Algorithmes Distribués

Master 1 Informatique – année 2005/2006 – Université de Bordeaux 1

Objectifs

Ce cours d'algorithmes distribués consiste à :

- Introduire le calcul distribué
- concevoir et analyser des algorithmes distribués
- programmer des algorithmes (c.f. TP) en utilisant un simulateur (VISIDIA)

Introduction

Qu'est ce que le calcul distribué?

Une machine ou architecture séquentielle permet le traitement d'une seule opération à la fois. L'extrait de programme ci-dessous est donc une suite d'instruction qui vont s'effectuer de manière séquentielle :

```
10      x := x + 1
20      y := 2*x – 4
30      Print y
```

Les machines parallèles permettent le traitement de plusieurs opérations en parallèle. Par exemple, les architectures 32 bits sont déjà des architectures parallèles dans la mesure ou elles traitent 32 bits simultanément. Lorsque le nombre de bits devient très important, on parle aussi de machines vectorielles.

Un système distribué est une machine parallèle à la différence qu'il y a plusieurs entités de calculs autonomes distantes. La différence principale avec la programmation d'architectures parallèles repose donc sur le fait que la distance entre les entités de calcul est plus importantes (échelle planétaire) et nécessite donc forcément l'utilisation d'un réseau de communication (type Internet).

Pourquoi utiliser des systèmes distribués?

Les objectifs rejoignent principalement ceux étudiés en programmation des architectures parallèles :

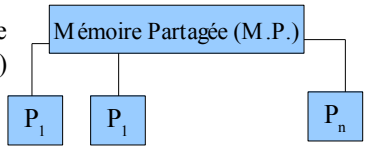
Augmenter la capacité de calcul (à moindre coup) en ajoutant graduellement des entités ou regrouper des ensembles d'entités déjà existant. Par exemple GRID5000 en France est un regroupement de 5000 calculateurs répartis sur tout le territoire.

Communiquer entre entités distantes (Telecommunication, Réseau LAN, Satellite, Wifi, ...) et mettre en rapport des banques de données réparties (Internet, Peer – To – Peer).

Un cadre de base pour les systèmes distribués

Les entités seront les processeurs, des ordinateurs, des PDA, des satellites. Il existe principalement deux modèles d'archirectures distribuées : le « passage de messages » et la « mémoire distribuée » :

- Passage de messages : ce modèle traite explicitement des communications. Lorsqu'un processeur veut communiquer avec un autre, il doit lui envoyer un message via le medium.
- Mémoire partagée : Au contraire, les processeurs ne communique pas entre eux directement. Ils ont accès à une zone mémoire (ou des variables) commune en lecture et/ou écriture.



En particulier, une mémoire partagée peut servir à communiquer.

Le modèle PRAM (Parallel Random Acces Machine) dont un schéma simplifié de fonctionnement est proposé ci-

contre, est une généralisation du modèle RAM classique (ou le processeur communique avec la mémoire vive). Le modèle PRAM ou mémoire partagée est utile pour l'étude du degré de parallélisme d'un problème.

Le modèle PRAM, et d'autres modèles dérivés tels que SIMD (Single Instruction Multiple Data) ou MIMD (Multiple Instruction Multiple Data) est un bon système. Il est cependant assez éloigné des systèmes distribués réels pour plusieurs raisons :

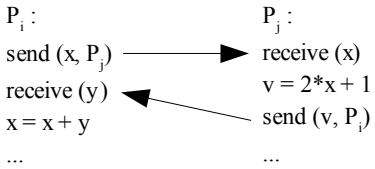
- Si n (nombre de processeurs) est grand, alors il n'est plus physiquement possible que tout le monde accède en même temps (unitaire) à une même variable.
- Si les distances entre les entités sont grandes alors l'accès à un lien commun (M.P.) avec des temps similaires n'est plus possible.

Dans le cadre de ce cours, nous allons choisir le mode point à point, qui se modélise par un graphe G = (V, E). Les sommets du graphe représentent les entités (processeurs) et les arêtes les liens de communications entre les entités (à priori, il s'agira toujours d'un graphe non orienté : les liens de communication sont bidirectionnels). On supposera de plus que G est toujours connexe (pas de sommets – entités isolés ou sous graphes isolés).

Particularité du calcul distribué

Il y a des communications et elles ne sont pas gratuites

Soient P_i et P_j deux programmes distants s'échangeant des données à une fréquence de 1 Ghz.



Sachant que 1 Ghz = 10^{-9} s et la vitesse de la lumière c = 3 . 10^8 m/s, la distance entre P_i et P_j doit donc être (pour ne pas interférer sur la vitesse de calcul) d'au plus : dist(P_i, P_j) < 10^{-9} 3.10^8 = 0,3 m.

Cette distance n'est évidemment pas respectable dans un vrai système distribué, ce d'autant plus qu'il s'agit ici de suppositions optimales (dans les composants électroniques, la vitesse des données n'atteind pas la vitesse de la lumière).

La plupart du temps, on considèrera donc que le temps de calcul local est négligeable devant les temps de communications.

Connaissance partielle

Un processeur doit faire son calcul avec une connaissance, à priori limitée, du système. Dans un système centralisé (ou séquentiel), l'état d'une partie de la mémoire est déterminé par le calcul du processeur unique. En particulier, une partie de la mémoire ne change pas si le processeur ne le modifie pas.

En distribué, P_i ne contrôle que son état local. Les données réparties sur d'autres processeurs changent sans aucun contrôle de P_i (une sorte d'adversaire qui modifie les données).

Parfois P_i connait la topologie du graphe en entier, parfois seulement ses voisins ou même, ne connait que lui-même (réseau anonyme : les processeurs ont tous le même programme).

On suppose généralement que les processeurs ont des identités uniques et connaissent leurs voisins, éventuellement le nombre de sommets du graphe.

Exemple : Diffusion d'un message M de x vers y

Il faut pour cela distinguer trois types de programmes : celui qui s'exécute sur x (pour envoyer le message), celui qui s'exécute sur y (pour recevoir le message) et celui qui s'exécute sur une machine qui n'est pas y (pour diffuser – transférer le message reçu aux voisins).

```
X :                                 Y :                 Z :
Pour tous les voisins v de x        Receive (M)         Receive (M)
        send (M, v)                                     Pour tous les voisins w de Z
                                                                send (M, w)
```

« Y » reçoit le message en un temps fini. Cependant, des messages peuvent boucler à l'infini (à cause de z qui rediffuse

sans distinction un message reçu à tous ces voisins). On peut corriger ce problème si l'on suppose que les processeurs connaissent n (le nombre de processeurs/sommets) : on met dans le message M un compteur C initialisé à n. On décrémente ce compteur à chaque passage sur un sommet z et on ne retransmet M que si C > 0.

Erreurs

Le cas des erreurs est relativement simple en séquentiel. Si un élément est défectueux (mémoire, périphérique, processeur, disque), on le change et on reprend/continue/recommence le programme. En distribué, on peut de plus avoir un lien de communication en panne. Cela peut donc causer de nombreux problèmes tels que la corruption des messages ou avoir des adversaires qui changent les messages.

