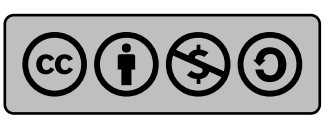
Algorithmes Distribués

Cyril Gavoille

LaBRI
Laboratoire Bordelais de Recherche
en Informatique, Université de Bordeaux

gavoille@labri.fr

20 mars 2025
– 244 pages –



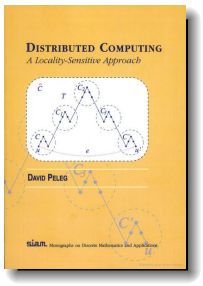
Algorithmes Distribués

– Master 1 & 2 –

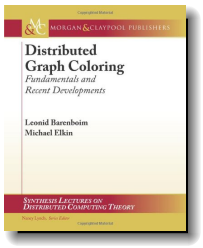
Objectifs : Introduire l'algorithmique distribuée ; concevoir et analyser des algorithmes distribuées (M2) ; présenter différentes applications et problématiques actuelles ; programmer des algorithmes sur un simulateur de calcul distribué (M1).

Pré-requis : Algorithmique ; notions de théorie des graphes

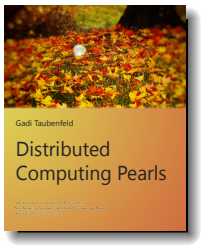
Quelques ouvrages de référence :



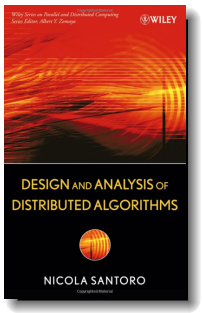
Distributed Computing : A Locality-Sensitive Approach
David Peleg
SIAM Monographs on Discrete Mathematics and Applications, 2000



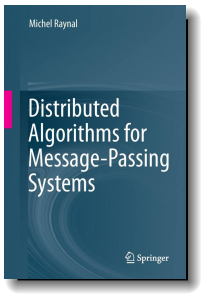
Distributed Graph Coloring : Fundamentals and Recent Developments
Leonid Barenboim et Mickael Elkin
Morgan & Claypool, 2013



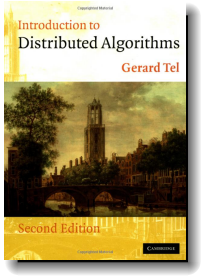
Distributed Computing Pearls
Gadi Taubenfeld
Morgan & Claypool, 2018



Design and Analysis of Distributed Algorithms
Nicola Santoro
Wiley Series on Parallel and Distributed Computing, 2006



Distributed Algorithms for Message-Passing Systems
Michel Raynal
Springer, 2013



Introduction to Distributed Algorithms (2nd Edition)
Gerard Tel
Cambridge University Press, 2000

An Introduction to Quantum Computing
Phillip Kaye, Raymond Laflamme et Michele Mosca
Oxford Press, 2006

Communication Complexity
Eyal Kushilevitz et Noam Nisan
Cambridge University Press, 1997

Table des matières

CHAPITRE

1 Introduction

Nuée d'oiseaux au-dessus du lac Baïkal.

Sommaire

Dans ce premier chapitre nous balayons les spécificités du calcul distribué en pointant les principales caractéristiques de ce vaste domaine (mémoire partagée, synchronie, connaissance partielle, défaillance, etc.).

Mots clés et notions abordées dans ce chapitre :

- machines séquentielles, machines parallèles, systèmes distribués,
- passage de messages, mémoire partagée, modèle PRAM,
- synchronisme.

1.1 Qu'est-ce que le calcul distribué ?

On a tendance à le définir par ce qu'il n'est pas : un calcul séquentiel ou bien parallèle.

Une machine (ou architecture) *séquentielle* permet le traitement d'une seule opération à la fois. Dans un programme séquentiel, comme l'extrait ci-dessous, les instructions s'exécutent les unes à la suite des autres, dans un ordre bien précis.

```
1  x = x + 1
2  y = 2*x - 3
3  print(y)
4  ...
```

Les machines *parallèles* permettent le traitement de plusieurs opérations en parallèle. En générale ces machines permettent d'exécuter en parallèle de nombreuses instructions élémentaires toutes similaires sur des données différentes (SIMD : *Single Instruction Multiple Data*). Les ordinateurs classiques, les architectures 64 bits ou plus par exemple, sont déjà des sortes de machines parallèles dans la mesure où elles traitent 64 bits simultanément (dans des registres et lors d'un même cycle machine). Le degré de parallélisme, qui mesure le nombre d'opérations réalisables en parallèles, est assez limité dans ce cas. Lorsque le nombre de bits manipulés en parallèle devient très important, on parle de machines *vectorielles* (comme par exemple les machines Cray dans les années 1990, voir figure 1.1). Sur de telles machines, ont peut par exemple calculer $A \cdot X + B$ où A, B sont des vecteurs de grandes tailles et X une matrice. Les machines parallèles possèdent un fort degré de synchronisation.

Un *système distribué* (on ne parle plus de machines) est similaire à une machine parallèle à la différence qu'il y a plusieurs entités de calculs autonomes distantes. Contrairement aux machines parallèles, les entités de calcul sont faiblement synchrones (voir complètement hétérogènes) et relativement distantes les unes des autres. Typiquement la distance entre les entités autonomes est plus grande de plusieurs ordres de grandeur que les dimensions des entités de calcul.

Bien sûr, tout est une question d'appréciation [1], car dans une machine séquentielle classique on peut trouver des unités de calcul parallèle (registres 64 bits, cartes GPU, etc.) et des processeurs multi-cœurs pouvant être considérés comme formant un mini-système distribué. Cependant, par *machines* séquentielles ou parallèles on sous-entend en général que tout ou presque tient sur une même carte mère, à l'intérieur d'une même machine justement, d'une même pièce ou d'un même bâtiment. Ce n'est pas le cas d'un système distribué comme Internet connectant ses routeurs via différents protocoles. On

1. Toute définition formelle est vouée à l'échec. On parle de *notion émergente*, comme une « dune ». À partir de quel moment un tas de sable devient-il une dune ? Dit autrement, après la suppression d'un seul grain de sable, quand est-ce qu'une dune redevient un tas de sable ?

FIGURE 1.1 – Super-calculateur vectoriel Cray-2 lancé en 1985, le premier a dépasser le Giga-flops (= 10^9 *floating operations per second*). Une des innovations est sa forme en arc de cercle permettant, en réduisant la longueur des connexions, une bande passante exceptionnelle et une meilleure synchronisation de la mémoire. En juin 2022, 37 ans plus tard, le premier supercalculateur Exa-flops (= 10^{18} flops) apparaît au classement top500. Il s'agit d'une machine HPE Cray EX225a possédant 8 730 112 cœurs.

va voir au paragraphe 1.3 que la distance entre entités de calcul est un facteur très limitant.

