

# PAF

## Decentralised Task-distribution in Peer-to-Peer Networks

22.06.2020

Groupe 1 : Anh-Vu Nguyen, Arthur Gourrin, Florian Le Mouël, David Gérard

Dépot : <https://gitlab.telecom-paris.fr/PAF/1920/prj16-1>

Paramètres internes	Description
Available	Si l'agent est en train de traiter une tâche (bool)
mytask	Tâche de l'agent (int, 0 si aucune tâche)
color	Bleu ou rouge en fonction de la tâche (1 ou 2) et vert si aucune

Registres in/out
Liste des identifiants des états des noeuds

Description d'un état : un état contient
Identifiant du noeud
Le statut du noeud
La liste des tâches à distribuer (Utile si on veut ajouter des tâches en cours d'exécution)

La date de cet état
---------------------

## Hypothèses:

- **Le graphe est connexe**
- **Les agents ont une connaissance limitée du réseau** (ils ne reçoivent de l'information qu'en communiquant avec leurs voisins)
- On peut prendre une racine au hasard dans le graphe (ce qui est plus compliqué à faire si ce sont les agents qui doivent se mettre d'accord sans vision globale du graphe) -> hypothèse non nécessaire avec l'algorithme "gossip" (n'a pas forcément de racine)
- Les tâches sont introduites au début et se répartissent au cours de l'algo sans nouvelle introduction de tâches ou influence extérieure (pourra être modifié selon les axes de développement)
- On cherche une répartition des tâches synchronisée (les tâches seront lancées au même moment, lorsque la solution sera établie)
- Les tâches ont la même importance / difficulté -> l'algorithme "deterministic" implémente une hiérarchisation des tâches
- 1 tâche = 1 agent
- Les agents ont tous les mêmes ressources de calcul
- 2 tâches différentes -> les algorithmes "gossip", "deterministic" et "probabilistic" peuvent implémenter un nombre quelconque de tâches
- Graphe quelconque (mais toujours connexe) -> on peut générer des graphes particuliers comme "small world", des graphes complets ou des arbres
- Autant de tâches que d'agents -> on peut avoir plus de tâches que d'agents et inversement (le comportement conséquent dépend de l'algorithme utilisé)

## Axes de développement:

- Hiérarchie des tâches (**implémenté**)
- Plus de tâches (**implémenté**)
- Différent types de tâches (nécessitant plus de temps de calcul ou plus de ressources)
- Un agent peut faire plusieurs tâche -> une solution simple consisterait à diviser un agent en plusieurs "sous-agents" formant un graphe complet (connection parfaite au sein de l'agent global)
- Une tâche peut être effectuée par plusieurs agents -> une solution simple consisterait à la diviser en "sous-tâches" qui formeraient un type à répartir avec l'algorithme habituel
- Plusieurs types d'agent (ex: ressources)
- Introduction d'une notion de temps
- Introduction de nouvelles tâches au cours du temps
- Recherche d'une racine optimale (**implémenté**)

- Générations de topologie (**implémenté**)
- Algorithme par propagation (**implémenté**)
- Algorithme avec convergence (**implémenté**)

## Répartition du travail

Nous avons chacun nos idées et avons donc décidé en partant d'une base commune de les développer de notre côté et de les fusionner le moment venu, à chaque fois qu'un axe de développement était mené à bout.

### Anh-vu

Algo probabiliste  
Algo gossip  
Intégration  
Code de test & Tests

### Arthur

Génération de topologies  
Algo estimation-adjustment  
Tests

### Florian

Algo deterministic  
Tâches multiples & hiérarchie des tâches  
Tests

### David

Recherche d'une racine optimale  
Tâches multiples  
Intégration

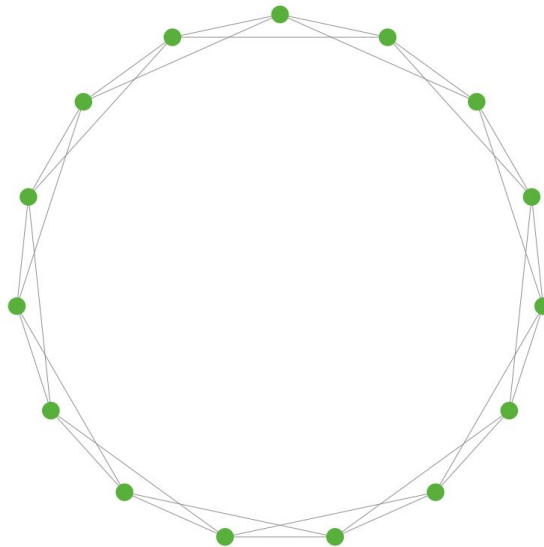
Pour les tâches tests et intégrations, chacun a apporté sa pierre à l'édifice, que ce soit dans l'explication de code ou dans les corrections nécessaires au bon fonctionnement du code intégré. Ces points en particulier ont été le fruit d'une collaboration au sein du groupe.

## Génération de graphes

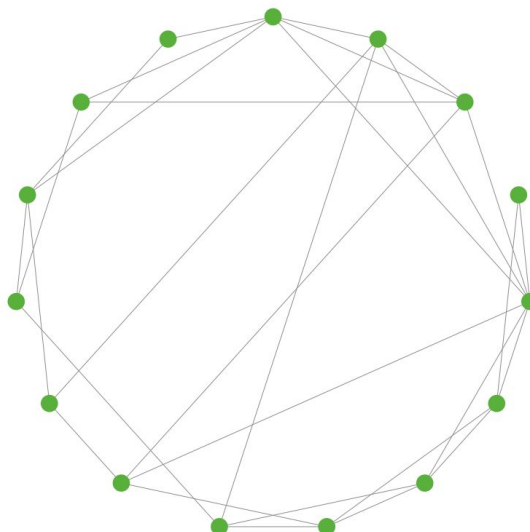
Afin d'étudier l'influence de la topologie du réseau sur les performances de nos différents algorithmes, nous avons mis en place plusieurs méthodes de génération de graphes. Ainsi, il est possible de générer les topologies suivantes : arbre, petit-monde, complet, connexe aléatoire.

## Détail pour petit-monde :

Afin de générer un graphe “petit-monde” (c’est à dire que chaque noeuds est relié aux autres via un petit nombre de noeuds intermédiaires, plus précisément le plus court chemin entre 2 noeuds varie comme  $\ln(\text{ordre du graphe})$ ), on s’inspire d’un algorithme existant. On commence par générer le graphe suivant :



Puis, pour chaque arête  $(e, s)$ , avec une certaine probabilité (dans notre cas, nous avons choisi 40 %) : on la supprime, puis on crée une arête  $(e, x)$  où  $x$  n’est pas déjà voisin de  $e$ . Cela donne des graphes de la forme suivante :



