Samy Bouchnaif
Valentin Dervieux
3A TSP

Pour le lundi 4 janvier 2016

Modélisations Statistiques & Applications

Compte-rendu de travaux dirigés n°2 – Détection de communautés, Graph embedding & Propagation d'épidémie

Au préalable, installons le package nécessaire à notre étude (la librairie *igraph*), et chargeons cette librairie :

```
library (igraph)
```

I - Détection de communautés :

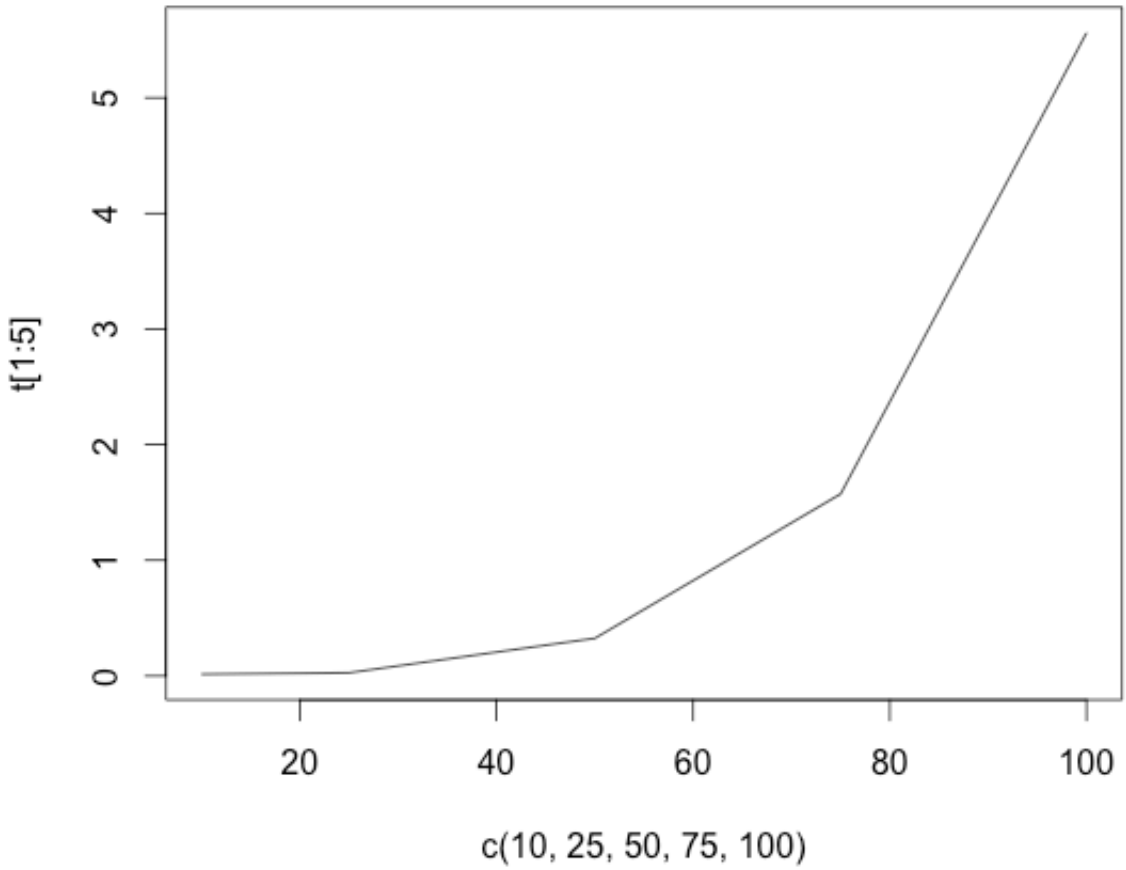
a) Observation de la complexité de calcul de deux algorithmes de détection de communautés (de notre choix), en utilisant les fonctions de génération de graphes aléatoires (vues dans le précédent TD) :

Nous prenons la probabilité égale à 0,01 (une probabilité plus importante engendre une détermination, pour chaque graphe, de cluster_optimal extrêmement longue ; pour une probabilité de 0,1, le calcul de cluster_optimal(g3) n'aboutit toujours pas au bout de 30 minutes).

```
g1<-erdos.renyi.game(10,0.01)
g2<-erdos.renyi.game(25,0.01)
g3<-erdos.renyi.game(50,0.01)
g4<-erdos.renyi.game(75,0.01)
g5<-erdos.renyi.game(100,0.01)
```

1er algorithme - *cluster_optimal* :

```
t<-c(0,0,0,0,0)
t[1]<-system.time(cluster_optimal(g1))[3]
t[2]<-system.time(cluster_optimal(g2))[3]
t[3]<-system.time(cluster_optimal(g3))[3]
t[4]<-system.time(cluster_optimal(g4))[3]
t[5]<-system.time(cluster_optimal(g5))[3]

plot(c(10,25,50,75,100), t[1:5], type='l')
```



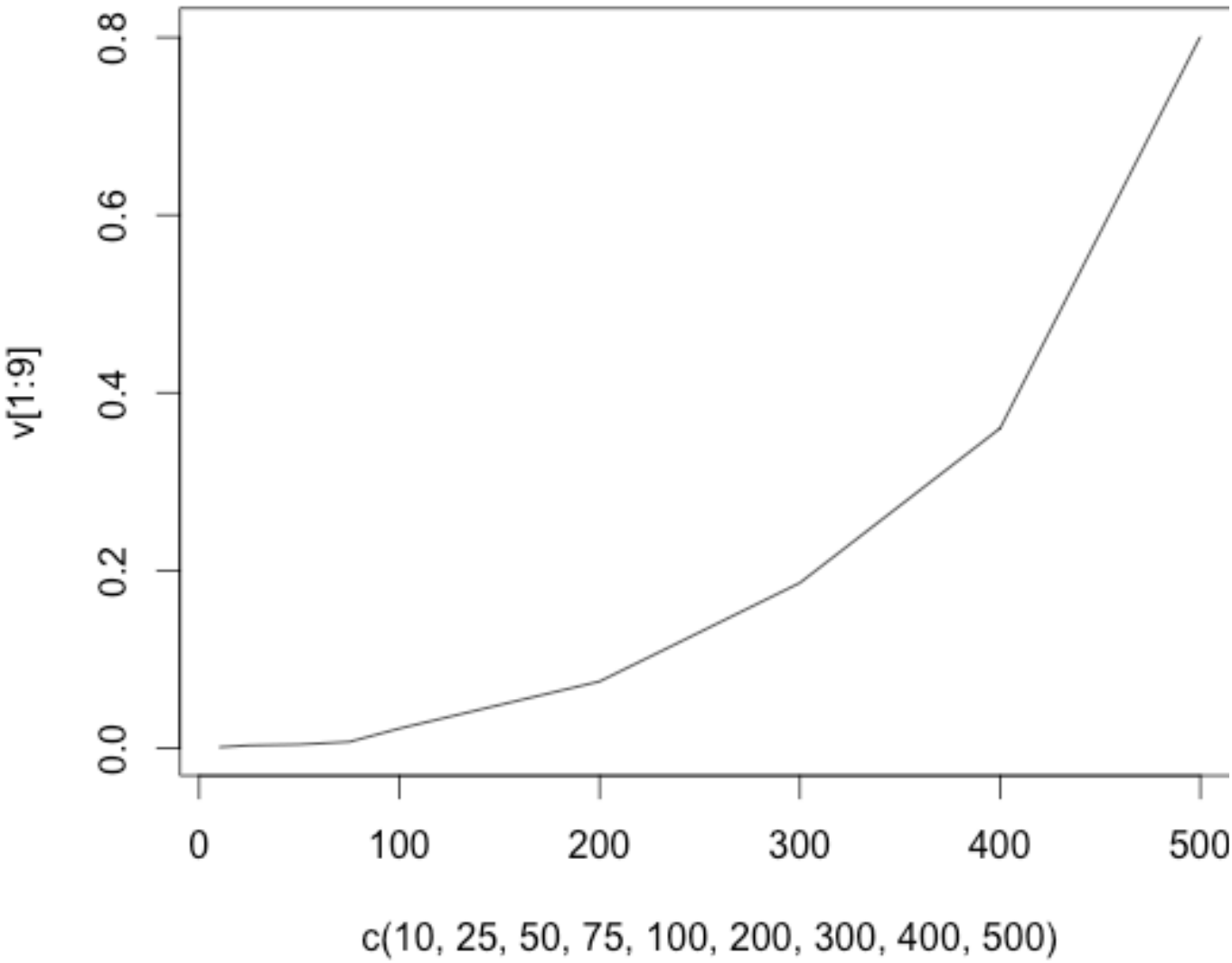
2e algorithme – *cluster_infomap* (pour le 2ème algorithme, nous avons étendu la plage du nombre de nœuds pour plus de visibilité sur le comportement de la complexité) :

```
g1<-erdos.renyi.game(10,0.01)
g2<-erdos.renyi.game(25,0.01)
g3<-erdos.renyi.game(50,0.01)
g4<-erdos.renyi.game(75,0.01)
g5<-erdos.renyi.game(100,0.01)
g6<-erdos.renyi.game(200,0.01)
g7<-erdos.renyi.game(300,0.01)
g8<-erdos.renyi.game(400,0.01)
g9<-erdos.renyi.game(500,0.01)
```

```
v<-c(0,0,0,0,0,0,0,0,0)
v[1]<-system.time(cluster_infomap(g1))[3]
v[2]<-system.time(cluster_infomap(g2))[3]
v[3]<-system.time(cluster_infomap(g3))[3]
v[4]<-system.time(cluster_infomap(g4))[3]
v[5]<-system.time(cluster_infomap(g5))[3]
```

```
v[6]<-system.time(cluster_infomap(g6))[3]
v[7]<-system.time(cluster_infomap(g7))[3]
v[8]<-system.time(cluster_infomap(g8))[3]
v[9]<-system.time(cluster_infomap(g9))[3]
```

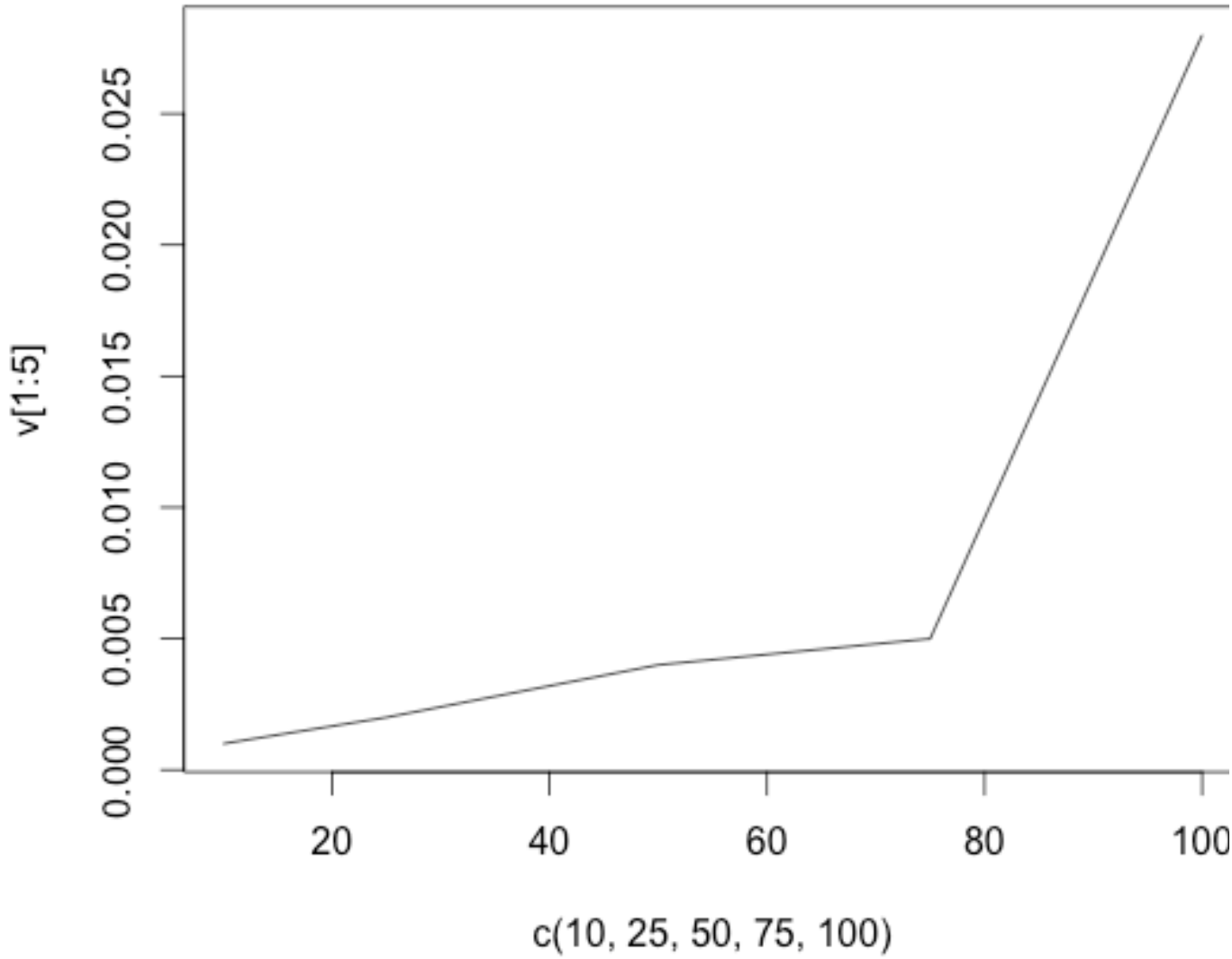
```
plot(c(10,25,50,75,100,200,300,400,500), v[1:9], type='l')
```



Comparaison des complexités (temps de compilation) entre les deux méthodes :

Pour nos deux méthodes, en augmentant le nombre de nœuds, nous remarquons une croissance exponentielle de la complexité (du moins cela s'apparente à une croissance exponentielle). La croissance exponentielle est rapidement apparente pour la méthode *cluster_optimal.* En revanche, pour la méthode *cluster_infomap*, cette croissance exponentielle n'est pas de suite

apparente. En effet, lors de l'étude de la complexité sur une plage allant de 0 à 100 nœuds, nous remarquions un comportement variable de l'évolution de la complexité, comme nous pouvons le constater ici :



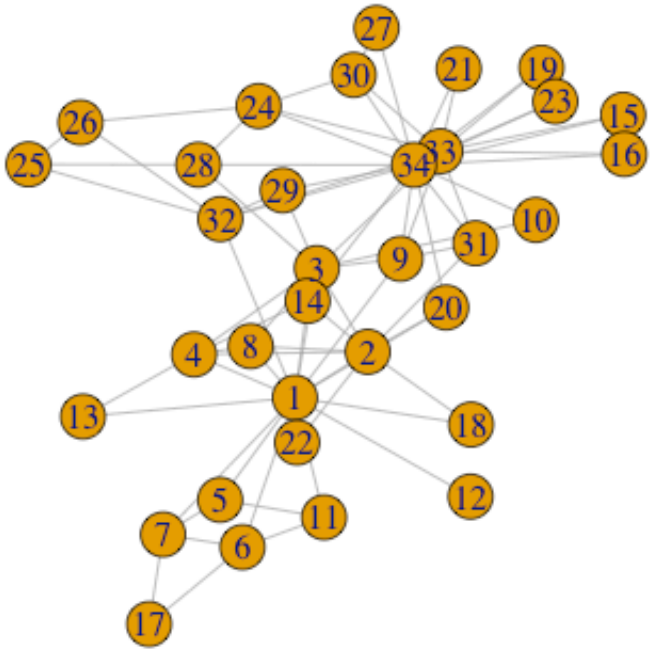
Toutefois, une fois arrivé à 100 nœuds, nous constatons par la suite un comportement s'apparentant à une croissance exponentielle pour la complexité de la méthode *cluster_infomap* : elle s'aligne sur la méthode *cluster_optimal*. C'est la raison pour laquelle nous avons fait le choix d'étendre notre plage de valeur en abscisse (nombre de nœuds). Nous avons dépassé les 100 nœuds pour la méthode *cluster_infomap* dans le but d'illustrer ce comportement également exponentiel (tardif certes, mais bien exponentiel une fois que nous dépassons les 100 nœuds) de la complexité.

De ce fait, bien que nous ayons mené notre étude seulement sur deux communautés, nous sommes amenés à penser que les algorithmes de détection de communautés (en particulier ceux que nous avons utilisé ici, aux complexités

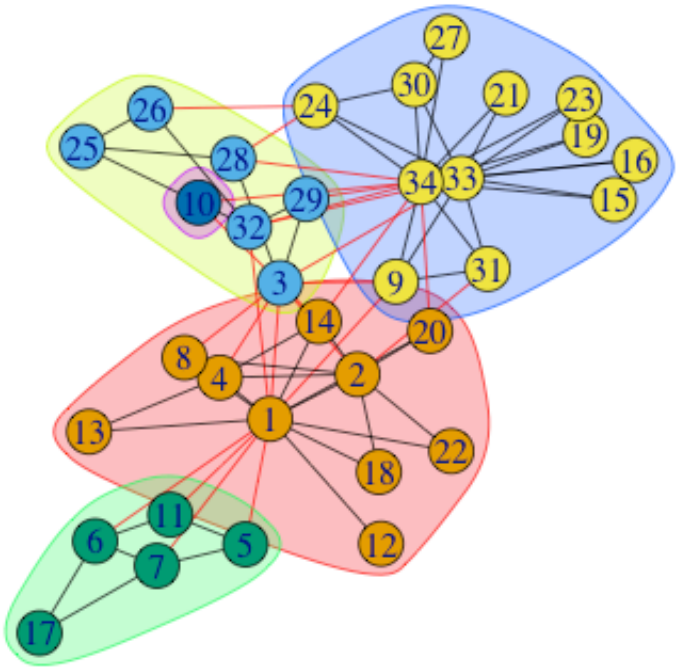
exponentielles) sont très coûteux en terme de complexité. Il va de soi qu'une méthode dont la complexité restera faible (et, comme cerise sur le gâteau, peu variable selon la quantité de nœuds) sera certainement préférée. Le problème est que dès que notre graphe a un nombre de nœuds excédant 200 nœuds, la méthode *cluster_optimal* est très (trop ?) gourmande en complexité : je suis contraint de laisser R tourner assez longtemps (parfois en vain) pour obtenir le comportement de la complexité. Pour *cluster_infomap* en revanche, nous obtenons le comportement de la complexité sans problème si nous prenons même jusqu'à 500 nœuds (après, cela se complique, en particulier si votre graphe *erdos.renyi.game* est de probabilité importante (donc un gain en complétude) ou si votre graphe tend vers un graphe Small-world, ce que nous allons remarquer au travers des analyses que nous apportons par la suite).

b) Utilisation d'un graphe fameux fourni dans *igraph* (graph.famous("Zachary")) et étude des deux algorithmes précédemment choisis (en utilisant la modularité comme mesure classique d'évaluation des communautés) :

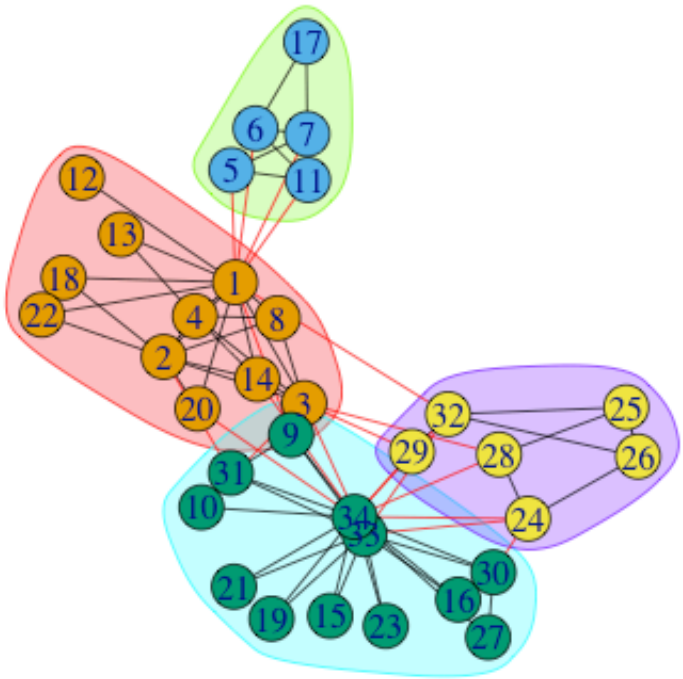
```
g<-graph.famous("Zachary")
plot(g)
```




```
c1<-cluster_edge_betweenness(g)
c1
plot(c1, g)
```




```
c2<-cluster_optimal(g)
plot(c2, g)
```




```
m<-membership(c1)
modularity(g, m)
```

```
[1] 0.4012985
```

```
m2<-membership(c2)
```

modularity(g, m2)

[1] 0.4197896

Rappel : la modularité est une mesure pour la qualité d'un partitionnement des nœuds d'un graphe, ou réseau, en communautés. Elle est principalement utilisée en analyse des réseaux sociaux. C'est aussi une fonction d'optimisation pour certaines tâches de détection de communautés dans les graphes. Le principe est qu'un bon partitionnement d'un graphe implique un nombre d'arêtes intra-communautaires important et un nombre d'arêtes inter-communautaires faible. Cette modularité est décrite comme la proportion des arêtes incidentes sur une classe donnée moins la valeur qu'aurait été cette même proportion si les arêtes étaient disposées au hasard entre les nœuds du graphe. La modularité prend ses valeurs entre -1 et 1 inclus. On peut considérer qu'un graphe a une structure de communautés significative quand une partition obtient un score de modularité supérieur à 0,3.