Redémarrer tout le système peut parfois être évité en rendant l'algorithme robuste (tolérance aux pannes). Bien évidemment, cette tolérance dépend ausi du réseau (graphe : y a-t-il plusieurs chemins possibles d'un sommet à un autre?).

Synchronisme

On considère deux modèles extrèmes :

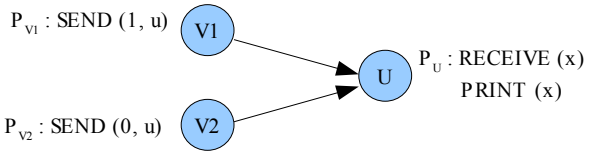
- Système synchrone : on suppose que le délai de transmission des messages est borné. Plus précisemment, chaque processeur a une horloge locale dont les tops d'horloges ont la propriété suivante : "un message envoyé de v vers un voisin u au top t (de v) doit arriver avant le top t + 1 de u". Donc, cela a le même effet qu'une horloge globale. Le cycle de calcul de chaque processeur ressemble donc à :
  - 1 : Envoyer des messages à 1 ou plusieurs voisins
  - 2 : Recevoir des messages de 1 ou plusieurs voisins
  - 3 : Effectuer le calcul local. Le calcul local est supposé prendre un temps négligeable devant l'étape 2. Donc le cycle de calcul passe son temps à attendre la réception de messages (les réceptions sont bloquantes).
- Système asynchrone : au contraire, l'algorithme est ici dirigé par des évènements. Les processeurs ne pouvant avoir accès à une horloge globale pour prendre leur décision. Les messages envoyés de v à u arrivent en un temps fini (si pas de faute) mais imprévisible. Par exemple, on ne peut pas déduire l'ordre d'arrivée des messages venant des voisins u_i de v d'après leur ordre d'émission. Généralement, un algorithme en asynchrone ressemble à :
  - Si évènement E1, faire X1 (avec E1 = arrivée d'un message d'un certain type : START, STOP, ...)
  - Si évènement E2, faire X2
  - ...

Il existe, dans la pratique, des systèmes intermédiaires, en connaissant une borne inférieure et supérieure de transmission des messages.

Asynchronisme et non déterminisme

En séquentiel, le non déterminisme est lié à l'usage de bits aléatoire (fonction RANDOM). Sinon, avec les mêmes entrées, on a les mêmes sorties (puisque, même exécution).

En distribué, on n'a pas besoin de fonction RANDOM pour avoir du non déterminisme. On peut exécuter deux fois le même algorithme et sur les mêmes entrées et obtenir deux résultats distincts. L'exécution d'un algorithme asynchrone est donc appelé scénario. Voici un exemple simple :



Ici, on ne sait pas si u recoit 1 ou 0 (en asynchrone). Le résultat du "print x" est donc potentiellement différent à chaque exécution.

Modèles du calcul distribué

On suppose un mode de communication point à point modélisé par un réseau (graphe).

Mesure de complexité

En séquentiel, il y a deux mesures de complexités : temps et espace. La chose est plus subtile en distribué.

Le temps

Définition en synchrone : La complexité en temps de l'algorithme P sur le graphe G, noté TEMPS (P, G) est le nombre de tops d'horloges (rondes) générés par P sur G dans le pire des cas (sur toutes les entrées légales possibles stockées sur les processeurs de G) entre le début du premier processeur qui démarre et la fin du dernier processeur qui s'arrête.

Définition en asynchrone : La complexité en temps de l'algorithme P sur G, noté TEMPS(P, G) est le nombre d'unité de temps du départ à la fin dans le pire des cas (sur toutes les entrées légales et tous les scénarii), chaque message prenant un temps unitaire.

Espace

C'est le nombre total de bits utilisés par P sur G dans le pire des cas (pour toutes les entrées légales possibles et pour tous les scénarii si l'on est en asynchrone).

Complexité en nombre de messages

Un message sur un arête coûte "1". On compte le nombre de messages échangés sur G par P (noté MESSAGE(P, G)) durant l'exécution de P sur G dans le pire des cas.

Quelques exemples

On considère quatre algorithmes P_A, P_B, P_C, et P_D qui s'exécutent sur K_N (graphe complet à n sommets) :

- P_A : envoie un message M de 0 à 1
- P_B : envoie un message différent M_i en parallèle depuis i vers i + 1(mod n), et ce, pour tout i
- P_C : envoie un message M depuis 0 vers 1, puis de 1 vers 2, ..., puis itérativement le long d'une chaine i => i+1
- P_D : envoie depuis 0 un mesage M_i vers le sommet i.

Si l'on regarde la complexité en temps, P_A, P_B et P_D sont de complexité 1, et P_C de complexité n – 1. En revanche, si l'on regarde la complexité en nombre de messages, P_A a une complexité de 1 et P_B, P_C et P_D une complexité n – 1.

Complexité globale et existancielle

Les notions de complexités O, Ω, Θ et o ne seront pas réexpliquées ici (voir cours "Algorithmes Avancés" et "Algorithmes des graphes" Licence 3). Nous supposons acquises ses notations pour la suite. Voici quelques définitions spécifiques à la complexité en distribué :

- Borne supérieure : "l'algorithme P a, par exemple, une complexité O(n.logn) pour les graphes planaires" <=> pour tout graphe planaire, Complexité(P, G) ≤ c.n.logn.
- Borne inférieure : "l'algorithme P a, par exemple, une complexité Ω(n) pour les graphes planaires" <=> il existe c tel que il existe un graphe G_0 (et tel qu'il existe un scénario – en asynchrone) tel que Complexité(P, G_0) ≥ c.n. La borne inférieure est donc une borne existancielle.
- Borne inférieure globale : "l'algorithme P a une complexité Ω(n) pour tout graphe" <=> il existe c (et un scénario dans le cas asynchrone) tel que pour tout graphe G, Complexité(P, G) ≥ c.n.

Modèles représentatifs

Avant d'expliquer les différents modèles représentatifs, il est important de noter ici quelques hypothèses qui seront utilisées par la suite. On suppose :

- qu'il n'y a aucune faute (pas de liens en panne, pas de changement de topologie du graphe, ...)
- qu'il y a une identité unique : les processeurs ont une identité unique (entier sur O(logn) bits)
- que les calculs locaux sont gratuits

Il existe alors 3 modèles représentatifs :

- Modèle LOCAL : on étudie la nature locale d'une exécution sur le TEMPS requis pour un problème. On ne se pose pas la question de l'asynchronisme (un modèle LOCAL est forcément synchrone) et de la congestion (plusieurs messages sur le même lien en même temps). Dans ce modèle, on suppose de plus que la taille des

messages est illimitée, que la communication est synchrone et que tout le monde se réveille en même temps (même top d'horloge).

