Algorithme auto-stabilisant efficace en mémoire pour la construction d'un arbre couvrant de diamètre minimum

Lélia Blin, Fadwa Boubekeur, Swan Dubois

*Algorithme auto-stabilisant efficace en mémoire pour la construction d'un arbre couvrant de diamètre minimum**

Lélia Blin[1], Fadwa Boubekeur[2] et Swan Dubois[3]

[1]*Sorbonne Universités, UPMC Univ Paris 6, CNRS, Université d'Evry-Val-d'Essonne, LIP6 UMR 7606, 4 place Jussieu, 75005 Paris*
[2]*DAVID, UVSQ, Université Paris-Saclay, 45 Avenue des Etats-Unis, 78035 Versailles Cedex*
[3]*Sorbonne Universités, UPMC Univ Paris 6, CNRS, Inria, LIP6 UMR 7606, 4 place Jussieu, 75005 Paris*

Le diamètre est l'un des paramètres les plus importants dans les réseaux. Calculer le diamètre d'un réseau est un problème fondamental dans l'analyse des réseaux à large échelle. De plus, cette métrique est utilisée dans d'importants domaines d'application dans les réseaux réels. Par conséquent, il est naturel d'étudier ce problème dans un réseau distribué, et plus généralement dans un réseau distribué tolérant aux fautes transitoires. Plus précisément, nous nous sommes intéressés au problème de l'identification d'un centre d'un graphe. Une fois trouvé, nous construisons un arbre couvrant de diamètre minimum enraciné sur ce centre. Ainsi, le problème principal revient à calculer le centre d'un graphe. Dans cet article, nous présentons un algorithme auto-stabilisant uniforme pour le problème de construction d'un arbre couvrant de diamètre minimum dans le modèle à états. Notre algorithme possède plusieurs avantages qui le rendent adapté à des fins pratiques. C'est le premier algorithme traitant ce problème qui opère sous un démon non équitable (environnement asynchrone). En d'autres termes, aucune restriction n'est faite sur le comportement distribué du réseau. Par conséquent, c'est le démon le plus difficile avec lequel le réseau peut composer. De plus, notre algorithme utilise une mémoire de $O(\log n)$ bits par processus (ou n est le nombre de processus). Cet algorithme améliore les résultats précédents d'un facteur n. Ces améliorations ne sont pas atteintes au détriment du temps de convergence, qui reste polynomial en nombre de rondes.

Mots-clefs : Auto-stabilisation, Arbre couvrant, Centre, Diamètre

1 Introduction

Dans le contexte des réseaux, une tâche cruciale est de maintenir l'efficacité des communications. Une approche classique pour résoudre ce problème est la construction d'un arbre couvrant du réseau, offrant ainsi un unique chemin entre toutes paires de nœuds. Il existe différentes sortes d'arbres couvrants, selon le paramètre que l'on veut optimiser. Dans cet article, nous nous concentrons sur le problème de la construction d'un arbre couvrant de diamètre minimum (MDST). Le MDST est une approche naturelle si l'on veut optimiser le délai de communication entre toutes paires de nœuds dans un réseau, puisque la distance entre une paire de nœuds est limitée par le diamètre de l'arbre, qui est minimal dans le cas du MDST. Nous présentons un nouvel algorithme auto-stabilisant de calcul d'un MDST qui offre plusieurs avantages en comparaison aux travaux existants, premièrement notre algorithme peut s'exécuter dans un réseau totalement asynchrone (démon inéquitable), deuxièmement il améliore la mémoire utilisée par chaque nœud d'un facteur n (où n est le nombre de processus). Notez que ce gain en mémoire ne se fait pas au prix de la performance en temps.

État de l'art La construction d'arbre couvrant a été largement étudiée dans les réseaux distribués dans un contexte sans fautes ou en présence de fautes. Il existe une large littérature sur la construction auto-stabilisante sur les arbres couvrants sous contraintes tels que les arbre couvrants de parcours en largeur

*. Ce travail a fait l'objet d'une publication à IPDPS'15 [BBD15].

(BFS) [CRV11], les arbres couvrants de parcours en profondeur [HC93], les arbres couvrants de poids minimum [KKM11], les arbres couvrants de plus court chemin [Hua05], les arbres couvrants de degré minimum [BPBR11], les arbres de Steiner [BPBR09], etc.

Le problème du MDST est étroitement lié au calcul des centres du réseau [HT95]. En effet, un centre est un nœud qui minimise son excentricité, autrement dit sa distance maximale à tous les autres nœuds du réseau. Un MDST est donc un arbre couvrant en largeur enraciné sur un centre du réseau. Comme il existe de nombreuses solutions auto-stabilisantes pour calculer un arbre couvrant en largeur, nous nous concentrons dans ce qui suit sur la partie la plus difficile du problème du MDST : le problème de calcul d'un centre du réseau.