Commentaires (sur les graphes et sur la modularité) : pour rappel, la fonction *cluster_edge_betweenness* permet de distinguer des communautés selon la mesure dite « d'entre-deux » d'une arête ; mesure qui caractérise la capacité d'une arête à être située le long des chemins les plus courts entre toutes les paires de noeuds. Pour ce qui est de la fonction *cluster_optimal*, elle permet également de distinguer des communautés en calculant les structures des communautés optimales d'un graphe (pour ce faire, elle maximise la mesure de modularité sur toutes les partitions possibles).

En conséquence, nos deux algorithmes précédemment choisis, de par leur bonne modularité, peuvent être considérés comme assez performants dans une optique de distinction de communautés. En effet, *cluster_edge_betweenness* et sa modularité de 0.4012985 n'ont rien à envier à *cluster_optimal* et sa modularité de 0.4197896 : *cluster_optimal* est certes de « meilleure » modularité, mais la différence reste très légère et surtout dérisoire, d'autant plus que par définition de *cluster_optimal*, nous sommes déjà censés obtenir une bonne modularité. Du coup, nous pouvons considérer que *cluster_edge_betweenness* nous offre également de belles performances en terme de modularité (surtout pour une fonction dont le but premier n'est pas de maximiser la modularité, mais qui arrive à approcher la modularité d'une fonction qui a cet objectif de maximisation de la modularité). Attention toutefois à la limite de résolution de la modularité en règle générale. En effet, si l'on est confronté à des communautés de tailles différentes à l'intérieur d'un même graphe, certaines communautés, même bien définies, pourront ne pas être

distinguées dans la partition de modularité optimale.

c) Chargement de nos données et construction d'un graphe tel que les nœuds représentent les acteurs et qu'une arête existe entre deux nœud si et seulement si les deux acteurs ont joué ensemble dans un film :

Pour charger nos données : RStudio -> Tools -> Import Dataset -> From Text File…, et on choisit le fichier « imdb_actor_edges.tsv ». Pour « imdb_actors_key.tsv » (comme nous souhaitons séparer nos colonnes par des tabulations):

```
imdb_actors_key<-
read.table("/Users/sam/Desktop/TD/imdb_actors_key.tsv", sep="\t",
header=T)
```

Nous créons ensuite le graphe en ne prenant que les deux premières colonnes des données *imdb_actor_edges*, au même titre que la deuxième partie du précédent TD :

```
g<-graph_from_data_frame(imdb_actor_edges[,1:2], directed=F)
```

d) Analyse des communautés de ce graphe en utilisant si besoin un sous-ensemble du graphe résultant (ainsi que la première question) :

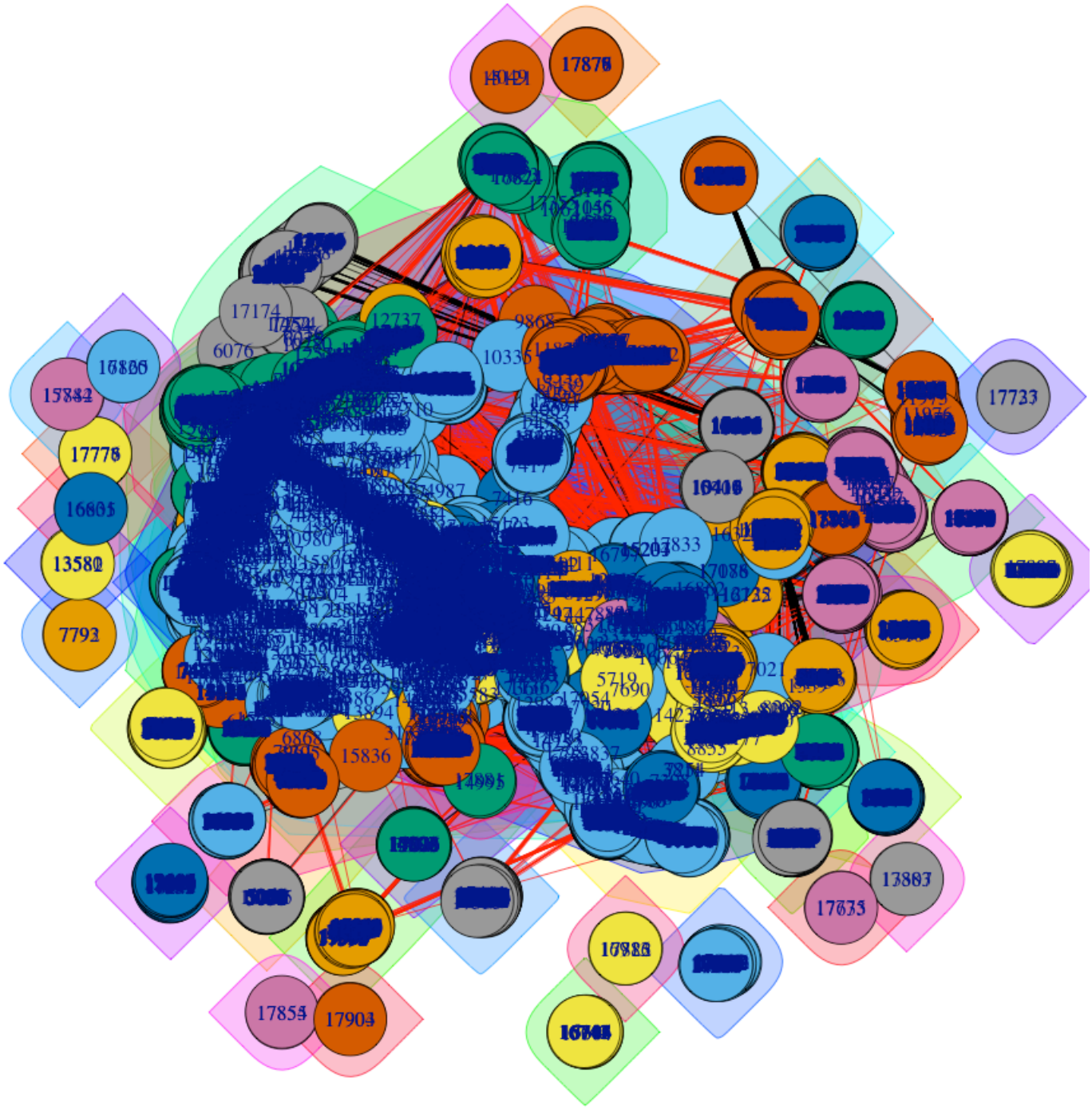
Notre approche utilise *cluster_louvain* pour distinguer des communautés « fortes » en terme de contribution à la modularité du graphe g, ainsi que *membership* pour mesurer l'adhésion de chaque nœud à la structure de la communauté :

```
c<-cluster_louvain(g)

#Cette fonction va plus loin que la fonction cluster_optimal. En
#effet, en plus de distinguer des communautés en calculant les
#structures des communautés optimales d'un graphe, elle est
#également basée sur une approche hiérarchique. Pour faire bref,
#elle fait de la modularité « multi-niveau » : initialement, un nœud
#est assigné à une communauté lui étant propre, mais au fur et à
#mesure de l'ajout des autres nœuds et arêtes, chaque nœud est
#déplacé à la communauté avec laquelle il contribue de manière
#maximale à la modularité du graphe. De ce fait, nous sommes
#censés récupérer un graphe doté d'une excellente modularité.
```

```
m<-membership(c)

#Traduit sous la forme d'un vecteur numérique l'adhésion de
#chaque nœud à la structure de la communauté.

modularity(g, m)

[1] 0.8308406

#Nous nous retrouvons bien avec une excellente modularité :
#0.8308406.
```

Analyse : à l'aide de la fonction *cluster_louvain*, nous distinguons des communautés qui contribuent toutes à obtenir une excellente modularité du graphe (par définition même de cette fonction). Par ailleurs, dans *cluster_louvain*, lorsqu'il n'y a plus aucun sommet qui puisse être réaffecté à une communauté (selon le principe vu précédemment dans l'explication de *cluster_louvain*), chaque communauté est considérée tel un sommet à part entière (pour l'algorithme de *cluster_louvain*), et le processus recommence avec les communautés fusionnées. Le processus se termine lorsqu'il n'y a qu'un seul sommet restant ou lorsque la modularité ne cesse de croître au cours d'une une étape. En guise d'illustration, voici ce que donne la commande *plot(c,g)* :

```
plot(c,g)
```



Remarque : des communautés semblent être « dominantes » comme les communautés « bleu ciel ». Pour le reste, difficile de distinguer d'autres communautés majoritaires : on voit plusieurs nœuds oranges foncés, verts, gris, ainsi que des communautés qui semblent minoritaires (les communautés « orange clair », « rose », « jaune », etc). Mais encore une fois, vu le nombre de nœuds et le maillage, il est difficile d'avoir un regard précis sur notre graphe. Quoi qu'il en soit, nous pouvons tout de même dire qu'il est très dense, en particulier au niveau de son cœur, et vu la modularité (0.8308406), nous sommes en présence de communautés « fortes » : elles se distinguent très bien les unes

des autres et offrent une partition du graphe d'une très grande qualité.

```
length(m)
```

```
[1] 17577
```

```
m[1:20]
```

```
#Nous n'extrayons qu'une partie du vecteur m, et nous allons en
#guise d'exemple voir l'appartenance de deux acteurs d'ID
#différents à des communautés semblables (donc partageant la
#même valeur numérique dans le vecteur m). Disons les acteurs
#d'ID 3423 et d'ID 11385 qui partagent tous deux la valeur 25 dans
#ce « sous-vecteur » de m (donc ayant au moins une communauté
#en commun ; communauté caractérisée par la valeur 25 dans le
#vecteur m).
```

```
17776  5578  1835 14790  8606 12376 13981 17378 14371  9504  5072  1523  3382  2300   402  3423
    4    28    10    12    14    51    26    32    37    21     8    41    26    10    26    25
11385  6198  2235 11494
   25    10    21    10
```

```
#À présent, déterminons leurs clés respectives.
```

```
which(imdb_actors_key[,1] == 3423)
```

```
[1] 16154
```

```
which(imdb_actors_key[,1] == 11385)
```

```
[1] 10642
```

```
#Voyons qui sont ces deux acteurs qui partagent au moins une
#même communauté.
```

```
imdb_actors_key[c(16154,10642), ]
```

```
         ID           name movies_95_04 main_genre
16154  3423   Milder, Andy           18      Drama
10642 11385 Malinger, Ross           12     Comedy
                                                                                genres
16154 Animation:2,Comedy:1,Documentary:1,Drama:4,Fantasy:3,Sci-Fi:4,Thriller:1
10642                  Comedy:4,Family:2,Fantasy:1,Mystery:1,NULL:3,Romance:1
```

Commentaires : ce résultat nous permet d'obtenir les noms des acteurs concernés (Andy Milder et Ross Malinger), ainsi que leurs genres (les genres des films dans lesquels ils ont tourné). De ce fait, nous pouvons voir que ces deux acteurs appartiennent tous deux aux genres « comédie », et « fantastique ».

En effet, les communautés ne sont pas caractérisées par un seul et même genre, ce serait trop fastidieux de faire ainsi étant donné que nombre d'acteurs appartiennent à plusieurs genres. D'ailleurs, de cette manière, nous perdrions nettement en modularité sur le graphe, ce qui est contraire à *cluster_louvain*. En effet, rappelons que lors de la construction des communautés dans ce programme, au fur et à mesure de l'ajout des autres nœuds et arêtes, chaque nœud est déplacé à la communauté avec laquelle il contribue de manière maximale à la modularité du graphe. Les communautés peuvent réunir plusieurs genres différents, là n'est pas le problème. Le programme *cluster_louvain* cherche justement le meilleur compromis entre communautarisme de genre et maximisation de modularité, quitte à fusionner des communautés au fur et à mesure. En clair, ce résultat obtenu sur les acteurs Ross Malinger et Andy Milder est parfaitement logique au regard du principe de *cluster_louvain*.

e) Utilisation du fichier complémentaire imdb_actors_key.tsv afin d'interpréter les résultats : nous avons déjà utilisé ce fichier complémentaire dans les questions précédentes afin de mieux interpréter nos résultats.

II – Graph embedding :

a) Choix d'une communauté de taille inférieure stricte à 1000 nœuds parmi celles précédemment trouvées :

```
c<-cluster_louvain(g)
c
sizes(communities = c)
```

```
Community sizes
   1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18   1
 207  338  133    3    8    4  181   43  101 1710   39  249  132  994   86  276   88  308  41
  20   21   22   23   24   25   26   27   28   29   30   31   32   33   34   35   36   37   3
  11 1245   37   61  312 1061 3982  252  640   94  482    2  218    2   33    7    3  345 144
  39   40   41   42   43   44   45   46   47   48   49   50   51   52   53   54   55
 201    2  439    2   16   22   23    2    2    2    3   66 1240    3    2    2    2
```

#Au regard de la taille des communautés (les 3 couples de lignes #correspondent chacun au couple communauté en haut et taille #indiquée juste en-dessous de la communauté), comme il nous en #faut une de taille inférieure stricte à 1000, nous allons choisir la #3ème (de taille 133), et nous allons l'appeler u pour la suite du #problème.

```
u<-c[3]
```

Faisons une brève étude des objets manipulés :

```
class(c)
```

[1] "communities"

```
attributes(u)
```

$names
[1] "3"

```
attributes(c)
```

$names
[1] "membership" "memberships" "modularity" "names" "vcount" "algorithm"

$class
[1] "communities"

Extrayons le contenu des noeuds de la communauté n°3 dans une liste :

```
new_liste <-u[[1]]
length(new_liste)
```

[1] 133

```
class(new_liste[1])
```

```
[1] "character"
```

Nous obtenons donc une liste de type *String*, que nous cherchons à convertir en liste d'entiers :

```
new_liste_num<-strtoi(new_liste, base = 0L)
length(new_liste_num)
```

```
[1] 133          #Elle garde la même longueur, jusqu'ici tout va bien.
```

À présent, nous allons créer un vecteur de booléens qui cherche à déterminer les indices du tableau traduisant si une arête fait partie de la communauté n°3 :

```
bool_good_indices<-(imdb_actor_edges[,1] %in% new_liste_num)
& (imdb_actor_edges[,2] %in% new_liste_num)
```

De même, étudions ce nouvel objet :

```
length(bool_good_indices)
```

```
[1] 287074
```

```
nrow(imdb_actor_edges)
```

```
[1] 287074
```

```
class(bool_good_indices)
```

```
[1] "logical"
```

```
bool_good_indices[1]
```

```
[1] FALSE
```

```
as.data.frame(table(bool_good_indices))
```

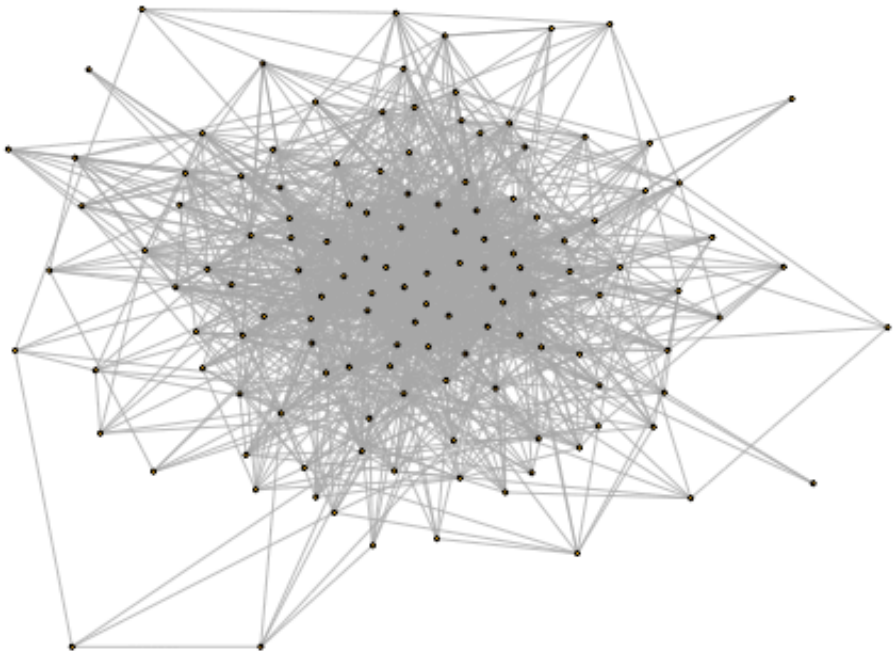
```
  bool_good_indices   Freq
1             FALSE 285661
2              TRUE   1413
```

Créons un nouveau tableau pour tracer le graphe à partir de la communauté extraite :

```
tableau=c()
for (i in 1:nrow(imdb_actor_edges)) {
   if (bool_good_indices[i]==TRUE) {
      tableau<-rbind(tableau, imdb_actor_edges[i,])
   }
}
tableau          #Vu sa longueur, je vais vous épargner son affichage.
```

Et enfin, exécutons la commande permettant d'obtenir le graphe de notre communauté extraite :

```
g_imdb<-graph_from_data_frame(tableau[,1:2], directed=FALSE)
plot(g_imdb, vertex.size=1, vertex.label=NA, asp=FALSE)
```



b) Application de l'algorithme *laplacian eigenmap* sur le graphe résultant :

Calculons à présent les valeurs propres et vecteurs propres de la matrice d'adjacence (matrice d'adjacence qui s'obtient, pour information, par le biais de la commande *get.adjacency(g_imdb)*) :

```
g_laplacien_eigenmaps<-embed_laplacian_matrix(g_imdb,2)
g_laplacien_eigenmaps
```

Cette fonction effectue la décomposition spectrale du laplacien du graphe de notre communauté extraite. Le résultat des valeurs propres et vecteurs propres étant trop long à afficher, nous nous focalisons directement sur la projection résultante en dimension 2 à la question suivante (d'où le choix, dans cet algorithme, de l'entier 2 en argument, pour la dimension de la projection spectrale). Présentons seulement ici le résultat des valeurs propres :

```
$D
[1] 64.69622 63.86249
```