- Modèle CONGEST : on étudie l'effet du volume des communications. 1 message = 1 message de base (O(logn) bits. Par exemple, on peut considérer la tache : "dans un arbre, un sommet interne v veut communiquer l'identité de ses fils à son père." Les d fils envoient leur identité, ce qui nécessite d messages et v envoie ensuite 1 à 1 les d identités à son père. On obtient un total de 2d messages envoyés sur G. Dans le modèle CONGEST, on peut être amené à distinguer le cas synchrone et le cas asynchrone.
- Modèle ASYNC : on étudie l'effet de l'asynchronisme (Le modèle ASYNC peut donc être vue comme l'inverse du modèle LOCAL).

Diffusion et concentration

La diffusion

La diffusion consiste à envoyer, depuis une source, le même message à tous les autres sommets. Soit G un graphe avec une source r_0 et n sommets, alors tout algorithme B de diffusion sur G vérifie :

- MESSAGE (B, G) >= n – 1
- TEMPS (B, G) >= ecc (r_0) (en synchrone ou asynchrone), et ou ecc est la plus grande distance à partir de r_0.

On notera que, pour tout sommet v, ecc(v) = Max (dist(v, x)), pour tout x. En notant D le diamètre d'un graphe, on a alors $\forall v, \frac{D}{2} \leq ecc(v) \leq D$.

Arbre de diffusion

Une stratégie courante, pour réaliser une diffusion est d'utliser un arbre : on ne diffuse que le long des arêtes de l'arbre (arbre de diffusion). Ici, on va supposer que l'on connait un arbre de diffusion T (à savoir : chaque sommet connait son père et ses fils). L'algorithme de diffusion est alors le suivant :

Algorithme Tcast(r_0, T) :

- Si u != r_0, Receive (M, Pere(u))
- Pour tout fils f de u, send (M, f)

Dans ce cas, si T est un arbre couvrant de G enraciné en r_0 :

- MESSAGE (Tcast(r_0, T), G) = n – 1
- TEMPS (Tcast(r_0, T), G) = h (hauteur de l'arbre) – que ce soit en synchrone ou en asynchrone.

On appelle arbre BFS (Breath First Search) un arbre de plus court chemin, c'est à dire un arbre T couvrant de G enraciné en v tel que $dist_T(u, r) = dist_G(u, r)$ pour tout u de V(G).

Si T est un arbre de plus court chemins enraciné en r_0, alors Tcast(r_0, T) est un algorithme de diffusion optimal en nombre de messages et en temps.

Innondation

On utilise cette technique en l'absence de structure précalculée comme un arbre de diffusion. L'algorithme d'innondation se résume ainci :

Algorithme FLOOD (source r_0) :

- Si on est r_0 : SEND (M, v) pour tout voisin v de r_0
- Pour tout sommet différent de r_0, à la réception du premier message sur l'arête e :
  - on stocke le message M
  - On diffuse le message M sur toutes les autres arêtes incidentes à u
- Pour tout autre message contenant M, ne rien faire

Cet algorithme vérifie les propriétés suivantes :

- MESSAGE (FLOOD, G) = $\Theta(|E|)$
- TEMPS (FLOOD, G) = $\Theta(ecc(r_0)) = \Theta(diam(G))$

Sur une arête, il circule au moins un message, et au plus 2. Après t unités de temps, M atteind tous les sommets à distances inférieure à t de r_0. Donc, TEMPS (FLOOD, G) =< ecc(r_0) et pour tout algorithme de diffusion B, TEMPS(B, G) >= ecc(r_0).

Le problème de la diffusion à partir d'une source r_0 est donc équivalent (en temps et nombre de messages) au problème de la construction d'un arbre couvrant enraciné en r_0.

Si on sait construire un arbre T, alors on le construit et on applique Tcast pour diffuser. Si on connait un algorithme de diffusion B depuis r_0, alors il suffit de définir le père de u comme le voisin v d'ou u a reçu M en premier.

Il y a une grande différence entre le mode synchrone et asynchrone quant à la structure de l'arbre T_{FLOOD} construit à partir de l'agorithme FLOOD :

- En synchrone, T_{FLOOD} est un BFS (plus court chemin) enraciné en r_0, donc avec une hauteur égale à ecc(r_0).
- En asynchrone, T_{FLOOD} peut avoir une hauteur bien supérieure à ecc(r_0). En réalité, cette hauteur peut atteindre n – 1.

Concentration

Le problème avec les algorithmes de diffusion vus précédemment, est qu'un sommet ne sait pas lorsque la tache globale est terminée. C'est le problème de la terminaison (ou détection locale de la terminaison).

La solution est de collecter l'information de terminaison des feuilles à la racine dans l'arbre de diffusion : il s'agit donc d'une opération de concentration. Puis, r_0 rediffuse un message de FIN à tous.

Dès que u reçoit le message M (de la diffusion), il fait :

- Si c'est une feuille de la diffusion, il repond par un message ACK à son père
- Sinon, il attend de recevoir tous les messages ACK de ses fils, puis l'envoie à son père.

Un tel algorithme assure les propriétés suivantes (en synchrone) :

- MESSAGE (CONCENTRE, T) = n – 1
- TEMPS (CONCENTRE, T) <= hauteur (T)

En asynchrone, le temps de diffusion et le temps de concentration peuvent être très différent (par exemple sur un graphe complet avec un arbre couvrant de hauteur n – 1).

Arbres couvrants

Arbre BFS

Un arbre BFS est un arbre T couvrant les sommets d'un graphe G de racine r_0 tel que $dist_T(x, r_0) = dist_G(x, r_0)$.

On a vu au chapitre précédent l'algorithme FLOOD permettant de construire un BFS dans le cas synchrone. Nous rappelons ici que MESSAGE (FLOOD, G) = $\Theta(|E|)$ et TEMPS (FLOOD, G) = $\Theta(ecc(r_0)) = \Theta(diam(G))$. Ces bornes sont les meilleures possibles si les sommets n'ont aucune connaissance sur le graphe G.

Nous allons admettre les propositions suivantes :

- Pour tout G et pour tout algorithme distribué A qui calcule un BFS sur G (sans connaissance sur G) alors MESSAGE (A, G) = $\Omega(|E|)$
- Pour tout G et pour tout algorithme distribué A qui calcule un BFS sur G alors TEMPS (A, G) = $\Omega(D)$.

Par conséquent, en synchrone, la meilleure solution pour construire un BFS est l'algorithme FLOOD. En asynchrone, FLOOD ne construit pas nécessairement un BFS sur G. Dans la suite, on va se concentrer sur le modèle asynchrone CONGEST.

Arbre BFS : algorithme distribué de Dijkstra

L'idée de cet algorithme est de construire un BFS en travaillant par niveau. A chaque étape, on essaye d'ajouter les sommets adjacents au dernier niveau qui ne sont pas déjà dans l'arbre. Dans l'implémentation distribuée, la racine r_0 va "synchroniser" la construction de chaque nouveau niveau. Au départ de chaque phase, r_0 génère un message "pulse" dans l'arbre déjà construit. On suppose qu'après p phases, il y a un BFS T_p enraciné en r_0 qui couvre tous les sommets à distance inférieure ou égale à p de r_0.