En résumé, ce qui caractérise les systèmes distribués par rapport aux machines parallèles, c'est :

1. potentiellement beaucoup plus d'entités de calcul ;
2. distance physique entre entités plus importante.

Le *calcul distribué* est donc l'art de faire des calculs avec de tels systèmes, de réaliser une tâche globale à l'aide de calculs locaux. Quant à son utilité, on fait des calculs avec des systèmes distribués pour essentiellement deux raisons :

1. augmenter à moindre coût la puissance de calcul et de stockage, en ajoutant graduellement des entités, ou en regroupant des ensembles d'entités déjà existantes (*clusters* ou grappes de PC, Grid5000, *data-centers*, *clouds* ...) ;
2. communiquer entre entités distantes via des protocoles (réseaux LAN, réseaux de satellites, Wifi, téléphonie mobile, réseaux sociaux, ...), mais aussi mettre en rapport des banques de données réparties (transactions bancaires, Internet, échange de fichiers, *streaming*, *peer-to-peer*, crypto-monnaie et *block-chain*, ...).

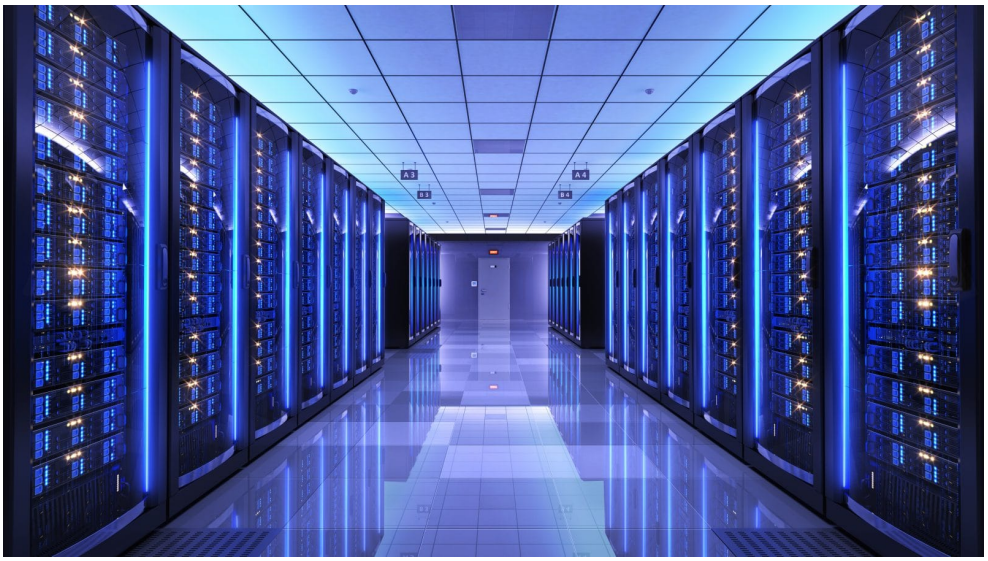


FIGURE 1.2 – *Data-center*.

1.2 Deux modèles de base

Les entités de calculs peuvent être incarnées par tout un tas d'objets différents (on parle parfois d'*IoT* : *Internet of Things*). Ce sont des téléphones portables, des tablettes,

des processus, des cœurs de calcul, des robots, des drones, des oiseaux [2], des insectes [3], des cellules biologiques [4] ... Pour simplifier, on choisira à partir de maintenant le terme de « processeur » comme synonyme d'« entité de calcul », bien que ce dernier terme soit plus générique.

En général, un modèle spécifique est associé à chaque situation : processus, oiseaux ou cellules ... tout n'est évidemment pas pareil ! Aussi, on distingue classiquement deux grands modèles (ou classes de modèles) de systèmes distribués selon la manière de communiquer des processeurs.

1.2.1 Mémoire partagée (*shared memory*)

> Dans ce modèle, les processeurs ne communiquent pas directement. Ils s'échangent des informations grâce à une mémoire ou des variables communes qu'ils peuvent tous lire et modifier à l'aide d'instructions du type Read et Write.

Dans ce modèle, les instructions Read/Write sont atomiques [5] et ont généralement un coût faible, voir unitaire, si bien que le coût des communications est passé sous silence. On utilise surtout ce modèle pour étudier des problèmes liés à l'assynchronisme des Read/Write et aux accès concurrents à la mémoire.

Cela se produit par exemple lorsqu'on souhaite améliorer l'efficacité de l'implémentation d'un objet partagé, tel un compteur ou une liste chaînée. Chaque processus peut vouloir incrémenter un compteur partagé (comme sur une page *web* par exemple), ajouter ou supprimer un élément d'une liste chainée partagée (comme une entrée dans une base de données). Il s'agit donc d'implémenter les primitives de ces objets (Inc(C), Add(L, e), Del(L, e), ...) à l'aide de Read/Write, dans ce contexte de concurrence de l'accès à ces objets. [*Question.* Pour l'incrémentation Inc(C) d'un compteur partagé C, décrivez un scenario où l'implémentation naïve `n=Read(C);Write(n+1);` fait qu'un utilisateur verrait son compteur diminuer, et donc ne pas refléter le nombre d'appel à Inc(C).]

Ce modèle sert aussi à utiliser pleinement la capacité d'un processeur multi-cœurs. Pour cela on pourra, par exemple, faire appel aux routines de *multithreading* et d'exclusion mutuelle (*mutex*) de la bibliothèque `#include <pthread.h>` de `C`. Ou encore utiliser une version simplifiée grâce à OpenMP (*Open Multi-Processing*) avec `#include <omp.h>`

2. Notamment via les protocoles de population.
3. Comme les fourmis par exemple [FK12][FK13].
4. Dans ce cas on utilise des modèles comme le *beeping model* [CK10], voir le paragraphe 1.2.5.
5. Cela veut dire qu'il est possible de *linéariser* chaque instruction Read/Write du systèmes (instructions qui peuvent avoir des durées différentes), ce qui revient à les dater ou à s'accorder sur une notion commune de temps. Ainsi deux Write exécutés successivement sur un même processeur renverront deux états successifs temporellement ordonnés de la mémoire, le dernier Write renvoyant le dernier état supposé de la mémoire. Ce n'est malheureusement pas le cas si les Read/Write ne sont pas linéarisables où les processeurs ne peuvent plus s'accorder sur les notions de passé et de futur.

et les directives `#pragma omp`.