Une façon naturelle de calculer l'excentricité de l'ensemble des nœuds d'un réseau, pour pouvoir en identifier les centres, est de calculer le plus court chemin entre toutes paires de nœuds. Il est important de noter qu'une solution de calcul de centre utilisant le calcul du plus court chemin entre toutes paires de nœuds utilise au moins $O(n \log n)$ d'espace mémoire par nœud, elle est donc inapplicable si l'on souhaite faire des économies de mémoire. Dans le cadre de l'auto-stabilisation, peu de travaux sont consacrés au calcul des centres du réseau. La plupart des travaux [AS97, BGKP99, DL13] proposent des solutions qui calculent le ou les centres sur une topologie arborescente uniquement. Seul le travail de Butelle *et al.* [BLB95] propose le calcul des centres sur une tpologie quelconque. Le principal écueil de ce résultat réside dans sa complexité en espace qui est de $O(n \log n)$ bits par nœud, ce qui est équivalent à des solutions basées sur le calcul du plus court chemin entre toutes paires de nœuds.

Contributions Dans cet article, nous répondons positivement à la question suivante : est-il possible de calculer un centre d'un réseaux d'une manière auto-stabilisante en utilisant uniquement une mémoire de $O(\log n)$ bits par nœud. Pour cela, nous donnons un nouvel algorithme auto-stabilisant déterministe qui ne nécessite que $O(\log n)$ bits par nœud, ce qui améliore les résultats existants d'un facteur n. De plus, notre algorithme fonctionne dans tout environnement asynchrone puisque nous ne faisons aucune hypothèse sur l'adversaire (le démon). Notre algorithme dispose aussi d'un délai polynomail de convergence en $O(n^2)$ rondes (ce qui est comparable avec les solutions existantes [BLB95]).

2 Modèle

L'auto-stabilisation est la capacité d'un réseau à retrouver un comportement correct de lui-même à partir d'une configuration quelconque (résultant de fautes transitoires) et ceci en un temps fini. Le pire temps mis par le réseau pour atteindre un comportement correct à partir de n'importe quelle configuration initiale est appelé le temps de stabilisation (ou temps de convergence).

Le réseau est modélisé par un graphe non orienté connexe $G = (V, E)$ où V est l'ensemble des nœuds du réseau et E l'ensemble des liens de communications. Nous considérons des réseaux *identifié*, autrement dit, il existe un identifiant id_v unique pour chaque processus v pris dans l'ensemble $[0, n^c]$ pour une certaine constante c.

Nous considérons le *modèle à états* (voir [Dol00]). Chaque nœud a un ensemble de variables partagées. Un nœud v peut lire ses propres variables et celles de ses voisins mais ne peut écrire que sur ses propres variables. L'état d'un nœud est défini par la valeur courante de ses variables. Une*configuration* est le produit des états de tous les nœuds du réseau.

L'asynchronisme du réseau est modélisé par un *adversaire* (*démon*) qui choisit, à chaque étape, le sous-ensemble de nœuds qui sont autorisés à exécuter une de leurs règles pendant cette étape. La littérature propose un grand nombre de démons en fonction de leurs caractéristiques [DT11]. À chaque étape, un démon choisit parmi l'ensemble des nœuds activables (les nœuds peuvant executer un calcul) ceux qu'il active. Dans cet article, nous supposons un *démon distribué non equitable*. Ce démon est le plus général possible car il ne pose aucune restriction sur le sous-ensemble de nœuds choisis par le démon à chaque étape. Cela permet de modéliser toute exécution asynchrone. Pour calculer la complexité en temps, nous utilisons la définition de ronde [DIM97]. Cette définition englobe le temps d'exécution du processus le plus lent dans n'importe quelle exécution de l'algorithme.

3 Description de l'algorithme

Nous devons calculer l'excentricité de chacun des nœuds du réseau afin de déterminer le ou les centres du réseau. Le centre du réseau (s'il y en a plusieurs, c'est celui d'identifiant minimum qui sera sélectionné) deviendra la racine du MDST. Si chaque nœud calcule simultanément la distance qui le sépare de tout autre nœud, il pourra facilement en déduire son excentricité mais, dans ce cas, l'espace mémoire nécessaire en $O(n \log n)$ bits. L'idée principale de notre algorithme est basé sur la remarque suivante : le calcul de l'excentricité d'un nœud peut être fait par l'intermédiaire d'un BFS enraciné à ce nœud (l'excentricité du nœud étant alors la profondeur de ce BFS), calcul d'un BFS pouvant être effectué en utilisant $O(\log n)$ bits de mémoire. Il suffit donc de séquentialiser les calculs des excentricités afin de maintenir une occupation mémoire de $O(\log n)$ bits.

