le code de la fonction donnée ci-dessous:

In [5]:
```
maxT=10

np.random.seed(1)
print(tme10.simulation(graph,[0],maxT))
print(tme10.simulation(graph,[0],maxT))
print(tme10.simulation(graph,[0],maxT))

np.random.seed(1)
```

```
print(tme10.simulation(graph,[0,1],maxT))
print(tme10.simulation(graph,[0,1],maxT))
print(tme10.simulation(graph,[0,1],maxT))
```

```
[ 0.          2.71669685 10.         10.         10.         10.
 10.         10.          3.19055869  3.17528764  2.86665883]
[ 0.          0.60940319 10.         10.         10.         10.
 10.         10.          2.36988928 10.         10.        ]
[ 0.          0.22787406 10.         10.         10.         10.
 10.         10.          1.27950225  3.42920125 10.        ]
[ 0.          0.          0.03983788  0.09306264  0.05063365  1.10889995
 10.          1.16647819 10.          1.16739272  0.03159079]
[ 0.          0.         10.         10.         10.         10.
  0.16359844 10.          1.71855838 10.         10.        ]
[ 0.          0.          3.08047501  1.49963044  3.25699405 10.
 10.         10.          0.83189232  2.23597755 10.        ]
```

Question 2.2

La méthode $getProbaMC(graph,sources,maxT,nbsimu)$ retourne les estimations de probabilités marginales d'infection des différents noeuds de $graph$, conditionnées à l'observation des $sources$. Pour être enregistrée, une infection doit intervenir avant la seconde $maxT$. Ainsi, si la méthode retourne 0.2 pour le noeud i, cela indique qu'il a été infecté avec un temps $t_i \in]0,maxT[$ dans 20% des $nbsimu$ simulations effectuées. Compléter la méthode ci dessous:

In [6]:
```
np.random.seed(0)

rInf=tme10.getProbaMC(graph,[0],maxT,100000)
print(rInf)

rInf=tme10.getProbaMC(graph,[0],maxT,100000)
print(rInf)

rInf=tme10.getProbaMC(graph,[0,1],maxT,100000)
print(rInf)

rInf=tme10.getProbaMC(graph,[2,8],maxT,100000)
print(rInf)
```

```
[1.      0.7785  0.25939 0.44694 0.23214 0.11123 0.15518 0.09145 0.58973
 0.36455 0.38976]
[1.      0.77994 0.25928 0.44709 0.23307 0.11118 0.155   0.09067 0.59052
 0.36201 0.38788]
[1.      1.      0.35724 0.58993 0.32084 0.17582 0.20088 0.13995 0.79891
 0.49967 0.49876]
[0.71818 0.79804 1.      0.93559 0.89997 0.49813 0.15957 0.35803 1.
 0.44108 0.39904]
```

Question 2.3

Cette méthode permet de bonnes estimations (malgré une certaine variance) lorsque l'on n'a pas d'observations autres que le vecteur de sources (i.e., on estime des probabilités de la forme: $P(t_i < maxT|\{(j,t_j),t_j=0\})$). Par contre, si l'on souhaite obtenir des probabilités d'infection du type $P(t_i < maxT|\{(j,t_j),t_j=0\}, \{(j,t_j), j \in {\cal O}\})$, c'est à dire conditionnées à des observations supplémentaires pour un sous-ensembles de noeuds ${\cal O}$ (avec $t_j > 0$ pour tout noeud j de ${\cal O}$), l'utilisation de la méthode de MonteCarlo précédente est impossible. Cela impliquerait de filtrer les simulations obtenues selon qu'elles remplissent les conditions sur les noeuds de ${\cal O}$, ce qui nous amènerait à toutes les écarter sachant que l'on travaille avec des temps continus.

Pour estimer ce genre de probabilité conditionnelle, nous allons nous appuyer sur des méthodes de type MCMC, notamment la méthode de Gibbs Sampling. Cette méthode est utile pour simuler selon une loi jointe, lorsqu'il est plus simple d'échantillonner de chaque variable conditionnellement à toutes les autres plutôt que directement de cette loi jointe. L'algorithme est donné par:

1. Tirage d'un vecteur de valeurs initiales pour toutes les variables X_i
2. Pour toutes les variable X_i choisies dans un ordre aléatoire, échantillonnage d'une nouvelle valeur: $X_i \sim p(x_i\mid x_1,\dots,x_{i-1},x_{i+1},\dots,x_n)$
3. Recommencer en 2 tant qu'on souhaite encore des échantillons

Notons qu'il est souvent utile d'exploiter la relation suivante, qui indique que pour échantillonner de la loi conditionnelle, il suffit d'échantillonner chaque variable proportionnellement à la loi jointe, avec toutes les autres variables fixées: $$p(x_j\mid x_1,\dots,x_{j-1},x_{j+1},\dots,x_n) = \frac{p(x_1,\dots,x_n)}{p(x_1,\dots,x_{j-1},x_{j+1},\dots,x_n)} \propto p(x_1,\dots,x_n)$$

Après une période dite de $burnin$ d'un nombre d'époques à définir, l'algorithme émet des échantillons qui suivent la loi jointe connaissant les observations. Lorsque l'objectif est d'estimer des probabilités marginales, on fait alors tourner cet algorithme pendant une certain nombre d'époques après la période de $burnin$, au cours desquelles on recence les différentes affectations de chacune des variables étudiées.