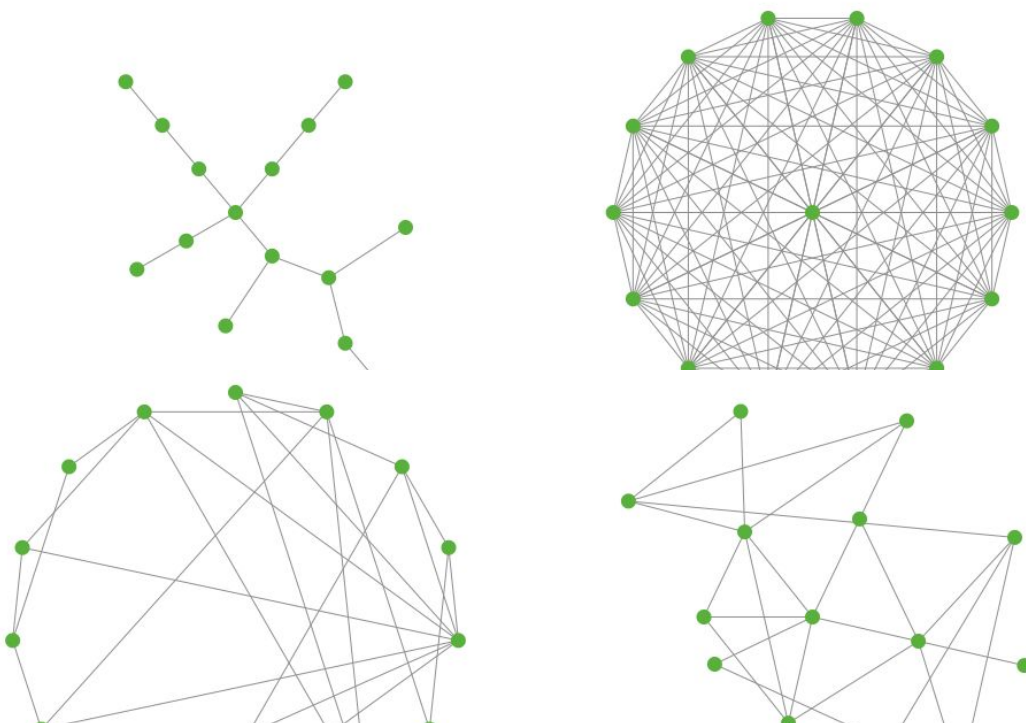
## Détail pour connexe aléatoire :

Ce que nous appelons “connexe aléatoire” est un graphe dont les arêtes sont créées aléatoirement, mais dont on est sûr qu’il est connexe. Pour créer un tel graphe d’ordre  $n$  et de taille  $c$  ( $c \geq n$ ) :

- on crée d’abord un arbre ( $n-1$  arêtes) qui est donc connexe. Pour cela, on place le premier noeud puis on place les autres un par un en les reliant à chaque à un noeud existant (pris au hasard) par une arête.
- A partir de cette arbre, on répète  $c - n + 1$  fois l’opération suivante : choisir deux noeuds puis les relier par une arête.

Le graphe obtenu n’est pas parfaitement aléatoire (car il est construit à partir d’un arbre) mais pour l’utilisation que nous en ferons ce sera une approximation satisfaisante. Il est à noter que la phase de création de l’arbre décrite ci-dessus est exactement celle utilisée pour créer des graphes de type arbre.

## Les 4 topologies implémentées



Dans l'ordre : arbre, complet, petit-monde, connexe aléatoire

## Algorithme par estimation de la distribution de tâches et ajustement

### Présentation

L'idée de cet algorithme est de faire estimer à chaque noeud la répartition globale des tâches sur tout le graphe. Pour cela, on suppose que chaque noeud connaît la répartition de tâches à atteindre (et, pour simplifier, on se limite ici à 2 tâches différentes). Par exemple, on peut vouloir 30 % de tâche 1 dans le réseau, et 70 % de tâche 2, et chaque noeud a connaissance de cet objectif. Chaque noeud  $n$  possède un attribut,  $[x_n y_n]$ , qui correspond à son estimation propre de la répartition dans le graphe : ce noeud estime qu'il y a une proportion  $x$  de tâche 1, et une proportion  $y$  de tâche 0. Chaque noeud  $n$  exécute la procédure  $P$  suivante :

- Remplacer  $x_n$  (et  $y_n$  de façon analogue) par :

$$\frac{\sum_v x_v}{\sum_v (x_v + y_v)}$$

où les  $x_v$  sont les estimations des voisins et celle du noeud  $n$ . Cela nécessite donc de demander aux voisins leurs estimations.

- Répéter cette première étape  $M$  fois. Nous avons choisi  $M = N / 3$ , où  $N$  est l'ordre du graphe. Ce choix est arbitraire, mais résulte d'un compromis entre précision de l'estimation et rapidité d'exécution. Il semble néanmoins nécessaire que le nombre de répétitions soit proportionnel à l'ordre du réseau.

Une fois cette procédure effectuée, le noeud  $n$  ajuste sa tâche en fonction de son estimation, et avec une certaine probabilité  $a$  (fixée, après divers essais, à 0.15 au début de l'algorithme). Par exemple s'il est de tâche 1, et s'il estime qu'il y a 50 % de tâche 1 dans le réseau, alors que l'on en voudrait 30 %, il va changer pour tâche 2 avec une probabilité  $a$ . Il décrémente ensuite  $a$  (nous avons choisi de décréementer par pas de 0.01).

Toutes ces opérations sont répétées tant que  $a$  est positif.

La complexité de cet algorithme est donc en  $O(N)$  pour chaque noeud, et elle varie selon le nombre de voisins d'un noeud. D'autre part, il s'échange peu d'information entre les noeuds (un noeud donné envoie seulement 2 flottants à chacun de ses voisins et à chaque itération de l'algorithme) donc il est a priori peu coûteux en bande passante. De plus, chaque noeud ne stocke que son estimation de répartition (2 flottants), ce qui est très économe en mémoire.

L'algorithme termine toujours puisque  $a$  est décrémentée à chaque itération. Nous allons tester ci-après s'il converge bien vers une solution convenable.

On peut enfin noter qu'il y a 2 paramètres importants pour l'algorithme :  $a_0$  (le choix de  $a$  initial) et  $M$ .

## Performances de l'algorithme

Nous avons fait des essais avec les paramètres présentés ci-dessus, pour des réseaux avec plus ou moins de noeuds et d'arêtes (ou connexions). Les noeuds sont initialisés avec un choix aléatoire de tâche qui respecte la loi de répartition fixée. Dans le tableau suivant, en ligne il y a le nombre de noeuds du réseau, en colonne le nombre de connexions. **Les résultats, présentés sous la forme  $a - b$ , représente l'erreur de répartition des tâches, en %. Cette erreur est donc l'écart entre l'objectif de répartition et la répartition obtenue effectivement.  $a$  est l'erreur avant l'exécution de l'algorithme (avec donc seulement une répartition aléatoire) et  $b$  est l'erreur après.** Ces données sont moyennées sur des centaines d'exécution de l'algorithme, et pour une topologie connexe aléatoire.

Noeuds \ Arêtes	800	1000	1500	2000
100	3,47 - 2,58	3,73 - 1,88	3,67 - 1,84	3,53 - 2,00
200	2,36 - 1,83	2,50 - 1,70	2,48 - 1,68	-
300	1,94 - 1,39	2,35 - 1,27	2,08 - 1,16	-
500	-	-	-	1,61 - 1,07

Tout d'abord, on voit que l'algorithme est systématiquement meilleur qu'une simple répartition aléatoire. Il converge donc bien vers la répartition de tâches voulue.

Ensuite, on peut voir 2 tendances dans ce tableau de résultats. Premièrement, si le nombre de noeuds augmente, l'algorithme est plus efficace (l'erreur diminue). Cette tendance s'observe aussi pour l'erreur avant l'algorithme s'observe aussi pour l'erreur avant l'algorithme. Ensuite, la même chose se produit avec le nombre de connexions : plus il est grand et plus l'erreur est petite. Cela est logique, puisque l'approche est probabiliste, elle fonctionne d'autant mieux que le nombre de noeuds est grand. D'autre part, plus de connexions implique un plus grand nombre de voisins par noeud : chaque noeud "voit" donc une plus grande partie du réseau et peut donc estimer plus finement la répartition globale des tâches.

## Influence de $a_0$

On fait maintenant varier  $a_0$ , et les tests sont effectués sur un réseau de 100 noeuds et 800 connexions, de topologie connexe aléatoire.