Mais le modèle de mémoire partagée sert aussi plus fondamentalement à l'étude des systèmes où des pannes peuvent se produire et ralentir arbitrairement les Read/Write. Comme on le verra dans la suite du cours, dans un environnement asynchrone, il y a une vraie difficulté à prouver qu'un protocole réalise effectivement la tâche désirée. C'est lié à la multitude des scenarios asynchrones possibles. De plus, les défaillances possibles imposent systématiquement aux protocoles d'être *sans-attente* (*wait-free* en Anglais). Cela signifie que l'algorithme doit toujours pouvoir progresser, c'est-à-dire décider de sa prochaine action, en un temps fini quel que soit le scenario. Sans cette condition, le protocole peut ne jamais terminer. Un protocole qui, par exemple, attendrait de voir apparaître une certaine valeur dans une certaine case de la mémoire n'est pas une solution sans-attente. Pour chaque processeur, les actions se résument donc à lire / écrire / calculer / terminer.

Consensus. Le problème typique étudié dans ce modèle est le Consensus et ses nombreuses variantes. Dans sa forme de base les processeurs ont chacun en entrée une valeur, 0 ou 1. Et ceux qui ne sont pas tombés en panne doivent s'accorder sur une valeur initialement proposée (potentiellement par un processeur tombé en panne). Donc, en plus de choisir la même valeur au bout d'un temps fini, les processeurs doivent choisir 0 si tous avaient 0 en entrée et choisir 1 s'ils avaient tous 1. Ici « choisir » signifie que le processeur écrit sa valeur de sortie de manière irrévocable.

Un exemple d'application est celui de prise de décision d'un pilote automatique à bord d'un avion, ou d'autres systèmes critiques similaires (navette spaciale, missile auto-guidé, véhicule autonome, etc.). Le plus souvent les calculs sont exécutés par trois calculateurs différents et indépendants (programmes différents conçus par des sociétés différentes). S'il faut trancher entre tourner à droite ou à gauche afin d'éviter un massif montagneux, on comprend alors toutes les contraintes du problème. (1) Il est important de se mettre d'accord sur une valeur proposée, car si tous les calculateurs considèrent qu'il faut tourner à gauche pour éviter la montagne, il ne faut certainement pas tourner à droite ! (2) Il est important de décider et sans attente, car l'avion file à vive allure et devant une montagne il faut réagir sans trop attendre, même si un des calculateurs est en rade ou qu'un calculateur pense que c'est un peu mieux de tourner à droite qu'à gauche. On pourrait en plus exiger une prise de décision à la majorité, mais ce n'est pas le débat. Cela rajoute une contrainte à un problème qui est déjà très difficile à résoudre sans cela.

Donc ici, le nombre n de processeurs est un peu secondaire. Et peu importe le nombre ou le coût des Read/Write, qu'on suppose d'une certaine façon négligeables par rapport aux délais liés à l'assynchronisme du système. Il s'agit de trouver un protocole permettant de résoudre le problème quel que soit le scénario, chaque processeur exécutant le protocole potentiellement à des vitesses différentes, variables dans le temps, et avec des pannes possibles.

Historiquement, c'est en 1978 dans [Gra78, page 73] que ce problème fût énoncé sous la forme de la coordination de deux armées pour une attaque commune, problème appelé *The Generals Paradox* (voir figure 1.3).

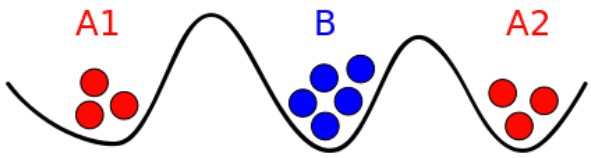


FIGURE 1.3 – Les armées A1 et A2 peuvent vaincre l'armée B seulement si elles coordonnent leur attaque (« A1 + A2 > B »), c'est-à-dire si elles décident d'une date et d'une heure commune d'invasion de la vallée où siège B. Cette coordination ne peut se faire que grâce à l'envoi de messagers qui peuvent malheureusement se faire capturer (perte par omission) puisqu'ils devront traverser la vallée ennemie. Source Wikipédia.

Le CONSENSUS se généralise au cas où il faut décider d'une valeur parmi k fixées et connues de tous, et non pas seulement deux. On parle de k-CONSENSUS. Lorsque $k = n$, il s'agit du problème LEADER ELECTION (l'élection de *leader*). Une autre généralisation, appelée k-SET AGREEMENT, impose que le nombre de valeurs différentes décidées soient au plus k à choisir parmi un ensemble de $k + 1$ valeurs. Donc pour $k = 1$, il s'agit du CONSENSUS.

Trouver un chef d'orchestre, une autorité ou encore élire un *leader*, se révèle être une brique essentielle de nombreux protocoles. Par exemple, tout protocole basé sur un vote à la majorité se résout simplement une fois un *leader* élu, ce qui, en passant, donne aussi une solution particulière au CONSENSUS. [*Question*. Comment ?]

Malheureusement, dans un système asynchrone et en présence d'une seule panne de processeur, le CONSENSUS n'a pas de solution sans-attente. C'est le fameux résultat d'impossibilité « FLP » du noms de ses auteurs [FLP85].

Sur l'impossibilité du consensus. L'idée du résultat d'impossibilité du consensus est la suivante : en fonction du nombre de pas de calculs effectués (en termes de READ/WRITE) et des pannes possibles, on se pose la question vers quelle valeur chaque processeur doit converger. Soit obligatoirement vers 0, soit obligatoirement vers 1, soit vers 0 ou 1 au choix (on parle de configuration bivalente). Un système à deux processeurs suffit pour montrer l'impossibilité. On représente une configuration possible global du système par un chemin où chaque sommet représente l'état d'un des processeurs. On met une arête entre deux sommets s'il existe une configuration globale du système, c'est-à-dire un scénario, dans laquelle l'état de ces sommets peut exister. Quand ces états ne sont pas bivalents, ces sommets sont coloriés par la valeur où il doivent converger, 0 ou 1 donc. Sans aucun pas de calcul, le chemin $0 - 1$ est possible, car l'un des

processeurs peut s'être vu proposer la valeur 0 et l'autre la valeur 1. Sans opération de Read/Write, ils ne peuvent pas deviner la valeur de l'autre et encore moins savoir s'il a crashé. Ils doivent conclurent qu'ils sont seuls au monde et converger vers leur valeur proposée : 0 et 1 respectivement. Cela n'est pas une solution correcte au problème si les deux processeurs n'ont finalement pas crashés. On montre alors que pour chaque pas de calcul supplémentaire il y a un scénario qui revient à insérer dans le chemin courant deux sommets colorés (donc non bivalents). On obtient alors un nouveau chemin où tous les sommets sont colorés et dont les extrémités sont 0 et 1. Il existe donc dans ce chemin une arête $0-1$ montrant qu'après un nombre arbitraire de pas de calculs, les deux processeurs sont dans un état tel que le problème n'est toujours pas résolu.