Pour mettre en oeuvre cet algorithme, nous aurons aurons besoin d'avoir accès rapidement aux prédecesseurs et successeurs dans le graphe. La méthode ci-dessous retourne un couple de dictionnaires à partir du graphe:

- $preds[i]$ contient la liste des prédécesseurs du noeud i, sous la forme d'une liste de triplets $(j,k_{j,i},r_{j,i})$ pour tous les j précédant i dans le graphe.
- $succs[i]$ contient la liste des successeurs du noeud i, sous la forme d'une liste de triplets $(j,k_{i,j},r_{i,j})$ pour tous les j pointés par i dans le graphe.

In [7]:
```
preds,succs=tme10.getPredsSuccs(graph)

print("preds=",preds)
print("succs=",succs)
```

```
preds= {1: [(0, 0.9, 0.2), (8, 0.8, 14)], 0: [(1, 0.9, 3)], 2: [(1, 0.2, 1), (3, 0.4, 0.5)], 3: [(2, 0.5, 0.2),
(4, 0.9, 2), (1, 0.5, 2)], 4: [(2, 0.9, 10)], 5: [(2, 0.5, 0.5)], 7: [(5, 0.7, 15), (6, 0.1, 4)], 6: [(1, 0.2,
3)], 8: [(1, 0.8, 0.8)], 9: [(8, 0.2, 0.1), (10, 0.9, 1)], 10: [(1, 0.5, 12)]}
succs= {0: [(1, 0.9, 0.2)], 1: [(0, 0.9, 3), (2, 0.2, 1), (3, 0.5, 2), (6, 0.2, 3), (8, 0.8, 0.8), (10, 0.5, 1
2)], 2: [(3, 0.5, 0.2), (4, 0.9, 10), (5, 0.5, 0.5)], 3: [(2, 0.4, 0.5)], 4: [(3, 0.9, 2)], 5: [(7, 0.7, 15)],
6: [(7, 0.1, 4)], 8: [(9, 0.2, 0.1), (1, 0.8, 14)], 10: [(9, 0.9, 1)]}
```

Pour calculer les probabilités conditionnelles, il faut prendre en compte les quantités suivantes:

- Probabilité pour j d'être infecté par i au temps t_j connaissant $t_i < t_j$:
  $$\alpha_{i,j}=k_{i,j}r_{i,j} exp(-r_{i,j}(t_j-t_i))$$
- Probabilité pour j de ne pas être infecté par i jusqu'au temps t: $$\beta_{i,j}=k_{i,j} exp(-r_{i,j}(t_j-t_i)) + 1 - k_{i,j}$$
- Probabilité pour j d'être infecté au temps t_j connaissant les prédecesseurs infectés avant t_j: $$h_{j}=\prod_{i \in preds[j], t_i<t_j} \beta_{i,j} \sum_{i \in preds[i], t_i<t_j} \alpha_{i,j} / \beta_{i,j}$$
- Probabilité pour j de ne pas être infecté avant $maxT$ connsaissant ses prédecesseurs infectés: $$g_{j}=\prod_{i \in preds[j], t_i<t_j} \left(k_{i,j} exp(-r_{i,j}(maxT-t_i)) + 1 - k_{i,j}\right)=\prod_{i \in preds[j], t_i<t_j} \beta_{i,j}$$

Dans la méthode $computeab(v, times, preds)$, on prépare le calcul et les mises à jour de ces quantités. La méthode calcule, pour un noeud v selon les temps d'infection courants donnés dans $times$, deux quantités a et b:

- $a= max(\epsilon, \sum_{i \in preds[v], t_i<t_v} \alpha_{i,v} / \beta_{i,v})$ si $t_v< maxT$ et $a=1$ sinon.
- $b=\sum_{i \in preds[v], t_i<t_v} \log \beta_{i,v}$.

Si v appartient aux sources, on retourne $(a,b)=(1,0)$

Écrire la méthode `compute_ab` . On prendra, par défaut, $\epsilon=10^{-20}$.