$a_0$	0,05	0,10	0,15	0,20	0,25
Erreur	3,57 - 2,17	3,72 - 2,32	3,47 - 2,58	3,70 - 2,36	3,22 - 2,91

Il ne se dégage pas de tendance claire dans ce premier tableau. Nous avons recommencé les mesures avec 300 noeuds et 1000 connexions :

$a_0$	0,05	0,10	0,15	0,20	0,25
Erreur	2,07 - 1,43	2,34 - 1,27	2,35 - 1,27	2,12 - 1,16	2,28 - 1,29

Ce second tableau éclaire les résultats du premier : il semble que prendre  $a_0$  petit n'est pas idéal. Une valeur de  $a_0$  entre 0,10 et 0,20 semble appropriée : le prendre trop grand est contre-productif (et cela augmente la durée de l'algorithme) et le prendre trop petit ne convient pas à des réseaux avec beaucoup de noeuds.

## Influence de $M$

On fait varier  $M$ , les tests sont effectués sur un réseau de 100 noeuds et 800 connexions, de topologie connexe aléatoire.

$M$	$N / 3$	$N / 2$	$N$
Erreur	3,47 - 2,58	3,61 - 2,19	3,38 - 1,93



On rappelle que  $N$  est l'ordre du réseau.

On voit une tendance qui se dégage de ces mesures : plus  $M$  est grand et plus l'algorithme est précis (erreur faible). Cela est logique, car  $M$  correspond au nombre de fois que chaque noeud ajuste son estimation de la répartition globale. Prendre  $M$  grand est donc gourmand en temps, mais permet de gagner en précision sur les estimations.

## Influence de la topologie

Pour un réseau de 100 noeuds petit-monde, on obtient pour l'erreur : 3,72 - 2,03. Ce résultat est comparable à ceux obtenus pour une topologie connexe aléatoire, ce qui porte à croire que l'algorithme est équivalent sur ces deux topologies.

Pour un réseau de 200 noeuds de type arbre, on obtient pour l'erreur : 2,62 - 1,82. A nouveau, ce résultat est comparable à ceux obtenus pour une topologie connexe aléatoire avec peu de connexions. Il semble donc que ce premier algorithme soit peu sensible à la topologie du réseau.

## Algorithme “deterministic”

L'algorithme a été nommé deterministic par opposition au premier algorithme que nous avons écrit qui était probabiliste.

Cet algorithme fonctionne selon une logique de propagation séquentielle (par opposition à une propagation parallèle). Le principe est le suivant:

- Un agent initial possède l'information des tâches à distribuer dans un tableau (contient un compteur du nombre de tâches restantes à distribuer pour chaque type)
- Il choisit une tâche parmi celles à effectuer (selon l'ordre de numérotation) qui devient sa nouvelle tâche
- Il fait passer le tableau à un voisin non occupé en enlevant une tâche au compteur correspondant à celle choisie
- Le voisin non occupé fait de même et retient l'agent qui lui a donné l'information
- Si tous ses voisins sont occupés, il fait remonter le tableau à l'agent parent qui le lui avait donné au départ
- L'agent parent vérifie si il a des voisins non occupés, ou fait remonter à son tour

Lorsque l'agent initial récupère le tableau et que tous ses voisins sont occupés on a effectué un parcours en profondeur du graphe, donc tous les agents sont occupés car le graphe est par construction connexe. Si il ne récupère pas le tableau, c'est que toutes les tâches ont été distribuées. A tout instant de l'algorithme, un seul agent possède le tableau à jour des

tâches restantes à distribuer, et c'est le seul qui est susceptible de se voir attribué une nouvelle tâche.

Une manière globale de visualiser l'algorithme est la suivante: le tableau contenant le nombre de tâche à distribuer parcourt le graphe (selon un parcours en profondeur) et distribue une tâche lorsqu'il rencontre un agent inactif.

Étant donné la construction de l'algorithme, on est assuré que le nombre de tâches à distribuer converge vers 0 (car on ne peut que soustraire au nombre de tâches, pas ajouter) et atteint 0 dans le cas où il y a assez d'agents.

Si il y a plus d'agents que de tâches à distribuer, certains agents seront simplement inactifs à la fin de l'algorithme, puisqu'il n'y aura plus de tâches à distribuer.

Si il y a moins d'agents que de tâches à distribuer, les dernières tâches (par ordre de numérotation) ne seront pas distribuées. L'algorithme distribuera un maximum de tâches, jusqu'à ce que tous les agents soient occupés.

#### Performances de l'algorithme:

Le temps d'exécution ne dépend pas du nombre de tâches à effectuer ou du graphe choisi. Le nombre de type de tâches influence néanmoins de manière linéaire ce temps d'exécution.

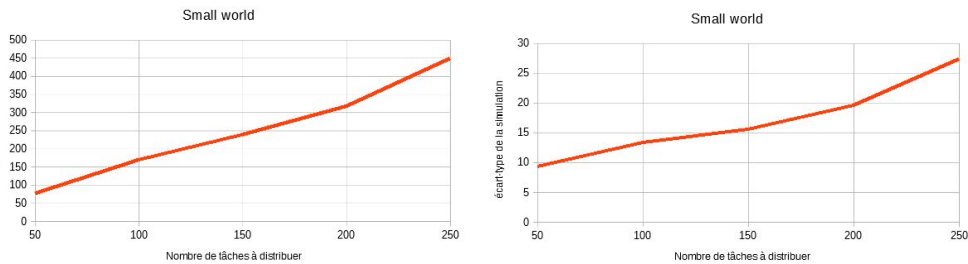
En terme de bande-passante, les seules données transitées sont un tableau contenant autant d'entiers que de types de tâches. Seul un échange est effectué par tick.

Chaque agent doit connaître:

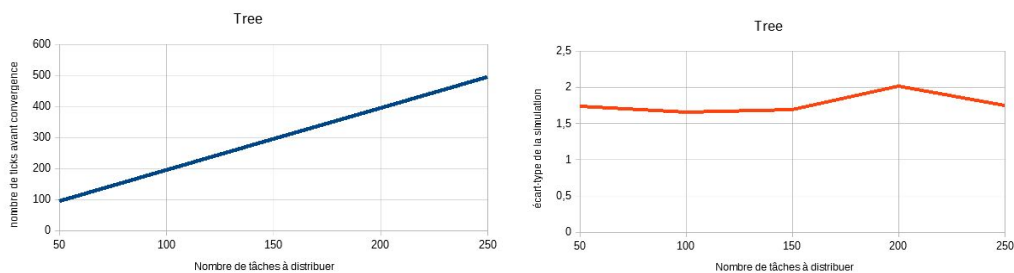
- sa tâche attribuée
- sa disponibilité (tâche attribuée ou non)
- la dernière valeur reçue des tâches à distribuer (le tableau contenant l'information)
- si il possède le tableau à jour (le update-agent qui parcourt le graphe dans l'algorithme)
- l'identifiant de l'agent qui le lui a fourni en premier

## Statistiques:

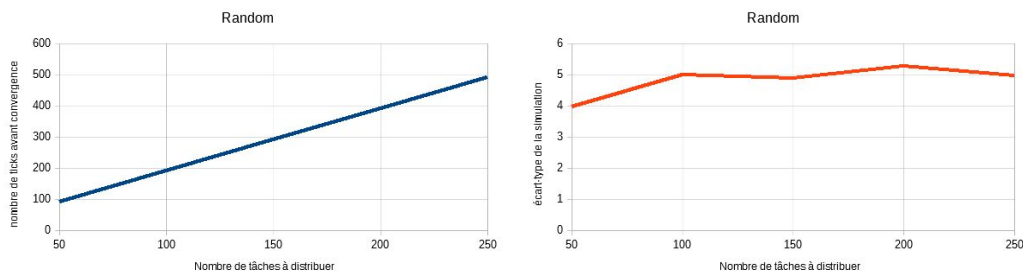
### Small world



### Tree



### Random graph

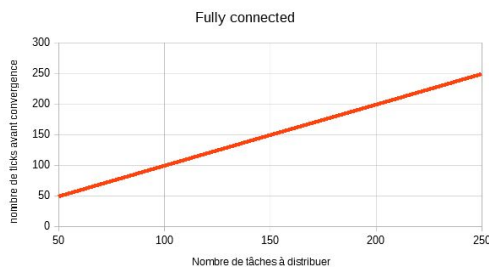


Dans les trois cas précédents, l'évolution du temps de propagation est linéaire en fonction du nombre de tâches à distribuer. Cela ne dépend pas du nombre de types de tâches différents.