En particulier, on suppose que chaque sommet de T_P connait son père et ses fils (de T_P). Ainsi, la phase p + 1 de l'algorithme DistDijkstra :

- r_0 génère un message "pulse" qu'il diffuse dans T_P
- A la réception d'un message "pulse", chaque feuille de T_p envoie un message d'exploration "layer" à tous ses voisins différent du père,
- Soit w un sommet qui reçoit pourla première fois un message "layer" de v, alors père(w) = v et w retourne à v un message lui indiquant qu'il a été choisi comme père. Tous les autres sommets essayant d'envoyer par la suite un message à w recevront un message "ACK" classique spécifiant que le sommet w est déjà dans l'arbre.
- Les feuilles w de T_p collectent les messages ("ACK" + "Père" des messages "layer" et choisissent comme fils ceux qui ont choisit v comme père).
- Lorsque v (feuille de T_P) a reçu un "ACK" de tout ses voisins, alors on retourne un "ACK3 à son père et les sommets internes de T_p font de même (concentration vers r_0)
- Lorsque r_0 a reçu tous les "ACK" de ses fils, on démarre la phase suivante.

On distingue cependant ici un problème majeur : bien que l'algorithme construise effectivement un BFS, l'algorithme lui même ne se termine pas.

Une solution consiste à ce que les feuilles comptent le nombre de nouveaux fils. S'il y en a au moins 1, on envoie 1 au père, et 0 sinon. Les sommets internes font une concentration vers r_0 en réalisant un OR. La racine continue tant que le résultat du OR de ses fils est différent de 0. On peut détecter la terminaison en chaque sommet en diffusant depuis r_0 un message FIN.

Après la phase p, l'arbre T_{p+1} construit à la phase p + 1 est bien un arbre BFS couvrant les sommets à distance inférieure ou égale à p + 1 de r_0.

La complexité en temps à la phase p est TEMPS (phase p) <= p + 2 + p.

On a donc $TEMPS(DistDijkstra) \leq \sum_{p=0}^{D} TEMPS(phase) = O(D^2)$.

La complexité en nombre de messages à la phase p est MESSAGE (phase) = O($|E(T_p)| + |E_p| + |E_{p,\,p+1}|$) avec :

- $|E(T_p)|$ représente le nombre de message "pulse" + messages de concentration (inférieur à n)
- $|E_p|$: messages d'exploration entre sommets de niveau p dans T
- $|E_{p,\,p+1}|$: messages d'exploration entre sommets de niveau p et p + 1 dans T

On obtient alors une complexité totale :

$$MESSAGE(DistDijkstra) \leq \sum_{p=0}^{D} MESSAGE(phase) = O(n.D + |E(G)|) \quad .$$

Pour résumer, on pourra donc dire en conclusion que l'algorithme de Dijkstra construit bien un arbre BFS de G enraciné en r_0 et ayant les propriétés suivantes :

- TEMPS (DistDijkstra) = $O(D^2)$
- MESSAGE(DistDijkstra) = O(nD + |E(G)|)

Arbre BFS : Algorithme de Bellman-Ford

Cette variante est basée sur une version optimiste de FLOOD. Chaque sommet v gère une variable L(v) qui est son niveau dans l'arbre construit par FLOOD. Cette variable est corrigée au fur et à mesure de l'exécution de FLOOD de façon à maintenir la longueur de la plus courte route de v à r_0.

Algorithme BF :

- $L(r_0) = 0$ et $L(v) = \infty$ pour tout v différent de r_0
- r_0 envoie un message Layer(0) à tous ses voisins

- A la réception d'un message Layer(d) d'un voisin w reçu en v. Si d + 1 < L(v) alors père(v) = w et L(v) = d + 1 et on envoie Layer(d + 1) à tous les voisins sauf le nouveau père w.

L'algorithme de Bellman-Ford vérifie :

$\forall$ d ≥ 1, après d unités de temps, tout sommet v à distance d de r_0 a déjà reçu un message Layer(d – 1) d'un de ses voisins. Donc L(v) = d et père(v) = w tel que L(w) = d – 1. Par conséquent TEMPS (DistBF, G) = O(D).

La première valeur bornée affectée à L(v) est au plus n – 1 (la longueur de plus long chemin possible). Donc L(v) va être modifiée auu plus n – 2 fois (L(v) ne peut que décroitre). A chaque modification de L(v), v envoie deg(v) – 1 messages. Donc au total v va envoyé au maximum n.deg(v).

On a donc $MESSAGE(DistBF, G) \leq \sum_v n.deg(v) = O(n|E(G)|)$

Par conséquent, dans le modèle asynchrone CONGEST, l'algorithme distribué de Bellman-Ford construit un BFS enraciné en r_0 avec comme complexité :

- TEMPS (DistBF) = O(D)
- MESSAGE (DistBF) = O(n.|E|)

En résumé :

	Messages	Temps
Borne inférieure (et modèle synchrone)	E	D
Dijkstra	E + n.D	D^2
Bellman-Ford	n.|E|	D
Meilleur connu	E + $n.\log^3 n$	$D.\log^3 n$

Arbre DFS

Il s'agit maintenant de visiter tous les sommets d'un graphe et de construire un arbre couvrant en longueur. En séquentiel l'algorithme DFS fonctionne comme suit : On démarre en r_0 et pour chaque v faire : si v a des voisins non encore visités, alors on en visite 1. Sinon, on revient vers le sommet qui a visité v en premier. S'il n'y a pas de tel sommet, la recherche est terminée. En séquentiel, cela prend un temps O(|E|).

En distribué, un algorithme consiste à simuler l'algorithme séquentiel en utilisant un jeton. Seul le sommet ayant le jeton pourra exécuter des actions. Pour savoir si w, voisin de v a été visité ou non, il est nécessaire d'envoyer un message sur l'arête {w, v} (mais on peut le faire qu'une seule fois). Le temps et le nombre de messages sont donc en O(|E|). Mais on peut améliorer le temps, en évitant de traverser les arêtes qui ne sont pas dans l'arbre.

Solution : lorsque v est visité la première fois :

- On gèle le DFS
- informer tous les voisins que v a été modifié (ie, les voisins de v stockent "v visité"). Cela permet de faire qu'ils n'aillent pas le visiter leur tour venu, pour améliorer les traitements.
- Attendre un accusé de réception de tous les voisins
- reprendre l'exécution

Ainsi, un sommet va savoir exactement les sommets voisins restant à visiter. Le jeton traverse uniquement les arêtes de l'arbre. Cela prend O(1) en temps par étape et O(deg(v)) messages. On obtient donc une complexité finale :

- MESSAGE(DistDFS) = O(|E|)
- TEMPS(DistDFS) = O(n)

MST : Minimum Stanning Tree

Soit G = (V, E, w), ou w est une fonction de poids. Soit G' le sous graphe de G et tel que $w(G') = \sum_{e \in E(G')} w(e)$.