In [8]:
```
nbNodes=len(graph[0])
times=np.array([maxT]*nbNodes,dtype=float)
times[0]=0
times[1]=1
times[2]=4

print(tme10.compute_ab(0,times,preds,maxT,eps=1e-20))
print(tme10.compute_ab(1,times,preds,maxT,eps=1e-20))
print(tme10.compute_ab(2,times,preds,maxT,eps=1e-20))
print(tme10.compute_ab(3,times,preds,maxT,eps=1e-20))
```

```
(1, 0)
(0.17610107365772135, -0.17810126145719926)
(0.012293749653343877, -0.2107736084094422)
(1.0, -1.12301187855188)
```

Question 2.4

La méthode `compute_ll` calcule la log-vraisemblance d'une diffusion (représentée par le tableau times), en appelant la méthode computeab sur l'ensemble des noeuds du réseau. Elle retourne un triplet (log-likelihood, sa, sb), avec sa et sb les tables des valeurs a et b pour tous les noeuds.

In [9]:
```
ll,sa,sb=tme10.compute_ll(times,preds,maxT)
print("ll=",ll)
print(times)
print("like_indiv=",np.exp(np.log(sa)+sb))
```

```
ll= -13.117139892397578
[ 0.  1.  4. 10. 10. 10. 10. 10. 10. 10. 10.]
like_indiv= [1.         0.14737154 0.00995741 0.32529856 0.1        0.52489353
 0.8        1.         0.20059727 1.         0.5       ]
```

Question 2.5

Afin de préparer les mises à jour lors des affectations successives des variables du Gibbs Sampling, on propose de définir une méthode $removeV(v,times,succs,sa,sb)$ qui retire temporairement du réseau un noeud v, en passant son temps d'infection à -1 dans times et en retirant sa contribution aux valeurs a et b (contenues dans sa et sb) de tous ses successeurs j tels que $t_j > t_v$ (y compris donc les non infectés qui sont à $t_j=maxT$).

In [10]:
```
def removeV(v,times,succs,sa,sb,eps=1e-20):
```

```
  succs=succs.get(v,[])
  t=times[v]
  if t<0:
    return
  times[v]=-1
  sa[v]=1.0
  sb[v]=0.0
  if len(succs)>0:
    c,k,r=map(np.array,zip(*succs))
    tp=times[c]
    which=(tp>t)

    tp=tp[which]
    dt=tp-t
    k=k[which]
    r=r[which]
    c=c[which]
    rt = -r*dt
    b1=k*np.exp(rt)
    b=b1+1.0-k

    a=r*b1
    a=a/b
    b=np.log(b)

    sa[c]=sa[c]-np.where(tp<maxT,a,0.0)
    sa[c]=np.where(sa[c]>eps,sa[c],eps)
    sb[c]=sb[c]-b
    sb[c]=np.where(sb[c]>0,0,sb[c])


#Test
print("sa=",sa)
print("sb=",sb)

nsa=np.copy(sa)
nsb=np.copy(sb)
ntimes=np.copy(times)
removeV(3,ntimes,succs,nsa,nsb)
print("diffa=",nsa-sa)
print("diffb=",nsb-sb)

nsa=np.copy(sa)
nsb=np.copy(sb)
ntimes=np.copy(times)
removeV(1,ntimes,succs,nsa,nsb)
print("diffa=",nsa-sa)
print("diffb=",nsb-sb)
```

```
sa= [1.         0.17610107 0.01229375 1.         1.         1.
 1.         1.         1.         1.         1.         1.        ]
sb= [ 0.         -0.17810126 -0.21077361 -1.12301188 -2.30258509 -0.64455983
 -0.22314355  0.         -1.60645602  0.         -0.69314718]
diffa= [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
diffb= [0.         0.         0.         1.12301188 0.         0.
 0.         0.         0.         0.         0.         0.        ]
diffa= [ 0.          0.82389893 -0.01229375  0.          0.          0.
  0.          0.          0.          0.          0.          0.        ]
diffb= [0.         0.17810126 0.21077361 0.69314717 0.         0.
 0.22314355 0.         1.60645602 0.         0.69314718]
```

La méthode addVatT fait l'inverse: elle rajoute un noeud qui était retiré du réseau, avec un temps $newt$. Il faut alors mettre à jour les valeurs a et b (dans sa et sb) de tous les successeurs de v tels que $t_j > newt$ et calculer les valeurs a et b du noeud v.

Écrire cette méthode.