Lorsque le consensus est possible. Une variante du Consensus a par contre une solution (qu'on ne donnera pas), même en présence de pannes : Approximate Agreement. Le problème est similaire à Consensus excepté que si les deux valeurs 0 et 1 ont été initialement proposées, alors les valeurs choisies doivent être des réels de $[0,1]$, et toutes comprises dans un intervalle de taille arbitrairement petite fixée à l'avance [DLP+86]. Ce problème peut être utile pour décider d'exécuter une certaine action à une date commune (où une petite marge d'erreur sur le moment précis est tolérable, comme la mise à jour d'un *OS* sur un ensemble de terminaux, de *smartphones*, etc.). L'idée principale de la solution à ce problème est de calculer la moyenne des valeurs proposées un certain nombre de fois.

Dans le cas synchrone le problème a une solution. [*Question.* Laquelle ?] Dans le cas synchrone, mais où les crashs peuvent être des pannes *Byzantinnes*, on peut le résoudre seulement si $n > 3f$ où n est le nombre de processeurs total et f le nombre de processeurs fautifs. Un processeur a un comportement Byzantin s'il est en panne, définitivement (*crash*) ou pas, ou si son fonctionnement est défectueux, volontairement (malicieux) ou pas. Les pannes Byzantines modélisent les pires pannes possibles. Le résultat FPL d'impossibilité a juste besoin d'une panne de type crash et de l'assynchronisme.

Dans le modèle de passage par message (modèle discuté au paragraphe 1.2.2 ci-après), et dans le cas synchrone, il est possible de résoudre le problème sans condition sur les pannes mais uniquement dans certaines topologies (voir par exemple [GP16]).

Le consensus en pratique. Il existe cependant une solution au consensus avec un algorithme *avec* attente (et donc avec des *timeouts* et donc des horloges). C'est évidemment fondamental en pratique puisqu'en pratique il faut que ça marche ! Garder la cohérence d'un système distribué est crucial. Il faut quelque part que l'état du système (comme une page *web* Wikipédia par exemple), disons codé par une variable X, corresponde à la vraie valeur, celle qui fait consensus justement. Il y a plusieurs solutions historiques dont RAFT et PAXOS (voir [HM20]. Tous les systèmes distribués tels Facebook ou Google sont basés sur ce principe. Voir aussi [Tau98, Chp. 4] et le site thesecretlivesofdata.com

pour une animation très réussie de l'algorithme RAFT.

Le modèle PRAM. Le modèle PRAM (pour *Parallel Random-Access Memory*, voir figure 1.4) est une généralisation du modèle RAM mais aussi un cas particulier du modèle à mémoire partagée. Il est exclusivement synchrone et a pour but l'étude du parallélisme. Il n'est pas question *a priori* de processeur en panne. Ce modèle se décline à son tour en de nombreuses variantes suivant la politique adoptée pour l'écriture simultanée dans une même cellule par plusieurs processeurs.

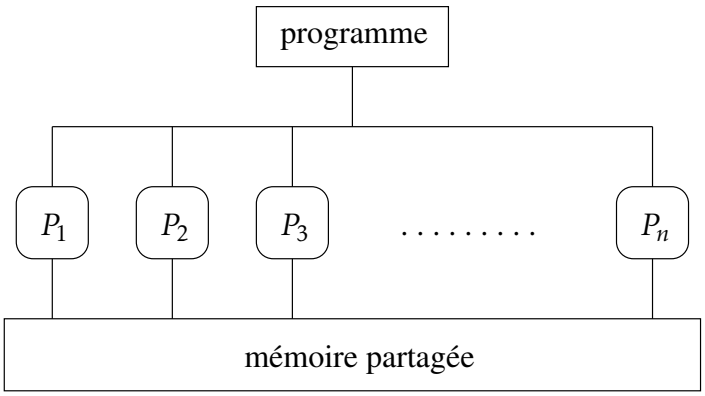


FIGURE 1.4 – Le modèle PRAM : un cas particulier de mémoire partagée.

Notons que le modèle séquentiel RAM est le modèle PRAM avec $n = 1$ processeur. Et les lectures/écritures se font par définition n'importe où en mémoire en temps unitaire (RAM=*Random-Access Memory*). Le modèle PRAM est assez éloigné de la réalité des systèmes distribués à grande échelle. En effet, lorsque n devient grand, il n'est plus physiquement possible de garantir la lecture simultanée d'une même cellule mémoire par les n processeurs, et plus généralement d'avoir un accès en temps uniforme à toutes les ressources par tous les processeurs.

1.2.2 Passage de messages (*message passing*)

> Ce modèle traite explicitement des communications. Lorsqu'un processeur veut communiquer avec un autre, il doit lui envoyer un message qui transite via un médium de communications, grâce à des instructions du type SEND et RECEIVE.

Le coût d'une instruction SEND est loin d'être unitaire, surtout si l'émetteur et le récepteur sont distant et n'ont pas de lien direct. C'est donc précisément l'efficacité des échanges de messages qu'on étudie avec ce modèle quitte à négliger le coût des calculs locaux des processeurs.

Ils existent de nombreux travaux, qu'on ne détaillera pas dans ce cours, qui s'intéresse à simuler sur un système à mémoire partagée un algorithme écrit pour un système par passage de messages, ou le contraire [ABND95]. Donc on peut toujours (au moins en théorie) exécuter sur un système distribué A un algorithme conçu pour un système B, avec, c'est presque certain, des performances dégradées.

Toute la suite du cours concerne le modèle par passage de messages. Dans ce modèle il existe plusieurs modes de communications [6].

1.2.3 Point-à-point

> Chaque processeur ne peut communiquer directement qu'avec un certain nombre de processeurs, ses voisins.

On modélise de tels systèmes naturellement par un graphe connexe (voir la figure 1.5). Les sommets sont les processeurs, et les arêtes les liens directs de communication. Sans précision particulière, on supposera que le graphe est non orienté (ou symétrique) et sans multi-arêtes.