L'écart-type évolue également de manière linéaire pour les small worlds, ce qui ne correspond pas aux écarts-type avec d'autres graphes qui eux sont non seulement plus bas mais ne semblent pas varier avec le nombre de tâches.

On le voit ensuite avec les graphes complets (fully connected), le nombre minimal de ticks de convergence est le nombre total de tâches à distribuer. Le nombre de ticks dans le pire des cas sera deux fois le nombre de tâches à distribuer. En effet, l'information ne parcourt pas plus de deux fois une arête: une fois pour propager et une fois pour remonter. Le parcours de l'information forme un arbre couvrant du graphe (car l'information effectue un dfs de ce dernier), ce qui assure que le nombre d'arêtes vaut le nombre de noeuds moins un, et donc dans le cas idéal le nombre d'agents.

## Fully connected



On observe un écart-type de 0 quelle que soit la situation. C'était prévisible, étant donné que le graphe est toujours le même et que le chemin suivi n'influence pas le déroulement de l'algorithme. La configuration du graphe est telle qu'à chaque tick, l'information est propagée à un agent non occupé. On pouvait donc prévoir que le nombre de ticks de convergence était égal au nombre de tâches à distribuer.

## Gossip algorithm

Cette algorithm peut être choisi sur le second menu déroulant et fonctionne de la manière suivante :

A chaque tour chaque noeud va choisir un voisin au hasard avec une probabilité uniforme. Ensuite, il lui envoie ses données. On choisit de ne recevoir les données du noeud qu'on vient de choisir. Ensuite, lorsqu'un noeud reçoit les données d'un autre noeud sous forme de liste, pour chaque élément de la liste reçue il va l'ajouter à ses données si elle n'existe pas (on ajoute donc un noeud au réseau connu) et si l'élément existe déjà, on compare les dates et on garde la plus récente.

Cette méthode permet à chaque noeud d'avoir une vision du réseau. Cette vision s'améliore au cours du temps avec les données qui se propagent le long du graphe.

### Initialisation

L'algorithme peut commencer de deux manières :

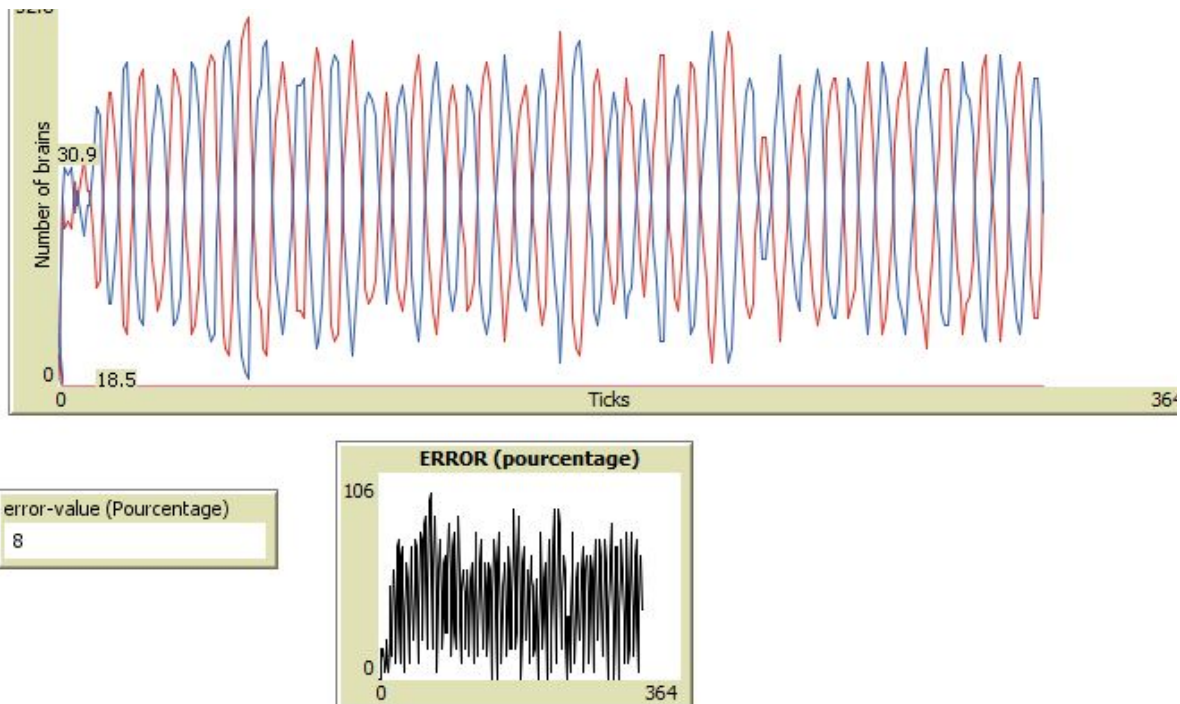
- Si *initialize-gossip* est activé, les noeuds choisissent une tâche au hasard au début de l'algorithme
- Si *initialize-gossip* est désactivé, seul un noeud (choisit au hasard ou par le leader election algorithm) connaît les tâches à distribuer. Les autres noeuds doivent donc attendre que l'information des tâches à distribuer se propage sur le réseau. Une fois que c'est le cas les deux cas se comportent de la même manière

Pour la suite on se supposera que *initialize-gossip* est activé et que les noeuds ont une tâche assignée au hasard dès le départ. Pour la suite, on

assignera 2 types de tâche aux noeuds qui seront répartis uniforméments.  
(Pour 60 agents on aura 30 tâches de chaque types)

### Comportement de l'algorithme sans rectification

En faisant tourner l'algorithme décrit ci-dessus sans modification, il s'avère qu'il n'y a généralement pas de convergence ni de stabilisation des répartitions.



Intuitivement cela semble logique puisque si les noeuds changent trop souvent d'état, les informations qu'on deux noeuds espacés du diamètre du réseaux n'ont pas le temps de d'être croisés.

Pour régler cela, on décide de se concentrer sur le modèle de graphe small-world qui possède un diamètre proportionnel à  $\log(n)$  où  $n$  est le nombre de noeuds.

L'idée est de limiter la fréquence de mise à jour des états du noeud en introduisant une condition: lorsqu'il le souhaite, un noeud donné ne peut changer d'état qu'avec une probabilité égale à  $1/n$ .

L'introduction de cette condition permet bien de converger vers une erreur nulle.

### Choix de la probabilité 1

Pour choisir la probabilité optimale qui permet de converger le plus rapidement, plusieurs probabilités sont envisagées : en  $\frac{a}{\log(n)}$  ,  $\frac{a}{n}$  ,  $\frac{a}{\log(n)*n}$  etc ...

Pour choisir la bonne fonction, on a essayé de déterminer la fonction **f** optimale telle que la probabilité qu'un noeud change d'état soit  $\frac{1}{f(n)*n}$  , l'idée est que si l'hypothèse de départ de prendre 1/n était bonne, la fonction f ne varie pas beaucoup.

## Choix de la probabilité 2

Pour choisir f, on a déterminé expérimentalement f pour n = 30, 40, 50 et 60. La vitesse d'exécution ne permet pas de prendre de plus grandes valeurs pour n.