In [11]:
```
# Tests:

nsa=np.copy(sa)
nsb=np.copy(sb)
c,_,_=map(np.array,zip(*succs[1]))
c=np.append(c,1)
ll=np.sum((np.log(nsa)+nsb)[c])           # somme des logvraisemblances pouvant être modifiées par la modificat
removeV(1,times,succs,nsa,nsb)
tme10.addVatT(1,times,2,preds,succs,nsa,nsb,maxT)
ll2=np.sum((np.log(nsa)+nsb)[c])          # somme des logvraisemblances pouvant avoir été modifiées par la modi
removeV(1,times,succs,nsa,nsb)
tme10.addVatT(1,times,1,preds,succs,nsa,nsb,maxT)
ll3=np.sum((np.log(nsa)+nsb)[c])          # somme des logvraisemblances pouvant  avoir été modifiées par la mod
llall=np.sum(np.log(nsa)+nsb)             # logvraisemblance globale
print(np.exp(ll),np.exp(ll2),np.exp(ll3),llall)

c,_,_=map(np.array,zip(*succs[0]))
c=np.append(c,0)
ll=np.sum((np.log(nsa)+nsb)[c])
```

```
removeV(0,times,succs,nsa,nsb)
tme10.addVatT(0,times,maxT,preds,succs,nsa,nsb,maxT)
ll2=np.sum((np.log(nsa)+nsb)[c])
removeV(0,times,succs,nsa,nsb)
tme10.addVatT(0,times,0,preds,succs,nsa,nsb,maxT)
ll3=np.sum((np.log(nsa)+nsb)[c])
llall=np.sum(np.log(nsa)+nsb)
print(np.exp(ll),np.exp(ll2),np.exp(ll3),llall)

c,_,_=map(np.array,zip(*succs[5]))
c=np.append(c,5)
ll=np.sum((np.log(nsa)+nsb)[c])
removeV(5,times,succs,nsa,nsb)
tme10.addVatT(5,times,1,preds,succs,nsa,nsb,maxT)
ll2=np.sum((np.log(nsa)+nsb)[c])
removeV(5,times,succs,nsa,nsb)
tme10.addVatT(5,times,maxT,preds,succs,nsa,nsb,maxT)
ll3=np.sum((np.log(nsa)+nsb)[c])
llall=np.sum(np.log(nsa)+nsb)
print(np.exp(ll),np.exp(ll2),np.exp(ll3),llall)
```

```
3.830251606174211e-05 8.555487921315824e-05 3.830251606174211e-05 -13.117139892397578
0.14737153555403676 1.0000000000169125e-21 0.14737153555403676 -13.117139892397578
0.5248935341839319 2.999999999999998e-21 0.5248935341839319 -13.117139892397578
```

Question 2.6

Pour échantillonner pour une variable i, il faudra être à même de comparer les vraisemblances selon les différentes affectations. Cela implique de calculer la somme de toutes ces vraisemblances. Mais pour réaliser cette somme, il faudrait que nous sortions de la représentation logarithmique: $\sum_{t_i} exp(log(p(t_1,\dots,t_i,\dots,t_n))$. Si on le fait de cette manière, on risque d'avoir des arrondis à 0 presque partout. Une possibilité (log-sum-exp trick) est d'exploiter la relation suivante:

$$\log\sum_i x_i = x^* + \log\left(\exp(x_1-x^*)+ \cdots + \exp(x_n-x^*) \right)$$ avec $x^* = \max{\{x_1, \dots, x_n\}}$

Compléter la méthode `logsumexp` suivante, qui réalise cette somme en évitant les problèmes numériques:

In [12]:
```
#Test:
x=np.array([[0.001,0.02,0.008],[0.1,0.01,0.4]])
r=np.log(np.sum(x,-1))
print(r)

x=np.log(x)
r2=tme10.logsumexp(x)
print(r2)
```

```
[-3.54045945 -0.67334455]
[-3.54045945 -0.67334455]
```

Question 2.7

On souhaite maintenant mettre en place une méthode $sampleV(v,times,newt,preds,succs,sa,sb,k,k2)$ qui sample un nouveau temps d'infection pour le noeud v, connaissant les temps de tous les autres noeuds dans $times$ (ainsi que leurs valeurs a et b correspondantes contenues dans sa et sb). Puisque le domaine de support de t_v est continu, on doit faire quelques approximations en se basant sur une discrétisation des valeurs possibles:

1. On découpe la plage de temps $[0;maxT]$ en k bins réguliers. Dans chaque bin i, on échantillonne uniformément un temps, pour obtenir k points $d_1,\dots,d_k$. Si $t_v < maxT$, on ajoute t_v à cet ensemble de points pour gagner en stabilité (inséré dans la liste de manière à conserver l'ordre croissant).
2. On considère chaque point d_i comme le prototype d'un bin $[(d_i+d_{i-1})/2,(d_i+d_{i+1})/2]$. Pour d_1 on prend $[0,(d_1+d_2)/2]$ et pour d_k on prend $[(d_k+d_{k-1})/2,maxT]$. On fait l'hypothèse que la densité de probabilité est constante sur l'ensemble de chaque bin i, que l'on évalue en $t_v=d_i$. La probabilité que l'on échantillonne dans le bin i est alors égale à: $p(t_v \in bin_i | \{t_u\}_{ u \in V\setminus v}) = \frac{z_i \times l_i}{\sum_j z_j \times l_j + z_{maxT}}$, avec z_i la vraisemblance calculée selon $t_v =d_i$, l_i la taille du bin i et z_{maxT} la vraisemblance calculée pour $t_v=maxT$. La probabilité que v ne soit pas infecté dans la diffusion est alors donnée par : $p(t_v = maxT | \{t_u\}_{ u \in V\setminus v}) = \frac{z_{maxT}}{\sum_j z_j \times l_j + z_{maxT}}$.
3. On échantillonne une variable x proportionnellement aux probabilités calculées à l'étape précédente. Si x ne correspond pas à $maxT$, v est alors infecté à un temps inclus dans l'intervale du bin correspondant à x. Il s'agit alors de re-échantillonner $k2$ points uniformément dans ce bin et de calculer les densités en ces points (pour gagner en stabilité on ajoute le prototype du bin d_i). Le nouveau temps de v est alors échantillonné proportionnellement à ces densités.

Le code de la méthode de sampling est donné ci-dessous:

In [13]:
```
np.random.seed(0)



def getLL(v,times,nt,preds,succs,sa,sb,maxT,onUsers=None):
  sa=np.copy(sa)
  sb=np.copy(sb)
```

```
  if onUsers is None:
    onUsers=range(len(times))
  tme10.addVatT(v,times,nt,preds,succs,sa,sb,maxT)
  times[v]=-1
  ll=np.sum((np.log(sa)+sb)[onUsers])
  return (ll,sa,sb)


def sampleV(v,times,preds,succs,sa,sb,maxT,k,k2):

  nbCandidateT=k
  bounds=np.linspace(0,maxT,nbCandidateT)
  newt=np.random.uniform(bounds[:-1],bounds[1:])

  if times[v]<maxT:
    idx = newt.searchsorted(times[v])
    newt=np.concatenate((newt[:idx], [times[v]], newt[idx:]),axis=0)
    nbCandidateT+=1
  newt=np.append(newt,[maxT])

  if v in succs:
    c,_,_=map(list,zip(*succs.get(v,[])))
  else:
    c=[]
  c.append(v)
  c=np.array(c)
  oldll=np.sum((np.log(sa)+sb)[c])
  otime=times[v]
  nsa=np.copy(sa)
  nsb=np.copy(sb)
  removeV(v,times,succs,nsa,nsb)
  lls=[getLL(v,times,nt,preds,succs,nsa,nsb,maxT,onUsers=c) for nt in newt]
  ll,la,lb=zip(*lls)
  ll=list(ll)
  ll=np.array(ll)

  diffsx=(newt[1:]-newt[:-1])/2.0
  diffsx[1:]=diffsx[1:]+diffsx[:-1]
  diffsx[0]+=newt[0]
  diffsx[-1]+=(maxT-newt[nbCandidateT-1])/2.0
  areas=np.log(diffsx)+ll[:-1]
  lln=np.append(areas,ll[-1])


  p=np.exp(lln-tme10.logsumexp(lln))


  i=np.random.choice(range(len(p)),1,p=p).sum()
  if i==(len(p)-1):
    times[v]=maxT
    np.copyto(sa,np.array(la[-1]))
    np.copyto(sb,np.array(lb[-1]))
  else:
      if i>0:
        bi=(newt[i]+newt[i-1])/2.0
      else:
        bi=0
      if i<(len(p)-2):
        bs=(newt[i]+newt[i+1])/2.0
      else:
        bs=maxT
      bounds=np.linspace(bi,bs,k2)
      newt=np.concatenate(([newt[i]],np.random.uniform(bounds[:-1],bounds[1:])))
      lls=[getLL(v,times,nt,preds,succs,nsa,nsb,maxT,onUsers=c) for nt in newt]
      ll,la,lb=zip(*lls)
      ll=np.array(ll)
      p=np.exp(ll-tme10.logsumexp(ll))

      i=np.random.choice(range(len(p)),1,p=p).sum()

      times[v]=newt[i]
      np.copyto(sa,np.array(la[i]))
      np.copyto(sb,np.array(lb[i]))


times=np.array([maxT]*nbNodes,dtype=float)
times[0]=0
times[1]=1
times[2]=4
np.random.seed(0)
print(times)
sampleV(5,times,preds,succs,sa,sb,10,10,maxT)
print(times)
sampleV(5,times,preds,succs,sa,sb,10,10,maxT)
print(times)
sampleV(5,times,preds,succs,sa,sb,10,10,maxT)
```

```
print(times)
sampleV(5,times,preds,succs,sa,sb,10,10,maxT)
print(times)
sampleV(5,times,preds,succs,sa,sb,10,10,maxT)
print(times)
sampleV(5,times,preds,succs,sa,sb,10,10,maxT)
print(times)
sampleV(5,times,preds,succs,sa,sb,10,10,maxT)
print(times)
sampleV(5,times,preds,succs,sa,sb,10,10,maxT)
print(times)
```

```
[ 0.  1.  4. 10. 10. 10. 10. 10. 10. 10. 10.]
[ 0.  1.  4. 10. 10. 10. 10. 10. 10. 10. 10.]
[ 0.  1.  4. 10. 10. 10. 10. 10. 10. 10. 10.]
[ 0.  1.  4. 10. 10. 10. 10. 10. 10. 10. 10.]
[ 0.  1.  4. 10. 10. 10. 10. 10. 10. 10. 10.]
[ 0.  1.  4. 10. 10. 10. 10. 10. 10. 10. 10.]
[ 0.  1.  4. 10. 10. 10. 10. 10. 10. 10. 10.]
[ 0.          1.          4.         10.         10.          4.20931617
 10.         10.         10.         10.         10.        ]
[ 0.  1.  4. 10. 10. 10. 10. 10. 10. 10. 10.]
```