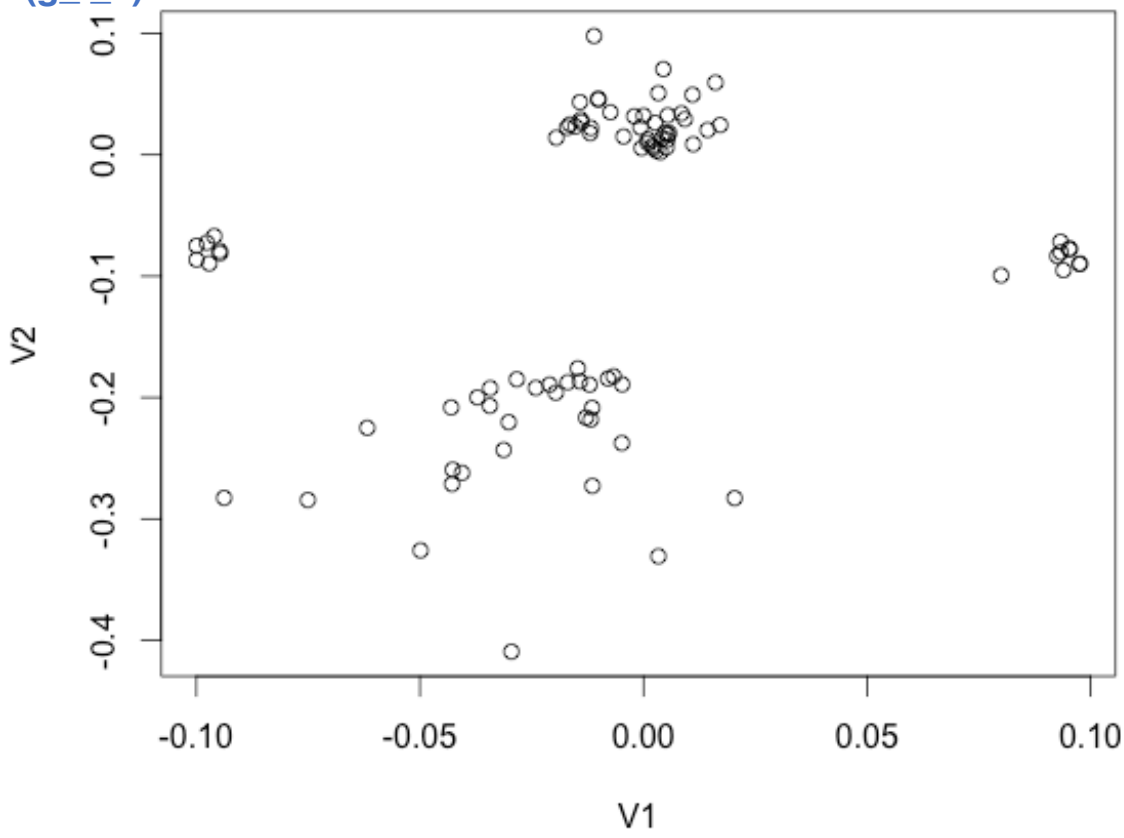
Après plusieurs autres calculs, nous remarquons que cet algorithme fait le choix des vecteurs propres dont les valeurs propres sont les plus grandes, un peu dans le même esprit que pour les analyses en composantes principales et factorielle des correspondances.

c) Visualisation de la projection résultante en dimension 2 :

```
g_l_e<-g_laplacien_eigenmaps[[1]]
g_l_e<-as.data.frame(g_l_e)
g_l_e<-g_l_e[(abs(g_l_e$V1)<0.1 & abs(g_l_e$V2<0.1)),]
```

Nous avons choisi 0.1 pour justement séparer les valeurs trop grandes. Dans un souci de visibilité, nous avons supprimées les valeurs trop écartées.

```
plot(g_l_e)
```



Remarques :

- Rappelons avant toute chose que la matrice laplacienne L est obtenue par le biais de l'opération L = D – A, où D est la matrice diagonale des degrés (di,i est égal au degré du sommet i ; soit le nombre de liens adjacents au sommet i), et A la matrice d'adjacence (ai,j valant 1 si (i,j) est une arête du graphe, 0 si cette arête n'existe pas dans le graphe).

- Le choix de la dimension 2 est motivé par le fait que nous souhaitions trouver un espace de dimension plus faible que la dimension originale de telle sorte que les nœuds proches dans l'espace originel restent proches dans l'espace réduit, et que les distances dans l'espace originel soient à peu près préservées (en vertu des propriétés du cours « Tools for the analysis of structure and information diffusion in graphs », page 23).

Analyse : notre fonction embed_laplacian_matrix réalise une représentation euclidienne en deux dimensions basée sur la matrice laplacienne L du graphe de notre communauté extraite. Cette représentation est déterminée par le biais de la décomposition en valeurs singulières de la matrice laplacienne L (décomposition spectrale effectuée précédemment). Nous souhaitons une représentation en 2 dimensions. De ce fait, nous allons prendre les deux vecteurs propres dont les valeurs propres sont les plus grandes (ceux de valeurs propres **63.86249 et 64.69622**) ; un lambda-vecteur propre f vérifiant : L*f = lambda*D*f. Ces deux vecteurs propres définissent des axes invariants pour l'application définie par la matrice L, donc un espace où cette application est « plus simple » (où la structure de A est plus simple). En conséquence, la partition obtenue est plus fidèle aux données empiriques que celle provenant d'une classification hiérarchique. Sans faire de rappel de cours, il est primordial pour notre algorithme de préserver les proximités de premier et second ordre, ainsi que l'information à grande échelle de notre réseau projeté (c'est plus ou moins de cette manière que nous pourrons construire une métrique et utiliser des notions de distance comme évoquées dans les remarques ci-dessus).

Ainsi, avec V1 et V2 qui sont nos deux vecteurs propres qui servent d'axe à notre repère en 2 dimensions, les points représentés sont les nœuds (auxquelles nous associons des coordonnées dans notre représentation graphique). Nous remarquons quatre grappes (ou classes) de nœuds, et une meilleure différenciation des nœuds comparé au *plot* du réseau (question a)), bien que la partie centrale-supérieure de cette représentation (ainsi que la partie gauche et droite, dans une moindre mesure) met en exergue des amas de nœuds. Cette

différenciation se voit surtout dans la partie inférieure de notre représentation graphique.

En conséquence, nous pouvons peut-être classer les nœuds de cette représentation en quatre catégories, ce qui offre une meilleure différenciation que le dernier *plot* du réseau. Et nous pouvons également remarquer dans la grappe située dans la partie inférieure de notre représentation que les nœuds sont plus éparpillés (donc une différenciation au sein-même d'une seule grappe).

Il serait tout de fois judicieux de pousser l'étude plus loin en analysant des caractéristiques de nos classes, qu'elles soient intra-classes (la compacité), inter-classes (la séparabilité), ainsi que les graphes associés à ces classes comme les graphes intrinsèques et les graphes de pénalité (l'analyse d'autres critères pourrait également nous aider). Cela pourrait nous permettre de mieux comprendre notre représentation graphique, et de mieux caractériser les classes qui partitionnent (ou du moins semblent partitionner pour notre exemple de projection en 2D) le graphe de notre communauté extraite.

V1 et V2 étant les vecteurs propres de valeurs propres maximales, pourrions-nous en déduire que les sous-communautés représentées ici en 2D sont les sous-communautés « dominantes » du graphe de notre communauté extraite ?

Rien n'est moins sûr. En effet, prudence est de rigueur ; cette projection en 2D a ses limites : elle ne représente pas l'ensemble des nœuds, mais cette approche nous est familière avec l'analyse en composantes principales et l'analyse factorielle des correspondances. De plus, l'utilisation du spectre du graphe est prometteuse, mais certains inconvénients peuvent être soulignés. En effet, il s'agit d'une méthode qui ne semble que peu adaptée pour comparer des graphes, étant donné que plusieurs graphes peuvent partager le même spectre, et une variation minime de la structure du graphe peut engendrer de très fortes variations du spectre correspondant (cela se voit assez bien après essai sur R). Le choix de la matrice laplacienne à utiliser est également discutable, car il faut également prendre en compte le fait que plus la distribution des degrés est hiérarchisée, plus les matrices laplaciennes (dépendantes de ces-mêmes degrés) diffèrent. Nos résultats obtenus par cette méthode de partitionnement utilisant le spectre du graphe sont donc à nuancer.

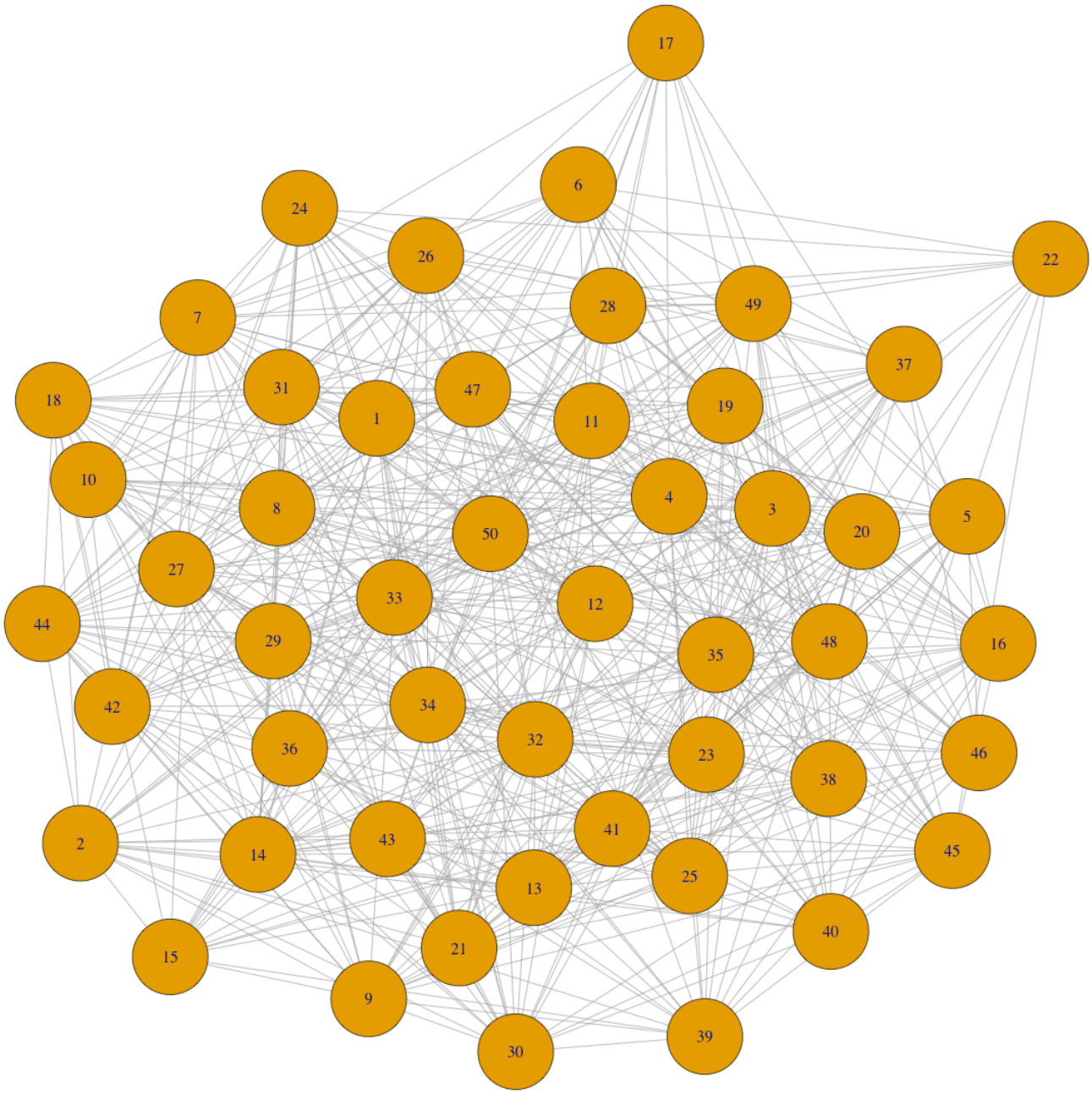
III – Propagation d'épidémie :

Dans cet exercice, pour les notations, nous allons juste remplacer p_0 par p, γ par y, β par b, et t par temps.

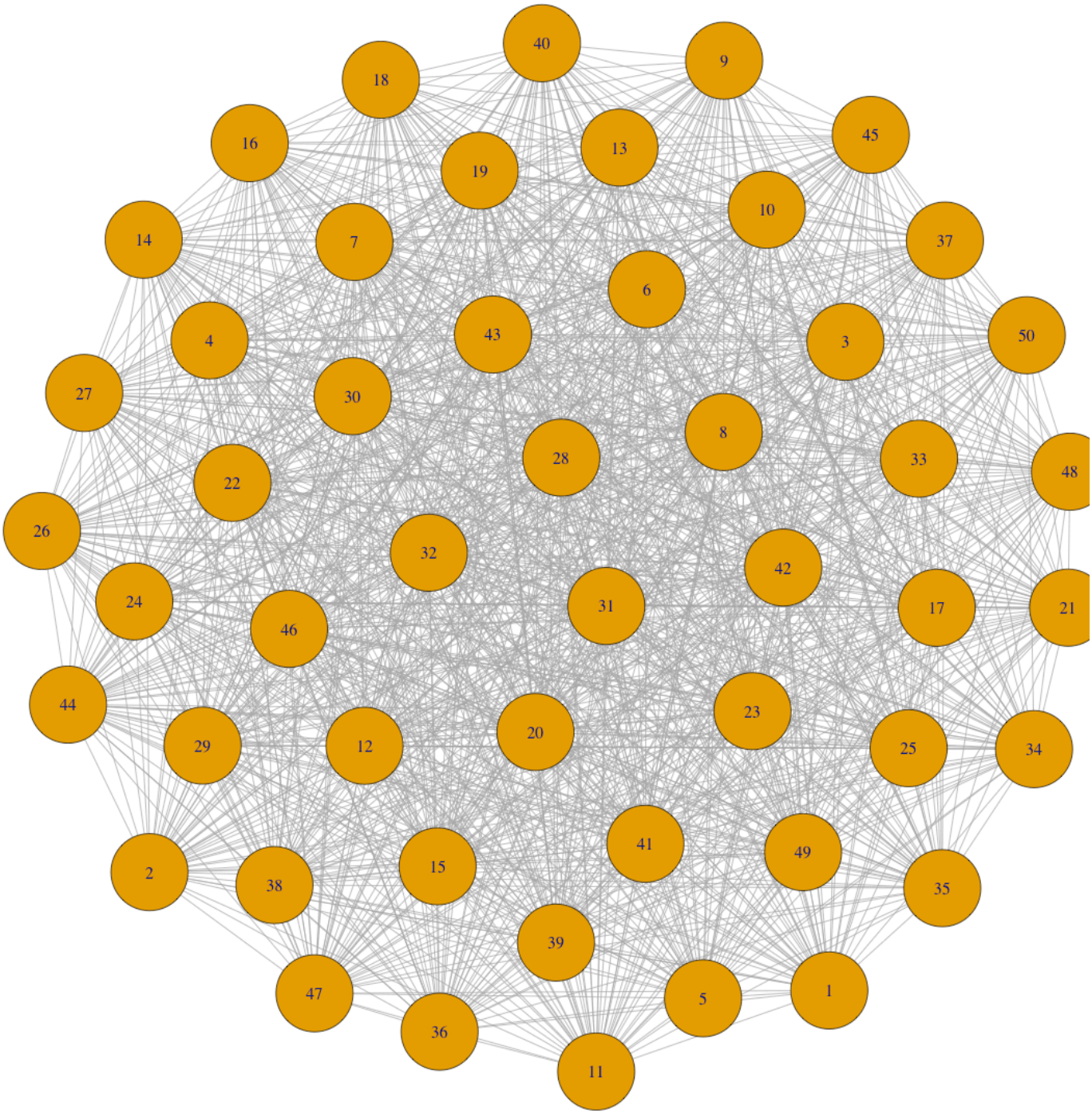
a) Choix de trois types de réseaux : nous faisons le choix *Erdös-Rényi, Graphe complet, & Watts–Strogatz.* Affichage de la proportion de nœuds infectés au cours du temps (cumulatif) dans le cadre de l'étude de la propagation, puis étude de l'influence du type de réseau à paramètres fixés.

```
n<-1000
p<-0.3
y<-0.7
b<-0.2
temps<-100
g1<-erdos.renyi.game(n,0.4)
g2<-erdos.renyi.game(n,1) #Équivalent à un graphe complet
g3<-sample_smallworld(1, n, 5, 0.05)

#Ces graphes n'étant pas du tout clairs en raison du nombre de
#nœuds et le maillage très complexe que cela induit, les plots de g1,
#g2, et g3 ont été réalisé avec seulement 50 nœuds, pour que les
#caractéristiques de chacun de ces graphes soient plus visibles.
#Pour la suite en revanche, nous repassons à 1000 nœuds.
```

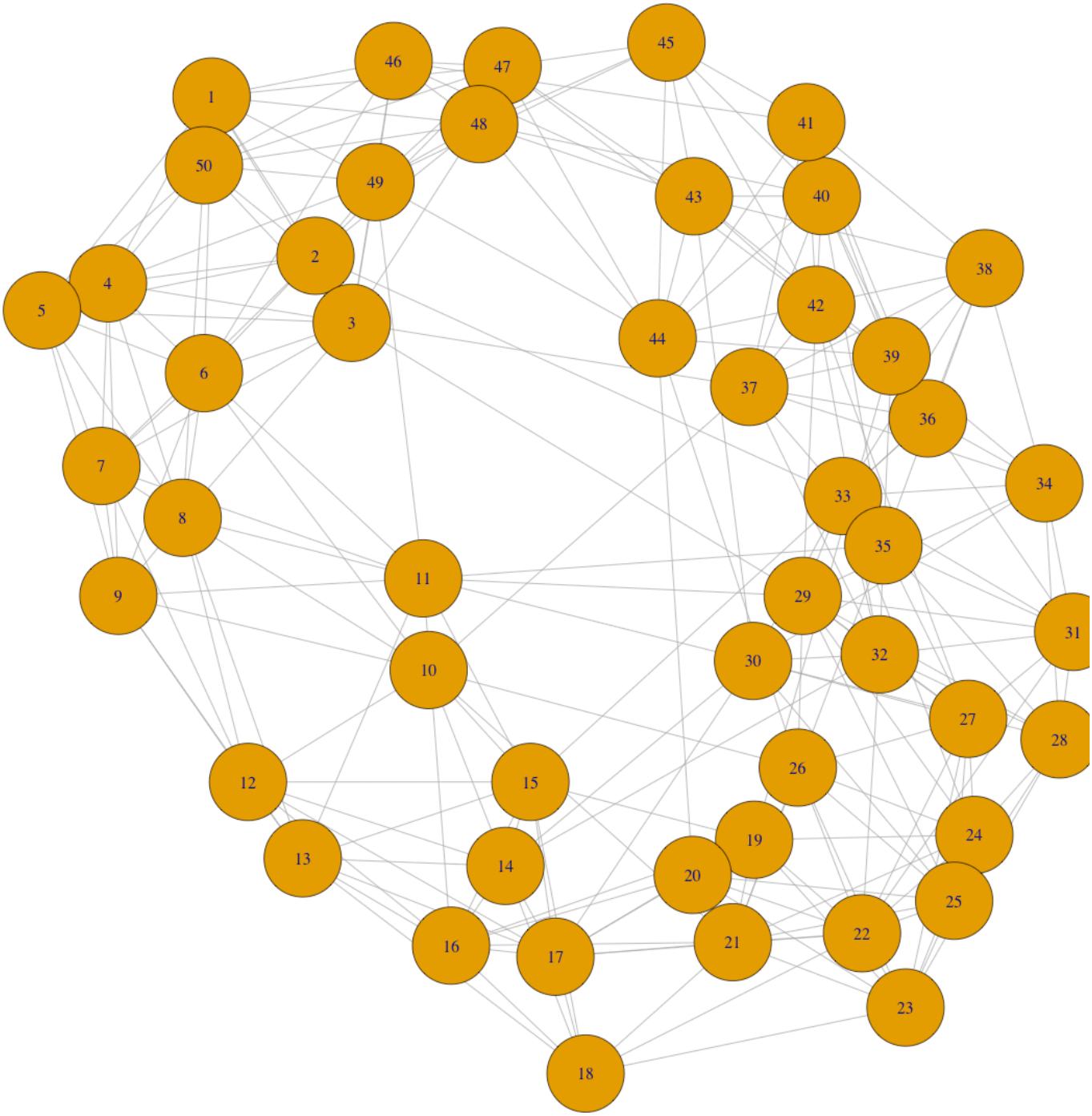
plot(g1)