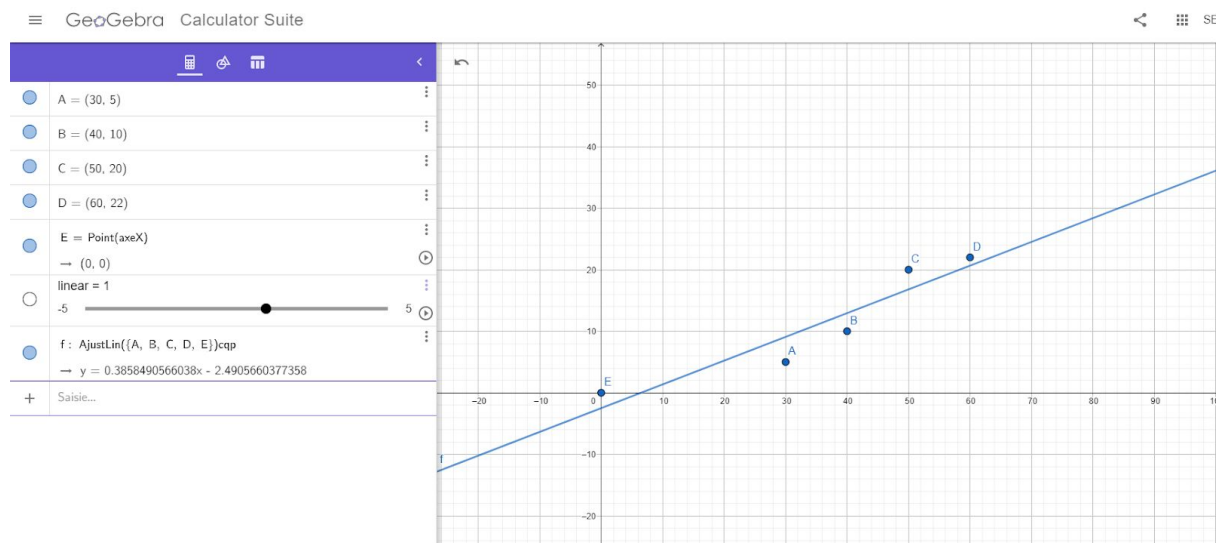
Cela a été fait par un algorithme dichotomique en démarrant par deux valeurs extrêmes de f. Voici un exemple pour n = 30.

f pour n = 30	Temps de convergence moyen (100 essais)	Écart-type (100 essais)
1	47.35	41?64
20	86.84	79.823
10	45.93	37.99
5	41.9	120.9
7	55.86	57.1
<b>6</b>	<b>40.25</b>	<b>47.67</b>
f retenue : 5		

**Voici les résultats:**

n	f
30	6
40	11
50	20
60	22

Malheureusement il faudrait avoir les valeurs de  $f$  pour un plus grand nombre de noeuds pour choisir une modélisation adéquate de  $f$ . Pour simplifier, il a été décidé de modéliser par



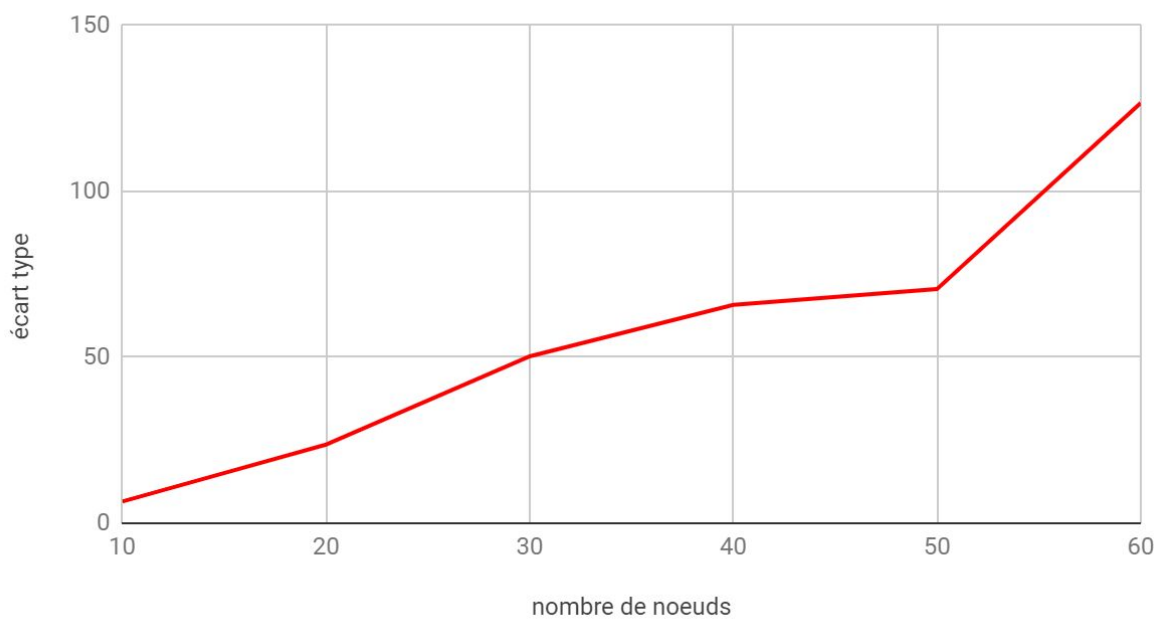
une fonction linéaire.

Performances de l’algorithme

Small word

nombre de noeuds	temps moyen de convergence	écart type
10	9.93	6.51
20	29.84	23.68
30	55.54	50.269
40	79	65.78
50	90.38	70.54
60	138.84	126.63

écart type vs nombre de noeuds

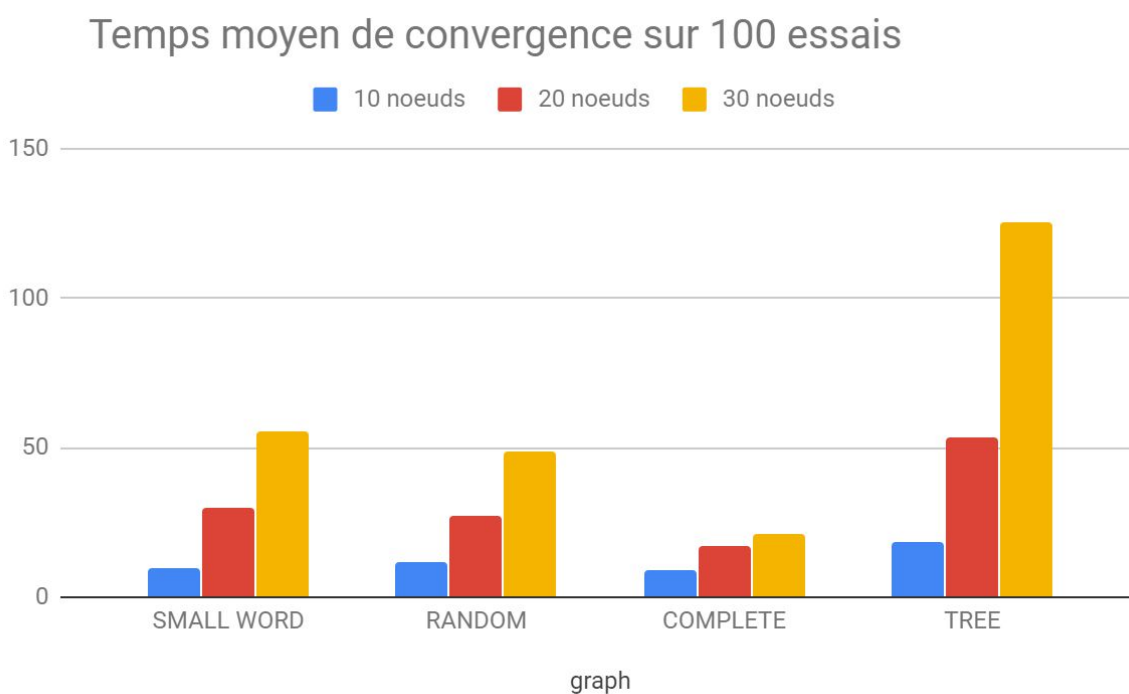


Comme attendu le temps de convergence et l'écart type augmentent avec le nombre de noeuds. Il faudrait plus de données pour modéliser la vitesse de convergence mais la vitesse d'exécution ne permet pas de tester pour un plus grand nombre de noeuds.



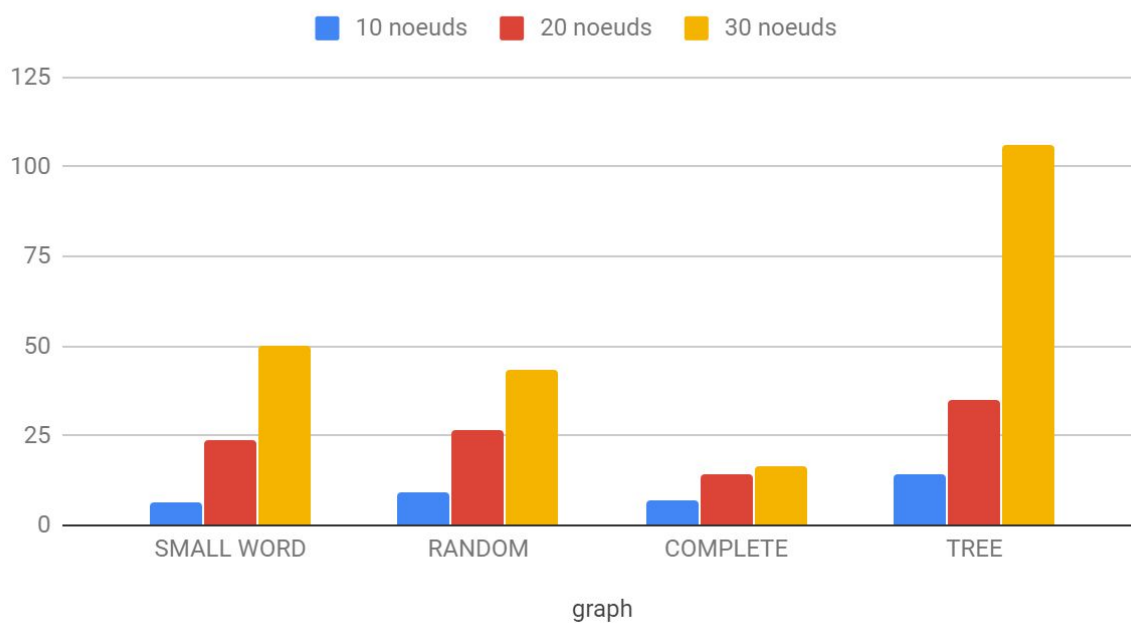
## Influence du type de graphe

Là encore rien de surprenant, le temps de convergence augmente avec le diamètre. Celui-ci est le plus grand avec l'arbre.



L'écart type semble suivre la même tendance.

## ECART TYPE



## Autres performances

Bande passante : Chaque agent aura vers la convergence une liste de données de longueur proportionnelle à la taille du réseau. Il envoie à chaque tick une liste de données de taille  $n$  et reçoit en moyenne une liste de taille  $n$ .

Complexité par agent : Pour savoir si une nouvelle donnée reçue existe déjà ou non, on fait un double parcours de liste d'où une complexité en  $O(n^2)$

Espace mémoire : la liste sera de taille  $n$  au max  $O(n)$  (cf page 1 : registres in/out)

## Pour $n$ types de tâches

Pour plus de 2 types de tâche, l'algorithme converge encore mais il existe des cas où la convergence ne se fait pas ou ne semble pas optimale.

La difficulté est d'étendre la fonction  $f(n,k)$  où  $n$  est le nombre d'agents et  $k$  le nombre de types de tâche. (cf **Choix de la probabilité 1** où est introduite la fonction  $f$ )

## Comparaison des algorithmes

L'algorithme estimation-adjustment apporte une amélioration à l'algorithme probabiliste en réduisant l'erreur de ce dernier, offrant une bonne référence. Celui-ci a pour avantage d'exiger peu de ressources.

Les algorithmes gossip et deterministic trouvent tous deux une répartition parfaite (s'il y a assez d'agents) des tâches. Ces deux derniers peuvent modéliser des situations différentes. L'algorithme gossip met en place une propagation parallèle de l'information alors que l'algorithme deterministic repose sur une propagation séquentielle. L'algorithme gossip peut partir d'une répartition initiale de tâches (obtenue avec l'algorithme estimation-adjustment par exemple) ou peut bénéficier de la désignation d'un sommet privilégié duquel partira l'information pour améliorer ses performances.

En regardant les performances de chacun de ses deux algorithmes, il semblerait que l'algorithme gossip dépende plus fortement de la topologie dans laquelle il est appliqué. Il se montre plus efficace sur un graphe complet par exemple, qui constitue une situation idéale. En revanche, dans le cas d'un arbre, l'exécution de gossip est vite ralentie par le nombre de noeuds (ou, de manière équivalente, de tâches car on prend le cas où les nombres de noeuds et de tâches sont égaux pour les tests). L'algorithme deterministic est assez peu influencé par la topologie, bien qu'il soit plus efficace sur un graphe complet. Dans les deux autres cas testés, les deux algorithmes ont des performances comparables en temps de convergence.

L'algorithme deterministic est le moins gourmand en bande-passante car il transmet des tableau de taille nombre de types de tâches contre des tableaux de taille nombre d'agent pour gossip. Deterministic n'utilise par ailleurs qu'un lien à la fois, ce qui n'est en général pas le cas de gossip.

La quantité d'information stockée sur gossip est plus importante et l'implémentation pour chaque agent est plus longue.

Les avantages de gossip par rapport à deterministic sont qu'il est applicable dans des situations plus large (tâches initiales sur les agents) et que l'on peut améliorer ses performances en faisant varier certains paramètres comme la topologie ou le point de départ de l'information si elle se propage depuis une source.

L'algorithme deterministic se démarque par ses exigences en terme d'espace mémoire et sa complexité d'exécution pour chaque agent.

## Leader designation algorithm

Tous les algorithmes ont besoins d'un point d'entrée où initialiser les tâches. Dans notre programme, ce point d'entrée peut être déterminé de deux façons : aléatoirement (bouton elect-root sur off) ou à l'aide d'un algorithme permettant de déterminer le brain le plus connecté aux autres brain (bouton elect-root sur on).

À l'initialisation, cet algorithme va calculer pour chaque brain la somme des distances qui le sépare des autres brains.

$$S_i = \sum_{j=0}^{n-1} d(\text{brain}_i, \text{brain}_j)$$

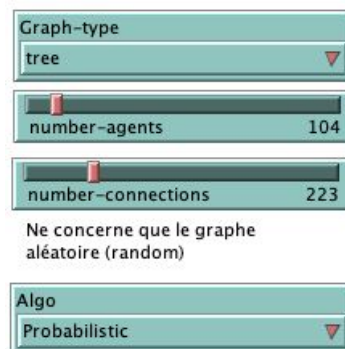
Le point d'entrée sera la brain qui minimise cette distance.

$$\text{brain}_{\text{entrée}} \Leftrightarrow S_{\text{entrée}} = \min_{i \in \{0, \dots, n-1\}} (S_i)$$

Pour un nombre de noeuds important, cette fonctionnalité ralentit considérablement l'étape de setup.

# Manuel d'utilisation du simulateur

## Paramétrer le type de graph



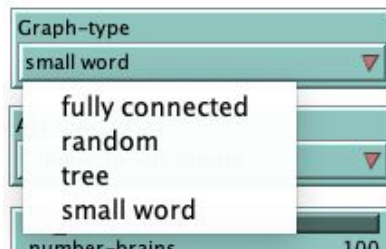
The image shows a configuration panel for graph parameters. It contains four main controls:

- Graph-type**: A dropdown menu currently set to "tree".
- number-agents**: A slider bar with a red indicator and a numerical value of 104.
- number-connections**: A slider bar with a red indicator and a numerical value of 223.
- Algo**: A dropdown menu currently set to "Probabilistic".

Below the sliders, there is a note: "Ne concerne que le graphe aléatoire (random)".

Pour paramétrer un graph on utilise les 4 boutons ci-dessus.

- **Graph type** permet de choisir parmi 4 topologies : fully-connected, random, tree et small world.



- **Number-of-agents** permet de choisir le nombre d'agents qu'on veut dans le graph.

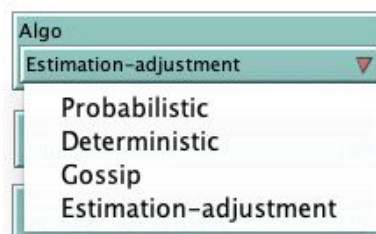


- **Number-connections** est un réglage qui n'intervient que lorsqu'on génère un graph avec le topologie "random". Si ce nombre est inférieur ou égal au nombre d'agents, le graph random généré sera un arbre de type tree. Quand **number-connections** augmente, cela rajoute des connexions aléatoires entre agents, jusqu'à ce que tous les agents soient connectés entre eux, formant alors un graph fully connected.

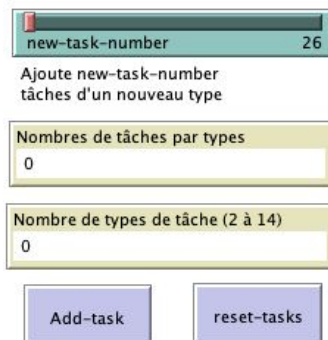


ne concerne que le graphe  
aléatoire

- **Algo** permet de choisir l'algorithme utilisé pour réaliser l'affectation des tâches parmi 4 choix : Probabilistic, Deterministic, Gossip et Estimation-adjustment.



## Paramétrer les tâches



new-task-number 26

Ajoute new-task-number tâches d'un nouveau type

Nombres de tâches par types  
0

Nombre de types de tâche (2 à 14)  
0

Add-task reset-tasks

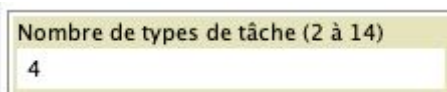
- Pour ajouter une tâche, vous devez sélectionner le nombre d'agents nécessaires à la réalisation de la tâche grâce au curseur **new-task-number**, puis cliquez sur **Add-task** pour ajouter la tâche.
- **Nombre de tâches par types** affiche une liste dans laquelle on trouve à l'indice  $i$  le nombre d'agent nécessaires pour réaliser la tâche  $i$ .



Nombres de tâches par types  
[74 42 95 132]

*Dans cet exemple, la tâche 1 nécessite 74 agents, la tâche 2 42 agents, la tâche 3 95 agents et la tâche 4 132 agents*

- **Number-of-types** indique le nombre de types de tâches que vous avez ajoutés.



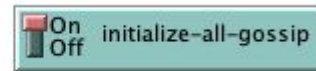
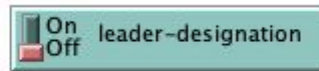
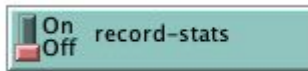
Nombre de types de tâche (2 à 14)  
4

- Enfin, **reset-tasks** supprime toutes les tâches que vous avez ajoutées.

**Attention :** Pour lancer une simulation, il faut rentrer au moins 2 tâches dans l'algorithme. Si vous n'en entrez pas assez, le message d'erreur "You must add at least two tasks !" apparaîtra dans la console au moment de l'initialisation de la simulation.

**Attention :** L'algorithme Estimation-adjustment ne fonctionne que pour 2 tâches uniquement. Si vous en entrez plus de 2, le message d'erreur "With Estimation-adjustment only 2 types of task can be added" apparaîtra dans la console si vous essayez d'en ajouter une troisième.

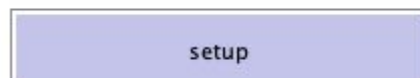
## Paramétrer la simulation

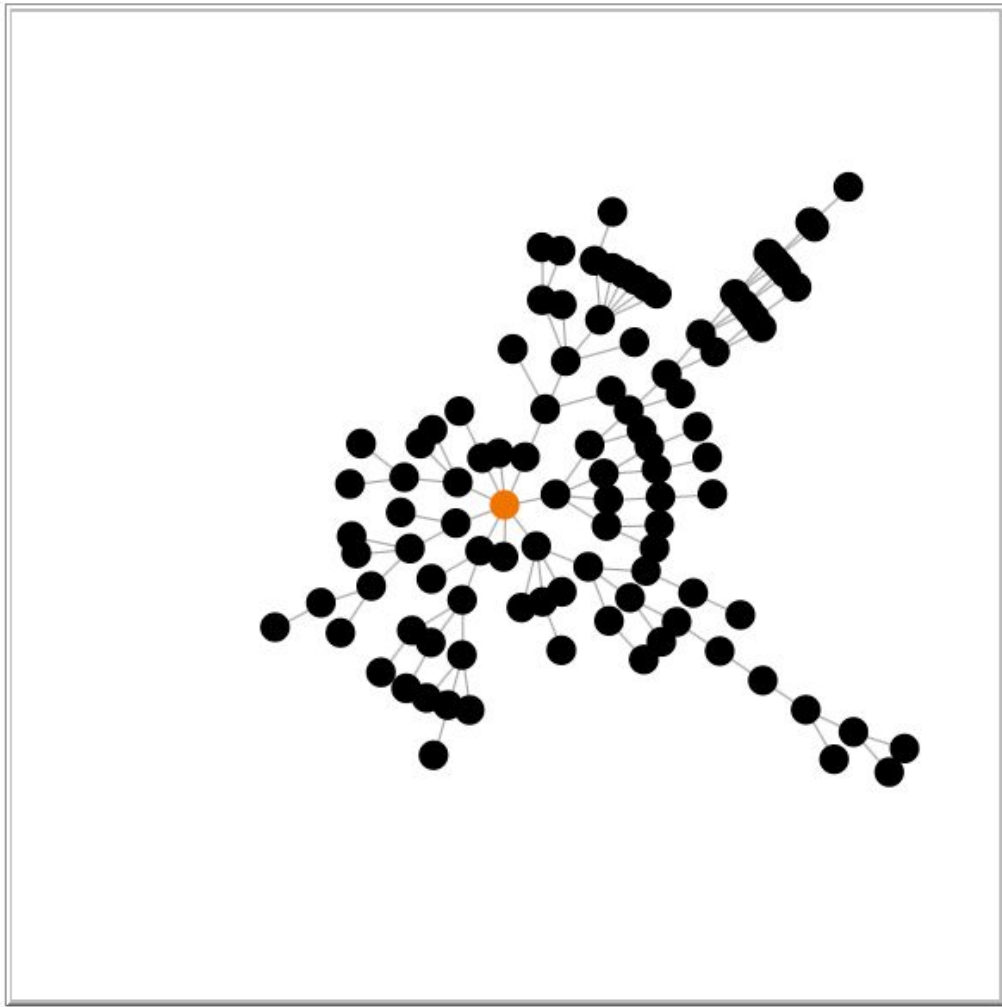


Avant de lancer la simulation, on peut choisir d'activer un certain nombre d'outils.

- **Record-stats** est une option très utile pour l'évaluation des performances d'un algorithme puisqu'elle permet de lancer 100 simulations à la suite et d'enregistrer les résultats dans un fichier text. Une simulation s'arrête quand l'erreur atteint 0% et la suivante commence instantanément sur un autre graph généré avec les mêmes paramètres.
- **Leader-designation** détermine la manière de choisir l'agent d'entrée de l'information quand on exécute les algorithmes probabilistic, deterministic et gossip (si initialize-all-gossip est off). Si **leader-designation** est on, le point d'entrée sera le noeud qui minimise le temps de propagation de l'information aux autres noeuds. Sinon, le point d'entrée sera choisi au hasard.
- **Initialize-all-gossip** est un réglage propre à l'algorithme gossip qui permet s'il est sur on d'initialiser les agents avec l'un des types de tâches de la simulation pris au hasards. Si **initialize-all-gossip** est sur off, alors tous les agents sauf le point d'entrée n'auront pas de tâche au début de la simulation

Voilà, maintenant que tout est paramétré on peut appuyer sur **setup** pour initialiser la simulation !





*Graph de topologie tree avec 104 agents initialisé avec le réglage leader-designation sur on pour une simulation avec 4 tâches utilisant l'algorithme Probabilistic.*

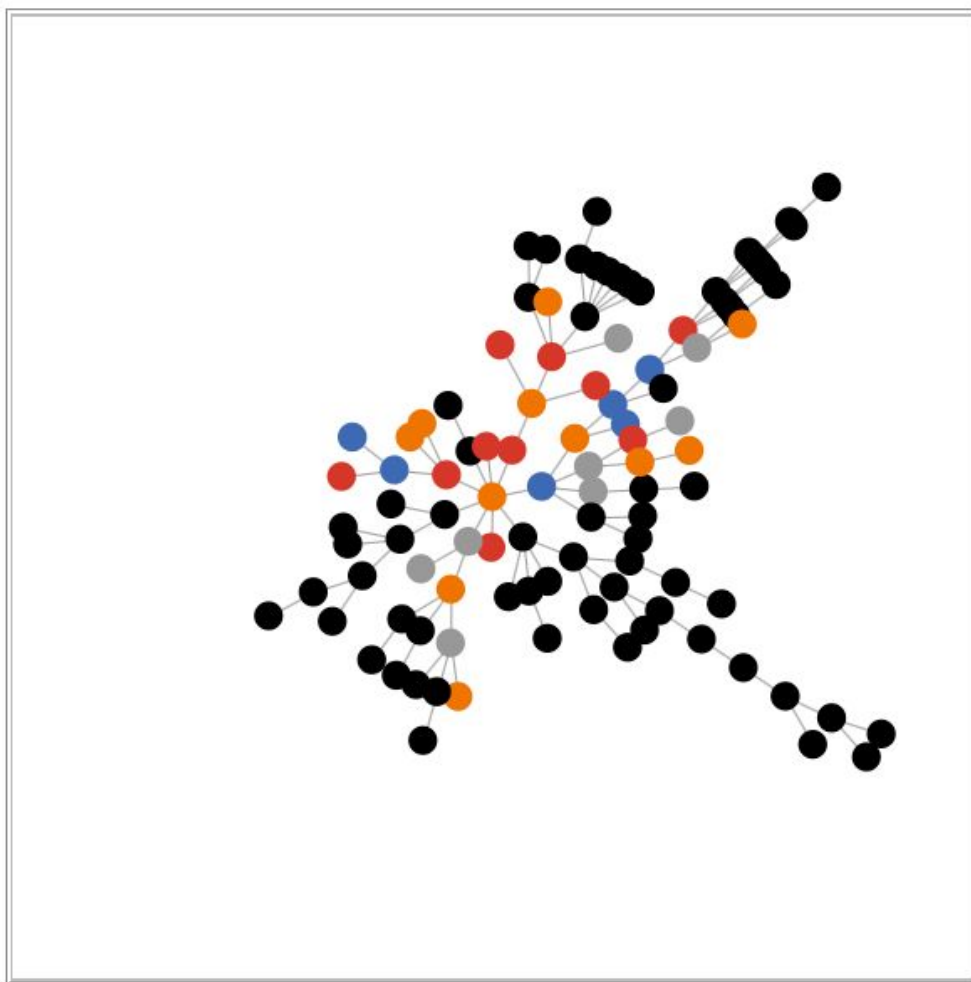


## Lancer et observer la simulation

Pour lancer 1 Tick de la simulation, appuyez sur **go (1 step)**. Pour laisser la simulation se dérouler en continue, appuyez sur **go**.

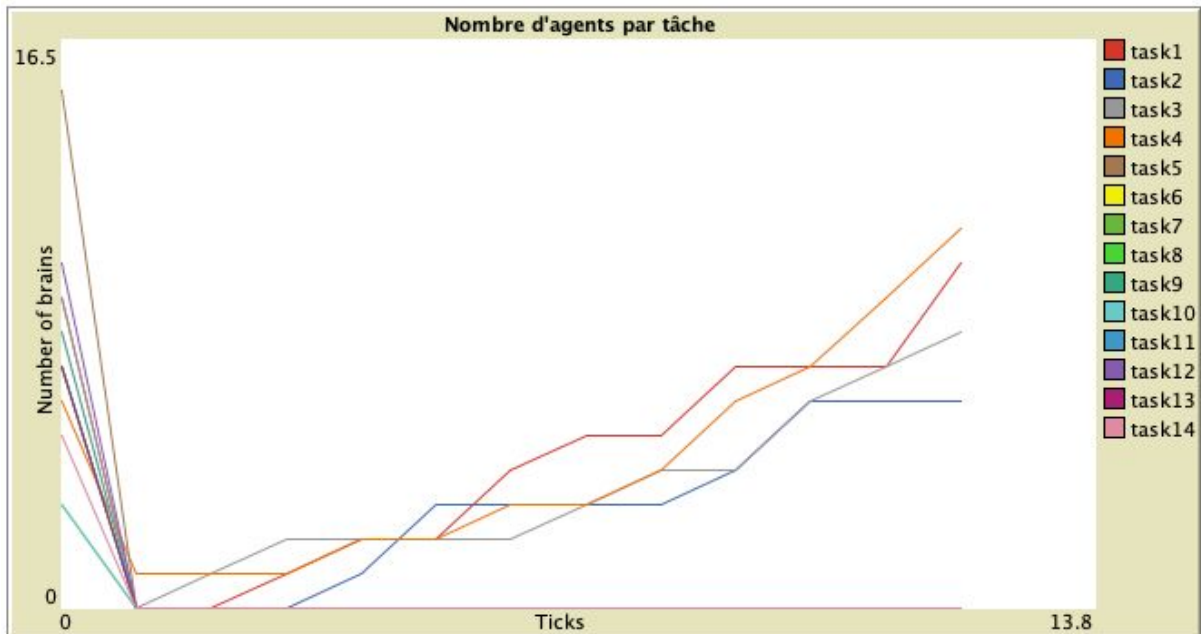


Dans la **fenêtre de visualisation du graph**, vous verrez au fur à mesure les agents prendre la couleur de la tâche auquel ils sont affectés.



*Meme simulation après 10 ticks.*

En même temps, vous pouvez observer le nombre d'agents affecté à chaque tâche en fonction du temps sur le graphique **Nombre d'agent par tâche**.



Enfin, les fenêtres **ERROR** permettent de visualiser la moyenne des erreurs relatives par type de tâche en fonction du temps.