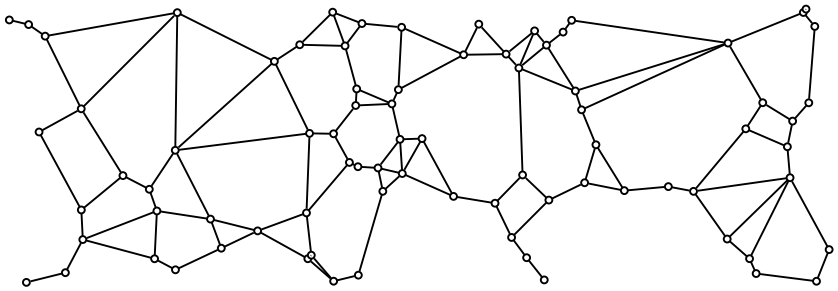


FIGURE 1.5 – Modélisation par graphe du modèle point-à-point. Ici un graphe de Gabriel sur 60 points du plan, où deux points u et v sont voisins si le disque de diamètre le segment uv ne contient aucun autre point.

1.2.4 Réseau de diffusion (*broadcast/radio networks*)

> Chaque processeur peut communiquer le même message simultanément à un certain nombre de récepteurs, comme un émetteur radio le ferait.

6. Malheureusement en calcul distribué il existe plusieurs centaines de modèles censés refléter une réalité visiblement difficile à cerner précisément. On y reviendra au chapitre 10 consacré à l'information quantique.

Là encore il y a de nombreuses variantes suivant que les réceptions multiples en un processeur créent ou non des collisions et si les collisions sont détectables en tant que telles.

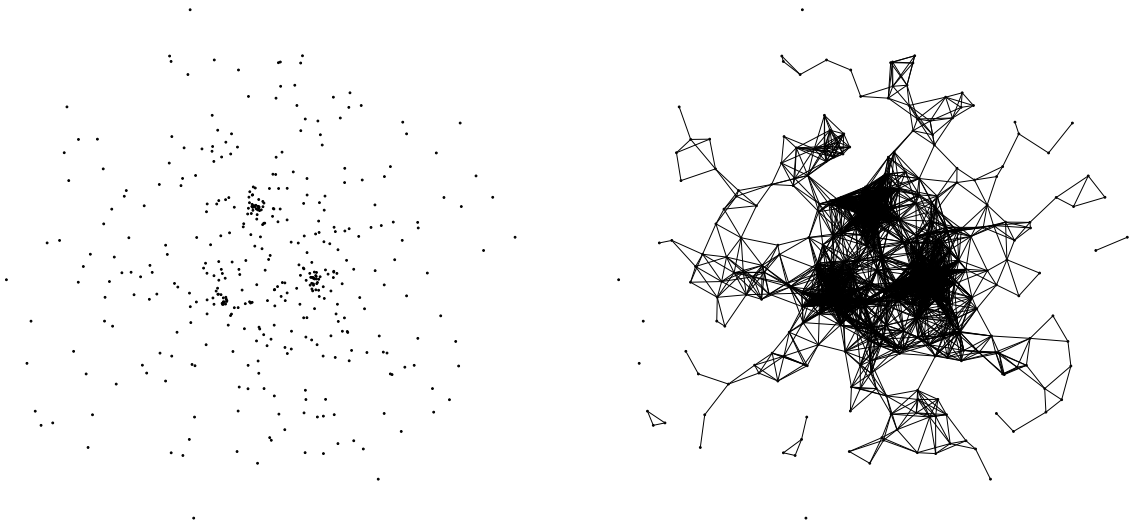


FIGURE 1.6 – Modélisation d'un réseau de diffusion par un graphe UDG (*Unit Disk Graph*) généré à partir d'un nuage de 400 points aléatoires (non uniformes) pris dans le carré $[0,10]\times[0,10]$ (figure de gauche). Un point peut communiquer à tous ceux situés dans sa boule de rayon unité (figure de droite). Notez que ce nuage de points ne produit pas un graphe connexe.

Parmi les variantes, il existe des modèles prenant en compte de manière plus ou moins sophistiquée la propagation et les interférences des signaux émis pour la transmission des messages. Par exemple, dans le modèle SINR (pour *Signal-to-Interference-plus-Noise Ratio*), la puissance d'un signal d'émission (et donc l'ensemble des points qu'il peut atteindre) dépend de l'interférence des signaux des autres émetteurs et du bruit ambiant.

Plus précisément, on note T l'ensemble des émetteurs (terminaux) qui souhaitent transmettre un message simultanément à un moment (ou slot) donné. Alors, un point u pourra recevoir un message d'un émetteur $v \in T$ dans ce slot si u écoute ($u \notin T$) et si le ratio « signal/bruit » de v vis-à-vis de u et de T dépasse un certain seuil β, une constante > 0. Et donc, la *cellule de diffusion* du sommet v vis-à-vis de T, c'est-à-dire ses voisins, est l'ensemble $\mathrm{Cell}(v) := \{u : \textsc{Sinr}(v,u,T) \geqslant \beta\}$ (cf. la figure 1.7). Le ratio $\textsc{Sinr}(v,u,T)$ est défini par la formule suivante :

$$\textsc{Sinr}(v,u,T) \;:=\; \frac{\mathcal{P}_v \cdot \mathrm{dist}(v,u)^{-\alpha}}{\mathcal{N} + \sum_{w\in T\setminus\{v\}} \mathcal{P}_w \cdot \mathrm{dist}(w,u)^{-\alpha}}\,.$$

Ici $\mathcal{N} \geqslant 0$ (pour *noise*) est le bruit ambiant, le signal minimal reçu lorsqu'il n'y a aucun

émetteur, $\alpha > 0$ est la constante d'atténuation du signal (dépendant du médium) et $\mathcal{P}_w > 0$ est la puissance de l'émetteur placé en $w \in T$.