Compléter la méthode de Gibbs Sampling gb ci-dessous, en lui passant en paramètre la fonction de sample (ici `sampleV`), k le nombre de bins à utiliser et $k2$ le nombre de points à échantillonner dans le bin choisi. Le paramètre ref correspond à un vecteur de probabilités marginales de référence (par exemple obtenu par MonteCarlo lorsque c'est possible) avec lequel on peut afficher la distance MSE au fur et à mesure du processus (tous les `period` pas de temps).

In [14]:
```
np.random.seed(1)

# On teste ici avec seulement des sources (i.e., des infectés au temps 0), car cela permet de comparer à la ref
ref=tme10.getProbaMC(graph,[0],maxT,nbsimu=100)
rate=tme10.gb(graph,[(0,0)],maxT,sampler=sampleV,burnin=100,ref=ref,period=1000)
print("------")
print(f"{rate=}")
```

```
100  burnin time   ll= -2.945801499181733
1100 [1.         0.76566757 0.27520436 0.44414169 0.24159855 0.10535876
 0.15894641 0.08446866 0.54405086 0.31880109 0.35785649] MSE = 0.0076659572958610025  ll= -6.216002333449634
2100 [1.         0.81437411 0.30318896 0.47881961 0.27177535 0.12660638
 0.16277963 0.10709186 0.6025702  0.37267968 0.407901  ] MSE = 0.013141187068735653  ll= -5.04069061083797
3100 [1.         0.79748468 0.29409868 0.46565624 0.26475331 0.12899065
 0.16317317 0.10802967 0.59464689 0.3737504  0.40761045] MSE = 0.00858942123817919  ll= -4.337507671104333
4100 [1.         0.78688125 0.26066813 0.44159961 0.23408925 0.11265545
 0.16020483 0.09485491 0.58863692 0.37771275 0.40819312] MSE = 0.004631919971702085  ll= -1.5059712919558215
5100 [1.         0.78651245 0.25485199 0.43795334 0.22799451 0.11252696
 0.15859635 0.09625564 0.59125662 0.38815918 0.41599686] MSE = 0.005449703524580764  ll= -3.9861881219992776
6100 [1.         0.78872316 0.25782659 0.44189477 0.23110965 0.10981806
 0.15554827 0.09293558 0.592198   0.39173906 0.4209146 ] MSE = 0.006229372118537535  ll= -5.228256308348173
7100 [1.         0.78636812 0.26855372 0.44669765 0.24010703 0.11406844
 0.15547106 0.09604281 0.59329672 0.38121391 0.40923814] MSE = 0.005193337244947123  ll= -5.134402804074275
8100 [1.         0.78076781 0.26910258 0.44463646 0.23972349 0.11652882
 0.15405505 0.09788915 0.58918652 0.38192816 0.40957906] MSE = 0.004611898852715847  ll= -1.5059712919558215
9100 [1.         0.78255137 0.26799253 0.4472036  0.23931436 0.11328425
 0.15404901 0.09460499 0.59224261 0.38204593 0.40951544] MSE = 0.005128462479826162  ll= -5.41414623733148
------
rate=array([1.        , 0.78255137, 0.26799253, 0.4472036 , 0.23931436,
       0.11328425, 0.15404901, 0.09460499, 0.59224261, 0.38204593,
       0.40951544])
```