17
6
24
26
22
7
28
49
37
18
31
47
1
11
19
10
4
3
20
5
8
50
27
33
12
44
29
35
48
16
42
34
32
23
46
36
38
2
14
43
41
45
25
13
40
15
21
9
39
30

plot(g2)


40
9
18
45
16
13
19
10
14
7
37
6
4
43
3
50
27
30
8
28
33
48
22
26
32
42
24
31
17
21
46
44
23
29
12
20
25
34
41
49
2
38
15
35
39
47
5
1
36
11

```
plot(g3)
```




```
m<-matrix (0, nrow=100, ncol=n)
colnames(m)<-1:n
m[1,]<-sample(c(0,1), n, replace=T, prob=c(1-p, p))
#m
dim(m)
```

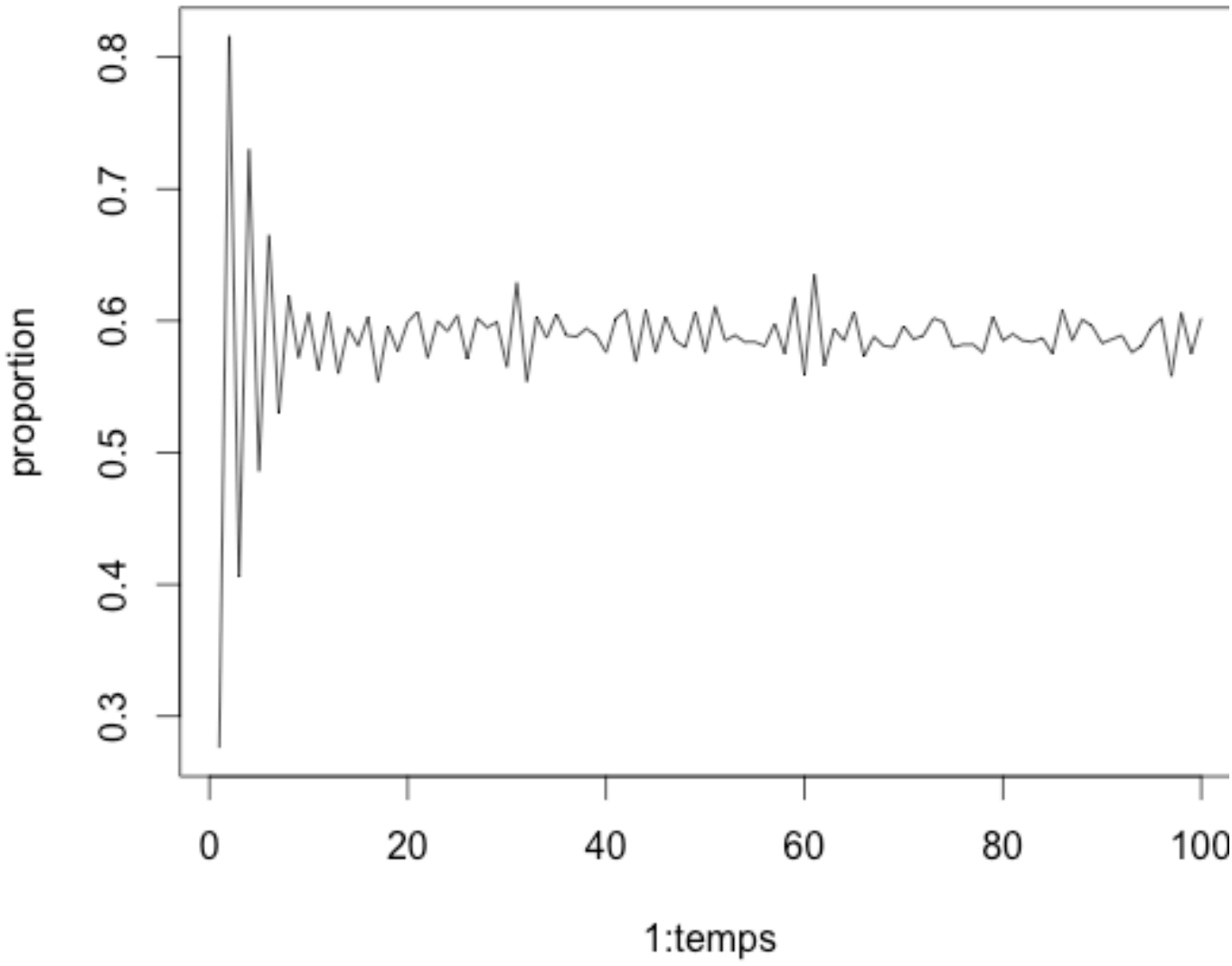
```
[1]  100 1000
#Pour étudier la proportion de nœuds infectés au cours du temps,
#Nous appliquons l'algorithme suivant pour chaque g (g1, g2, g3).

for(t in 1:(temps-1)) {

for (i in 1:n) {

if(m[t,i]==1){

voisins<-neighbors(g, v=i)
for(j in voisins){
if( m[t,j]==0){
if ( runif(1) < b){
m[t+1,j]<-1
}
}
}

if (runif(1) >y){
m[t+1,i]<-1
}
}
}
}
m

proportion<-rowSums(m)/n
proportion
plot(1:temps,proportion, type='l')
plot(density(proportion))
```

Pour g1 :

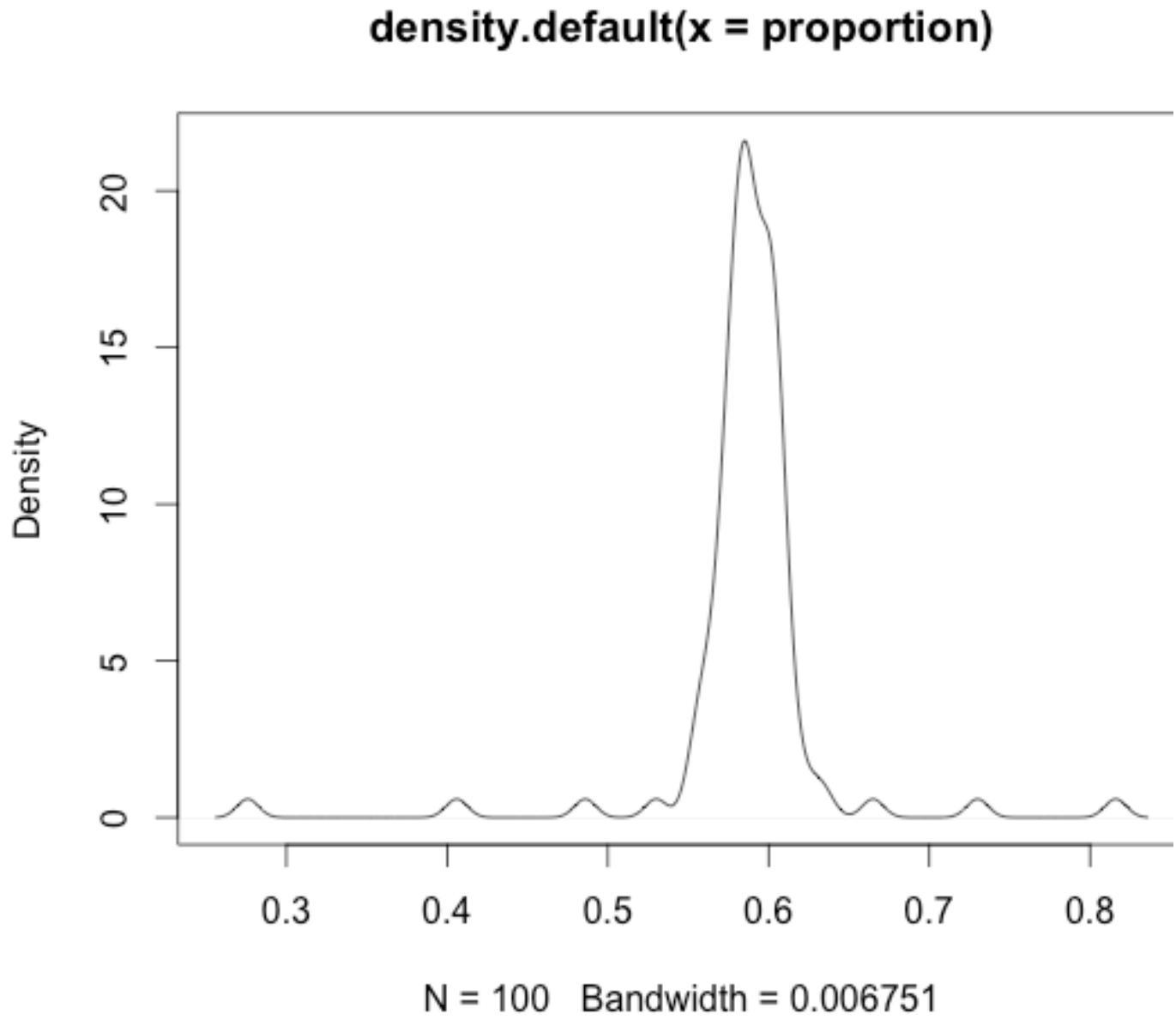
proportion

```
 [1] 0.276 0.816 0.406 0.730 0.486 0.665 0.530 0.619 0.572 0.606 0.562 0.607 0.560 0.595 0.58
[16] 0.603 0.554 0.596 0.577 0.599 0.607 0.572 0.600 0.592 0.604 0.571 0.602 0.595 0.599 0.56
[31] 0.629 0.554 0.603 0.587 0.605 0.589 0.588 0.594 0.589 0.576 0.602 0.608 0.569 0.608 0.57
[46] 0.603 0.585 0.580 0.607 0.576 0.611 0.585 0.589 0.584 0.584 0.581 0.598 0.575 0.618 0.55
[61] 0.635 0.566 0.594 0.585 0.607 0.573 0.588 0.581 0.580 0.596 0.586 0.589 0.602 0.599 0.58
[76] 0.582 0.582 0.576 0.603 0.585 0.590 0.585 0.584 0.587 0.575 0.608 0.585 0.601 0.596 0.58
[91] 0.586 0.589 0.576 0.581 0.595 0.602 0.558 0.606 0.575 0.602
```

plot(1:temps,proportion, type='l')



```
plot(density(proportion))
```

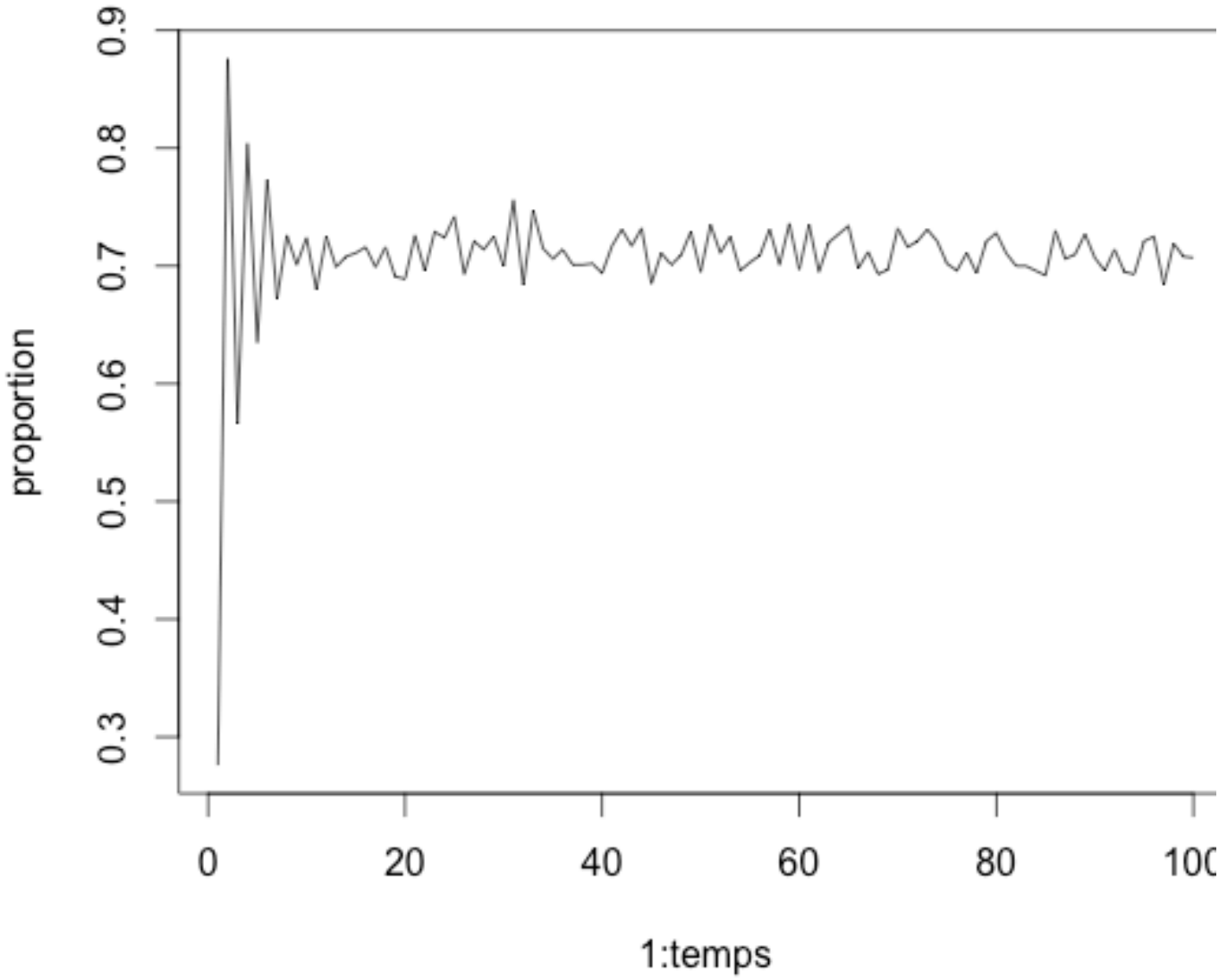


Pour g2 :

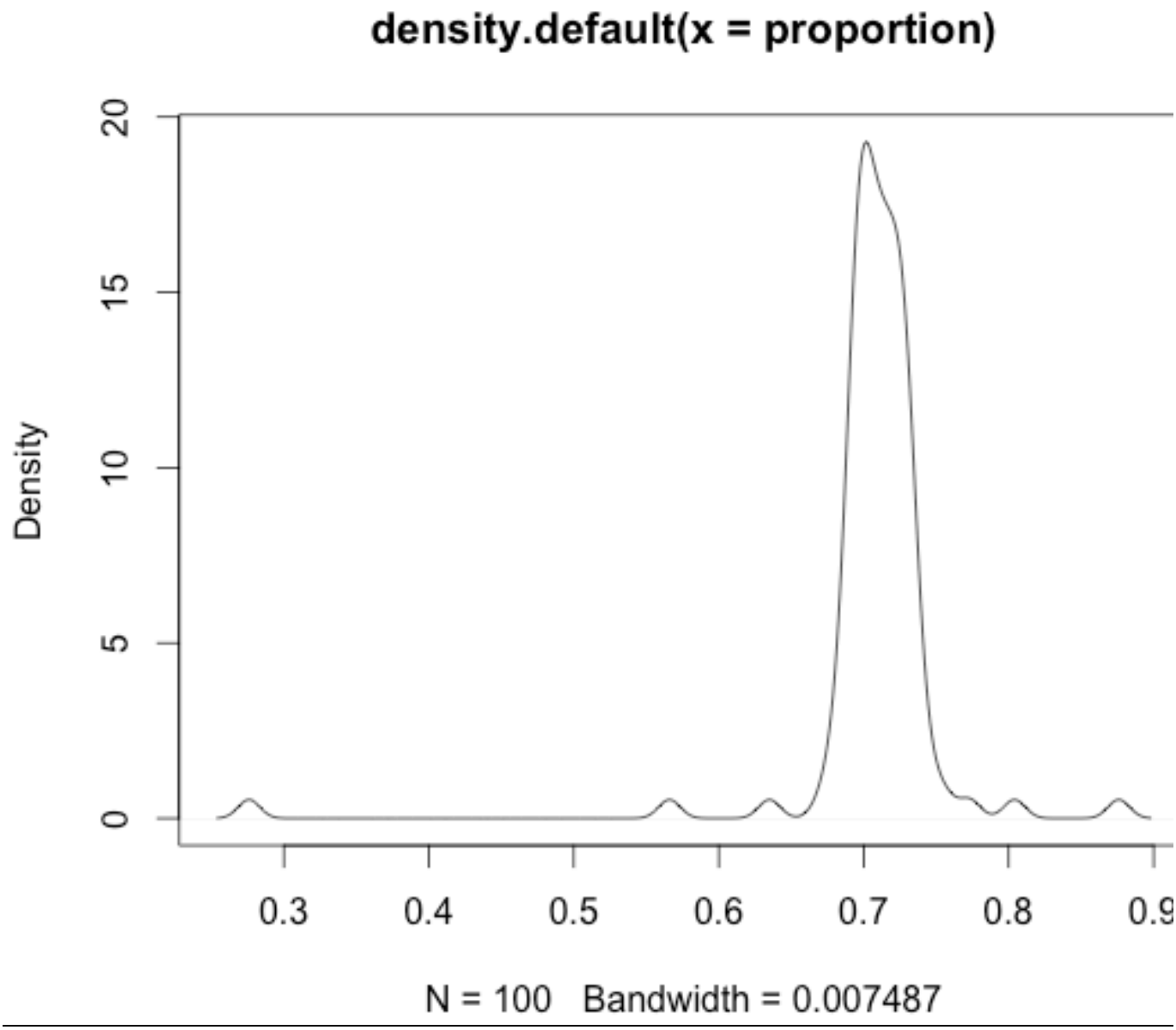
proportion

```
 [1] 0.276 0.876 0.566 0.804 0.635 0.773 0.672 0.726 0.701 0.724 0.680 0.725 0.699 0.708 0.71
[16] 0.716 0.699 0.716 0.691 0.689 0.726 0.696 0.729 0.724 0.742 0.693 0.721 0.714 0.725 0.70
[31] 0.756 0.684 0.747 0.715 0.706 0.714 0.701 0.701 0.702 0.694 0.717 0.731 0.717 0.732 0.68
[46] 0.711 0.701 0.709 0.729 0.695 0.735 0.711 0.725 0.696 0.703 0.709 0.731 0.701 0.736 0.69
[61] 0.735 0.695 0.720 0.727 0.734 0.698 0.712 0.693 0.697 0.732 0.716 0.721 0.731 0.721 0.70
[76] 0.696 0.711 0.694 0.721 0.728 0.711 0.700 0.700 0.696 0.692 0.730 0.706 0.710 0.727 0.70
[91] 0.696 0.714 0.695 0.693 0.721 0.725 0.684 0.719 0.708 0.707
```