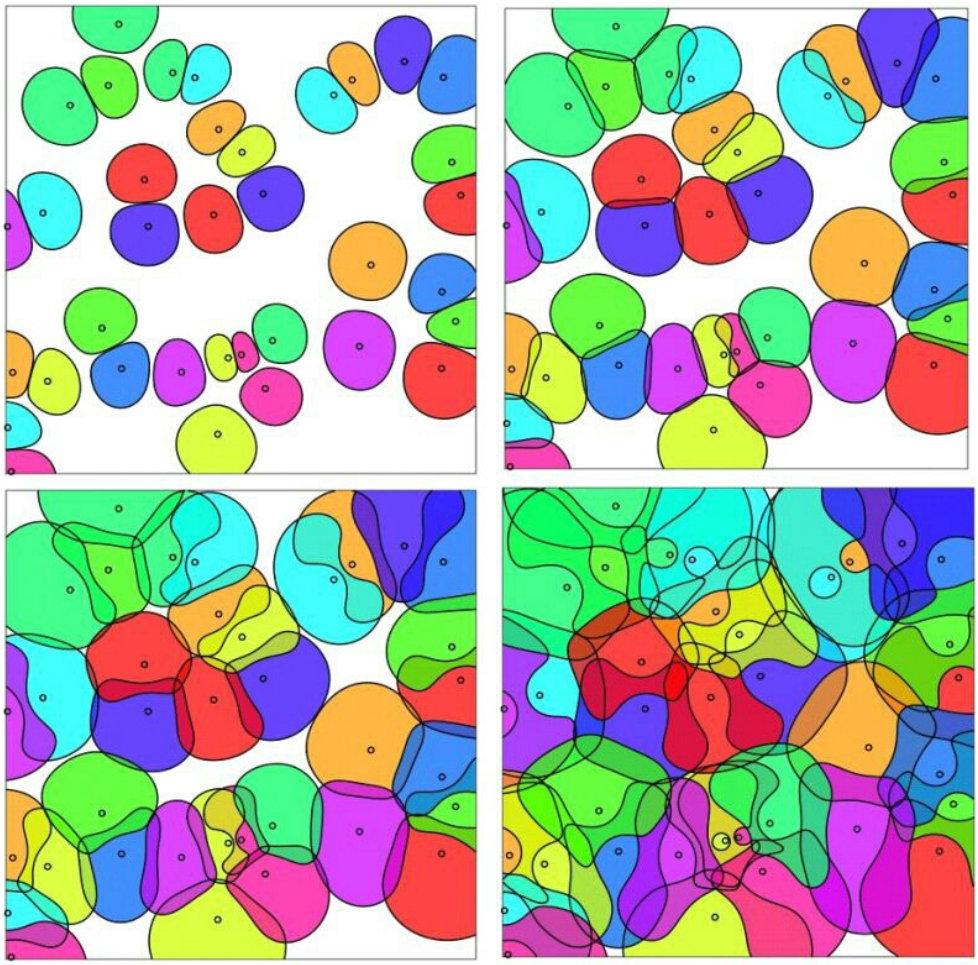


FIGURE 1.7 – Cellules de diffusions (ou couverture) dans le modèle SINR en fonction de la puissance des émetteurs (source Wikipédia).

Contrairement au modèle UDG (cf. figure 1.6) les cellules de diffusion, c'est-à-dire l'ensemble des points u pouvant recevoir un message de v, ne sont pas des disques parfaitement ronds. Un point à l'intersection de plusieurs cellules pourra recevoir de plusieurs émetteurs, tout comme dans les UDG. Un modèle de collision peut alors restreindre ou non les réceptions simultanées.

Lorsque $\beta \geqslant 1$, les cellules de diffusions sont disjointes et laissent apparaître des zones blanches, ce qui n'est pas forcément le cas sinon. Et lorsque les puissances $\mathcal{P}_w$ sont égales (cas uniforme), alors les cellules sont convexes et « épaisses » (ce qui signifie que le ratio entre le cercle inscrit et circonscrit est constant). Dans le cas non uniforme, il est possible que les cellules soient non connexes et même incluses les unes dans les autres (cf. figure 1.8).

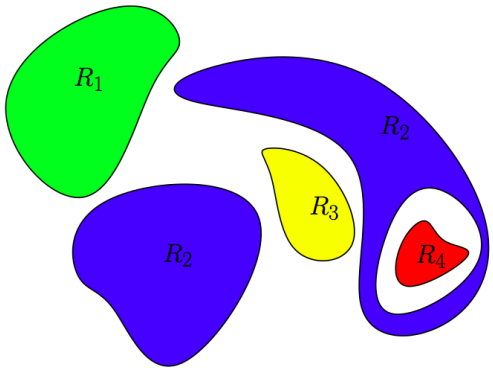


FIGURE 1.8 – Cellules de diffusions dans le cas de puissances non uniformes [KLPP11].

La situation peut devenir particulièrement complexe si les émetteurs se déplacent (selon un certain modèle de mobilité) et/ou les puissances changent au cours du temps.

1.2.5 D'autres modèles

Il existe beaucoup d'autres modèles qu'on ne pourra pas présenter, chacun correspondant à un ensemble spécifique d'hypothèses. Il y a par exemple le *beeping model*, inspiré de systèmes biologiques cellulaires qui communiquent par diffusion de substances chimiques ou par flashs. Il suppose un mode de communication minimaliste du type réseau de diffusion où les processeurs (les cellules donc) ne peuvent émettre qu'un seul type de message : des *beeps* (une certaine molécule chimique). Il suppose également un réveil asynchrone des processeurs et la détection ou pas des collisions [CK10][AAB+11].

La plupart des modèles peuvent aussi être contraints par des critères concernant : la dynamicité de la topologie (ajout/suppression de liens/sommets au cours du temps), la mobilité (le déplacement des émetteurs le long de certaines trajectoires), la tolérance aux pannes [Ray18], la puissance du *scheduler* (i.e., l'entité produisant la variété des scenarios possibles [7]), la détections de collision, ... produisant autant de raffinements et de variantes de ces modèles.

Bien sûr, le choix du modèle, qui peut être plus ou moins fin pour calquer au mieux la réalité, a un impact très important sur la conception des algorithmes distribués. En général, ce qu'on attend d'un modèle, c'est de pouvoir (pré)dire : *si* un système vérifie telle ou telle propriété, *alors* tel algorithme produira une solution avec telle ou telle performance ou qualité. Que le système vérifie réellement les hypothèses (ou jusqu'à quel point) est un autre sujet d'étude : l'*expérimentation*, qui laisse quant à elle une large part aux statistiques. Et, ne nous trompons pas, c'est une science à part entière (avec ses dérives comme la *p-value*, cf. [WSL19]).

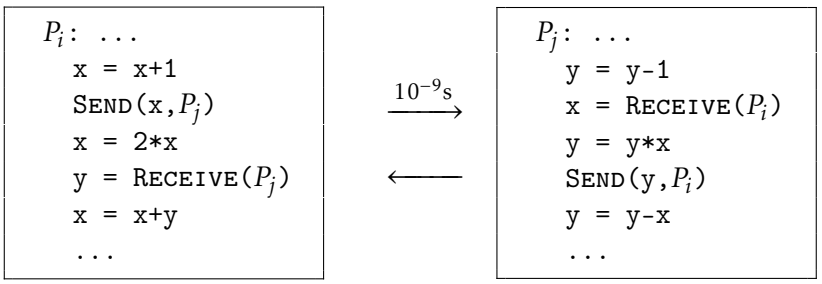
7. On en reparlera page 24 et page 31.