Partie optionnelle

L'algorithme de Metropolis-Hasting est une autre méthode de type MCMC qui utilise une distribution d'échantillonnage pour se déplacer dans l'espace des points considérés. Il s'agit de définir une distribution $q(y_{t+1}|x_t)$ de laquelle on sait générer un déplacement. L'algorithme procéde alors de la manière suivante:

1. Générer y_{t+1} selon $q(y_{t+1}|x_t)$
2. Calculer la probabilité d'acceptation $\alpha(x_t,y_{t+1})=\min\left\{\frac{\pi(y_{t+1})q(x_t|y_{t+1})}{\pi(x_t)q(y_{t+1}|x_t)},1 \right\} \,\!, \text{ avec } \pi(x_t) \text{ la densité de probabilité de } x_t$
3. Prendre $x_{t+1}=\begin{cases} y_{t+1}, & \text{avec probabilité}\,\,\alpha \\ x_t, & \text{avec probabilité}\,\,1-\alpha \end{cases}$

Dans notre cas, on propose de travailler avec des déplacements correspondants à des permutations d'un temps d'infection à chaque itération, comme dans le cadre du Gibbs Sampling. A chaque étape on choisit donc une variable à modifier, on choisit un nouveau temps pour cette variable et on calcule la densité correspondante. La probabilité d'acceptation est ensuite calculée selon cette densité et la probabilité du déplacement selon la distribution q qui a servi à générer le nouveau temps d'infection. On se propose de choisir $maxT$ avec une probabilité de 0.1. La probabilité $q(t_v|t)$ pour $t< maxT$ est alors égale à $0.9\times \frac{1}{maxT}$.

Implémenter l'approche d'échantillonnage par Metropolis-Hasting pour notre problème d'estimation de probabilités marginales d'infection.