plot(1:temps,proportion, type='l')



```
plot(density(proportion))
```

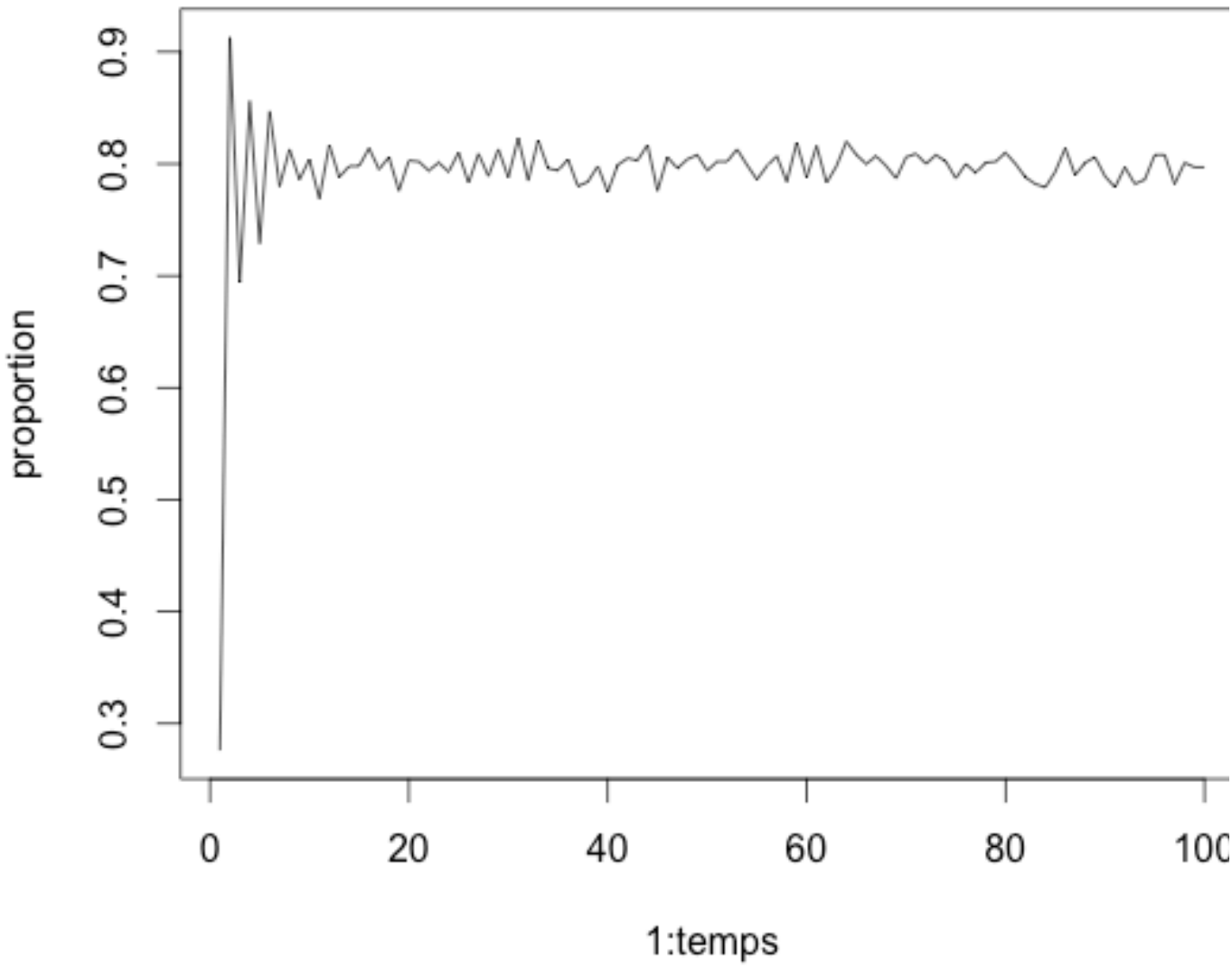


Pour g3 :

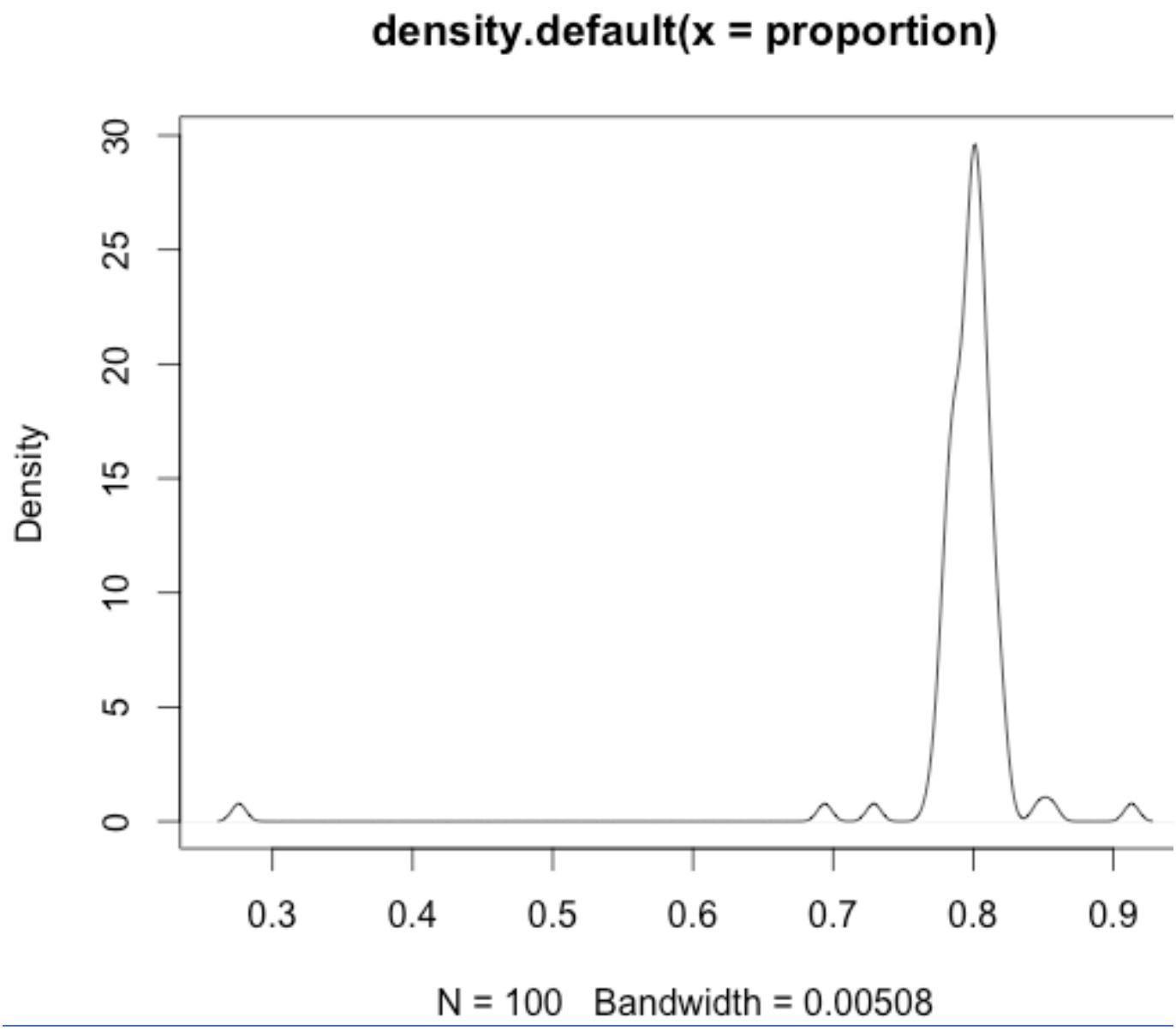
proportion

```
 [1] 0.276 0.913 0.694 0.856 0.729 0.847 0.780 0.813 0.786 0.804 0.769 0.817 0.788 0.798 0.79
[16] 0.814 0.795 0.806 0.776 0.803 0.802 0.794 0.801 0.793 0.810 0.783 0.809 0.789 0.813 0.78
[31] 0.823 0.785 0.821 0.796 0.794 0.804 0.780 0.784 0.798 0.775 0.799 0.805 0.803 0.817 0.77
[46] 0.806 0.796 0.804 0.808 0.794 0.802 0.802 0.813 0.799 0.786 0.798 0.807 0.784 0.819 0.78
[61] 0.816 0.783 0.798 0.820 0.808 0.800 0.807 0.798 0.787 0.806 0.809 0.800 0.808 0.802 0.78
[76] 0.800 0.792 0.801 0.802 0.810 0.800 0.788 0.782 0.779 0.793 0.814 0.790 0.801 0.806 0.78
[91] 0.779 0.797 0.782 0.786 0.808 0.808 0.782 0.801 0.797 0.797
```

plot(1:temps,proportion, type='l')



plot(density(proportion))



Analyse (g1 désigne le graphe Erdös-Rényi de probabilité 0.4, g2 le graphe complet (le graphe Erdös-Rényi de probabilité 1), et g3 le graphe Small-world) :

Nous remarquons que pour nos trois réseaux, la proportion de nœuds infectés explose très vite (passage de 0.13 à 0.81 pour g1, de 0.13 à 0.875 pour g2, et de 0.13 à 0.91 pour g3, le tout en moins de 2 unités de temps), puis décroît en moins de 2 unités de temps (de moitié pour g1, d'environ 1,56 fois sa valeur pour g2, et d'environ 1,3 fois sa valeur pour g3), pour ensuite (jusqu'à l'instant 'temps') osciller légèrement et relativement normalement (à 2-3 pics près relativement légers ; à 30 et 60 unités de temps pour g1 et à 30 unités de temps pour g2).

Cette oscillation se fait autour des proportions 0.58, 0.71 et 0.8 respectivement pour g1, g2, et g3. De ce fait, nous en déduisons un comportement assez constant de la proportion de nœuds infectés d'un graphe à l'autre. La différence réside toutefois dans les valeurs de ces proportions de nœuds infectés. En effet, nous tournons autour de 6 nœuds infectés sur 10 pour g1, 7 nœuds infectés sur 10 pour g2, et 8 nœuds infectés sur 10 pour g3. Ces proportions sont très importantes, et la croissance de cette proportion (en passant de g1, à g2, et à g3) semble tout à fait logique lorsqu'un graphe gagne en complétude ou lorsqu'un graphe gagne en capacité de jonction d'un nœud à l'autre par le biais d'un court chemin (peu d'arêtes ; justement ce que permet Small-world, graphe dans lequel tout nœud peut être joint par un autre nœud par le biais d'un court chemin) : en effet, le nombre de nœuds infectés risque d'être encore plus important s'ils sont tous connectés les uns aux autres, ou s'ils sont tous joignables entre eux par le biais d'un chemin (qui plus est si le chemin est court, le risque d'infection est plus grand, ce qui est parfaitement logique et facilement visible en termes probabiliste et graphique).

Nous pouvons être déçus de notre choix de réseaux, mais n'allons pas trop vite en besogne. Partant du principe que certains nœuds sont déjà infectés, les proportions obtenues ne sont pas nécessairement si élevées que cela. La déception réside surtout dans le fait que nous avons infecté « seulement » 30% de nos nœuds au départ, chaque nœud avait 70% de chance de guérir et 20% de chance de tenter d'infecter chacun de ses voisins.

Les paramètres fixés au début de cet exercice ont nécessairement leur part de responsabilité dans l'importance de cette proportion de nœuds infectés. Nous allons y voir plus clair en jouant sur ces paramètres à la question suivante. Nous n'allons toutefois pas toucher au nombre de nœuds dans chaque réseau (1000) et au temps (100).

b) Etude de l'influence des paramètres :

Dans un premier temps, nous nous efforçons de faire varier seulement le paramètre b. Ensuite nous ferons varier le paramètre y. Nous verrons plus tard si la proportion initiale de nœuds infectés p a une influence comparable à celles d'y et de b.

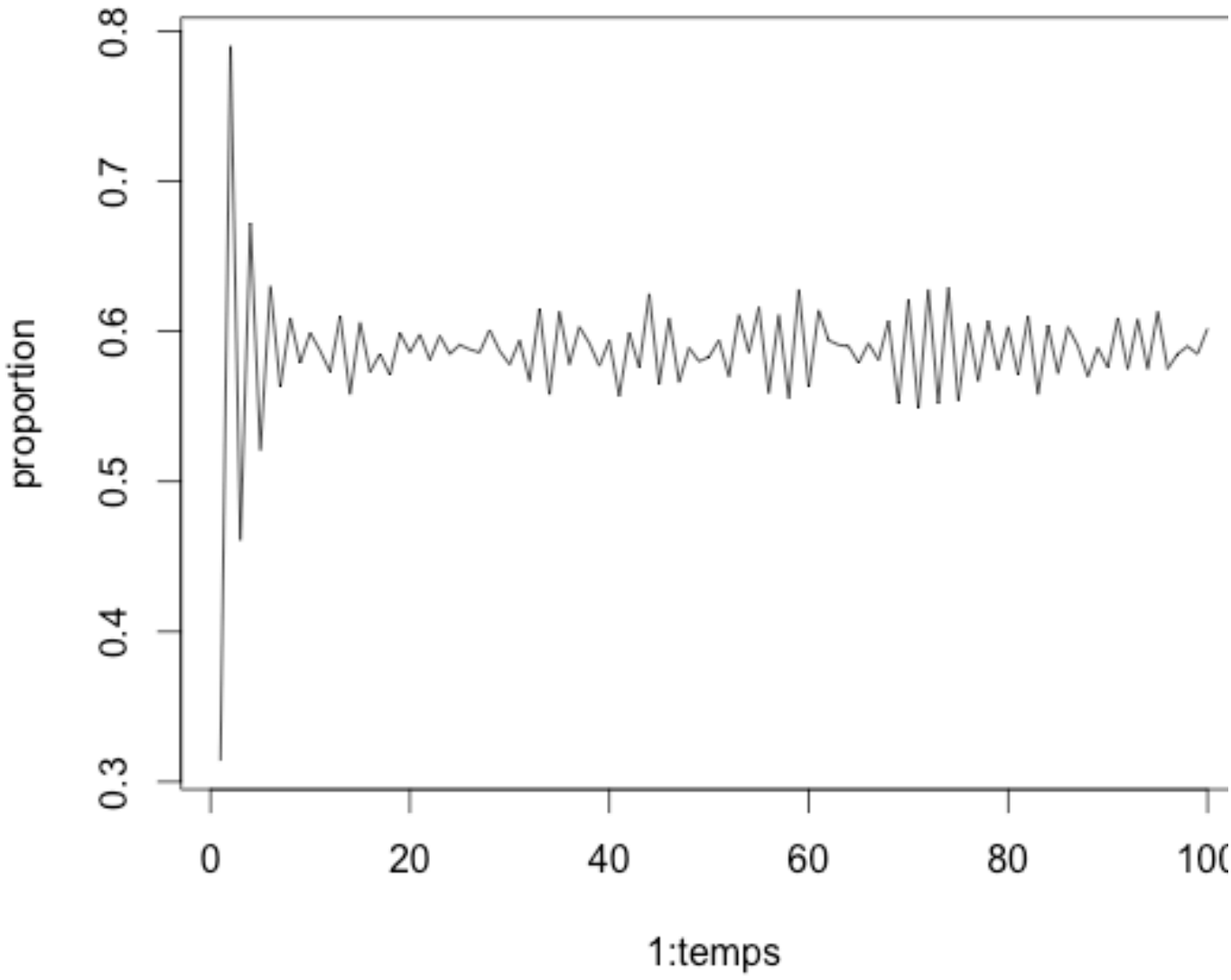
Nous réutilisons le même programme que précédemment en attribuant à b la valeur 0.05, et cela nous donne :

Pour g1 :

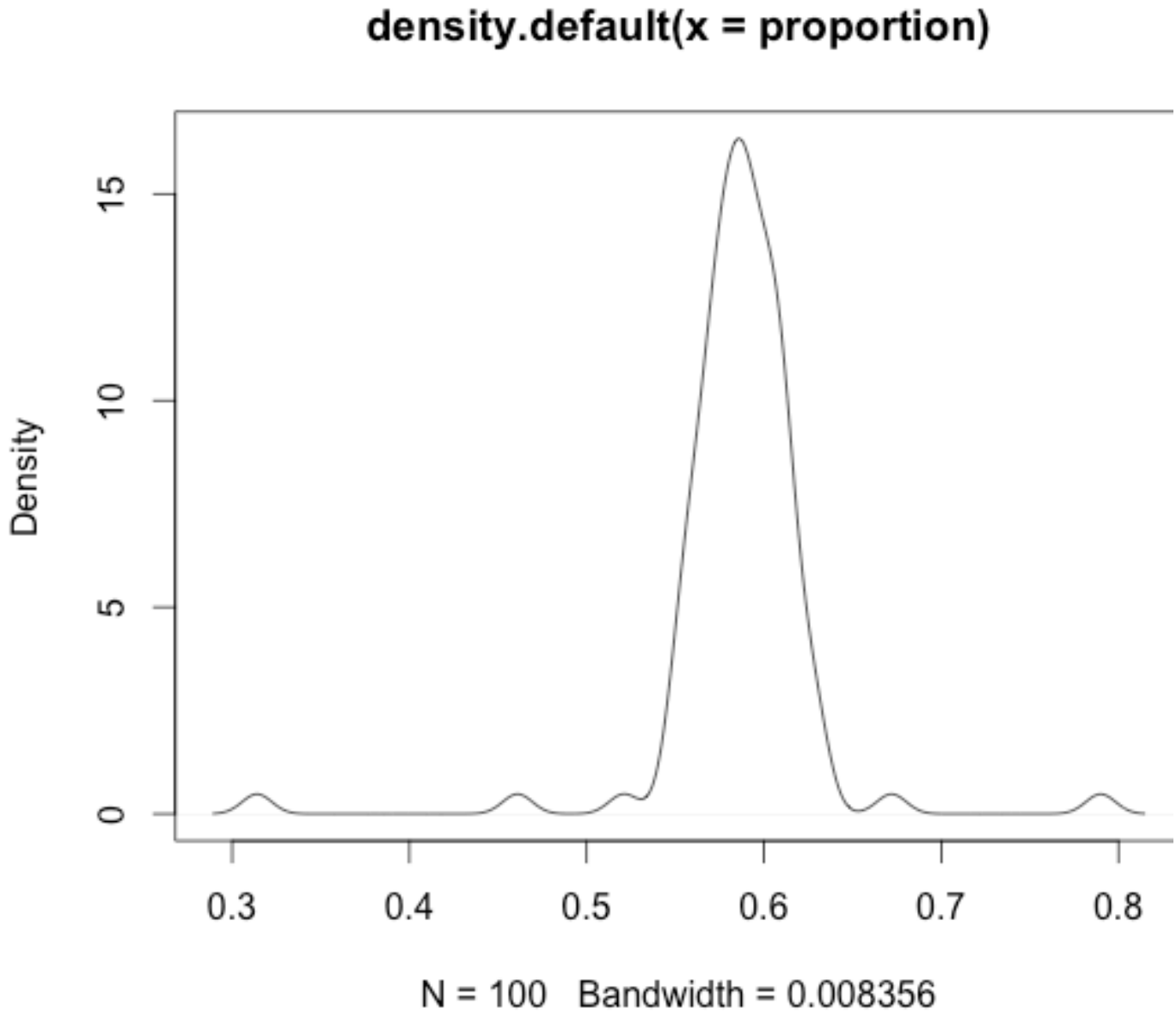
proportion

```
 [1] 0.314 0.790 0.461 0.672 0.521 0.630 0.563 0.609 0.579 0.599 0.587 0.573 0.610 0.558 0.60
[16] 0.573 0.585 0.571 0.599 0.586 0.598 0.581 0.597 0.585 0.591 0.588 0.586 0.601 0.587 0.57
[31] 0.594 0.567 0.615 0.558 0.613 0.578 0.603 0.592 0.577 0.594 0.557 0.599 0.576 0.625 0.56
[46] 0.609 0.566 0.589 0.580 0.583 0.594 0.570 0.611 0.586 0.616 0.559 0.611 0.555 0.628 0.56
[61] 0.614 0.594 0.591 0.590 0.579 0.592 0.581 0.607 0.552 0.621 0.549 0.628 0.552 0.629 0.55
[76] 0.605 0.567 0.607 0.574 0.603 0.571 0.610 0.558 0.604 0.572 0.603 0.590 0.570 0.589 0.57
[91] 0.609 0.575 0.608 0.575 0.613 0.575 0.585 0.590 0.585 0.602
```