Il y a malheureusement un compromis entre la fidélité du modèle à la réalité et son *explicabilité*, c'est-à-dire son pouvoir prédictif et sa capacité à expliquer les résultats. Plus un modèle est simple, c'est-à-dire moins il contient d'hypothèses ou de paramètres, plus grands sont ses pouvoirs prédictif et explicatif permettant de dégager des principes généraux, asymptotiques. La contrepartie est qu'un modèle trop simple est souvent éloigné de la réalité et risque de produire au final des prédictions, certes fines, mais peu réalistes. Inversement, un modèle se voulant fidèle à la réalité risque d'être trop complexe pour être analysé autrement que par des simulations. Les simulations fines, qui peuvent avoir leur intérêts, reviennent à rejouer la réalité sans comprendre les comportements limites et asymptotiques (car dans la réalité et les simulations, tout est fini !). Dans ce cas l'explicabilité du modèle est faible.

1.3 Spécificités du calcul distribué

1.3.1 Les communications

Elles ne sont pas gratuites ! Pour s'en convaincre, considérons un système où les processeurs synchrones fonctionnent à une fréquence d'horloge de 1 GHz = 10^{-9} s, soit une cadence de 1 millard de cycles (ou instructions) par seconde. Si les processeurs souhaitent faire aussi des SEND et RECEIVE à cette cadence comme ci-dessous, alors la distance entre deux processeurs P_i et P_j est au plus $10^{-9} \times c$, où $c < 300\,000$ km/s = $300\,000\,000$ m/s = 3×10^8 m/s est la vitesse de la lumière dans le vide [8]. Cela fait moins de $10^{-9} \times 3 \times 10^8 = 0.3$ m = 30 cm.



Lorsque le nombre de processeurs dépasse l'ordre du millier, il est difficile de tous les mettre dans une même boîte de 30 cm de diamètre, surtout que dans la pratique, les vitesses de transmission sont assez loin d'être réalisée à la vitesse de la lumière. L'information est transportée à environ 175 200 km/s dans le cuivre (soit $0.58c$) et les conversions opto-électroniques ne sont pas instantanées.

Si les processeurs sont maintenant équidistant de 3 km la fréquence des processeurs doit être alors diminuée d'un facteur 3km/30cm = 10 000. Dit autrement, à fréquence

8. Pour être précis, $c = 299\,792\,458$ m/s par définition depuis 1983.

égale, des processeurs distant de 3 km peuvent effectuer au moins 10 000 instructions avant de pouvoir interagir. À cette distance, les communications coûtent 10 000 fois plus chère que les calculs locaux.

On supposera donc que le calcul local est négligeable devant les communications. Notons que dans le modèle PRAM, c'est évidemment le contraire. Toutes les communications prennent un temps supposé constant, et donc le coût réside dans les calculs des processeurs et donc dans la parallélisation de ces calculs. Pour schématiser, le parallélisme néglige les communications, alors que le distribué néglige les calculs.

1.3.2 Connaissance partielle

Un processeur doit faire son calcul avec une connaissance limité du système. Dans un système centralisé (machine séquentielle) l'état de la mémoire est déterminé par le calcul du processeur unique. En particulier, cet état ne change pas si le processeur ne le modifie pas.

En distribué, le processeur P_i ne contrôle que son état local. Les données réparties sur d'autres processeurs changent sans aucun contrôle direct de P_i.

La *connaissance a priori* est l'état dans lequel se réveillent les processeurs. Parfois cette connaissance comprend la topologie du graphe. Les processeurs peuvent savoir, par exemple, qu'ils appartiennent à une clique de n processeurs car n est connu des processeurs et l'algorithme est spécifique à cette topologie. Ils pourraient connaître un arbre couvrant et l'exploiter efficacement comme on le verra dans le chapitre 3 concernant la diffusion. La connaissance du diamètre (d'un majorant ou d'un minorant) de la topologie, ou encore le *hop-count* maximum, est une connaissance importante en pratique qui permet de résoudre le problème suivant.

Count-to-infinity. Ce problème se produit dans le protocole de mise à jour du vecteur de distance (*distance-vector routing protocol*) où chaque nœud cherche à déterminer la meilleure route vers tous les autres, un ingrédient important du protocole BGP d'Internet. Pour cela, à chaque changement topologique qu'il détecte (lien vers un voisin qui tombe en panne), le nœud diffuse à ses voisins une mise à jour de ces nouvelles distances.

Pour router les messages selon les meilleures routes, les sommets se basent sur ce vecteur de distance [9] mais aussi sur d'autres informations de routage (tables de routage et autres mesures sur les routes) dont ils sont propriétaires et qu'ils ne transmettrent pas. Par exemple, si un nœud C, qui pense être à distance d_0 de A, apprend de son voisin B que A est en fait à distance d_1 de B, alors C sera en mesure de mettre à jour sa distance à A (et de ses tables de routage) si $1 + d_1 < d_0$: car en passant par B, C pourra atteindre

9. La distance correspond souvent au nombre de *hops*, soit le nombre minimum d'arêtes d'un chemin connectant deux nœuds.

A plus rapidement. Donc, si $1 + d_1 < d_0$, C diffusera à ses voisins son nouveau vecteur de distance. Et de proche en proche les routes pourront se raccourcir.

Le problème *count-to-infinity* apparaît lorsque des liens tombent en pannes et que le protocole de mise à jour fait augmenter les distances infiniment. Par exemple, considérons qu'une partie du réseau contienne un chemin

$$\cdots - A \not- B - C - \cdots$$

où s'effectue régulièrement des mises à jour des vecteurs de distance. Si le lien $A - B$ est momentanément inactif [10], B va mettre à jour sa table pour indiquer qu'il ne sait plus aller en A (distance vers A mise à $+\infty$). Au même moment C était en train de mettre à jour sa table et pense que A est à distance 2 de lui (car, avant la panne, B lui avait indiqué une mise à jour comme quoi A était à 1 *hop* de B). Mise à jour qu'il diffuse à ses voisins dont B. B apprend de C qu'une nouvelle mise à jour vient d'avoir lieu, notamment pour A. Ainsi B pense que C est maintenant à 2 *hops* de A. B met donc à jour sa distance vers A qui devient 3. Mise à jour qu'il diffuse à tous ces voisins dont C. Pendant ce temps C a reçu de B la valeur $+\infty$ pour A depuis B, ce qui pose un problème à C qui route vers B pour aller vers A. Il passe donc sa distance à A à $+\infty$ qu'il transmet immédiatement à B. Mais C reçoit plus tard une mise à jour de B qui lui dit que A est finalement à distance 3 de A (via C, mais C l'ignore puisque seules les distances transitent). Donc C met à jour son vecteur à 4 pour A et diffuse son vecteur vers B. Le sommet B va recevoir une valeur $+\infty$ de C pour A, qu'il rediffusera à C, mais c'est trop tard car B a déjà envoyé 3 à C et va bientôt de surcroît recevoir 4 de C. Il pensera donc que C a finalement réussit à découvrir une nouvelle route vers A, de 4 *hops*, et donc B passera sa distance à 5, qu'il diffusera vers C, etc.