Notre algorithme utilise plusieurs couches, chacune d'elles exécutant une tâche spécifique. Avant de présenter plus en détail notre algorithme, en voici un rapide survol. La première couche est consacrée à la construction d'un *arbre couvrant enraciné*, appelé `Backbone`. La deuxième couche maintient *la circulation d'un jeton* sur le `Backbone`. La troisième couche est dédiée au *calcul des excentricités* : un nœud qui possède le jeton calcule son excentricité avant de passer le jeton à son voisin sur le `Backbone`, qui en fait de même et ainsi de suite. La dernière couche est consacrée au *calcul du centre* : le `Backbone` collecte les excentricités des feuilles vers la racine. La racine du `Backbone` calcule le centre du graphe, et diffuse l'identifiant de ce centre par l'intermédiaire du `Backbone`. Pour converger vers un MDST, nos couches doivent avoir différentes priorités. En effet, la construction du `Backbone` doit avoir la plus haute priorité, le reste de notre algorithme ne pouvant pas s'exécuter correctement si notre `Backbone` n'est pas correct. De même, pour la circulation de jeton, notre algorithme doit assurer l'unicité du jeton pour effectuer un calcul correct des excentricités. Enfin, le calcul des excentricités et le calcul du centre du graphe peuvent être fait de manière concurrente. Les principales difficultés rencontrées pour implémenter cette approche sont l'utilisation des mêmes variables pour le calcul de l'excentricité de nœuds différents et l'organisation des différentes couches afin qu'elles fonctionnent avec un démon totalement asynchrone. Nous allons maintenant détailler les différentes couches, pour la première et la deuxième couche nous utilisons des résultats existants, le cœur de notre travail est donc l'organisation entre ces couches et l'algorithme des deux dernières couches.

La première couche est consacrée à la construction d'un *arbre couvrant enraciné*. À notre connaissance, seul l'algorithme de construction d'arbre couvrant enraciné proposé par [DLV11] correspond à nos critères, à savoir, un démon distribué non équitable, $O(\log n)$ bits de mémoire par nœud et une convergence en $O(n)$ rondes. Cet algorithme construit un arbre BFS appelé `Backbone`enraciné au nœud d'identité minimum.

La deuxième couche est la *circulation de jeton* sur le `Backbone`. Nous avons adapté l'algorithme de Petit et Villain [PV99]. Le but de la circulation de jeton est de synchroniser le multiplexage temporel des variables de la troisième couche de notre algorithme qui calcule l'excentricité de chaque nœud. En effet, afin de réduire l'espace mémoire de notre algorithme à $O(\log n)$ bits par nœud, tous les nœuds calculent leurs excentricités en utilisant les mêmes variables, mais de façon séquentielle. Pour éviter les conflits, nous gérons l'accès en exclusion mutuelle à ces varaibles par la circulation de jeton.

La composition entre la couche de construction du `Backbone` et la couche de circulation du jeton de notre algorithme doit résister à la non équité du démon. En effet, nous devons nous assurer que le démon ne peut pas choisir exclusivement les nœuds qui souhaitent exécuter la circulation de jeton (rappelons que nous supposons que la construction du `Backbone` a la priorité sur la circulation de jeton) car cela pourrait empêcher le `Backbone` d'être construit. Pour faire face à cette question, nous avons choisi de bloquer la circulation du jeton au nœud v si v a détecté une incohérence dans le `Backbone` (cycle ou non connexité de l'arbre).

La troisième couche de notre algorithme est dédiée au *calcul des excentricités*. Nous différencions la circulation du jeton en descente et en montée. Quand un nœud v reçoit le jeton en descente, il commence une construction auto-stabilisante d'un arbre BFS enraciné sur lui-même. Quand la construction du BFS de v est accomplie, la profondeur maximale du BFS est calculée des feuilles vers v. Cette profondeur maximal donne l'excentricité de v. Une fois que le nœud v a recueilli son excentricité, il libère le jeton pour le nœud suivant dans le `Backbone`. L'algorithme de cette troisième couche est une contribution en soi

car le calcul de l'excentricité d'un nœud à l'aide d'un BFS (mémoire $O(\log n)$) pose plusieurs défis dus à l'asynchronisme. En effet, certaines branches du BFS peuvent commencer à calculer leur profondeur avec une mauvaise estimation de leurs distances à v. Pour résoudre ce problème, nous avons mis en place une priorité sur le calcul du BFS par rapport au calcul de la profondeur, tout changement de valeur de la distance dans le BFS entraînant une purge du calcul de la profondeur.