plot(1:temps,proportion, type='l')



```
plot(density(proportion))
```

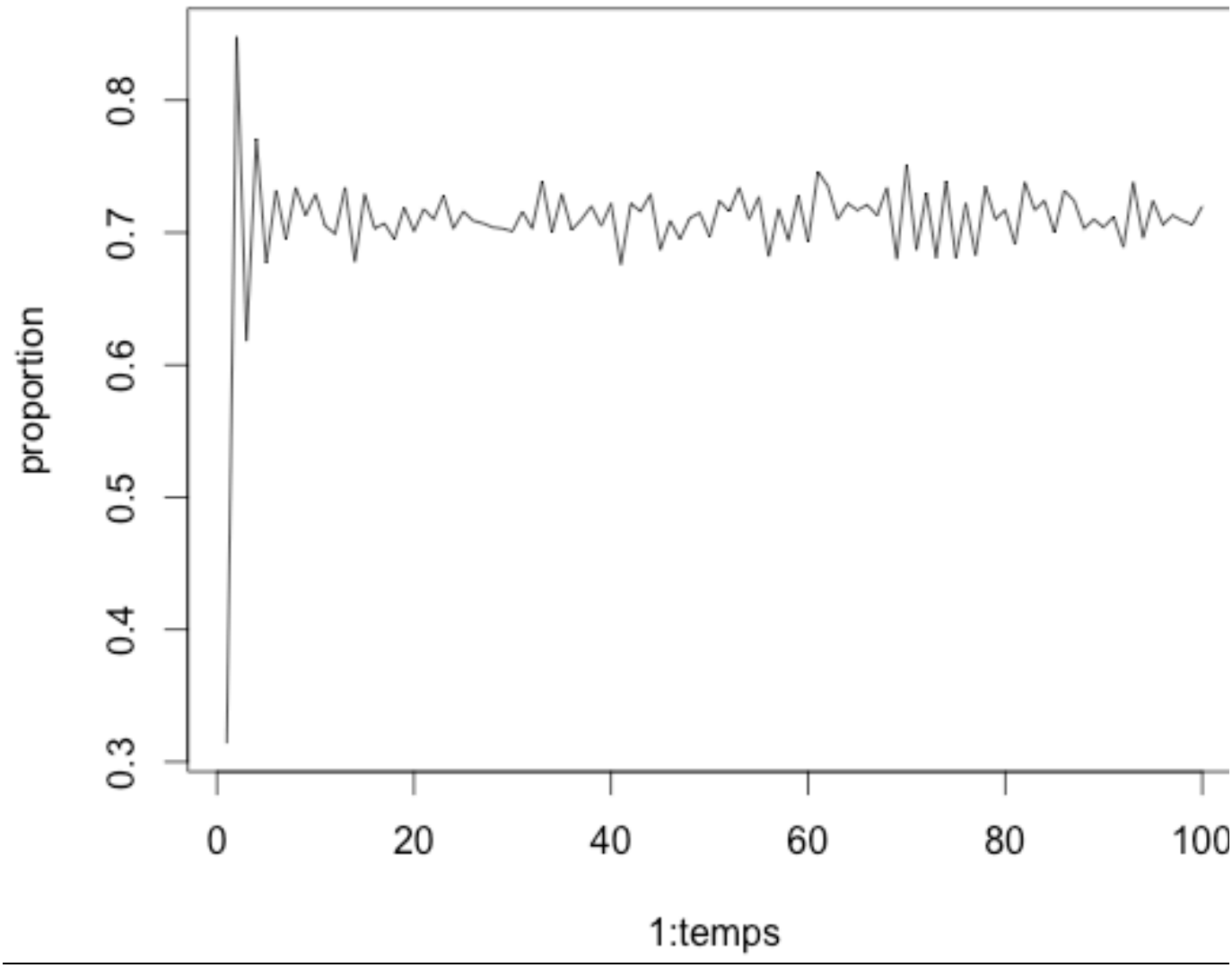


Pour g2 :

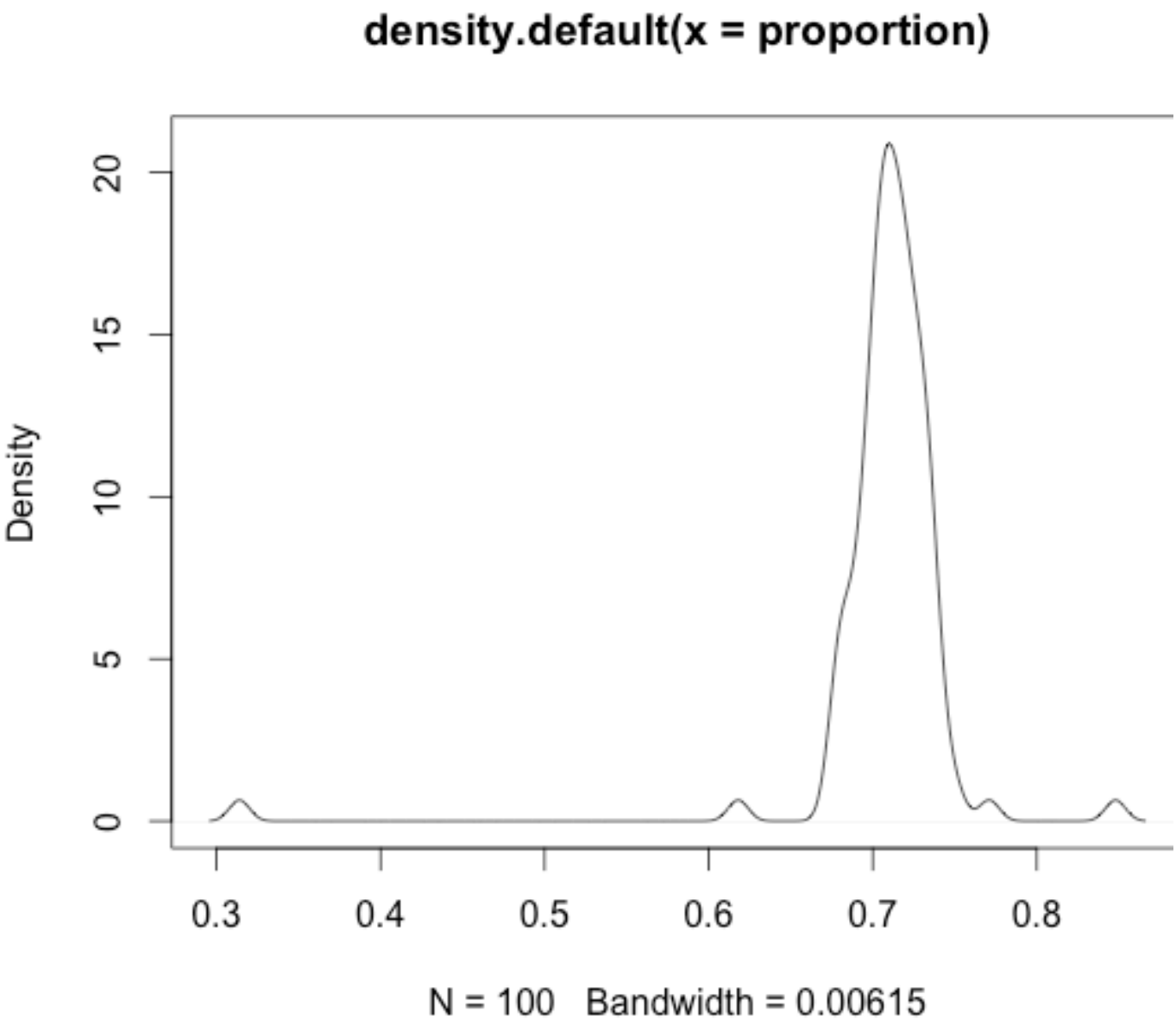
proportion

```
 [1] 0.314 0.848 0.618 0.771 0.677 0.732 0.695 0.734 0.713 0.729 0.705 0.699 0.734 0.678 0.72
[16] 0.703 0.707 0.695 0.719 0.701 0.718 0.710 0.728 0.703 0.716 0.709 0.707 0.704 0.703 0.70
[31] 0.716 0.703 0.739 0.700 0.729 0.702 0.710 0.720 0.705 0.722 0.676 0.722 0.716 0.729 0.68
[46] 0.709 0.695 0.711 0.715 0.697 0.724 0.716 0.734 0.710 0.727 0.682 0.718 0.694 0.728 0.69
[61] 0.746 0.735 0.710 0.722 0.717 0.721 0.713 0.734 0.680 0.751 0.687 0.730 0.681 0.739 0.68
[76] 0.722 0.683 0.735 0.710 0.717 0.691 0.738 0.717 0.724 0.700 0.732 0.724 0.703 0.710 0.70
[91] 0.712 0.689 0.738 0.696 0.724 0.706 0.713 0.709 0.706 0.720
```

plot(1:temps,proportion, type='l')



```
plot(density(proportion))
```

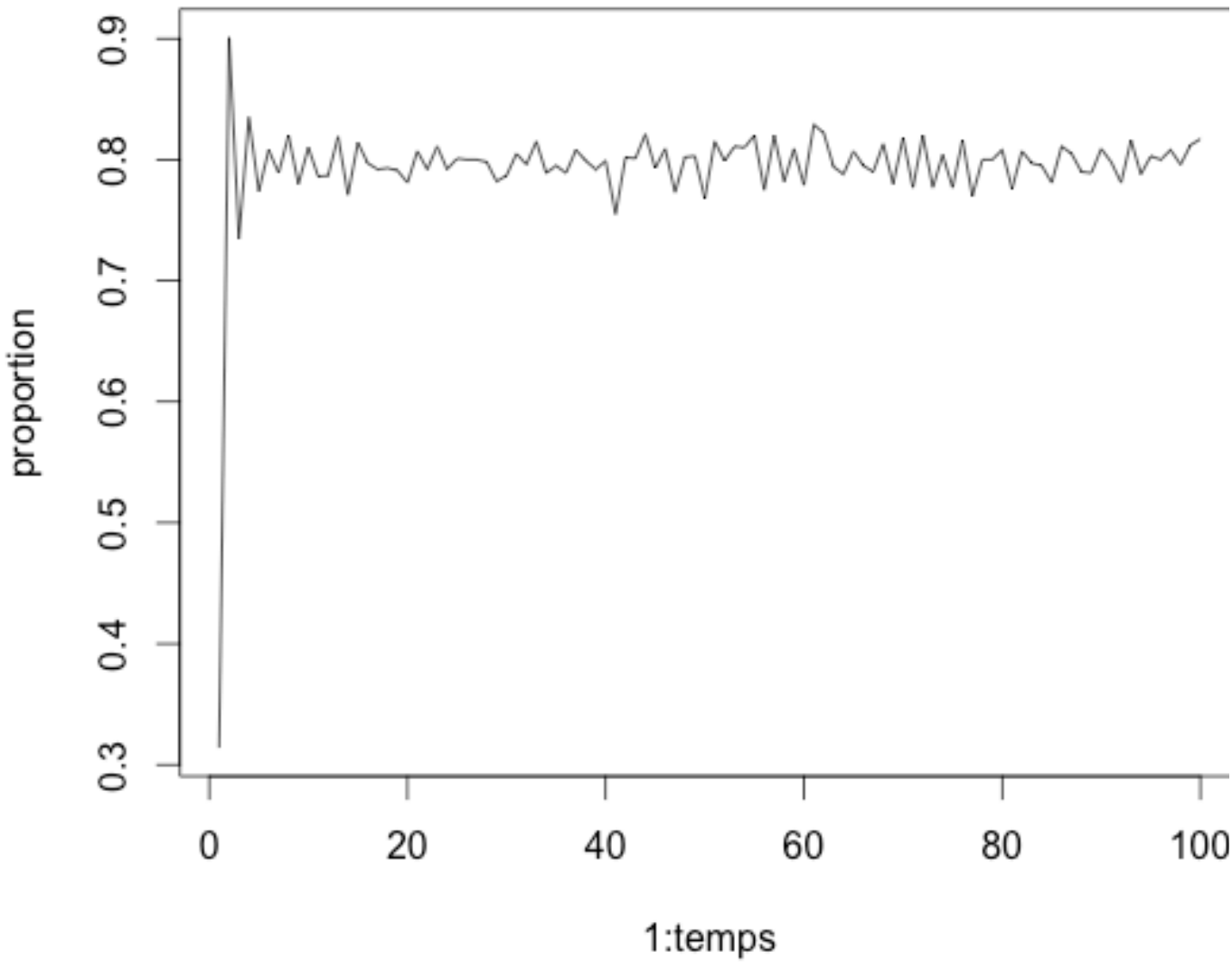


Pour g3 :

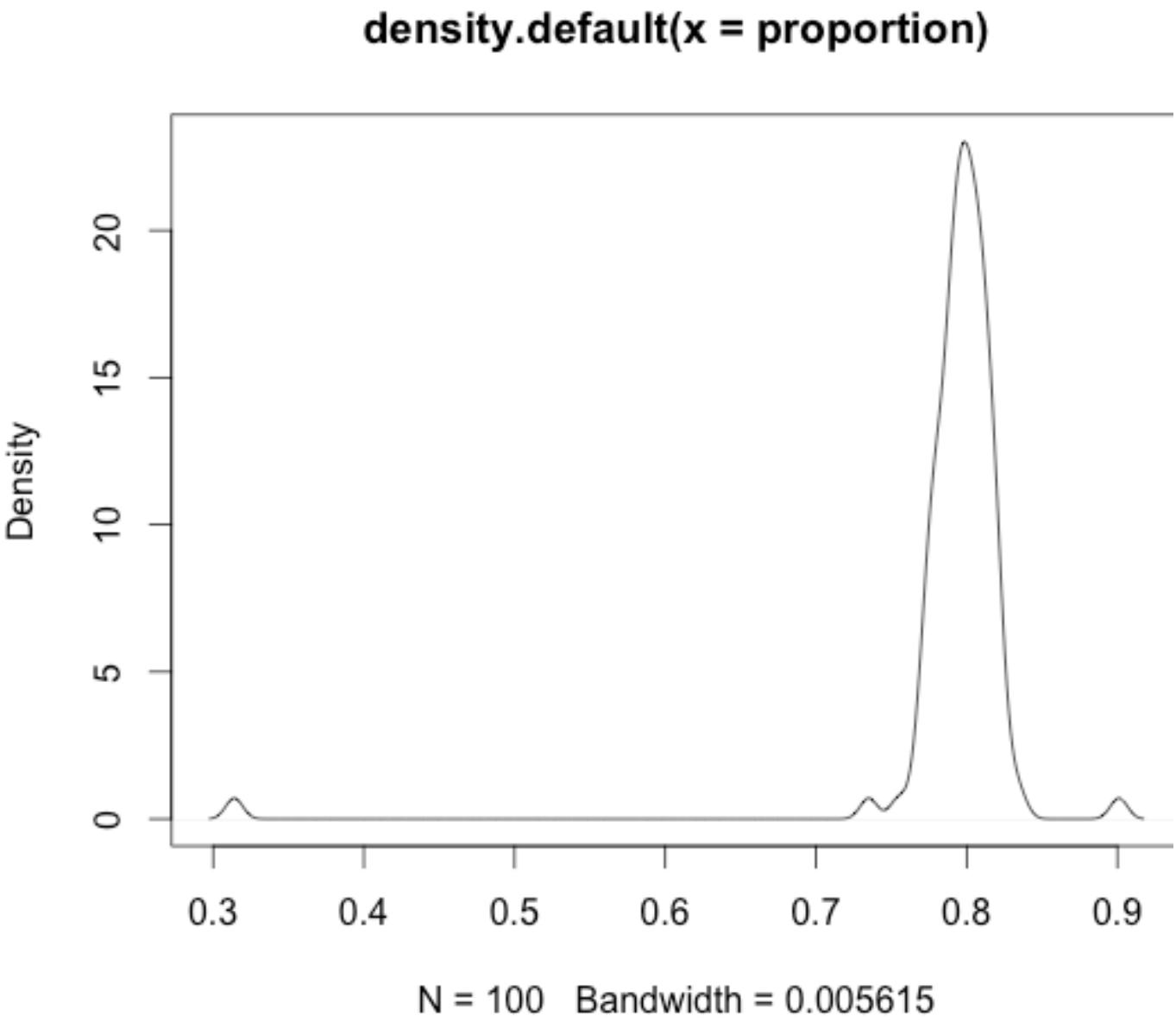
proportion

```
 [1] 0.314 0.901 0.735 0.835 0.774 0.808 0.789 0.820 0.780 0.810 0.786 0.787 0.819 0.771 0.81
[16] 0.797 0.792 0.793 0.791 0.781 0.807 0.792 0.811 0.792 0.801 0.800 0.800 0.798 0.782 0.78
[31] 0.805 0.796 0.815 0.789 0.795 0.789 0.808 0.799 0.792 0.799 0.755 0.802 0.801 0.821 0.79
[46] 0.809 0.773 0.802 0.803 0.768 0.815 0.799 0.811 0.810 0.820 0.775 0.820 0.782 0.809 0.77
[61] 0.829 0.822 0.794 0.788 0.807 0.795 0.790 0.813 0.780 0.818 0.777 0.820 0.777 0.804 0.77
[76] 0.816 0.770 0.800 0.800 0.808 0.776 0.807 0.797 0.795 0.781 0.811 0.805 0.790 0.789 0.80
[91] 0.798 0.781 0.816 0.788 0.803 0.800 0.808 0.796 0.812 0.817
```

plot(1:temps,proportion, type='l')



plot(density(proportion))



Nous ne notons pas de différence remarquable (elle est même presque inexistante) avec le jeu précédent de paramètres (tous invariables à l'exception de b qui valait 20%, et ici il vaut 5%). Visiblement, la probabilité de tentative d'infecter ses voisins, même réduite, ne semble pas avoir d'influence sur la proportion de nœuds infectés dans le réseau.

