

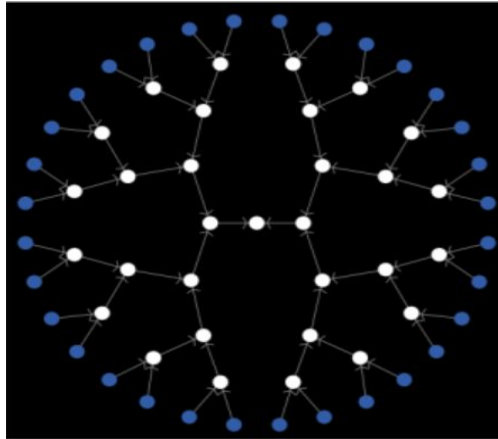


Autonomous Growth and Task-distribution in Hierarchical Organisations

Groupe [2] :

- DEVAUX Etienne
- EL AISSAOUI Anas
- EL KHEIR Yassine
- DE PORCARO Loïc
- LOH Johnathan

Ce projet vise à développer un modèle multi-agents pour simuler la croissance progressive et le possible rétrécissement des organisations hiérarchiques. Il vise à étudier les taux de réussite des différents processus de coordination des agents ainsi que la distribution des tâches d'une façon optimale et selon la nécessité alors que la topologie évolue.



❖ **Modèle de croissance et Modèle de retrait**

- Simulation optimale en terme de nombres des sommets.
- Simulation Hiérarchique Parfaite (Workers au même niveau).

❖ **Distribution des tâches**

- Distribution à l'aide d'un système de communication programmé.
- Distribution aléatoire en fonction d'une liste des ratios.
- Distribution aléatoire Bernoulli.

❖ **Etude de la Complexité et la Convergence**

- Etude de la complexité (Ressources de Communication)
- Etude de la convergence

❖ Modèle de croissance et modèle de retrait

- Simulation optimale en terme de nombres des sommets.

Il s'agit dans un premier temps de produire un algorithme permettant de modéliser une structure hiérarchique en fonction du nombre de workers, de managers et du nombre de worker par manager (Capacité). est défini et à chaque itération Afin d'aboutir à cette organisation, le nombre d'enfants par manager et à chaque itération un travailleur peut être ajouté ou supprimé, et les interconnexions sont ajoutées directement. Si ce worker se trouve avec un fils, il devient manager. Notre modèle peut également ajouter plusieurs travailleur en même temps.

Explication de l'algorithme :

Chaque turtle (Worker/Manager) est défini par des attributs qui le rend unique :

- *id* :: Un nombre pour s'assurer de l'unicité du sommet.
- *task* :: -1 (manager) , $n \geq 0$ (worker)

On utilise également une variable globale comptant le nombre d'agents dans la structure : *number*.

On observe qu'avec une construction où les id sont affectés de manière croissante, la détermination de l'id du manager se fait à l'aide d'une simple division euclidienne par le nombre d'agent que peut gérer un manager *c*. Ainsi, lorsque l'on crée un nouvel agent, il suffit de lui donner pour *id* la valeur *number* (que l'on incrémente ensuite de 1) et de le lier avec son manager dont l'id est $\lfloor \text{number} / c \rfloor$. Si ce manager est un worker, alors on le passe en manager.

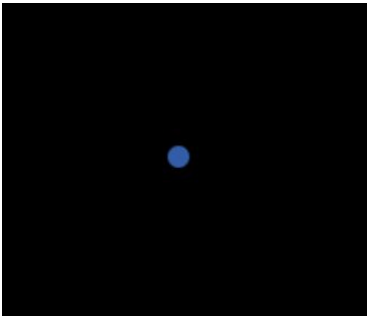
Dans le cas d'une suppression, le processus est similaire et dans le cas où un manager n'a plus de worker, celui-ci devient lui-même un worker.

Pour ajouter ou supprimer un nombre précis de travailleur, deux cas particuliers sont à traiter :

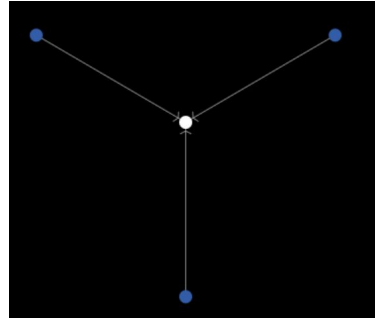
- lors d'un ajout, si on transforme un worker en manager, alors il faut à nouveau ajouter un agent afin de "vraiment" ajouter un worker à la structure
- lors d'une suppression, si un manager devient un worker, alors il faut à nouveau supprimer un agent afin de réduire le nombre de worker

Finalement, pour ajouter ou supprimer plusieurs workers, il suffit d'itérer un ajout ou une suppression de workers un nombre suffisant de fois.

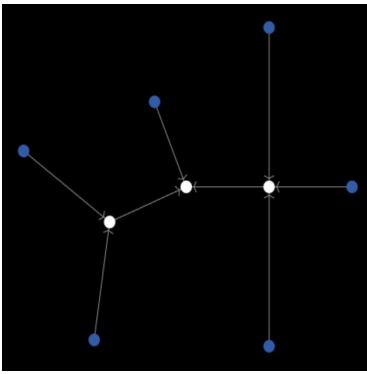
Modélisation simple : (Capacité = 3)



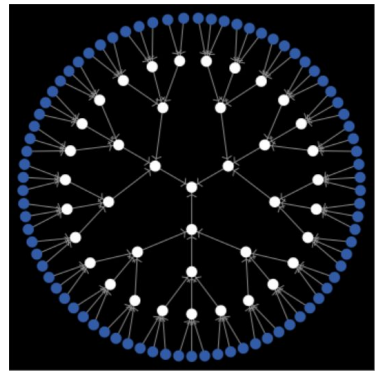
Ajouter un seul Worker



Lors de l'ajout de deux workers
l'algorithme ajoute un manager



On voit que la structure est optimale
en terme de nombres sommets



Arbre de 80 Workers
avec une capacité de 3

- Simulation Hiérarchique (Workers au même niveau)

La procédure de l'algorithme est différente, on lance au début les workers qu'on veut sans manager, et on ajoute selon la nécessité des managers sous la contrainte de capacité (nombre des workers que le manager peut manager), et que les workers soient tous au même niveau.

La procédure de "Delete Worker" est implémenté comme l'inverse de la procédure l'ajout d'un worker, on supprime un worker (de plus grand ID), ainsi également l'algorithme update la hauteur et le nombre nécessaire des managers.

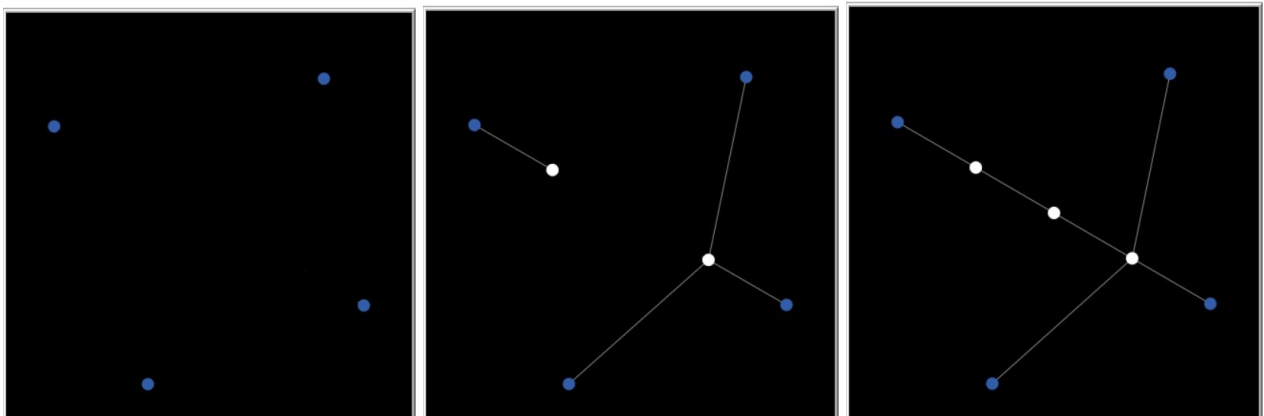
Explication de l'algorithme :

Chaque worker/manager (Turtle) est bien défini par :

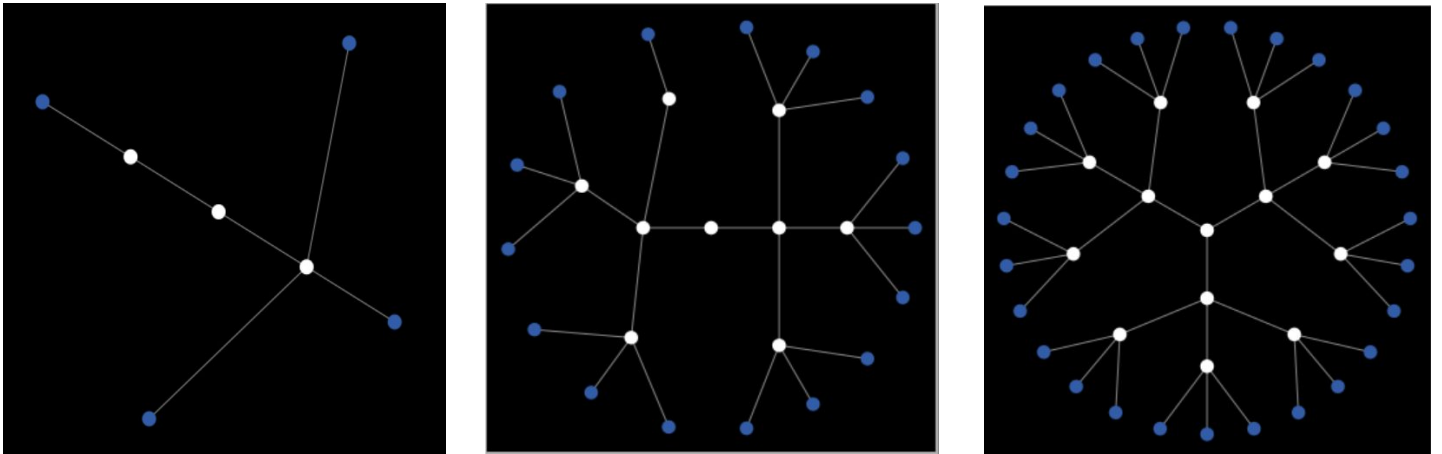
- id :: Un nombre pour s'assurer de l'unicité du sommet.
- isWorker :: Un Booléen pour différencier un worker et un manager
- level :: La profondeur (à quel niveau se trouve)
- parent-id :: id du parent
- child-id-list :: une liste des ID des fils (Workers ou Managers)

L'algorithme implémenté est équivalent à un algorithme combinatoire, on lance les workers sans managers initialement, et on regroupe des ensembles des workers selon la capacité (par exemple si la capacité égale à 3 : générer des ensembles de 3 workers différents), l'algorithme ajoute pour eux directement un manager, qui forme les managers du plus haute niveau des workers. Après la création des managers, une couche au-dessus des workers, on refait la même procédure pour les vertex de haut niveau (Managers intermédiaires), jusqu'à arrivé à la racine. Cela signifie que tous les workers auront donc la même distance de la racine.

Pour illustrer notre propos, on prendra un exemple avec 4 workers avec une capacité de 3 dans la figure de gauche ci-dessous. Les 3 premiers workers seront affectés à un manager, et le dernier sera également affecté à un (dans la figure du milieu). Enfin, les 2 managers seront affectés à un manager aussi de plus haute niveau comme la figure de droite ci-dessous.



On fait notamment un update de cette méthode lors de l'ajout ou la suppression d'un worker (un ensemble).



Pour les deux approches, on a ajouté des boutons pour lancer un nombre données des workers, ainsi que les supprimer à la fois dans un seul tick (Add-Workers, Delete-Workers)

◆ Distribution des tâches

Distribution à l'aide d'un système de communication programmé:

Données : une structure hiérarchique évolutive, p le nombre de tasks, c le nombre d'agents que peut gérer un manager et $ratios$ la liste des poids de ces tasks.

Hypothèses : chaque manager connaît la liste $ratios$ ¹ et l'information se propage sur une distance de 1 chaque tour.

Note : dans tous les exemples qui suivent, on considère que chaque manager peut manager 3 agents et que l'on travail avec 4 tâches.

Cette distribution repose sur un système de communication parents-enfants. On peut la décomposer en deux procédures qui s'effectuent à des instants différents :

- la communication, c'est-à-dire lorsque les couples fils-pères s'envoient des informations, qui a lieux lors de la transition entre deux tours
- le traitement des informations par chaque agent qui a lieux lors du tour

Pour conserver cette organisation en deux temps, nous avons implémenté deux fonctions. *Update* pour simuler une phase de communication entre deux tours et *communicate* qui traite les informations reçues. Nous avons procéder de cette manière en deux temps distincts afin de ne pas avoir à se préoccuper de l'ordre avec lequel les agents effectuent le traitement de l'information et pour être certain que l'information ne se propage que d'une distance 1 à chaque tour, ce qui était une de nos hypothèses.

Afin de réaliser cette organisation, nous avons pourvu nos agents de plusieurs blocs d'informations :

❑ **info** : liste du nombre de travailleur descendants de l'agent associé à chaque tâche

ex : [0 2 3 0] => l'agent a sous ses ordres (directs ou indirects) 0 travailleurs de tâche 0, 2 travailleurs de tâche 1, 3 travailleurs de tâche 2 et 0 travailleurs de tâche 3.

❑ **info-from-childs** : liste des listes *info* de tous les enfants directs de l'agent

ex : [[0 1 0 0] [0 1 1 0] [0 0 2 0]] => le fils 0 de l'agent a pour info [0 1 0 0], c'est-à-dire qu'il a sous ses ordres uniquement un travailleur de tâche 1 (dans ce

¹ *ratios* : liste de longueur le nombre de tâches contenant les poids de chaque tâche. Dans le cas de 2 tâches, si *ratios* = [1 2], alors l'objectif est d'avoir deux fois plus de worker alloués à la tâche 0 qu'à la tâche 1.

cas, il est lui-même travailleur), le fils 1 de l'agent a pour info [0 1 1 0] et le fils 2 de l'agent a pour info [0 0 2 0].

- ❑ **info-from-childs-new** : liste possédant le même format que la liste *info-from-childs*. Cette list sert à réaliser la communication lors de l'intervalle entre deux tours.
- ❑ **orders-received** : liste du nombre de travailleurs descendants de l'agent que celui-ci doit associer à chaque tâche
ex : [1 1 2 1] => l'agent doit associer 1 travailleur sous ses ordres à la tâche 0, 1 à la tâche 1, 2 à la tâche 2 et 1 à la tâche 3.
- ❑ **orders-received-new** : liste possédant le même format que la liste *orders-received*. Cette list sert à réaliser la communication lors de l'intervalle entre deux tours.
- ❑ **orders** : liste des listes au format *orders-received* que l'agent transmet à chacun de ses enfants directs.
ex : [[1 0 0 0] [0 1 0 0] [0 0 2 1]] => le fils 0 de l'agent reçoit pour ordres [1 0 0 0], c'est-à-dire qu'il doit associer un travailleur sous ses ordres à la tâche 0 (dans ce cas, il est lui-même ce travailleur et doit dès lors effectuer la tâche 0), le fils 1 reçoit les ordres [0 1 0 0] et le fils 2 reçoit les ordres [0 0 2 1]

Lors d'un tour, on appelle d'abord la fonction *update* chez tous les agents et ce dans un ordre quelconque. Chaque agent remplace alors respectivement ses valeurs *info-from-childs* et *orders-received* par *info-from-childs-new* et *orders-received-new* et réinitialise ces dernières à des listes vides. Cela permet, comme nous le préciserons plus loin, de simuler une communication prenant place lors des intervalles entre les tours.

Puis chaque agent exécute la fonction *communicate*, toujours dans un ordre quelconque. Cette opération se décompose en 2 étapes distinctes et indépendantes permettant une communication vers le haut de la hiérarchie et vers le bas de celle-ci. Ces fonctions s'exécutent différemment si l'agent en question est un worker ou un manager.

- Dans le cas d'un worker, si celui-ci reçoit des ordres, c'est-à-dire que *orders-received* est non vide, alors il change de task et applique celle donnée par l'ordre en question, liste de la forme [0 0 1 0] avec un unique 1 et des 0.
Puis il constitue sa liste *info* en créant une liste de 0, [0 0 0 0], et en plaçant un unique 1 à l'emplacement correspondant à sa tâche.
- Dans le cas d'un manager, si celui-ci reçoit des ordres, c'est-à-dire *orders-received* non vide, il va alors compter le nombre de workers descendant de chacun de ses fils directs grâce à la liste *info-from-childs*² et distribuer les tâches qu'il doit allouer à chacun de ses

² Calcul : nombre de worker sous le fils i = somme des éléments de la liste *info-from-childs[i]* (info du fils i).

enfants. S'il ne reçoit pas d'ordres, il va en générer lui-même à l'aide de la liste *ratios*³ puis les distribuer.

Puis, il constitue sa liste *info* si sa liste *info-from-childs* est non vide à partir des listes *info* de ses enfants contenues dans la liste *info-from-childs*⁴.

Ensuite, que l'agent soit un manager ou un worker, il envoie les informations qu'il vient de traiter. Il va donc envoyer *info* (si *info* est non vide) à son manager et directement le placer à l'emplacement qui lui correspond dans la liste *info-from-childs-new* de son manager.

Si l'agent considéré est un manager, il envoie ses ordres à chacun de ses enfants sous forme d'une liste *orders-received*. Pour cela, il lui suffit de parcourir sa liste *orders*⁵ et pour chaque élément de cette liste envoyer celle-ci au fils correspondant.

On a ainsi pu implémenter un système de communication couplé à un traitement de la demande permettant d'obtenir une répartition des tasks optimale pour la valeur de *ratios* entrée par l'utilisateur.

Distribution aléatoire en fonction d'une liste des ratios

Données: *n* nombre de workers et *p* le nombre de tasks.

Avant de présenter la logique de cette distribution aléatoire, on veut bien mettre en exergue la liste de convergence attendue par ces différentes méthodes aléatoires.

$$X_n \xrightarrow{\mathcal{L}} X$$

On appelle cette liste, la liste de distribution parfaite, cette dernière on la calcule par une méthode mathématique bien défini,

- Soit *n* le nombre des workers, et *p* le nombre de task
- Soit *q* la partie entière de la division de *n/p* ($E(n/p)$)
- Soit *r* le reste de la division de *n/p* ($r = n - (E(n/p) * p)$)

la liste parfaite de distribution sera :

$$L = [q \text{ for } i \text{ in range } (n)]$$

*après on ajoute aux *r* premiers termes un plus*

³ Calcul : pour chaque tâche, on alloue la partie entière de la valeur voulue (généralement non entière). Puis s'il reste des workers sans tâches, on ajoute à la task de plus grand poids un worker, puis s'il reste encore des workers sans task, on ajoute à la task avec le deuxième plus grand poids un worker et ainsi de suite jusqu'à ce que tous les workers aient une task.

⁴ Calcul : *info*[*i*] du parent = somme des *info*[*i*] de ses fils, c'est-à-dire somme des *info-from-childs*[*child*][*i*] pour tous les enfants du parent.

⁵ Rappel : les éléments de la liste *orders* sont des listes de forme *orders-received*.

Exemple :

- soit le nombre des workers égale à 13
- soit le nombre des tasks égale à 4

❑ la partie entière : 3

❑ le reste : 1

⇒ Donc la liste parfaite : [4 3 3 3]

Donc, dans un premier temps, on a réalisé une distribution parfaite.

On obtient alors une liste parfaite de distribution, il s'agit de la liste des workers par task

Pour la procédure de cette première distribution, au lieu de bien d'enchaîner la liste des ordres (comme celle de la méthode auparavant), les managers renvoient des ordres aléatoires selon une loi de probabilité en fonction des ratios définis auparavant, ce qui enchaîne le système à un état stable (après un nombre raisonnable de tentative (on va étudier la convergence après) cela conduit à une distribution égale à la liste de distribution parfaite).

Distribution aléatoire Bernoulli

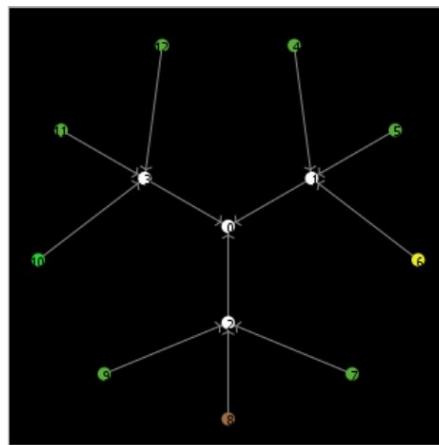
- ❑ Cette distribution est plutôt semi-Hiérarchique, on a lancé cette distribution pour comparer les différentes méthodes développées ainsi que discuter les faillites du système non hiérarchiques.
- ❑ C'est plutôt le grand manager (avec un id bien connu = 0 (le premier sommet créé)) qui distribue les tâches aux workers.
- ❑ Les tâches seront distribuées, par le grand manager, selon une loi de Bernoulli, une probabilité p pour que la tâche soit bien distribuée, et $1-p$ pour une tâche différente.
- ❑ Comment s'assurer que c'est la bonne tâche attribuée au worker ? , les managers du premier niveau (qui gèrent directement les workers), sont bien définies par une liste des tâches attendues de la part de leurs workers. (Bien détaillé dans l'exemple au dessous)
- ❑ En termes de procédure et structures utilisés, pour les managers du premier niveau sont évalués par une liste des tâches de leur workers. le manager est bien défini par son id unique égale à 0. pour connaître les workers, on fait un parcours linéaire par la méthode "ask turtles with" pour chercher les sommets qui n'ont pas la couleur "White" (qui ne sont pas des managers). pour chaque worker sélectionné, le grand manager lui attribue une tâche selon une loi Bernoulli. le worker dont on a attribué la tâche remonte à son manager (Chaque Worker connaît bien l'id de son père "previous"), compare la tâche attribuée par le grand manager par la

tâche attendue par son manager (On a dit auparavant que les managers du premier niveau ont une liste des tâches attendues par leurs workers)

Illustration :

Soit initialement la distribution :

- ❑ Le grand manager (la racine) est bien d'id égale à 0
- ❑ chaque manager pointe vers ces workers/Managers



Manager	Tâches attendues
ID = 1	[0 0 0]
ID = 2	[1 1 1]
ID = 3	[2 2 2]

Comment ces listes seront attribués et c'est quoi la procédure ?

- on a créé une méthode qui permet de diviser la liste parfaite vu au dessous, selon les managers du premier niveau selon la nécessité des workers et selon la priorité des id (s) (le manager qui a l'id le plus faible est le plus prioritaire (car c'est lui qui aura un nombre complet des workers))

ex.: soit la liste parfaite de répartition $L = [2 \ 1]$

et il y'a deux managers 1 : avec le nombre de workers = 2

2 : avec le nombre de workers = 1

Le manager 1 : reçoit la liste des tâches $[2 \ 0]$

Le manager 2 : reçoit la liste des tâches $[0 \ 1]$

- après la division des listes des tasks, le grand manager sélectionne un worker (par préférence de notre programme, on commence par celui de plus petit id), la racine lui attribue une tâche selon une loi bernoulli, après le worker vérifie si cette tâche appartient à la liste des tâches de son manager, si oui, on enlève cette task de la liste des tasks chez le manager, et on attribue plus la tâche à ce worker, sinon, on refait la même procédure jusqu'à tomber sur une task appartenant à la liste des tasks chez son manager.

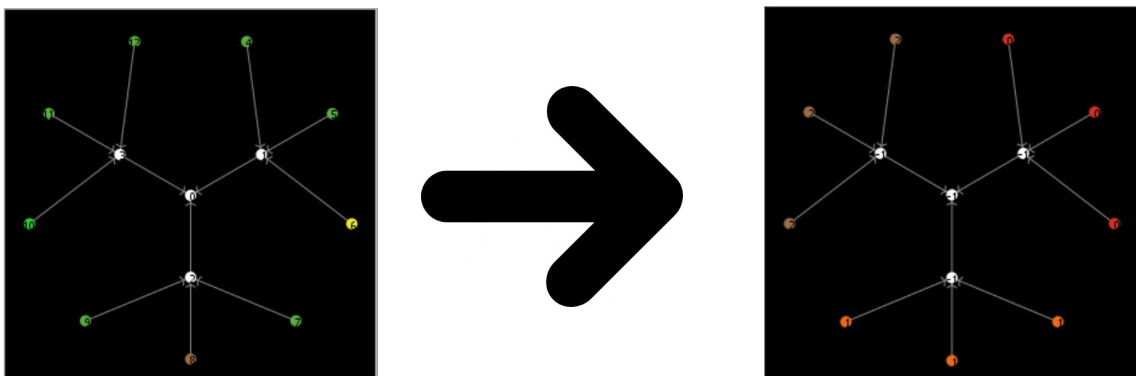
ex.: soit le manager **1** avec la liste des tâches $L = [2 \ 0]$

soit Worker **0** son premier agent (l'agent du manager d'id = **1**)

la racine lui attribue une tâche aléatoire selon la loi de bernoulli,

- ❑ premier essai : la tâche attribuée au Worker est **1**
la tâche **1** n'appartient pas à la liste (car $L[1] = 0$)
Donc on refait la procédure
- ❑ deuxième essai : la tâche attribué au Worker est **0**
la tâche **0** appartient à la liste (car $L[0] = 2$)
On supprime cette tâche de liste donc L devient **[1 0]**
On fixe la tâche du Worker **0**

Résultat Final



❖ Etude de la Complexité et la Convergence

Etude de la complexité spatiale (Ressources de Communication)

- **Distribution à l'aide d'un système de communication programmé et la première Distribution aléatoire.**

Il s'agit de déterminer la place mémoire nécessaire pour que l'algorithme fonctionne correctement. Il faut optimiser cette donnée.

Nous avons listé au cours de ce rapport les différentes variables stockées par les agents composant notre structure hiérarchique. Parmi celles-ci, les ressources de communication sont les suivantes :

- α **info** : liste d'entiers de longueur le nombre de tasks t
- α **info-from-childs** et **info-from-childs-new** : liste de listes d'entiers listant les *info* des enfants d'un agent. La longueur de la liste est égale au nombre d'agents que peut gérer un manager c et la longueur d'un élément de cette liste est égale au nombre de tasks t => taille mémoire : $c*t$ (deux fois avec *info-from-childs* et *info-from-childs-new*)
- α **orders-received** et **orders-received-new** : liste d'entiers de longueur le nombre de tasks t (deux fois avec *orders-received* et *orders-received-new*)
- α **orders** : liste de liste d'entiers listant les *orders-received* qu'il faut fournir à chacun des enfants de l'agent. La longueur de la liste est égale au nombre d'agents que peut gérer un manager c et la longueur d'un élément de cette liste est égale au nombre de tasks t => taille mémoire : $c*t$

Taille mémoire totale pour chaque agent :

$$3*t + 3*t*c = O(t*c)$$

- **Distribution aléatoire Bernoulli.**

Il s'agit de déterminer la place mémoire nécessaire pour que l'algorithme fonctionne correctement. Il faut optimiser cette donnée.

L'idée de la distribution semble minimiser les ressources de système de communication puisque la structure n'est pas totalement hiérarchique (Semi Hiérarchique, puisque la communication se fait juste entre la racine, les workers, et les managers du premier niveau), mais en pratique cela prend beaucoup

d'espace, car pour chaque turtle (Worker), est bien identifié par son id et sa propre task chez la racine, donc la complexité réduite en hiérarchie se multiplie dans cette structure chez la racine.

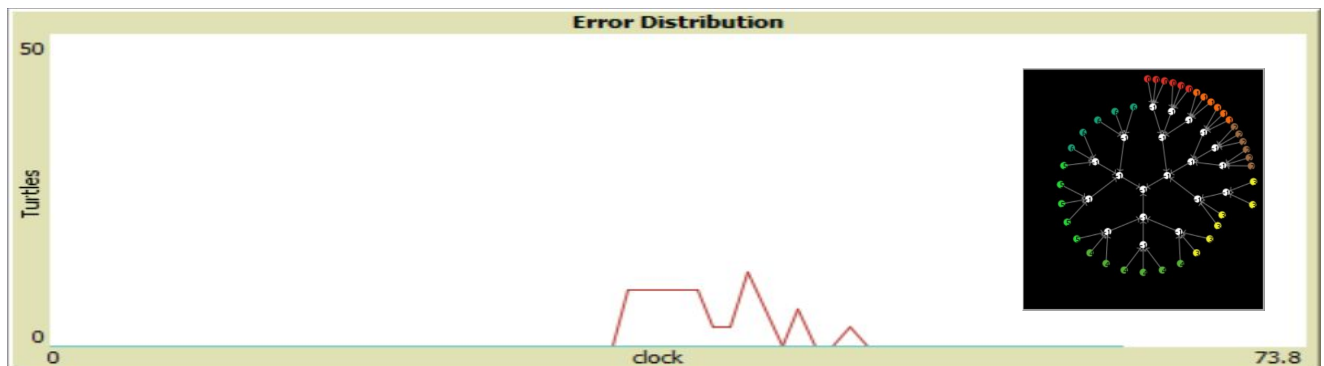
Taille mémoire totale pour assurer la communication :

- soit t le nombre des tasks
- soit c la capacité de chaque manager
- soit n le nombre des workers

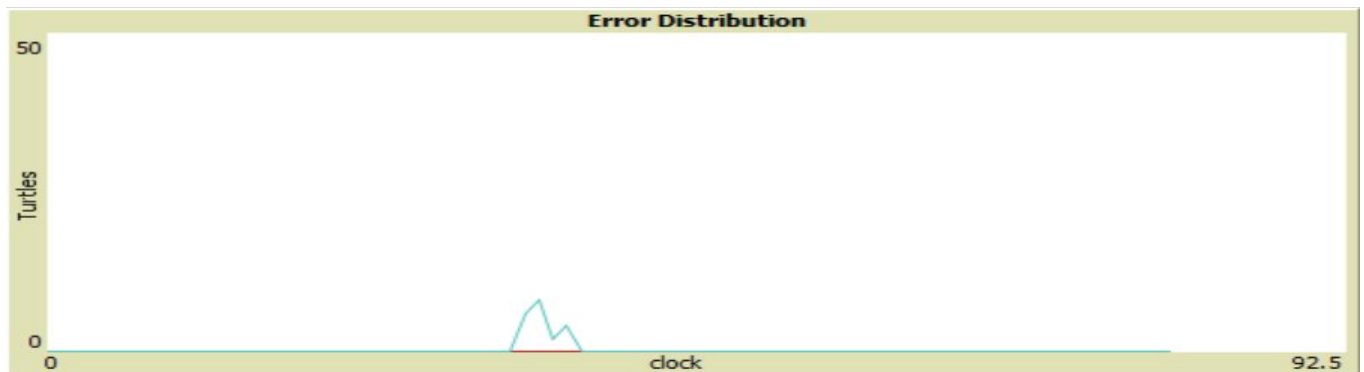
Le parcours des workers, l'attribution de la tâche et la vérification faite par le worker résultent une complexité en fonction $O(n*c*t*t)$

Etude de la convergence:

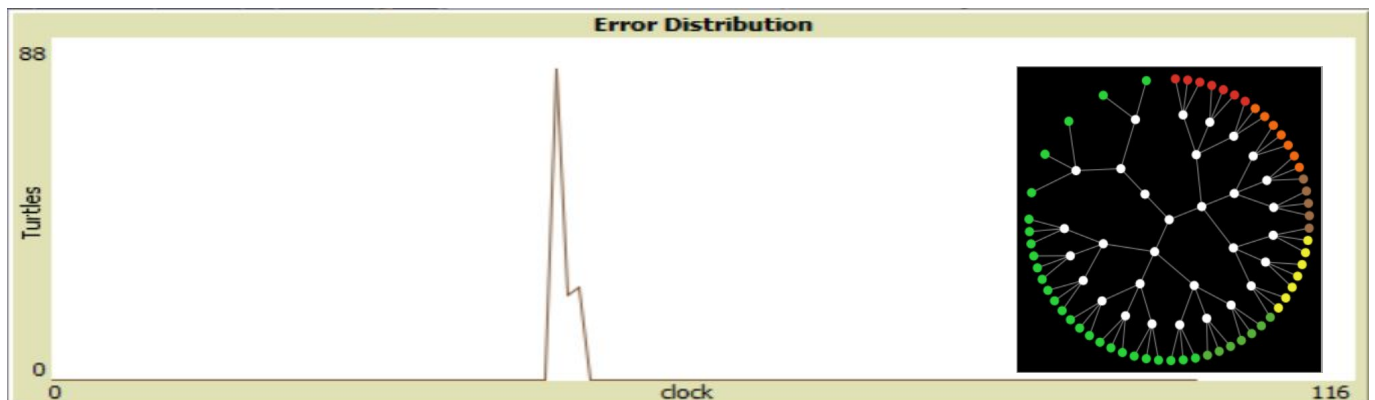
- Etude de l'erreur : (écart entre la liste de distribution des tâches et la liste de distribution parfaite)
 - ❑ Nombre de Vertex égale à 90
 - ❑ Nombre de task égale à 9
- ❖ Distribution à l'aide d'un système de communication programmé :



- ❖ Distribution aléatoire en fonction d'une liste des ratios :



- ❖ Distribution aléatoire Bernoulli :



→ Comparaison et Conclusion :

	Système de Communication	Première Distribution Aléatoire	Deuxième Distribution Aléatoire (Bernoulli)
Complexité	$O(t*c)$	$O(t*c)$	$O(t*t*c)$
Convergence	raisonnable	rapide	rapide pour une petite hiérarchie (100 Workers et pour un nombre de task fini)
Pic d'erreur	uniforme le long de la distribution	décroissant uniforme	un pic très grand au début mais une chute rapide (pour un petit système)
Communication	Hiérarchique (Système de communication sophistiqué)	Hiérarchique (Système de communication sophistiqué)	Semi-Hiérarchique

- Le premier Système de communication :

Une complexité de ressources raisonnable à l'égard du système hiérarchique sophistiqué, car le manager ne distribue la tâche qu'aux ces managers, qui eux même transmettent les listes aux autres, ainsi de suite, ça revient à une méthode classique "diviser pour régner" qui permet de réduire la condensation des informations dans un seul manager (souvent la racine), si on voit aussi, le graphe de distribution des tâches, on trouve que les tâches sont souvent regroupées chez un seul manager.

- La Première Distribution Aléatoire :

Il n'y a pas de grande différence avec la première distribution (Système de communication), sauf que la convergence est beaucoup plus rapide, puisque la distribution n'est pas contrôlée, on travaille juste sous la contrainte que la liste de distribution tend vers la liste de distribution parfaite. les tâches ne sont plus regroupées chez un seul manager ce qui n'est pas réaliste.

- **La deuxième Distribution aléatoire (Bernoulli) :**

En terme de rapidité, cette méthode est plus rapide (en terme de Convergence) dans le cas où le système est petite (Nombre de workers et la task), mais tant que le système évolue, la convergence augmente.

Ce système montre la faillite du système non-hiérarchique, même si on utilise des méthodes de distribution aléatoire plus rapide (Bernoulli), mais la condensation des ressources sur un seul manager (la racine), gâche tout.

Point d'amélioration :

Après les différentes simulations étudiés et les variants résultats trouvés, on constate que le modèle hiérarchique (Premier Modèle) est le plus robuste, avec un système de communication sophistiqué, comme point d'amélioration, on voulait coder les tâches en modes binaires, à l'aide d'un dictionnaire, en utilisant le codage de Huffman, qui permet d'assurer l'unicité des variables (tâches) et réduit également la complexité, donc au lieu de faire une liste, on pourrait faire juste un str de binaire: "1011101", et sera directement étudié par le dictionnaire, qui renvoie directement la liste des tâches chez le manager.

ex:

Soit un codage de Huffman d'ordre 2 :

tâche 0 : 00

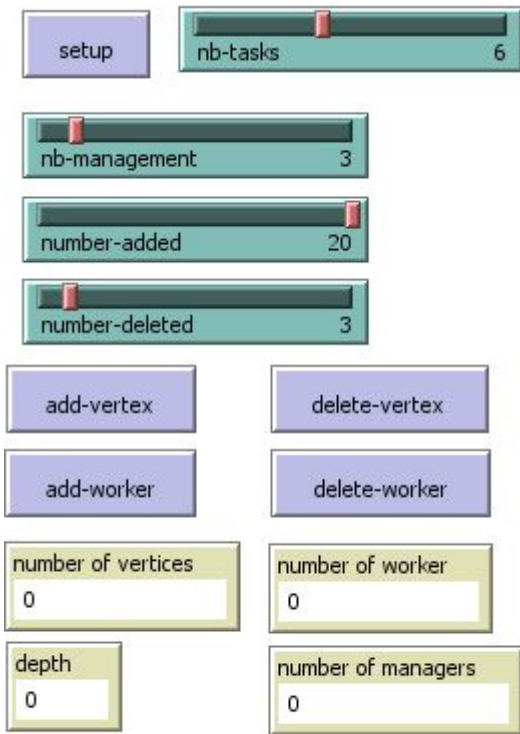
tâche 1 : 010

tâche 2 : 011

Le manager a un mot de code "01000": dû à l'unicité par le système de Huffman, on déduit la liste des tâches du manager : [1(00) 1(010) 0(011)] et cela va rendre le parcours linéaire des listes (notamment liste de liste) à un travail réduit sur des structures "String" ce qui réduit la complexité en $1/n$ avec n : le nombre de workers.

Note : Durant tout le programme, On a rajouté des sémaphores qui sont des outils qui permettent de synchroniser les méthodes implémentés, et pour éviter ainsi les interférences entre les différentes distributions.

Mode d'utilisation du premier Modèle (Système de communication Sophistiqué avec la première distribution aléatoire, ainsi que la seconde simulé dans l'erreur)



- Dans un premier temps, appuyer sur le bouton "Set Up".

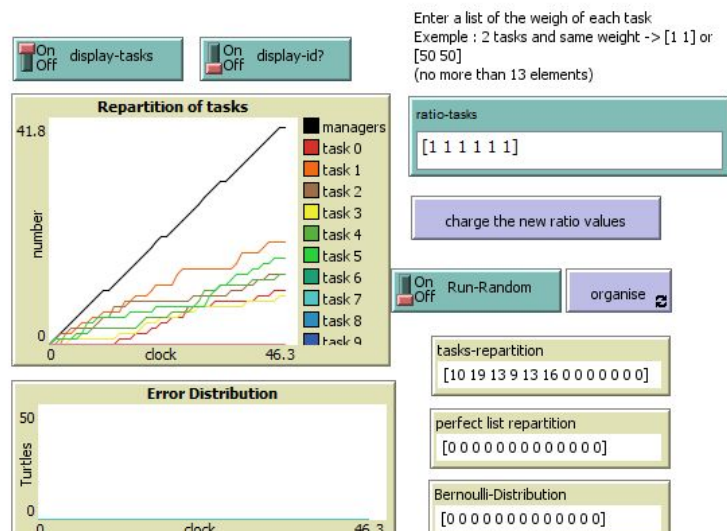
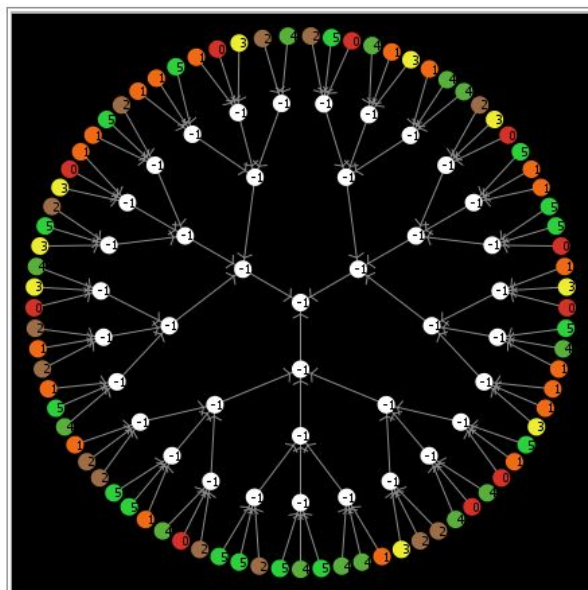
- Ensuite, l'utilisateur peut régler le nombre de travailleurs qu'il veut ajouter en un clic. Il doit aussi régler le paramètre C: le nombre de worker par manager afin d'établir une bonne distribution.

- En appuyant sur le bouton, "add-vertex" l'utilisateur ajoute un worker. Il peut choisir de le supprimer avec le bouton "delete-vertex".

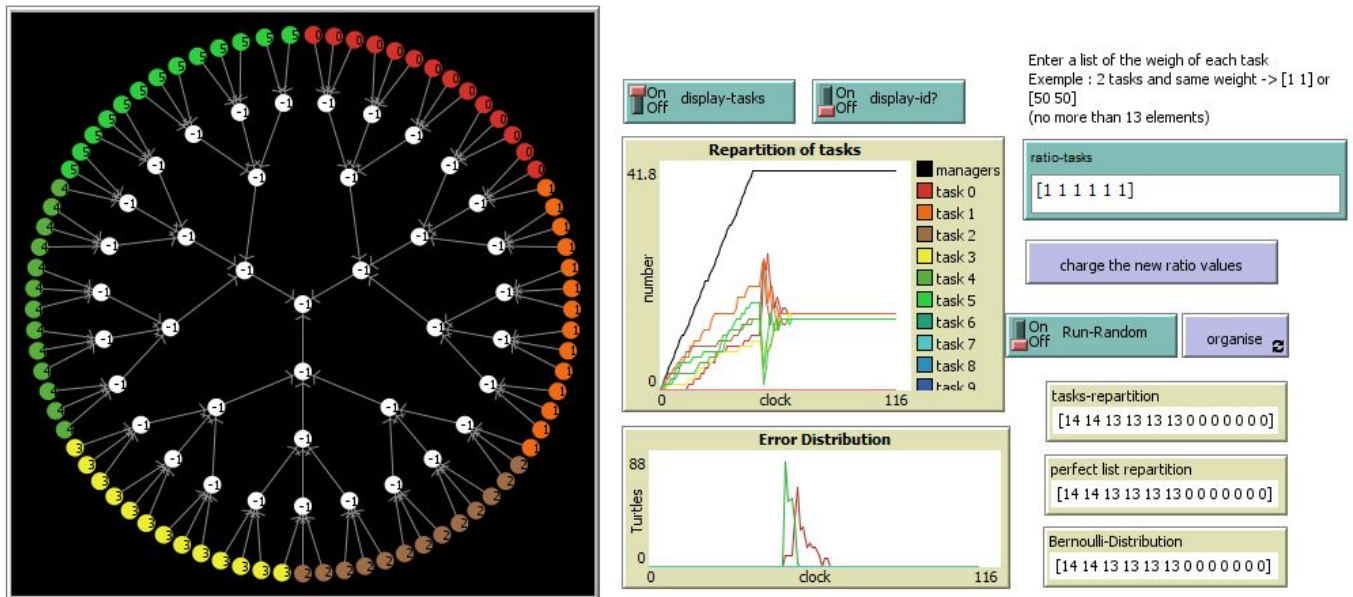
- En appuyant sur le bouton "add- worker", l'utilisateur peut implémenter un nombre de worker qu'il aura défini auparavant.

Ensuite, une fois ces paramètres définis, nous allons alors faire apparaître la hiérarchie et cliquant sur les boutons.

Au début, les tâches sont répartis de manière aléatoire, comme on peut le voir dans le graphique de la répartition des tâches.

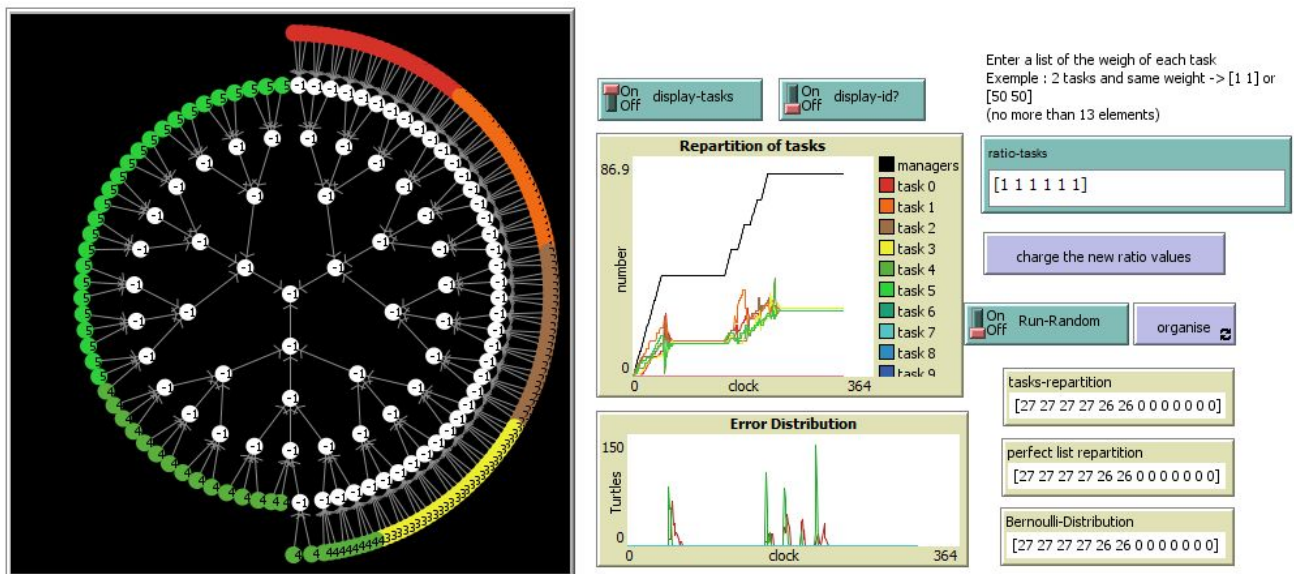


En appuyant sur le bouton “organise”, la simulation permet de répartir les tâches selon le matrice ratio des tâches définie par l'utilisateur au préalable, comme on peut le voir ci dessus.



L'utilisateur peut modifier les différents ratios. Pour cela, il doit modifier les différents coefficients et appuyer sur le bouton “change the new ratio values”. Appuyer sur le bouton “organise” pour faire apparaître la nouvelle simulation.

L'utilisateur peut également rajouter des worker en gardant la fonction “organise”.

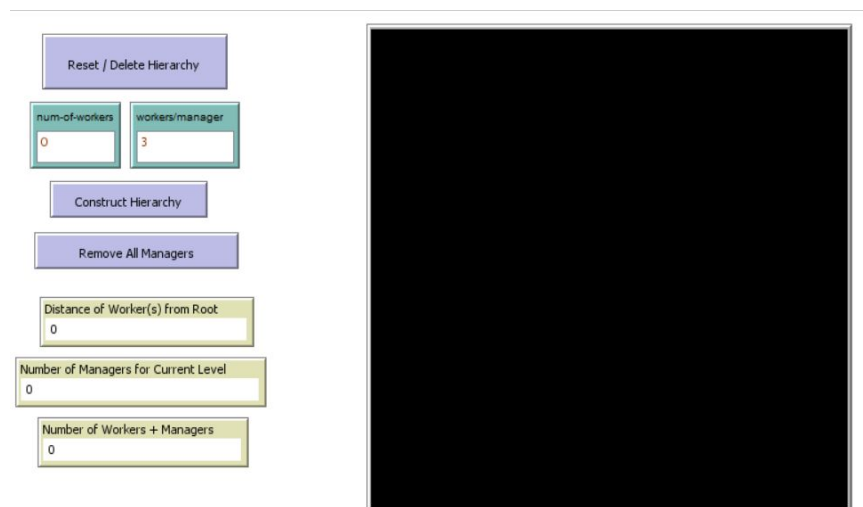


Le bouton “run-random” permet de choisir quelle méthode de distribution appliquée au système. (on : distribution aléatoire du premier modèle , Off distribution parfaite).

Enfin, plusieurs indicateurs sont présents. Un compteur de worker, de manager, le nombre de vecteurs, la profondeur, ainsi que la distribution des différentes tâches.

Deuxième Modèle Semi-Hiérarchique:

Voici l'interface graphique qui facilite la construction du graphe:



Dans l'interface graphique, il doit avoir 3 boutons : reset / Delete Hierarchy, Construct Hierarchy et Remove All Managers.

Le bouton reset permet de retirer toutes les personnes (workers et managers) de la hiérarchie d'un clic simple.

Pour construire le graphe, il faut d'abord remplir le nombre des workers dans l'input num-of-workers et la capacité des workers pour chaque manager dans workers/manager. Ensuite, il suffira de cliquer sur Construct Hierarchy, qui va construire la hiérarchie à partir des 2 inputs données. On peut aussi facilement trouver plus d'information sur notre arbre d'hiérarchie, comme ci-dessous:

- La distance entre la racine et chaque workers (Distance of Worker(s) from Root)
- Le nombre des managers à chaque itération pour un niveau, qui doit toujours être égal à 1 une fois le graphe de la hiérarchie est complété (Number of Managers for Current Level)
- Le nombre collectif des workers et des managers (Number of Workers + Managers)

Le bouton Remove All Managers va retirer tous les managers dans la hiérarchie. Donc, il ne sera utilisable qu'une fois la hiérarchie a été construit.

Rapport d'implication des membres:

Suite à l'assignation de ce projet, nous avons décidé de travailler chacun de son côté afin de se pencher sur le nouveau système NetLogo. Ensuite, nous avons proposé différentes solutions et méthodes afin de mener à bien la résolution de ce problème.

Nous avons, dans un premier temps, décidé de travailler chacun de son côté sur un algorithme. Ensuite, nous prenons les meilleurs idées et les algorithmes qui fonctionnent le mieux. Après contestation, nous avons gardé le modèle de Yassine et Loïc pour simulation optimale en terme de nombres des sommets, que Anas et Etienne travaillaient dessus. Yassine a également participé au développement de l'autre simulation des workers à la même profondeur avec Jonathan.

Ensuite, pour la distribution des tâches, Loïc a vite mis en place un modèle permettant une répartition parfaite à l'aide d'un système de communication programmé qui a été amélioré par la suite par Yassine. Yassine, Anas et Etienne quant à eux ont réussi à avoir un modèle quasi-similaire mais avec une grosse complexité. En s'y penchant plus, nous avons obtenu une autre méthode de distribution aléatoire en fonction d'une liste des ratios, que Yassine a intégré sur la simulation principale. Ensuite, il s'est penché sur la distribution aléatoire Bernoulli dont il a intégré dans les deux modèles faites avec Jonathan et Loïc, les Sémaphores ont été ajoutés à la fin pour synchroniser les méthodes.

Dépôt Git : <https://gitlab.telecom-paris.fr/PAF/1920/prj17-2.git>