L'arbre T est MST pour G si w(T) est minimum pour tout arbre T de G.

Hypothèses

Chaque sommet connait le poids de ses arêtes incidentes. On suppose que les poids des arêtes sont tous différents. Par

conséquent, MST est unique. Si tous les poids sont identiques, et toutes les identités des sommets sont identiques, alors aucun algorithme distribué existe (avec un nombre fini de messages).

On admet les propriétés suivantes, pour tout graphe :

- Tout algorithme distribué calculant un MST pour G sans connaissance (sauf les poids) nécessite $\Omega(|E|)$ messages.
- Tout algorithme distribué sur le graphe à n sommets nécessite $\Omega(n.\log n)$ messages.

Définition "Fragment" : Un arbre T d'un graphe G = (E, V, w) est un MST fragment de G s'il existe un MST T_M de G tel que T est un sous arbre de T_M. Un arête e est sortante du fragment T si exactement une extrémité de e est dans T. L'arête sortante de poids minimum du fragment T est notée MWOE(T).

Si e = MWOE(T) alors T U {e} est un fragment.

Par conséquent, on obtient l'algorithme suivant :

- Répéter jusqu'à obtenir un arbre couvrant :
  - Choisir un fragment T et ajouter son arête MWOE(T)

Nous allons donc maintenant voir deux algorithmes permettant de calculer un MST.

Algorithme de PRIM

Au départ, on choisit comme fragment un sommet r_0. La source r_0 va synchroniser la construction de T U {e} avec e = MWOE(T). La source envoie un message "Pulse" à tous les sommets du fragment T ($r_0 \in T$) pour indiquer que la phase p + 1 a commencé. (T possède alors p sommets et chaque sommet de T connait ses voisins dans T).

Chaque sommet détermine son arête (u, v) telle que $v \notin T$ de poids minimum et informe la racine du résultat. Les résultats sont concentrés en r_0 et on en profite pour calculer le minimum de ses descendants (pour minimiser le nombre de messages). Il reste alors à r_0 d'ordonner l'ajout de MWOE(T).

D'ou l'algorithme DistPRIM :

- r_0 génère un message "Pulse" et le diffuse sur T. Pour la toute première phase, ce message va aussi informer les sommets de T du nouveau sommet y et de l'arête e = (x, y) ajoutée à l'arbre à l'étape d'avant.
- A la réception d'un message "Pulse", chaque sommet v de T connecté dans G par e = (v, y) à y marque l'arête e comme interne.
- Le sommet y interroge ses voisins pour connaitre ceux qui sont dans T et les marque.
- Si v est une feuille, il envoie à son père dans T le message (e, w(e), Ze) ou e est l'arête la plus légère sortante de v et pas dans T et elle que e = (v, Ze).
- Ces triplets sont concentrés sur T vers r_0 en calculant le minimum w(e) reçu (un sommet interne $v \in T$ reçoit w_1, w_2,... et calcule le minimum – y compris ses propres valeurs – avant la transmission vers le père).
- La racine choisit le triplet (e, w(e), Ze) avec w(e) minimum. Le minimum est calculée avec les valeurs de r_0.

Cet algorithme est correct (il calcule bien un MST) car il applique la règle e = MWOE(T). Nous allons maintenant voir sa complexité. Etudions d'abord la complexité pour la phase p + 1 :

- TEMPS (phase p + 1) = O(p), pour la diffusion et la concentration
- MESSAGE (phase p + 1) = 2|E(T)| + O(deg y) = O(p + deg(y))

On obtient alors une complexité totale :

- $TEMPS(DistPRIM) = \sum_{p=1}^{n-1} O(p) \leq O(n^2)$
- $MESSAGE(DistPRIM) = \sum_{p=1}^{n-1} c.(p + deg(y_p)) \leq O(n^2 + |E(G)|) = O(n^2)$

Algorithme GHS (Synchrone) : Gallager, Humblet, Spira (1983) – variante de Kruskal

A une certaine étape, soient F_1, F_2, ..., F_k les fragments obtenus (chaque sommet de F_i connait la racine). On détermine MWOE(F_i) pour chaque fragment F_i en utilisant DistPRIM telle que e_i = MWOE(F_i) = (v_i, y_i). Le sommet v_i va alors envoyer un message "demande de fusion".

Si on a par exemple $e_1 = e_2$, on fusionne F_1 et F_2 par e_1 et cela forme un nouveau fragment composé de F_1 U F_2. Sinon, les arêtes e_i forment une structure particulière. On note <F> l'ensemble des fragments et <E> les arcs induits par les messages "demande de fusion". On obtient alors le graphe <G> = (<F>, <E>).

Chaque composante de <G> est un graphe orienté (les arcs pointent vers la racine) saf que la racine est un double fragment F_1 <=> F_2. Tous les fragments d'une même composante sont fusionnés. On décide que la racine du fragment F_1 <=> F_2 ayant le plus petit identifiant devient la racine du nouveau grand fragment. Pour cela, il faut alors faire une diffusion de v_1 et v_2 vers tous les fragments rattachés.

Cela peut se faire car les sommets extrémités d'un arc de <E> ont reçu un message "demande de fusion" et vont pouvoir diffuser sur cette arête la nouvelle racine du grand fragment.

On continue ainsi jusqu'à obtenir un seul fragment. On peut détecter cette terminaison car tous les voisins des sommets du fragment seront dans l'arbre (il n'y a plus d'arête MWOE(F)).

Voyons maintenant la complexité de cet algorithme :

- TEMPS (phase i) = O(n)
- MESSAGE (phase i) = O(n + |E(G)|) = O(|E(G)|), ou n représente la diffusion concentration dans tous les fragments et |E(G)| la dernière étape de Prim.

Il s'agit maintenant de remarquer qu'entre la phase i et la phase i + 1, le nombre de fragments est deux fois plus faible (au moins). Il y a donc logn phases au plus (avec n fragments au départ). On obtient donc une complexité globale :

- TEMPS(GHSSynchrone) = O (n.logn)
- MESSAGE(GHSSynchrone) = O(|E(G)|.logn)

Synchronisation

Ecrire un algorithme pour un système asynchrone est plus difficile que pour un système synchrone. L'idée est donc de transformer automatiquement un algorithme synchrone (pour un système symétrique) en algorithme asynchrone.

Cela va impliquer un surcoût en temps et en nombre de messages. Mais, dans la plupart des cas, ce surcoût ne change pas les complexités.

Méthodologie

Soit Π_s un algorithme synchrone écrit pour un système synchrone et σ un synchroniseur. On souhaite $\Pi_A = \sigma(\Pi_s)$ de façon à ce que Π_A puisse s'exécuter sur un système synchrone. La simulation doit être correcte, c'est à dire que l'exécution de Π_A doit être similaire à celle de Π_s. Le synchroniseur a 2 composantes :

- la composante originale c.o
- la composante de synchronisation c.s

L'idée de base est que chaque processeur v possède un générateur de "pulses", $P_v = 0, 1, 2, ...$ Le processeur v exécute Π_A à la phase p uniquement si $P_v = p$. En asynchrone, pour passer de $P_v = p$ à $p + 1$, il faut communiquer avec ses voisins.