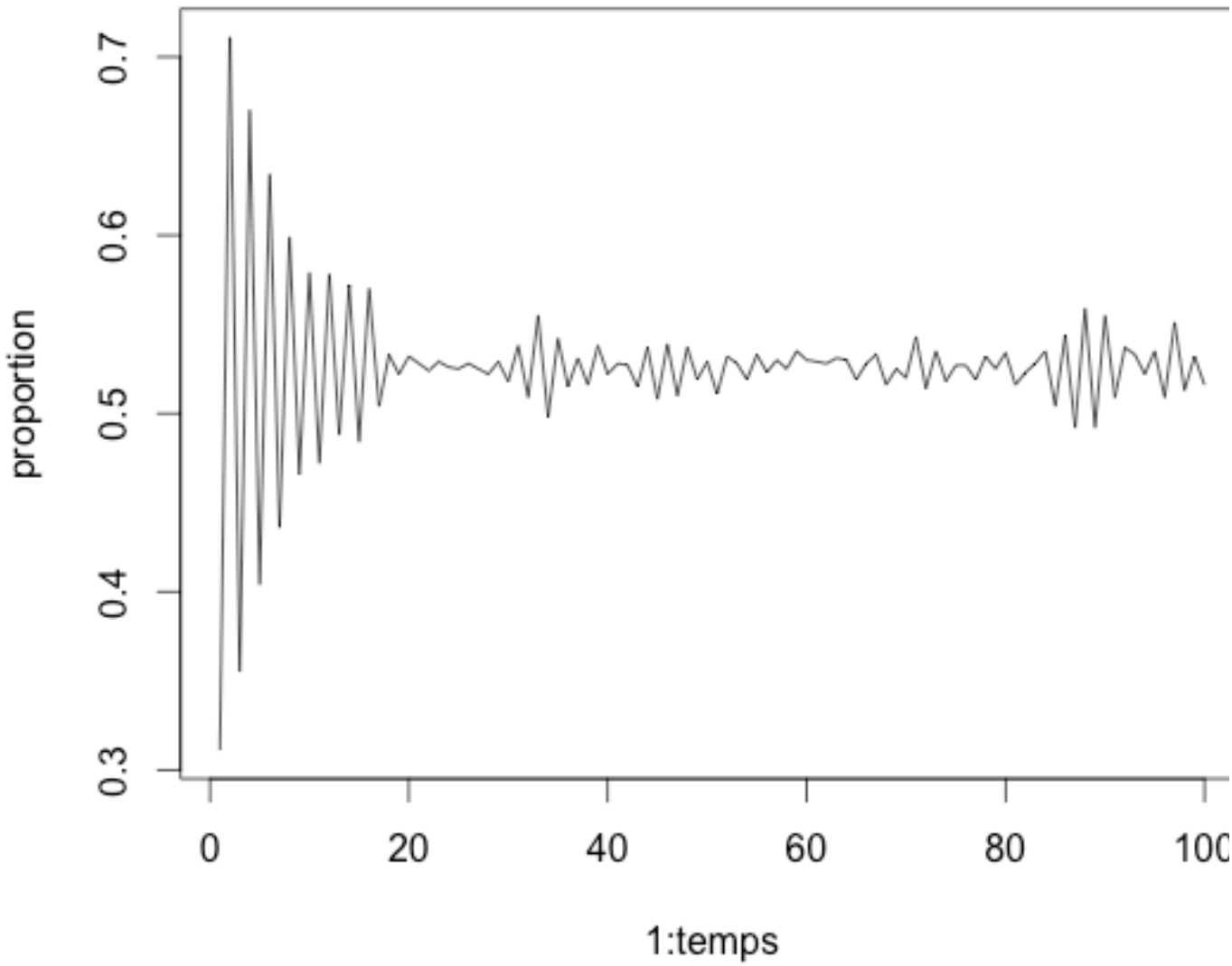
Nous allons donc cette fois jouer sur la probabilité de guérison d'un nœud infecté, à savoir y. Faisons passer y de 70% à 90% (et revenons volontairement à 20% pour b, pour voir si à elle seule, cette probabilité de guérison d'un nœud infecté peut changer les choses), en réutilisant toujours le même programme que précédemment. Cela nous donne :

Pour g1 :

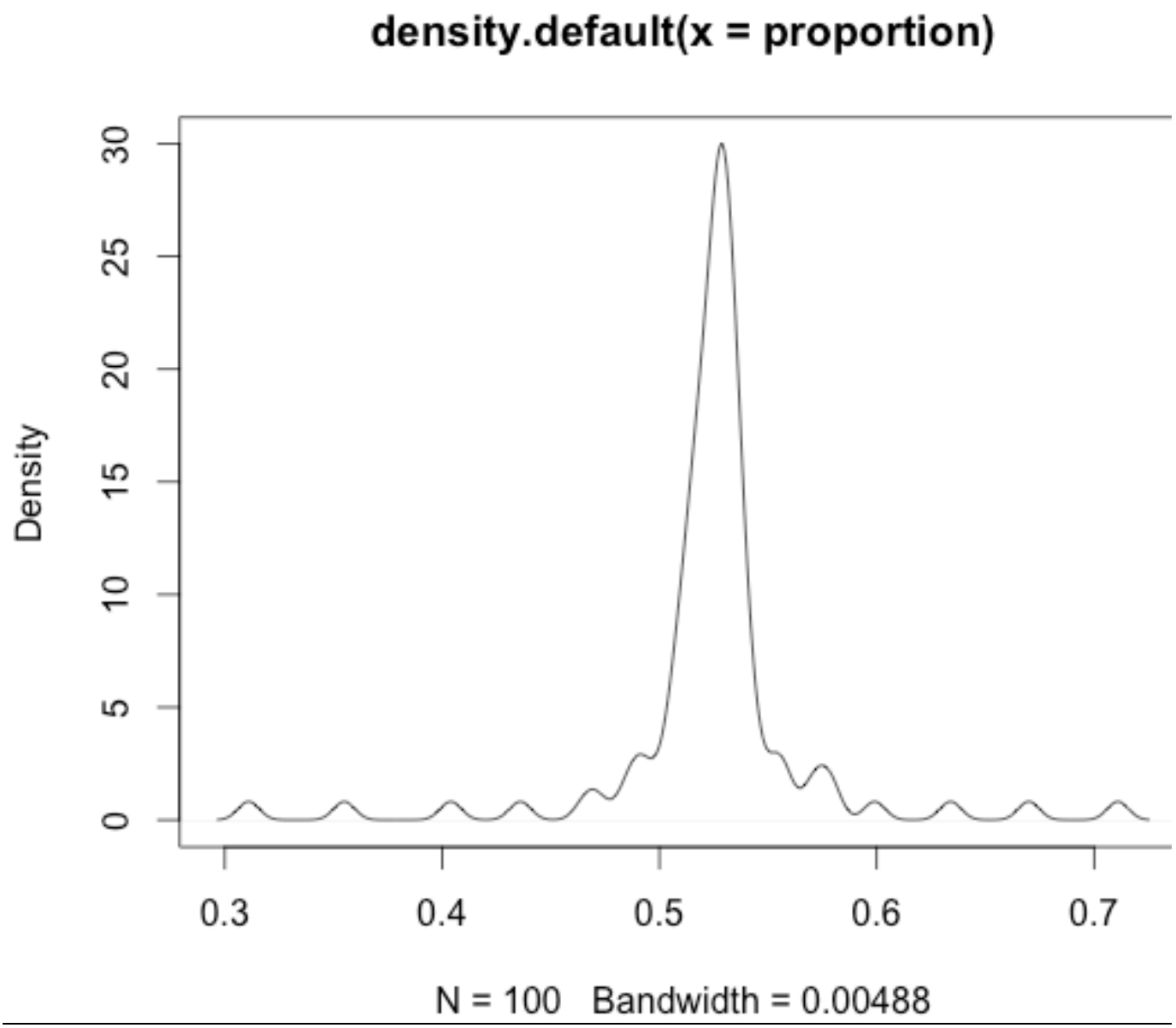
proportion

```
 [1] 0.311 0.711 0.355 0.670 0.404 0.634 0.436 0.599 0.466 0.579 0.472 0.578 0.488 0.572 0.48
[16] 0.570 0.504 0.533 0.522 0.532 0.528 0.524 0.529 0.526 0.525 0.528 0.525 0.522 0.529 0.51
[31] 0.538 0.509 0.555 0.498 0.542 0.515 0.531 0.516 0.538 0.522 0.528 0.527 0.515 0.537 0.50
[46] 0.539 0.510 0.537 0.519 0.529 0.511 0.532 0.528 0.519 0.533 0.523 0.530 0.525 0.535 0.53
[61] 0.529 0.528 0.531 0.530 0.519 0.528 0.533 0.516 0.525 0.520 0.543 0.514 0.535 0.518 0.52
[76] 0.527 0.519 0.532 0.525 0.534 0.516 0.523 0.528 0.535 0.504 0.544 0.492 0.559 0.492 0.55
[91] 0.509 0.537 0.533 0.522 0.535 0.509 0.551 0.513 0.532 0.516
```

plot(1:temps,proportion, type='l')



```
plot(density(proportion))
```

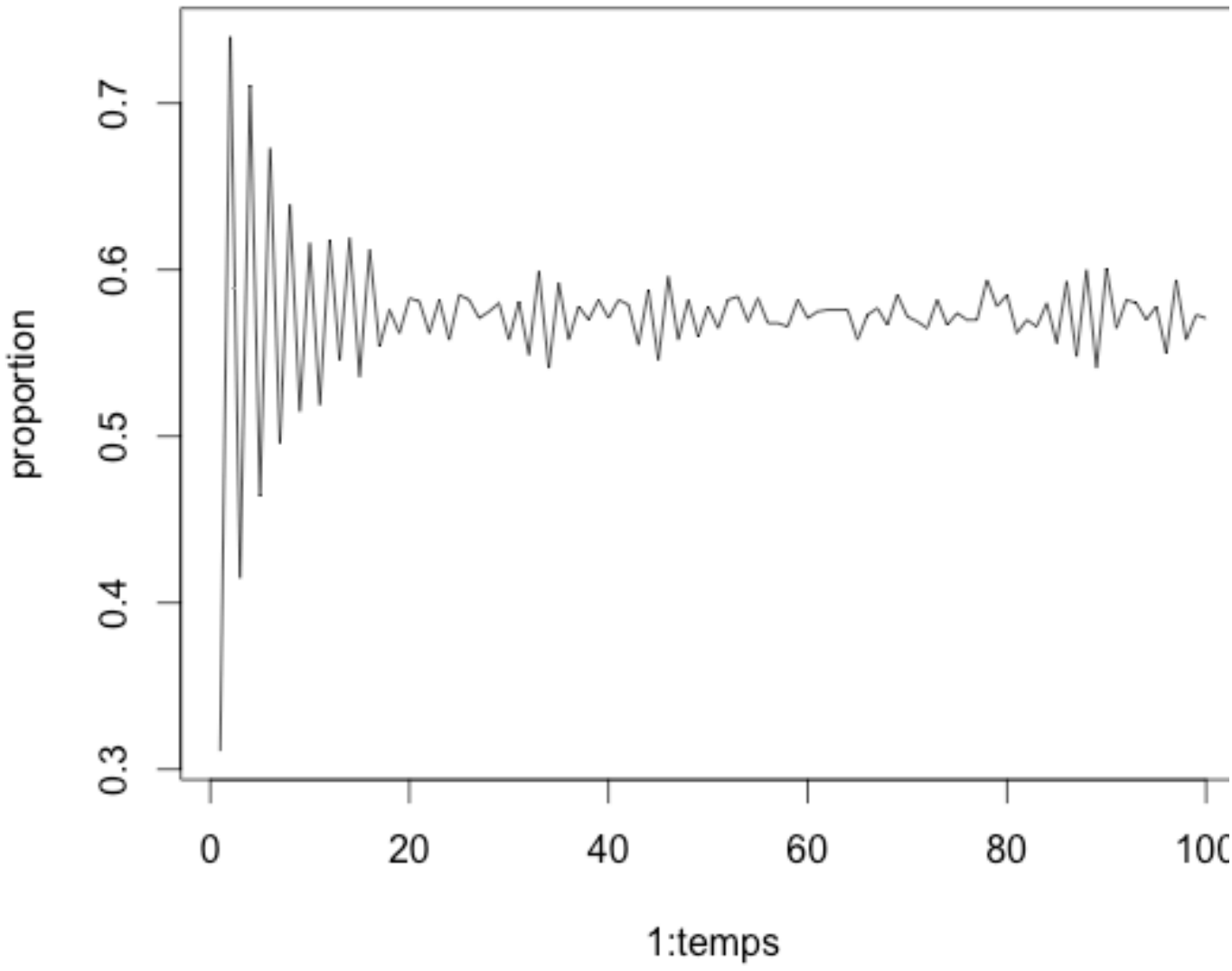


Pour g2 :

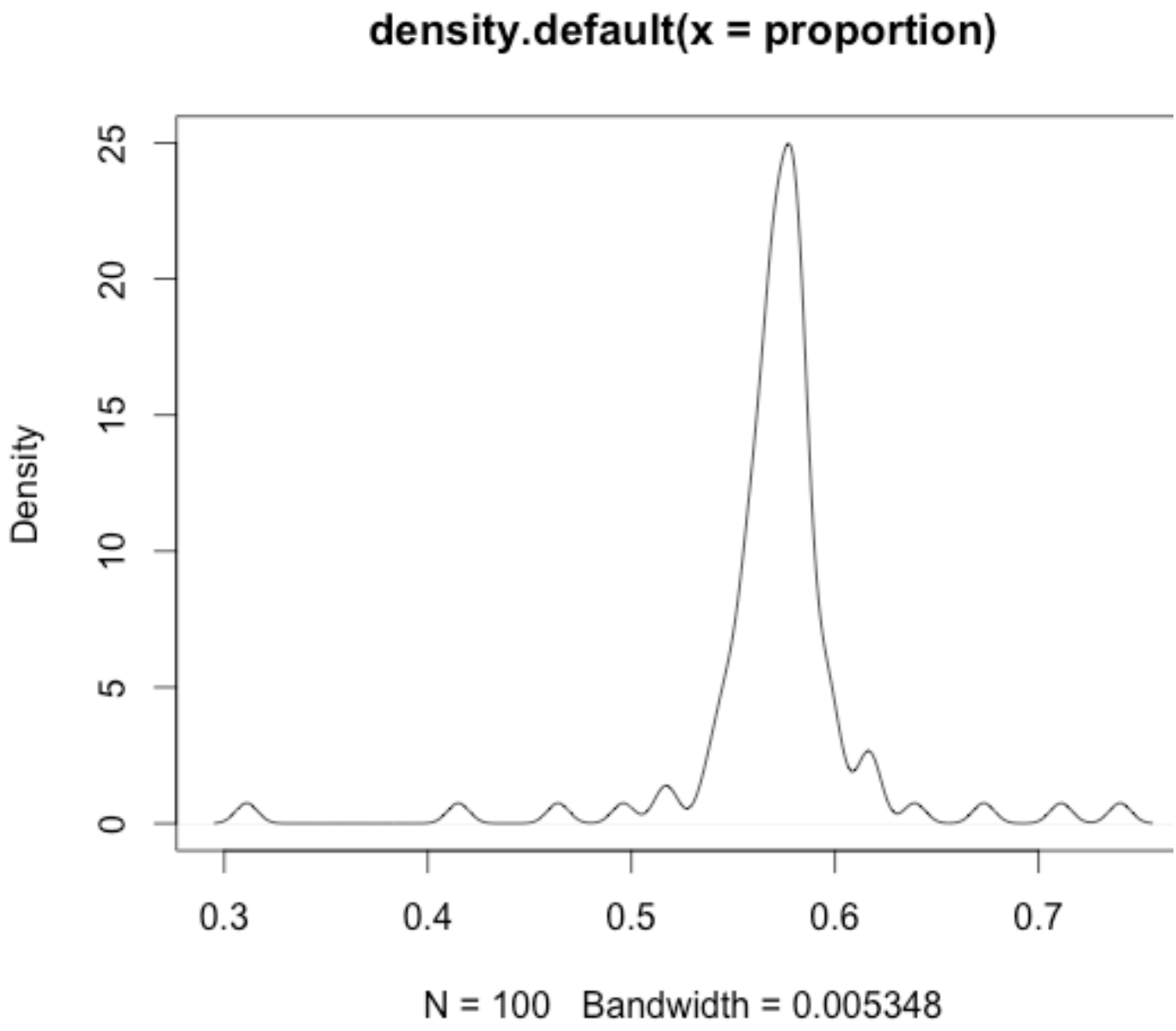
proportion

```
 [1] 0.311 0.740 0.415 0.711 0.464 0.673 0.496 0.639 0.515 0.616 0.519 0.618 0.546 0.619 0.53
[16] 0.612 0.554 0.576 0.562 0.583 0.581 0.562 0.582 0.558 0.585 0.582 0.571 0.575 0.580 0.55
[31] 0.581 0.549 0.599 0.541 0.592 0.558 0.578 0.570 0.582 0.571 0.582 0.579 0.555 0.588 0.54
[46] 0.596 0.558 0.582 0.560 0.578 0.565 0.582 0.584 0.569 0.583 0.568 0.568 0.566 0.582 0.57
[61] 0.575 0.576 0.576 0.576 0.558 0.573 0.577 0.567 0.585 0.572 0.569 0.565 0.582 0.567 0.57
[76] 0.570 0.570 0.594 0.578 0.585 0.562 0.570 0.566 0.580 0.556 0.593 0.548 0.600 0.541 0.60
[91] 0.565 0.582 0.580 0.570 0.578 0.550 0.594 0.558 0.573 0.571
```

plot(1:temps,proportion, type='l')



```
plot(density(proportion))
```

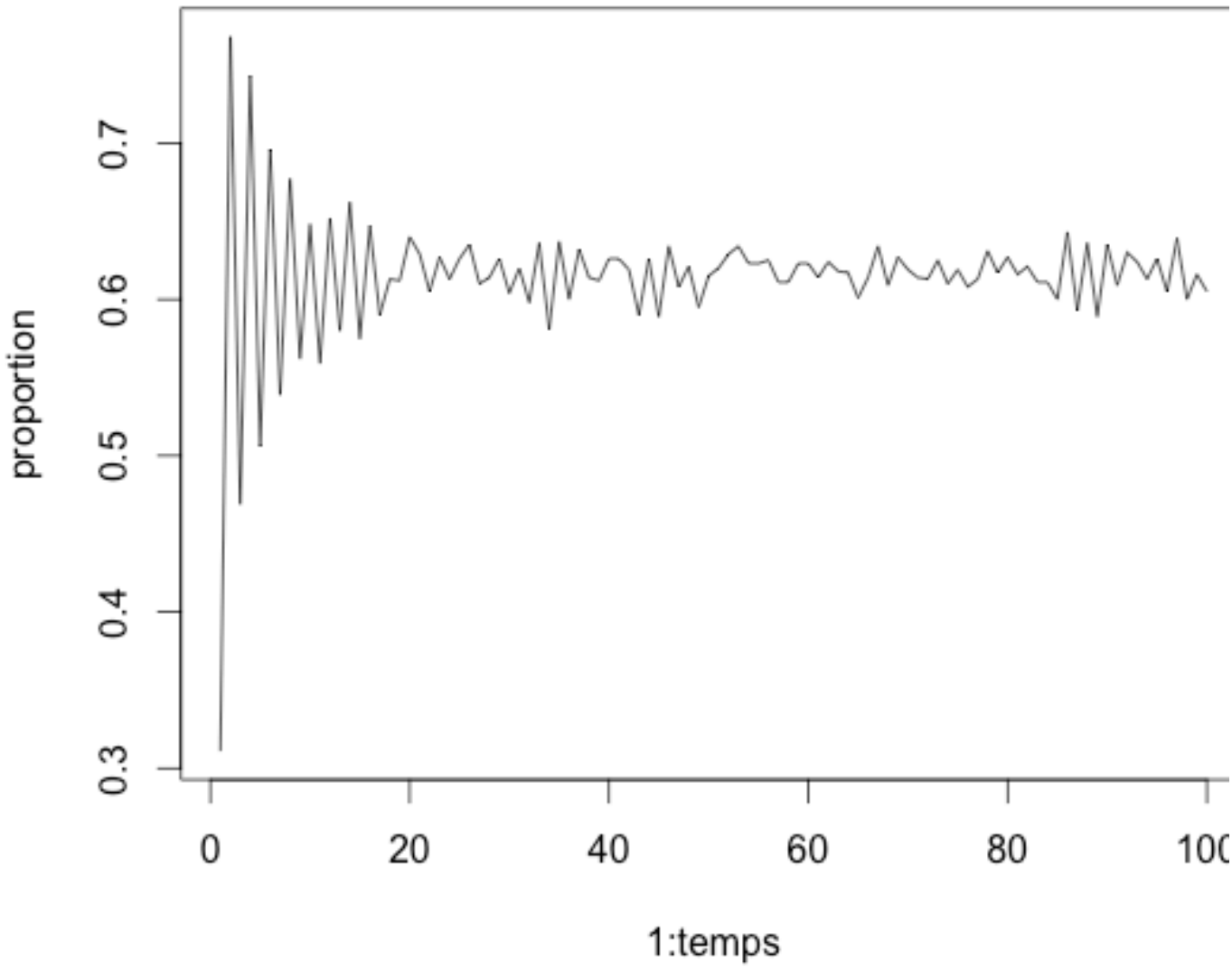


Pour g3 :