Enfin, la quatrième couche est dédiée au *calcul du centre*. L'excentricité de chaque nœud est collectée à chaque instant des feuilles vers la racine du `Backbone`. Ensuite, la racine propage cette excentricité minimale à tous les nœuds le long du `Backbone`. Le nœud avec l'excentricité minimale et l'identifiant minimum devient le centre du réseau. Ce nœud construit un BFS, c'est un arbre couvrant de diamètre minimum. Notre solution évite une configuration arbitraire de départ construisant plusieurs BFS (enracinés sur plusieurs nœuds se considérant de façon abusive comme centre) en dédiant uniquement trois variables à la construction du MDST (racine, parent, distance), évitant ainsi l'utilisation excessive de mémoire.

Discussion Une propriété désirable pour un algorithme auto-stabilisant est d'être silencieux, c'est-à-dire de garantir que l'état de chaque nœud reste identique dès qu'un état (global) légal a été atteint. En effet, une telle propriété garantit que l'auto-stabilisation ne surcharge pas le réseau avec un trafic important entre les nœuds lorsque ce réseau est dans un état légal. L'algorithme présenté ici n'est pas silencieux, car le jeton circule en permanence entraînant le recalcul des excentricités. Notez qu'une fois les excentricités calculées, leurs re-calculs donnent le même résultat, donc le centre et le MDST ne changent pas. La question naturelle qui se pose après le résultat de cet article est : existe t-il un algorithme auto-stabilisant silencieux pour la construction d'un MDST utilisant $O(\log n)$ bits de mémoire ?

Références

[AS97] G. Antonoiu and P. Srimani. A self-stabilizing distributed algorithm to find the center of a tree graph. *Parallel Algorithms and Applications*, 10(3-4) :237–248, 1997.

[BBD15] Lélia Blin, Fadwa Boubekeur, and Swan Dubois. A self-stabilizing memory efficient algorithm for the minimum diameter spanning tree under an omnipotent daemon. In *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS'15)*, pages 1065–1074, 2015.

[BGKP99] S. Bruell, S. Ghosh, M. Karaata, and S. Pemmaraju. Self-stabilizing algorithms for finding centers and medians of trees. *SIAM Journal of Computing*, 29(2) :600–614, 1999.

[BLB95] F. Butelle, C. Lavault, and M. Bui. A uniform self-stabilizing minimum diameter tree algorithm (extended abstract). In *WDAG'95*, pages 257–272, 1995.

[BPBR09] L. Blin, M. Potop-Butucaru, and S. Rovedakis. A superstabilizing log (n)-approximation algorithm for dynamic steiner trees. In *SSS'09*, pages 133–148, 2009.

[BPBR11] L. Blin, M. Potop-Butucaru, and S. Rovedakis. Self-stabilizing minimum degree spanning tree within one from the optimal degree. *J. of Parallel and Distributed Computing*, 71(3) :438–449, 2011.

[CRV11] A. Cournier, S. Rovedakis, and V. Villain. The first fully polynomial stabilizing algorithm for BFS tree construction. In *OPODIS'11*, pages 159–174, 2011.

[DIM97] S. Dolev, A. Israeli, and S. Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM J. Comput.*, 26(1) :273–290, 1997.

[DL13] A. Datta and L. Larmore. Leader election and centers and medians in tree networks. In *SSS'13*, pages 113–132, 2013.

[DLV11] A. Datta, L. Larmore, and P. Vemula. An o(n)-time self-stabilizing leader election algorithm. *J. Parallel Distrib. Comput.*, 71(11) :1532–1544, 2011.

[Dol00] S. Dolev. *Self-stabilization*. MIT Press, March 2000.

[DT11] S. Dubois and S. Tixeuil. A taxonomy of daemons in self-stabilization. Technical Report 1110.0334, ArXiv eprint, October 2011.

[HC93] S.-T. Huang and N.-S. Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7(1) :61–66, 1993.

[HT95] R. Hassin and A. Tamir. On the minimum diameter spanning tree problem. *Information Processing Letters*, 53(2) :109–111, 1995.

[Hua05] T. Huang. A self-stabilizing algorithm for the shortest path problem assuming read/write atomicity. *J. Comput. Syst. Sci.*, 71(1) :70–85, 2005.

[KKM11] A. Korman, S. Kutten, and T. Masuzawa. Fast and compact self stabilizing verification, computation, and fault detection of an MST. In *PODC'11*, pages 311–320, 2011.

[PV99] F. Petit and V. Villain. Time and space optimality of distributed depth-first token circulation algorithms. In *WDAS'99*, pages 91–106, 1999.