In [15]:
```
def move(v,times,preds,succs,sa,sb,maxT):
  x=np.random.rand(1)
  if x<0.1:
    newt=maxT
  else:
    newt=np.random.rand(1)*(maxT)


  if v in succs:
    c,_,_=map(list,zip(*succs.get(v,[])))
  else:
    c=[]
  c.append(v)
  c=np.array(c)
  oldll=np.sum((np.log(sa)+sb)[c])
  otime=times[v]
  nsa=np.copy(sa)
  nsb=np.copy(sb)
  removeV(v,times,succs,nsa,nsb)
  nll,la,lb=getLL(v,times,newt,preds,succs,nsa,nsb,maxT,onUsers=c)

  alpha=nll-oldll
  if otime>=maxT:
    prt=0.1
  else:
    prt=0.9*(1.0/(maxT))
  if newt>=maxT:
    pnt=0.1
  else:
    pnt=0.9*(1.0/(maxT))

  alpha+=np.log(prt)-np.log(pnt)
  x=np.random.rand(1)
  if x<np.exp(alpha):
    times[v]=newt
    np.copyto(sa,np.array(la))
    np.copyto(sb,np.array(lb))
  else:
    times[v]=otime
  #print(newt,alpha,times)



def mh(graph,infections,maxT,burnin=1000,nbEpochs=1000000,ref=None,period=1000):
   names,_,_ = graph
   preds,succs=tme10.getPredsSuccs(graph)
   nbNodes=len(names)
   times=np.array([maxT]*nbNodes,dtype=float)
   for x,t in infections:
     times[x]=t
   variables=np.array(range(nbNodes),dtype=int)[times==maxT]
   nbInf=np.array([0]*nbNodes)
   ll,sa,sb=tme10.compute_ll(times,preds,maxT)
   nbt=0
   for epoch in range(nbEpochs):
     for i in np.random.choice(variables,len(variables),replace=False):
        move(i,times,preds,succs,sa,sb,maxT)

     if epoch>=burnin:
       nbInf=nbInf+np.where(times<maxT,1,0)
       nbt+=1
     if (epoch-burnin)%period==0:
       ll,sa,sb=tme10.compute_ll(times,preds,maxT)
       rate=nbInf/(1.0*nbt)
       print(epoch,rate if epoch>burnin else " burnin time", "MSE = "+str(np.sum(np.power(rate-ref,2))) if ref :
   return rate

### ATTENTION C'EST TRES LONG !!!
ref=tme10.getProbaMC(graph,[0],maxT,nbsimu=100)
rate=mh(graph,[(0,0)],maxT,burnin=100,ref=ref,period=50000)
print("------")
print(f"{rate=}")
```

```
100  burnin time MSE = 1.7226  ll= -1.5059712919558215
/var/folders/r1/pj4vdx_n4_d_xpsb04kzf97r0000gp/T/ipykernel_33721/4260472198.py:35: DeprecationWarning: Conversio
n of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single ele
ment from your array before performing this operation. (Deprecated NumPy 1.25.)
  times[v]=newt
```

```
50100 [1.         0.7351853  0.17093658 0.3698326  0.15117698 0.06319874
 0.13649727 0.05041899 0.550009   0.31877362 0.34465311] MSE = 0.06209647341326489  ll= -5.913531445757639
100100 [1.         0.75704243 0.19616804 0.39735603 0.17606824 0.0800292
 0.14315857 0.06561934 0.57236428 0.3397066  0.36591634] MSE = 0.03499369424087582  ll= -6.515330019406871
150100 [1.         0.76758155 0.2048653  0.41026393 0.18241212 0.08353944
 0.14758568 0.06851288 0.58185612 0.3600376  0.38869074] MSE = 0.024950552972078127  ll= -9.917569133798311
200100 [1.         0.77675612 0.20566397 0.41351293 0.18204909 0.08764456
 0.14811926 0.07212964 0.59088705 0.36449318 0.39197304] MSE = 0.023313757759404568  ll= -1.5059712919558215
250100 [1.         0.77742089 0.21978312 0.42324231 0.19661521 0.09606362
 0.14859941 0.07952768 0.59288963 0.35905856 0.38550646] MSE = 0.018137466775537202  ll= -3.8798280975836055
300100 [1.         0.77479075 0.21606595 0.42060526 0.19213269 0.09376302
 0.1492595  0.07809641 0.59155803 0.36070546 0.38590871] MSE = 0.019507414126829076  ll= -5.833430645657625
350100 [1.         0.77948634 0.22794506 0.43020734 0.20190228 0.100314
 0.15042243 0.08317691 0.59558687 0.36396182 0.38967317] MSE = 0.014760828261290253  ll= -4.060132498700488
400100 [1.         0.77609306 0.22328444 0.42460894 0.198292   0.09605476
 0.15128462 0.0796148  0.59145602 0.35560661 0.38102905] MSE = 0.018022581826869482  ll= -6.212302114412958
450100 [1.         0.77154495 0.21805507 0.41976573 0.19355513 0.09312202
 0.15005744 0.07791538 0.58852091 0.34961478 0.37379695] MSE = 0.021410607650539773  ll= -4.833840052469784
500100 [1.         0.77563445 0.22728355 0.42716315 0.2020056  0.0975198
 0.1512457  0.08080984 0.59106282 0.35311529 0.37872124] MSE = 0.017163506938313994  ll= -3.8651172157658835
550100 [1.         0.77685677 0.22844868 0.42844468 0.2036269  0.09708891
 0.15100518 0.08031622 0.59170074 0.35217391 0.37730477] MSE = 0.016996555706163066  ll= -3.426339945748547
600100 [1.         0.77676204 0.23174961 0.43082262 0.20697632 0.10201316
 0.15125308 0.08602986 0.59202401 0.35518941 0.37962937] MSE = 0.015077783396298464  ll= -4.871460727639844
650100 [1.         0.77733419 0.23797656 0.43472241 0.2124689  0.10493676
 0.15108438 0.08815525 0.59286832 0.35483945 0.37896865] MSE = 0.013531995620295298  ll= -3.938779152563864
700100 [1.         0.77698746 0.2434868  0.43735223 0.2175854  0.10807127
 0.15151978 0.09020416 0.59220201 0.35410949 0.37875089] MSE = 0.012340134879472308  ll= -11.43485649735892
750100 [1.         0.77596963 0.24211434 0.43654075 0.21637438 0.10659052
 0.15160646 0.08879588 0.59164588 0.35433553 0.37900349] MSE = 0.012629665544980722  ll= -5.732771652701642
800100 [1.         0.77639778 0.24145595 0.4361232  0.21564973 0.10647362
 0.15181981 0.08872739 0.59233051 0.35809205 0.38286202] MSE = 0.012087451739980831  ll= -8.835411897489571
850100 [1.         0.77818967 0.24766677 0.44043831 0.22171503 0.10969634
 0.15185982 0.09139519 0.59354165 0.35977605 0.38490778] MSE = 0.010293343565962964  ll= -4.704349356958716
900100 [1.         0.7798858  0.25085528 0.44233395 0.22493308 0.10941543
 0.15317539 0.09136879 0.59376045 0.36006738 0.38599957] MSE = 0.00954844977195489  ll= -6.576471998827114
950100 [1.         0.78146549 0.25283342 0.44413848 0.22682397 0.11016831
 0.1538051  0.09176201 0.59559937 0.36258593 0.38833643] MSE = 0.008784760974368939  ll= -5.362944065678735
------
rate=array([1.        , 0.78146549, 0.25283342, 0.44413848, 0.22682397,
       0.11016831, 0.1538051 , 0.09176201, 0.59559937, 0.36258593,
       0.38833643])
```

In [ ]:

In [ ]: