M1 Informatique 2023/2024 - Algorithmes distribués
Devoir à la maison

L'algorithme YoYo pour l'élection d'un leader

L'algorithme YoYo pour l'élection d'un leader dans un graphe a été proposé par Nicola Santoro [1] On suppose que le graphe est non-vide, connexe, ne contient pas d'auto-boucles et que les nœuds du graphe sont numérotés par des entiers naturels uniques, appelés les identifiants des nœuds. Chaque nœud maintient un ensemble de voisins actifs, partitionnés en *voisins entrants* et *voisins sortants*. (Un nœud peut communiquer avec les deux types de voisins.) Au début, tous les nœuds sont actifs ; les voisins sortants (resp., entrants) du nœud avec identifiant i sont les voisins j de i tels que $i < j$ (resp., $j > i$). On appelle *sources* les nœuds du graphe qui n'ont pas de voisins entrants, *puits* les nœuds qui n'ont pas de voisins sortants et *nœuds internes* tous les autres nœuds.

Chaque nœud répète les deux phases suivantes jusqu'à ce qu'il devienne inactif.

Phase Yo-. Une source envoie son identifiant à tous ses voisins sortants. Les autres nœuds attendent de recevoir les valeurs de tous leurs voisins entrants, puis envoient la valeur minimale reçue à tous leurs voisins sortants. (Les puits n'envoient pas de message dans cette phase.)

Phase -Yo. Un puits envoie *yes* à ceux parmi ses voisins entrants qui ont envoyé la valeur minimale pendant la phase Yo-, et il envoie *no* aux autres voisins entrants.

Un nœud interne attend de recevoir des messages (*yes* ou *no*) de tous ses voisins sortants [2]. Si au moins un *no* a été reçu, il envoie *no* à tous ses voisins entrants. S'il n'a reçu que des messages *yes* alors il envoie *yes* à ceux parmi ses voisins entrants qui ont envoyé la valeur minimale pendant la phase Yo- et *no* aux autres voisins entrants.

Une source attend d'avoir reçu des messages (*yes* ou *no*) de tous ses voisins sortants mais n'envoie pas de message dans cette phase.

L'exécution de la phase -Yo modifie aussi la répartition des voisins : quand un nœud envoie *no* à un voisin entrant, il le transfère dans l'ensemble de ses voisins sortants. De même, un nœud qui reçoit un *no* de la part d'un de ses voisins sortants transfère ce nœud dans l'ensemble de ses voisins entrants.

On observe que la phase Yo- part des nœuds source et se propage aux nœuds puits alors que la phase -Yo part des puits et se propage aux sources. Cette structure de l'algorithme induit une synchronisation des nœuds : un nœud ne peut envoyer un message pour la phase Yo- qu'après avoir reçu les messages de tous ses voisins entrants pour cette phase, et inversement un nœud ne peut envoyer un message pour la phase -Yo qu'après avoir reçu les messages de tous ses voisins sortants pour cette phase. On peut donc considérer que le calcul s'effectue par *rondes* qui consistent chacune en deux phases Yo-Yo consécutives.

Question 1.

1. Argumentez que l'algorithme vérifie l'invariant suivant : au début de chaque ronde d'exécution, le nœud j est voisin sortant du nœud i si et seulement si j est voisin entrant de i.
2. On considère alors le graphe dirigé donné par les voisins entrants et sortants des nœuds. Argumentez qu'au début de chaque ronde d'exécution de l'algorithme, ce graphe ne contient pas de cycle.
3. Supposez que le graphe contient au moins deux sources au début d'une ronde. Argumentez qu'au moins une des sources deviendra un nœud interne ou un puits après l'exécution de la ronde et qu'aucun nœud interne ou puits ne peut devenir une source après l'exécution de la ronde. Le nombre de sources décroit donc au fur et à mesure de l'exécution de l'algorithme.

1. N. Santoro. Design and Analysis of Distributed Algorithms, chap. 3.8. Wiley (2007).
2. Ici et dans la suite de la description on fait référence aux voisins entrants et sortants au début de la phase.

Question 2. En partant de la version incomplète de l'algorithme que vous trouverez sur Arche et *après avoir lu les conseils plus loin*, implantez l'algorithme YoYo tel que décrit jusqu'ici en DistAlgo. Faites exécuter votre programme sur des graphes différents et assurez-vous (en insérant des instructions `output` appropriées) que la seule source qui survit au cours de l'exécution correspond au nœud du graphe dont l'identifiant est minimal.

Question 3. Vous aurez remarqué que la version de YoYo décrite jusqu'ici ne termine pas. Nous allons étendre l'algorithme par des mécanismes d'élagage qui garantissent que les nœuds deviennent inactifs au fur et à mesure de l'exécution. Concrètement, un paramètre booléen *couper* est ajouté aux messages envoyés lors de la phase -Yo de l'algorithme pour indiquer si le lien par lequel est envoyé le message doit être coupé, et cette phase est modifiée comme suit :

- Un puits avec un seul voisin entrant répond par un message où *couper* est vrai et devient inactif (c'est à dire, il termine son exécution). En effet, un tel nœud répond toujours par *yes* à son unique voisin entrant et ne contribue donc pas de manière utile au calcul du résultat.
- Lorsqu'un nœud a reçu des valeurs identiques de la part de $u > 1$ voisins entrants dans la phase Yo-, il met le paramètre *couper* à vrai dans sa réponse à $u - 1$ voisins et à faux dans sa réponse au dernier voisin ayant envoyé cette valeur.
- Dans tous les autres cas, le paramètre *couper* sera faux dans la réponse.
- Quand un nœud envoie un message où *couper* est vrai, il retire le destinataire de l'ensemble de ses voisins entrants. Quand un nœud reçoit un tel message, il retire l'émetteur de l'ensemble de ses voisins sortants.
- Si un nœud n'a plus de voisins entrants ou sortants, il se déclare gagnant et devient inactif.

Modifiez votre programme DistAlgo pour implémenter ce mécanisme d'élagage et assurez vous du bon fonctionnement de l'algorithme.

Consignes. Vous rendrez un fichier texte ou pdf en réponse à la première question, ainsi que les programmes DistAlgo correspondant aux questions 2 et 3. Le code devra être proprement commenté et contenir suffisamment de messages à imprimer sur la console pour pouvoir suivre l'exécution du programme. Si besoin, vous pouvez ajouter une page de description expliquant votre implémentation.

Vos réponses sont à rendre au plus tard le **23 février 2023** par email à `stephan.merz@loria.fr`. Je suis aussi à votre disposition pour répondre à vos questions concernant cet exercice.

Conseils.

1. DistAlgo requiert Python 3.7, il est donc nécessaire d'installer cette version sur votre machine. Pour utiliser la bonne version de python, vous pouvez adapter le script `dar` (et éventuellement `dac`) en remplaçant la première ligne par une ligne comme

   ```
   #!/usr/bin/env python3.7
   ```
2. Le squelette DistAlgo fourni sur Arche contient un générateur de graphes aléatoires qui prend deux paramètres optionnels pour indiquer le nombre de nœuds et le nombre d'arêtes du graphe. Ce générateur vous sera utile pour tester votre implémentation.
3. Observez que le statut d'un nœud (source, puits ou nœud interne) peut changer au cours de l'exécution de l'algorithme à cause de l'évolution de l'orientation des arêtes et de l'élagage du graphe. Il convient de recalculer ce statut au début (ou à la fin) de chaque ronde.
4. Vous allez forcément devoir utiliser des variables locales à chaque processus, par exemple pour stocker les valeurs reçues, et ces variables doivent être réinitialisées périodiquement. Il faut réfléchir au bon endroit pour le faire afin de ne pas perdre des messages reçus d'un processus en avance par rapport à ses voisins et induire un blocage. Éventuellement utiliser des variables temporaires dans le reste du code et/ou utiliser des points de collecte pour retarder la réception de certains messages.