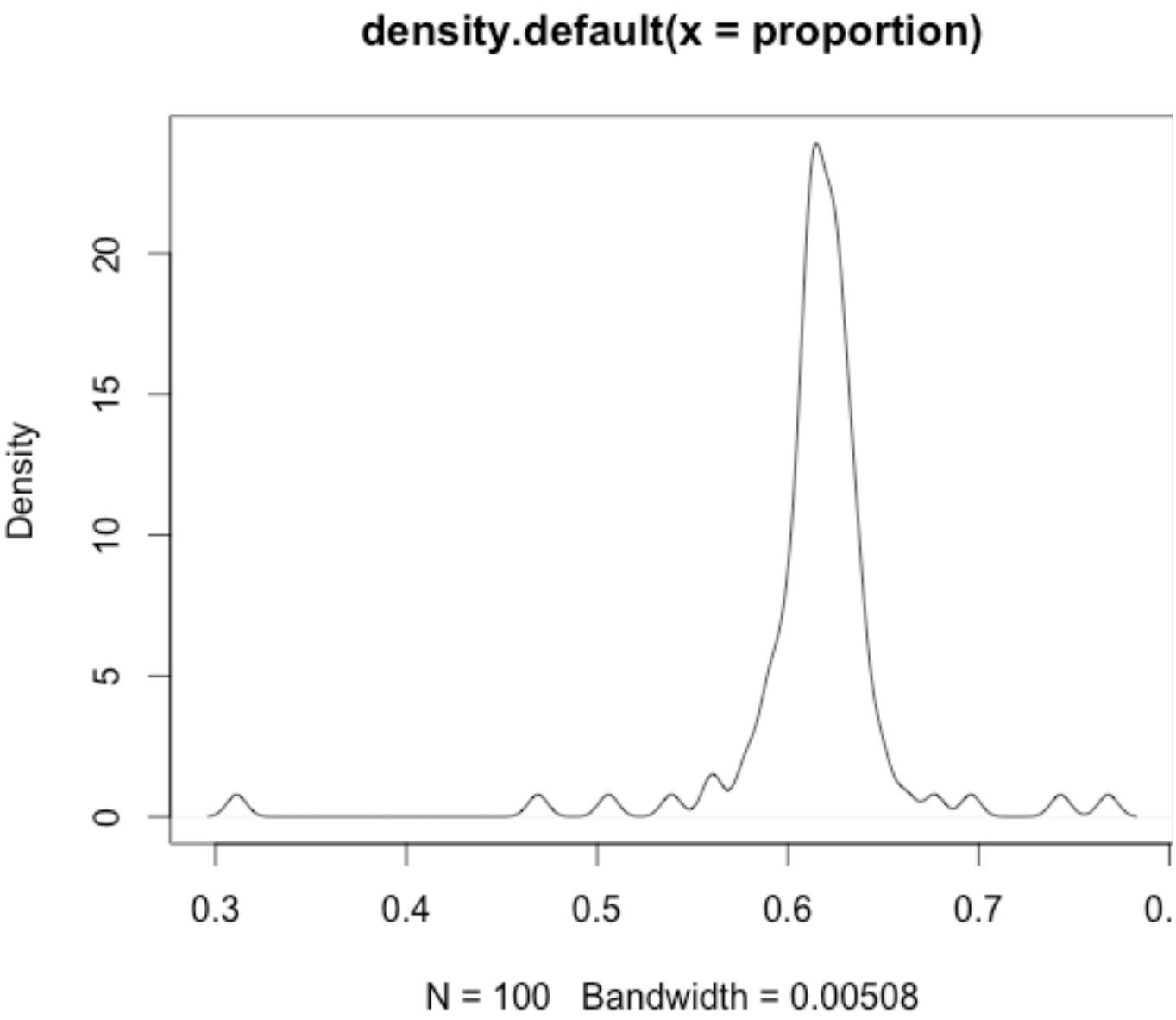
proportion

```
 [1] 0.311 0.768 0.469 0.743 0.506 0.696 0.539 0.677 0.562 0.648 0.559 0.652 0.580 0.662 0.57
[16] 0.647 0.590 0.613 0.612 0.640 0.629 0.605 0.627 0.613 0.626 0.635 0.610 0.614 0.626 0.60
[31] 0.620 0.598 0.636 0.581 0.637 0.600 0.632 0.614 0.612 0.626 0.626 0.619 0.590 0.626 0.58
[46] 0.634 0.608 0.621 0.595 0.615 0.620 0.629 0.634 0.623 0.623 0.625 0.611 0.611 0.623 0.62
[61] 0.614 0.624 0.618 0.617 0.601 0.614 0.634 0.609 0.627 0.619 0.614 0.613 0.625 0.610 0.61
[76] 0.608 0.613 0.631 0.617 0.627 0.616 0.621 0.611 0.611 0.600 0.643 0.593 0.636 0.589 0.63
[91] 0.609 0.630 0.624 0.613 0.626 0.605 0.639 0.600 0.616 0.605
```

plot(1:temps,proportion, type='l')



plot(density(proportion))



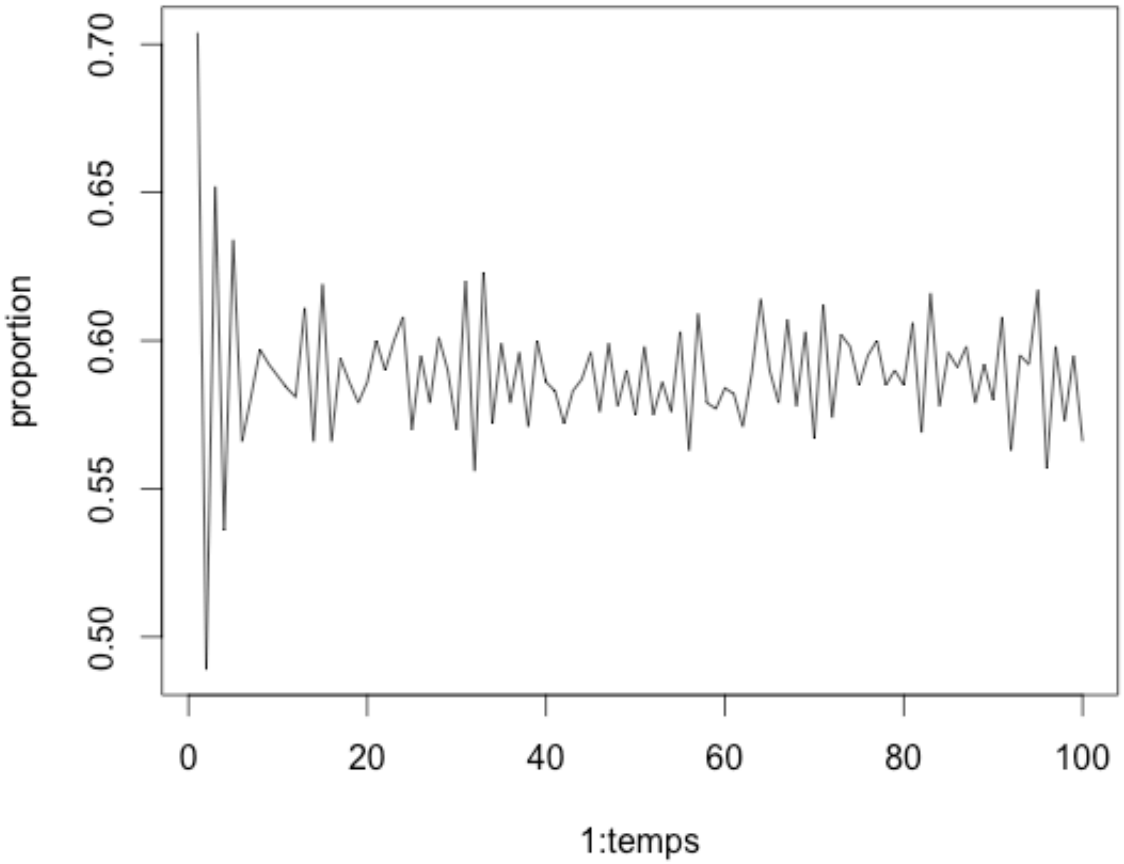
Analyse : pour g1, le comportement de la proportion semble similaire au cas initial (lorsque y valait 70%), mais l'oscillation se fait par la suite autour de 52% (au lieu de 58%). Il en est de même pour g2 (nous finissons par osciller autour de 57% au lieu de 71%) et pour g3 (nous oscillons autour de 61% au lieu de 80%). Cela nous montre que la probabilité de guérison d'un nœud infecté a un impact notable, en particulier sur les graphes plus complets et sur les graphes où le nombre de nœuds joignables rapidement (par le biais d'un court chemin) est plus important (en particulier dans le graphe Small-world, graphe dans lequel tout nœud peut être joint par un autre nœud par le biais d'un court chemin). Il peut donc être très intéressant de jouer sur ce paramètre y, en particulier dans

les graphes complets et Small-world où le risque de contamination est plus élevé que dans d'autres types de graphes.

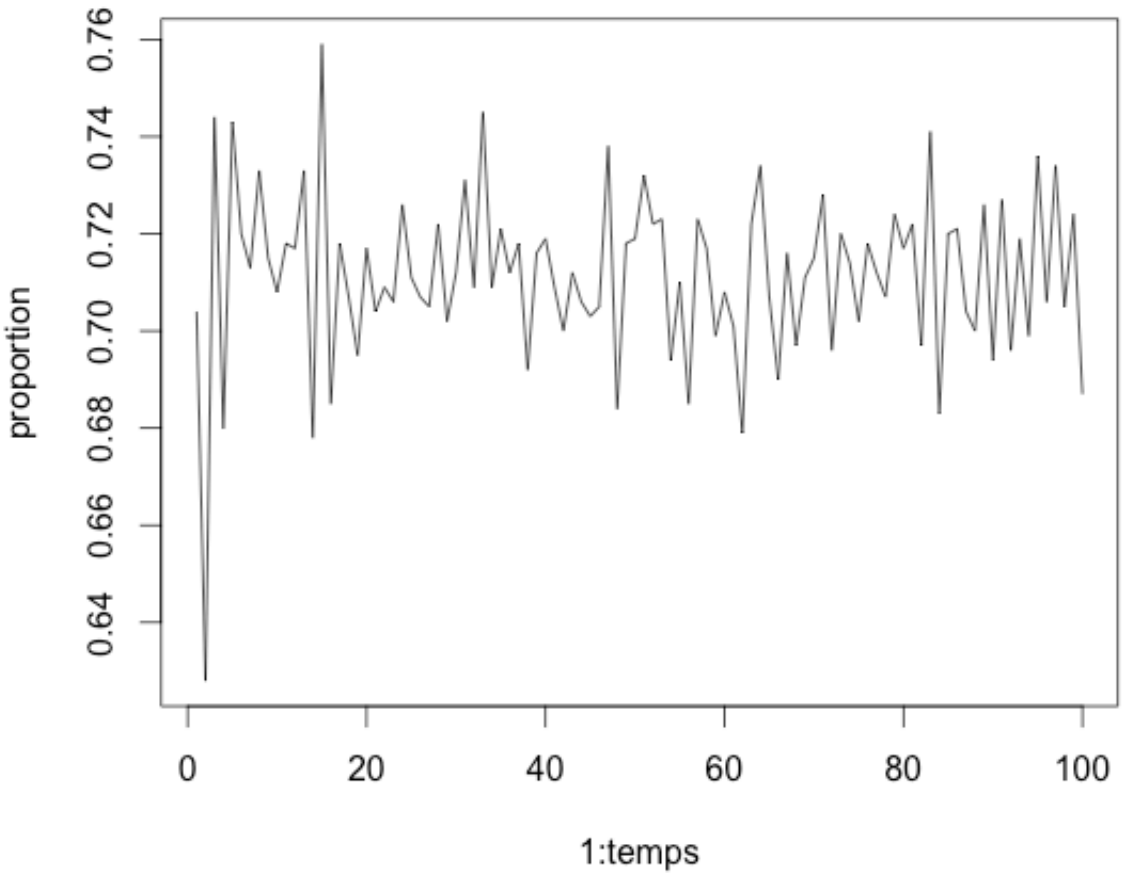
Notons que dans l'ensemble des graphes, la proportion de nœuds infectés commence par osciller, puis au bout d'un certain temps, elle semble converger vers une valeur. Donc nous jugeons inutile de changer le temps (il ne favorise ni la propagation de l'infection, ni la guérison des nœuds). Il suffit juste de prendre un temps assez important pour voir cette convergence, car si nous nous arrêtons à un temps insuffisant et que nous observons les oscillations, cela peut fausser les résultats (la proportion a besoin d'un certain temps pour se stabiliser).

<u>Conclusion</u> : Il semble y avoir un lien clair entre la contamination d'un graphe et sa modularité dans certains types de graphes. En effet, nous remarquons qu'il est difficile de faire flancher la propagation de l'infection dans un graphe Erdös–Rényi classique (probabilité 0.4, et de bonne modularité, même si elle n'est pas très grande) ; seule la probabilité *y* de guérison d'un nœud semble jouer sur la diminution de la proportion de nœuds infectés. En revanche, plus notre graphe gagne en complétude ou en capacité de jonction d'un nœud à l'autre par le biais d'un court chemin (peu d'arêtes ; justement ce que permet Small-world, graphe dans lequel tout nœud peut être joint par un autre nœud par le biais d'un court chemin), plus notre graphe gagne donc en modularité. La force de la modularité semble donc se combiner au pouvoir de guérison d'un nœud pour contrer de manière efficace l'infection sur un temps suffisamment important. A contrario, une modularité faible (bien qu'elle ne favorisera pas l'infection) ne semble avoir que peu d'influence sur la contamination.

Si nous changeons la proportion initiale de nœuds infectés (passage de 30% à 70%), nous remarquons pour g1 qu'au départ, la proportion explose également (mais à une valeur un peu moins importante que la valeur atteinte lorsque p0 valait 30%), mais par la suite, elle oscille également autour de 59%, ce qui nous laisse penser que la valeur de la proportion initiale de nœuds infectés n'a pas de réelle importance sur la valeur finale de cette proportion pour un graphe classique de type Erdös-Rényi (probabilité 0.4). Toutefois, elle semble avoir de l'influence sur l'agitation des variations de cette proportion au fil du temps (bien que cette agitation ne soit pas réellement prononcée). Nous pensons que plus cette proportion initiale est importante, plus la proportion va varier de manière agitée au fil du temps (sans que sa valeur finale ne change). Voyons plutôt :

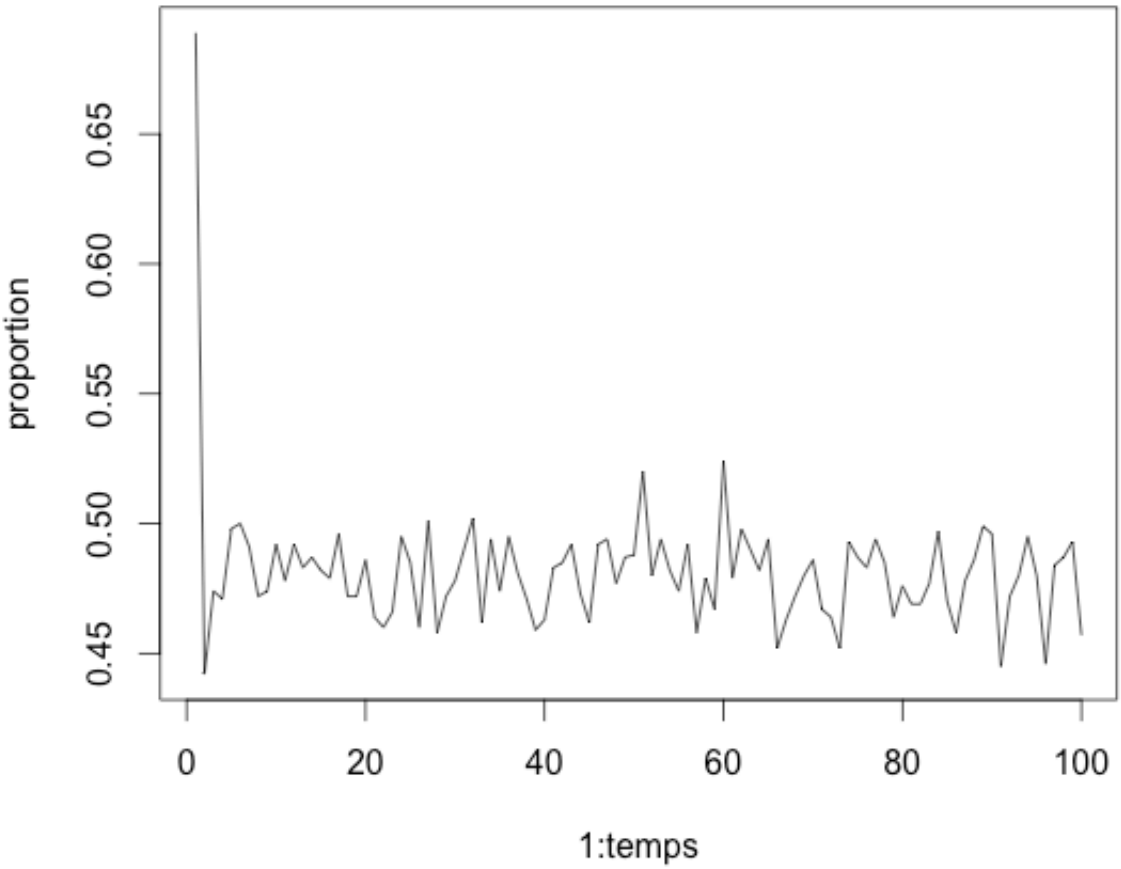


Pour un graphe de type complet (Erdös-Rényi, probabilité 1), à savoir g2, nous remarquons qu'au départ, la proportion n'explose pas (elle démarre directement à une valeur avoisinant les 70%). Par la suite, bien que son oscillation semble plus agitée que dans le cas où p0=30%, elle semble osciller autour de la même valeur, à savoir 0.71. La proportion initiale de nœuds infectés n'a pas d'influence sur sa valeur finale, néanmoins, au même titre que pour g1, elle semble avoir de l'influence sur l'agitation des variations de cette proportion au fil du temps (avec une agitation bien plus prononcée que pour g1). Nous pensons que plus cette proportion initiale est importante, plus la proportion va varier de manière agitée au fil du temps (sans que sa valeur finale ne change). Le gain en complétude de notre graphe (au passage de g1 à g2) peut expliquer cette forte agitation. En effet, le fait que chaque nœud soit connecté à tous les autres implique une variation importante de la proportion, à chaque instant (visuellement parlant, c'est somme toute assez logique). Voyons plutôt :



Pour un graphe de type Watts–Strogatz (g3), nous remarquons qu'au départ, la proportion n'explose pas (elle démarre directement à une valeur avoisinant les 70%). Il est également clair que l'augmentation de la proportion initiale de nœuds infectés provoque aussi une agitation des variations de cette proportion au fil du temps (elles restent un peu moins prononcées que l'agitation pour g2, mais elles marquent une différence avec le graphe Watts–Strogatz lorsque p0 valait 30%). Toutefois, nous notons la nouveauté assez surprenante : en faisant passer p0 de 30% à 70%, la proportion, qui oscillait autour de 80% lorsque p0 valait 30%, oscille à présent autour de 47% ! Cela nous étonne : pour un tel graphe (avec les propriétés que nous lui connaissons, qui ont été évoquées précédemment, et que nous n'allons pas répéter ici), ce serait limite bénéfique de contaminer dès le départ de manière plus ample nos nœuds, dans une optique de réduction de cette infection à l'instant terminal. Le binôme Watts–Strogatz a construit son graphe de telle sorte que plus vous l'infestez au départ, moins il sera malade à la fin. Cela semble très paradoxal, mais c'est peut-être dans ce genre de graphes que le pouvoir guérisseur d'un nœud est le plus important. Les courts chemins entre chaque nœud, ce n'est peut-être pas qu'un atout pour l'infection, c'est peut-être aussi un réel atout pour la guérison. En effet, les architectures « small-world » sont peut-être plus robustes aux perturbations que d'autres architectures réseaux, et leur

prévalence dans les systèmes biologiques apporterait un avantage évolutif clair aux systèmes biologiques sujets aux mutations ou infections virales. Voyons plutôt :



Nous n'avons volontairement pas touché au nombre de nœuds de nos réseaux, car il est imposé pour l'étude et le faire varier n'apporterait que peu d'intérêt (nous considérons très intéressant l'étude de la contamination d'un réseau volumineux en terme de nombre de nœuds). Nous pouvons le changer, mais un nombre plus faible ferait perdre de l'intérêt à notre étude de contamination de réseaux volumineux, et un nombre plus important nous imposerait des complexités algorithmiques colossales.