Notation : t(v, p) = processeur v met sa variable $P_v = p$.

Le processeur v est à la phase p si v est dans la période [t(v, p), t(v, p + 1) [.

Compatibilité de pulse

Si le processeur v envoie un message "original" M à un voisin w durant la phase $P_v = p$ alors M est reçu en w durant $P_w = p$.

Exécutions similaires

Soit Π_S un algorithme synchrone et $\Pi_A = \sigma(\Pi_S)$ une simulation (un algorithme asynchrone combiné avec le synchronisation σ). On considère une exécution $E^S = E^S(G, I)$ dans un système G synchrone et l'exécution E^A correspondante de Π_A dans le système G asynchrone avec les mêmes entrées (I = état initial des processeurs). Les exécutions sont similaires si pour tout v et voisin w et la variable locale X en v à la phase p >= 0, on a :

- La valeur stockée en X au début de la phase p dans E^A est la même qu'au début de la phase p dans E^S.
- Les messages originaux envoyés par v à w durant la phase p dans E^A (s'il y en a), sont les mêmes que ceux envoyés de v à w durant la phase p de E^S.
- Les messages reçus par v de w durant la phase p dans E^A sont les mêmes que ceux reçus par v de w durant la phase p dans E^S.

- Le résultat final de v dans E^A est le même que dans E^S.

Simulation correcte

Π_A est une simulation correcte de Π_S si pour tout graphe G et toute entrée I, les exécutions sont similaires. Si $\Pi_A = \sigma(\Pi_s)$, alors le synchroniseur est dit correct, et Π_A simule Π_s.

Si le synchroniseur σ garanti la compatibilité de "pulse", alors il est correct. La question principale : Quand v doit-il incrémenter P_V?

Mesures de complexités des synchroniseurs

On note respectivement $TEMPS_{init}(\sigma)$ et $MESSAGE_{init}(\sigma)$ le temps et le nombre de messages pour initialiser le composante de synchronisation.

On note $MESSAGE_{pulse}$ (σ) le nombre total de messages envoyés par σ durant une phase.

On note $TEMPS_{gap}(\sigma) = max_{v,\,p}\{t(v, p+1) – t(v, p)\}$. $t_{max}(p) = maxt_v(v, p)$ est le moment ou le dernier processeur passe à la phase p.

$TEMPS_{pulse}(\sigma) = max_{p>=0}\{tmax(p+1) – tmax(p)\}$

Clairement, on souhaite construire un synchroniseur σ avec $TEMPS_{pulse}(\sigma)$ et $MESSAGE_{pulse}(\sigma)$ le plus faible possible. Propriété (pour $\Pi_A = \sigma(\Pi_s)$) :

- $MESSAGE(\Pi_A) =< MESSAGE_{init}(\sigma) + MESSAGE(\Pi_s) + TEMPS(\Pi_s).MESSAGE_{pulse}(\sigma)$.
- $TEMPS(\Pi_A) =< TEMPS_{init}(\sigma) + TEMPS(\Pi_s).TEMPS_{pulse}(\sigma)$

Deux synchroniseurs

Synchroniseur alpha

C'est le plus simple. On effectue une synchronisation à la fin de chaque phase p en informant à tous ses voisins qu'on a fini la phase p (tous les messages de la phase p ont été reçus).

Le synchroniseur alpha doit être envoyé d'une racine r_0 vers tous les sommets (innondation) :

- $MESSAGE_{init}(alpha) = O(|E(G)|)$
- $TEMPS_{init}(alpha) = O(diam(G))$
- $MESSAGE_{pulse}(alpha) = O(|E(G)|)$
- $TEMPS_{pulse}(alpha) = O(1)$

Synchroniseur beta

Il suppose l'existance d'un arbre couvrant T. Chaque sommet, lorsqu'il apprend que tous ses descendants ont terminé, il en informe son père? On effectue une concentration vers la racine. Ensuite, la racine diffuse dans l'arbre T le message "next pulse" et tous les sommets incrémentent alors P_V :

- $MESSAGE_{init}(beta) = O(n.|E(G)|)$
- $TEMPS_{init}(beta) = O(diam(G))$
- $MESSAGE_{pulse}(beta) = O(n)$
- $TEMPS_{pulse}(beta) = O(diam(G))$ (si T est BFS)

En fait, $MESSAGE_{init}$ et $TEMPS_{init}$ dépendent de l'arbre (Dijkstra).

Coloration

La motivation de la coloration est de casser la symétrie (choisir un leader) et créer du parallélisme en distinguant des sommets (MDS : Maximal Dependant Set).

Problème de coloration

Une coloration propre d'un graphe G = (V, E) est une fonction c : V --> N tel que si {u, v} appartiennent à E alors c(u)!=c(v). Le plus petit nombre k pour que G soit coloriable (proprement) avec k couleurs est noté $\chi(G)$ et s'appelle le nombre chromatique de G. Dans toute la suite, on se placera dans le modèle local.

Réduction de palette

On va décrire une procédure basique qui va réduire le nombre de couleurs de m à $\Delta + 1$ ou Δ est le degré maximum du graphe. Il est toujours possible d'avoir $\chi(G) =< \Delta + 1$.

Si P est un ensemble de couleurs et W un ensemble de sommets.

FIRST_FREE
Soit c la plus petite couleur de P qui n'est pas utilisée par aucun sommet de W.
Return (c)

Soit P_m = {1, ..., m}, N(v) = {voisins de v}

Procedure REDUCE (pour chaque sommet de v)
Pour i = Δ + 2 à m faire
Si c(v) = i alors
c(v) = FIRST_FREE(N(v), $P_{\Delta+1}$)
Informer tous les voisins de ce choix

Propriétés :

- REDUCE produit une coloration propre avec au plus $\Delta + 1$ couleurs.
- TEMPS (REDUCE) = $m - \Delta + 1$

On remarque que 2 sommets voisins de u et v ne peuvent pas être actifs en même temps (sur le choix de la couleur) car au départ c(u) != c(v).

3-coloration pour les arbres et les graphques de degré borné

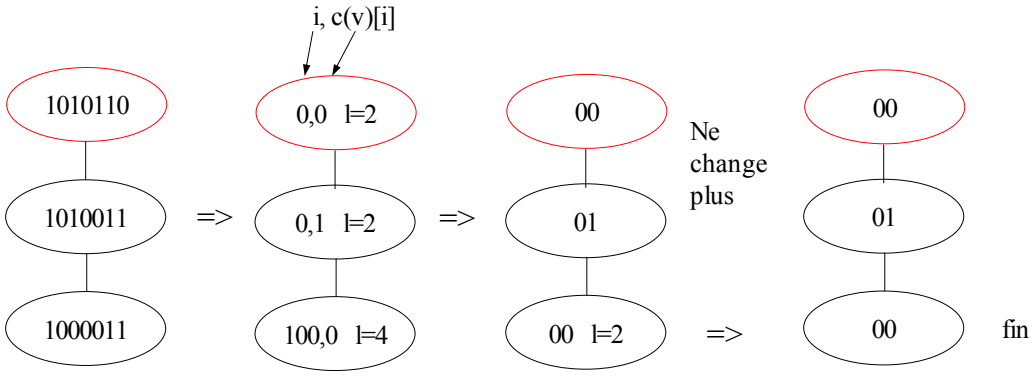
On parle de k-coloration si G a une coloration propre avec un nombre couleurs inférieur à k. Si T est un arbre, alors $\chi(T) = 2$. On note :

- $\log^{(i)} n$ = loglog...log n (i fois)
- $\log^{*} n = \min\{i / \log^{(i)} n =< 2\}$

Si c appartient à N, bin(c) est la représentation binaire standard de c. |bin(c)| = longueur de bin(c) (en bits). A o B est la concaténation des chaines binaires A et B.

6-COLOR(T)
c(v) = bin(ID(v))
Répéter :
l = | c(v) |
Si v est la racine, I = 0
Sinon, I = min {i / c(v)[i] != c(pere(v))[i]} (i est le premier bit ou c(v) et c(pere) diffèrent)
c(v) = bin(i) o c(v)[i]
informer tous les voisins
jusqu'à l = | c(v) |

Exemple :



Propriété :

A chaque itération, c est une coloration propre

Preuve :

Soit u = prec(v) (précédent) et soient I, J les indices choisis par 6-COLOR. Si Iu != Iv alors bin(Iu) != bin(Iv). Donc la coloration est propre. Si Iu = Iv alors c(u)[Iu] != c(v)[Iv] (car Iu est le premier bit différent). Soit K_i = | c(v) | après la $i^{ième}$ itération. K_0 = K = O(logn) bits et K_{i+1} = [partie entière supérieure de (log Ki)] + 1. $K_1 \sim$ loglog n.

D'ou : Si $K_i >= 4$ alors $K_{i+1} < K_i$.

Propriété :

Si K_i =< [partie entière supérieure de $(\log^{(i)} K)$] + 2. Pour tout i tel que $\log^{(i)} K >= 2$. (preuve par induction)

Propriété :

REDUCE produit une 6-Coloration.

Preuve :

A l'étape finale K_i =< 3. la dernière couleur est formée d'une position {0, 1, 2} et d'un bit. Cela fait donc au plus 6 possibilités de couleurs.

Passage de à 3 couleurs :

SHIFT_DOWN : chaque sommet (différent de la racine) se recolorie avec la couleur de son père. La racine choisit une nouvelle couleur. SHIFT_DOWN produit une coloration propre et tous les frères ont la même couleur.

SIX2THREE :

Pour x appartenant à {4, 5, 6} faire :

SHIFTDOWN

Si c(v) = x, c(v) = FIRST_FREE(N(v), P_3 = {1, 2, 3})

Propriété :

SIX2THREE est une 3-coloration.

Preuve :

Tous les frères sont de la même couleur. Lorsque SHIFT_DOWN est appliqué, FIRST_FREE termine car le père de v et ses fils n'utilisent que 2 couleurs.

Théorème 1 :

Il existe un algorithme distribué (LOCAL) qui produit une 3-coloration pour tout arbre à n sommets en O(log*n) temps. La technique s'étend aux graphes de degré au plus Δ. On utilise une variante de l'algorithme 6-Color.

- c(v) = bin(I) o c(v)[I]
- I = min{i / c(v)[i] != c(père(v))[i]}
- c(v) = (bin(I_1) o c(v)[I_1], ..., bin(I_Δ) o c(v)[I_Δ])
- I_j = min {i / c(v)[i] != c(v_j)[i]} ou v_j est le $j^{ième}$ voisin de v.

Si c(v) est sur K-bit c'(v) est sur Δ o (partie entière supérieure de (log K) + 1) bits. Après un temps $\log^*(\Delta_n) \sim \log^* n$, on s'arrête avec une couleur de taille 2Δ bits. Ceci produit les $2^{2\Delta}$ couleurs différentes. Pour réduire de $m = 2^{2\Delta}$ à Δ + 1 couleurs, on utilise REDUCE en temps $O(2^{2\Delta} - \Delta) = O(2^{2\Delta})$.

Théorème 2 :

Il existe un algorithme distribué (LOCAL) qui produit une Δ + 1 coloration pour tout graphe à n sommets et de degré maximum Δ en temps $O(\log^* n + {}^{2\Delta})$. Il existe un autre algorithme en O(Δ.log n).

Borne inférieure pour la 3-coloration d'un cycle

D'après le théorème 1, on peut calculer une 3-coloration pour un cycle en temps O(log*n). On va voir qu'il faut Ω(log*n) temps dans le pire des cas (mauvaise distribution des identités).

En temps t, un sommet ne peut connaître que les identifiants à distance inférieure à t de lui. Le sommet v connait un (2t + 1) uplet $(x_1, x_2, ... x_{2t+1}) \in W_{s, n}$ avec x_{t+1} = ID(v), x_t = ID(voisin de v), x_{t-1} = ID(voisin de x_t), ...

$W_{s, n}$ = {$(x_1, ..., x_s)$ / $1 =< x_i =< n, x_i != x_j$} (couleurs /ID distinctes deux à deux)

Soit Π_t un algorithme de coloration pour le cycle avec c_{max} couleurs et en temps t = t(n). Sans perte de généralité, on suppose que Π_t obéit aux 2 règles suivantes :

Pendant t phases, il collecte sa boule de rayon t. A la fin de cette étape, v connait $w(v) \in W_{2t+1, n}$.

Il choisit sa couleur $c(v) = \varphi_\Pi(w(v))$ ou $\varphi_\Pi = W_{2t+1, n}$ -> {1, ..., c_{max}} (la fonction de couleur de Π_t.

On définit le graphe $B_{s, n}$ = l'ensemble des arêtes ($(x_1, x_2, ..., x_s), (x_2, ..., x_s, x_{s+1})$) tel que $x_1 != x_{s+1}$. Intuitivement, 2 s-uplets de $W_{s, n}$ sont adjacents si il est possible de trouver 2 voisins d'un cycle x et y pour une certaine distribution des identifiants.

Propriété :

Si Π_t produit une coloration propre pour tout cycle alors φ_Π est une coloration pour $B_{2t+1, n}$.

Preuve (par l'absurde) :

Si $w = (x_1, ..., x_{2t+1})$ et $w' = (x_2, ..., x_{2t+2})$ voisin dans $B_{2t+1, n}$ tel que $\varphi_\Pi(w) = \varphi_\Pi(w')$. Et donc φ_Π n'est pas une coloration propre pour x_{t+1} et x_{t+2} d'ou contradiction.

Corollaire :

Si les cycles à n sommets sont coloriés en t étapes avec c_{max} couleurs alors le nombre chromatique $\chi(B_{2t+1, n}) =< c_{max}$.

Propriété :

$\chi(B_{2t+1, n}) >= \log^{(2t)} n$.

Soit t_0 le plus petit entier tel que $\log^{(2t)} n > 3$: $t_0 = ½ (\log^* n) – 3$. => $\chi(B_{2t+1, n}) > 3$ => $c_{max} > 3$. L'algorithme n'a pas pu 3-colorier le cycle en temps =< t_0.

Objectif :

Il faut démontrer que $\chi(B_{2t+1, n}) \geq \log^{(2t)} n$. D'ou $\log^{(2t)} n \leq c_{max}$.

- Si $2t \geq \log^* n – 1$ alors $\log^{(2t)} > 3$
- $\log^{(2t)} n \leq 3$ => $t > ½ (\log^* n – 1)$

Soit $\sim B_{s, n} = (\sim W_{s, n}, \sim E_{s, n})$ avec :

- $\sim W_{s, n}$ = { $(x_1, ..., x_n) / 1 \leq x_1 \leq ... \leq x_s \leq n$ }
- $\sim E_{s, n}$ = { $((x_1, ... x_s) (x_2, ... x_{s+1}))$ }

La version non orientée de $\sim B_{s, n}$ est un graphe de $B_{s, n}$.

Définition :

Soit H un graphe orienté = (U, F). Le LINE de H, noté DL(H) est le graphe avec ensemble de sommets : F (ensemble des arcs) e, e' ∈ F adjacents dans DL(H) si il existe v de U tel que e arrive en V et e' part de v.

Propriété :

$\sim B_{1, n}$ est le graphe complet à n sommets (2 sommets sont connectés par un arc dans chaque sens).

Théorème :

Tout algorithme distribué de 3-coloration d'un cycle à n sommets nécessite un temps supérieur à ½ (log*n – 1)

Ensembles indépendants Maximaux

Dans ce chapitre, on considérera le modèle LOCAL.

Algorithme de base

Soit G un graphe M inclu dans V(G). M est un MIS (Maximal Independant Set) si :

- Pour tout u != v de M, d(u, v) >= 2
- M U {x} n'est pas indépendant

En séquentiel, on peut calculer un MIS en temps O(n + m) avec un algorithme glouton :

- U = V(G), M = {}
- Tant que U != {} faire :
  - choisir arbitrairement v de U
  - U = U – N(V) (N(V) = {w / d(v, w) =< 1})
  - M = M U {v}

Le résultat de l'algorithme calculant un MIS est matérialisé par une variable (locale à chaque sommet) notée ^b :

- ^b = 1 si le sommet appartient à un MIS
- ^b = 0 si le sommet n'appartient pas à un MIS
- ^b = -1 initialement

L'algorithme naif MIS_{DFS} consiste à simuler l'algorithme séquentiel en faisant un parcours DFS des sommets du graphe. On parcours tous les sommets et si ^b = -1, alors on choisit u de MIS, ^b=1 et pourtous ces voisins, ^b = 0.

lemme :

- MESSAGE (MIS_{DFS}) = O(|E|)
- TEMPS(MIS_{DFS}) = O(n)

Algorithme MIS_{RANK} :

Il existe une variante : MIS_{RANK}. L'idée consiste à décider de joindre le MIS que si tous ces voisins qui ont une ID plus grand ont déjà décidé de ne pas joindre le MIS.

On peut écrire la procédure JOIN comme suit :

Procedure JOIN :
Si tout voisin w de u tel que ID(w) > ID(u) et ^b(w) = 0 alors
^b(u) = 1
envoyer "DECIDED 1" à tous ses voisins

En utilisant cette procédure, on peut alors écrire l'algorithme MIS_{RANK} :

Algorithme MIS_{RANK} :
Appliquer JOIN
A la réception d'un "DECIDED 1" venu de w faire :
^b = 0
Envoyer "DECIDED 0" à tous ces voisins
A la réception d'un "DECIDED 0", appeler la procédure JOIN

Lemme :

MESSAGE (MIS_{RANK}) = O(|E|)

TEMPS (MIS_{RANK}) = O(n)

MIS_{RANK} est basé sur le fait que les ID sont différents. MIS_{DFS} est basé sur le fait qu'un sommet se réveille.

Lemme :

Il n'existe pas d'algorithme déterministe distribué qui calcule un MIS sur un cycle anonyme (pas d'ID) si tous les sommets se réveillent en même temps.

Réduction de la coloration au MIS

Dans une coloration, chaque couleur induit un ensemble indépendant. Mais ce n'est pas forcément un MIS. Cependant, il existe une procédure qui trnasforme une m-coloration et produit un MIS.

COLOR2MIS :
Pour tout i = 1 à m faire :
Si c(v) = i alors :
Si aucun voisin de v n'a rejoint le MIS :
^b(v) = 1
Informer tous les voisins
Sinon ^b(v) = 0

Propriétés :

La procédure COLOR2MIS construit un MIS en temps O(m). L'indépendance est assurée par le fait que la procédure ^b(v) passe à 1 si tous les voisins ne sont pas dans le MIS. Il s'agit d'un ensemble maximal car tous les sommets sont passés en revue et sont ajoutés au MIS si possible.

Corollaire :

Si en temps t(G) on peut colorier G avec f(G) couleurs, alors en temps O(t(G) + f(G)) on peut construire un MIS

Corollaire :

Pour les arbres et les graphes de degrés bornés à n sommets, on peut un MIS en temps O(log*n)

Théorème :

Tout algorithme distribué qui construit un MIS sur un cycle à n sommets nécessite au moins un temps ½(log*n – 3).

Preuve :

On sait que ½ (log*n – 1) étapes sont nécessaires pour calculer une 3 coloration sur un cycle. On va montrer que tout MIS sur un cycle peut être transformé en 3-coloration en 1 seule étape. Les sommets du MIS envoient "2" à gauche et se colorie 1. Les sommets qui n'appartiennent pas au MIS se colorient 2 s'ils reçoivent le message "2", sinon ils se colorient "3".

Si les sommets ne connaissent pas leur droite et gauche, alors c'est encore possible en une étape. Il suffit d'envoyer 2 de chaque côté lorsqu'un sommet est dans le MIS et le sommet passe à 1. Sinon, on envoie l'ID (à droite et à gauche). Si deux sommets adjacents ne sont pas dans le MIS (c'est le seul cas possible), celui dont l'ID est la plus petite se colorie en 2, l'autre en 3 par exemple.

Ensemble dominant :

M est un ensemble dominant pour le graphe G si tout sommet est soit dans M, soit à un voisin dans M. Un MIS est un ensemble dominant. L'objectif est de construire un ensemble dominant de taille la plus petite possible (NP-COMPLET). On veut construire un ensemble dominant de cardinalité inférieur à n/2.