Le problème paraît ridicule étant donné que B et C sont voisins. On se dit qu'ils pourraient se coordonner peut être un peu mieux pour éviter une boucle infinie entre voisins. Malheureusement le même genre de boucle pourrait apparaître sur une ou plusieurs chaînes de routeurs entre B et C, chaînes qui pourraient à leur tour varier dans le temps, ce qui rend difficile la détection de tels phénomènes. (Par exemple, A, B, C pourraient être trois sommets espacés d'une grille où beaucoup de chemins $A - B$, $B - C$ et $A - C$, co-existent.)

En pratique, la plupart des protocoles limite à 16 le nombre de *hops* maximal dans un vecteur de distance. Bien sûr, en toute généralité, il faudrait connaître un majorant sur le nombre de *hops* maximal du réseau.

Anonymité. Dans certains cas, on supposera que les processeurs connaissent d'autres paramètres de la topologie : son degré maximum (pour savoir combien de messages maximum il est censé recevoir en une ronde), sa dimension (pour un graphe géométrique issu de points de $\mathbb{R}^d$), ... Mais, le plus souvent le processeur ne connaît que

10. Dans le protocole, un nœud est censé échanger régulièrement de petits messages afin de détecter les pannes entre voisins.

lui-même, c'est-à-dire son identité s'il en a une, et le programme qu'il doit exécuter. Notons qu'en pratique, le même programme est copié sur tous les processeurs du système. Mettre un programme différent à chacun des processeurs revient à leur donner une identité. Dans ce cas, il est plus simple de leur affecter une identité puis d'écrire un seul et même programme comprenant éventuellement des alternatives suivant l'identité des processeurs. Le maintient et les mises à jour du programme peuvent alors être réalisées par simple diffusion.

Système (ou réseau) anonyme

> Tous les processeurs ont exactement la même connaissance *a priori*. Ils ne possèdent pas d'identité, seulement une copie du même programme. Ils se réveillent donc dans le même état.

Une tâche impossible à résoudre dans un système anonyme est celle de Leader Election, qui, plus généralement, fait partie des problèmes de « brisure de symétrie ». Il s'agit de décider qui sera le maître et qui sera les esclaves (voir figure 1.9), sachant qu'il ne doit avoir qu'un seul maître.

Figure 1.9 – Dans le problème Leader Election, il s'agit de décider lequel des sommets est le maître. En l'absence d'identifiant, aucun algorithme distribué déterministe ne peut résoudre le problème pour ce graphe. [*Question.* Pourquoi ?]

1.3.3 Erreurs, pannes et défaillances

C'est simple dans le cas séquentiel : si un élément est défectueux (mémoire, processeur, disque-dur, ...) on le change, et on continue ou on recommence le calcul. En distribué, et en l'absence de tout contrôle centralisé, la détection même d'une défaillance peut être difficile. Comment détecter qu'un lien est en panne ou seulement très lent en l'absence d'horloges synchrones ? Un lien défectueux peut causer de grave problème (omission, ajout ou corruption de messages). D'autre part, redémarrer tout un protocole sur un système entier peut simplement être infaisable, comme le cas du routage dans Internet. Le redémarrage de tout le système peut être évité en rendant robuste l'algorithme. C'est le concept de tolérance aux pannes où le programme peut, même avec un (voir plusieurs) lien ou processeur en panne dans le réseau, encore fonctionner peut-être avec des performances moindres.

1.3.4 Synchronie

La synchronie est un élément impactant fortement le fonctionnement d'un système distribué. On considère deux fonctionnements extrêmes : synchrone ou asynchrone.

Mode synchrone

> Les tops de l'horloge[11] interne (donc locale) de chacun des processeurs vérifient la propriété suivante : un message envoyé au top t de P_i vers son voisin P_j est reçu[12] au top t de P_j, et ne peut pas dépendre d'autres messages envoyés à un top $\geqslant t$.

On parle souvent de « ronde » (*rounds* en Anglais) pour designer la période de temps où le compteur de l'horloge à la même valeur. Donc la propriété est que les messages émis à une ronde donnée sont reçus par leurs voisins pendant la même ronde[13]. De plus, il n'est pas possible de répondre à un message reçu lors de la même ronde. La réponse devra être émise à une ronde ultérieure.

En quelque sorte le délais de transmission de tous les liens est borné par une valeur connue de tous : la durée d'une ronde. Avec cette définition, rien ne dit que les horloges soient parfaitement synchrones, qu'elles incrémentent leurs tops au même moment. Mais c'est tout comme, car si au top t on demande à son (ou ses) voisin(s) :

> « Quelle heure as-tu ? (à réception de ce message) »

Invariablement on recevra « t » comme réponse, réponse reçue au cours du top $t+1$. Et tous les voisins interrogés répondront la même chose. Ce qui est important ici c'est bien la stabilité de la réponse en temps et en espace : il n'y aura pas de décalage au cours du temps et ce quel que soit le ou les processeurs considérés.

Notons que puisque la notion de simultanéité dépend de l'observateur (cf. figure 1.10), la définition doit être locale. La mauvaise définition serait de dire qu'en synchrone « toutes les horloges sont synchrones, c'est-à-dire changent de top horloge simultanément ».

L'exécution de chaque programme est alors dirigée par les tops horloges ou les rondes. Et entre chaque top successif on exécute un certain cycle d'instructions qu'on appelle aussi *ronde*. Typiquement un programme en mode synchrone ressemblera à :

```
P_i :
  au top 1, faire ronde_1
  au top 2, faire ronde_2
  ...
```

où $\texttt{ronde}_t$ est un cycle Send/Receive/Compute. Plus précisément au top t :

11. Les « tops horloge » sont les valeurs de l'horloge qui est vue comme un compteur croissant d'entiers naturels. L'expression « au top t » désigne un moment où ce compteur vaut t.
12. S'il n'y a pas de panne.
13. On parle parfois de ronde *close*.

```
ronde_t :
  1. SEND à zéro, un ou plusieurs de ses voisins
  2. RECEIVE de zéro, un ou plusieurs de ses voisins
  3. calculs locaux sur les messages reçus
```

Le cycle s'exécute complètement en une ronde quels que soient les calculs locaux, y compris une boucle du type : « pour chaque voisin P_j de P_i, faire SEND(M, P_j) ». Vis-à-vis des communications, les calculs locaux sont gratuits (comme si la boucle ci-dessus sur les voisins prenait un temps nul). Les tops horloge égrainent les rondes de communications, pas les calculs qui, comme on l'a vu dans le paragraphe 1.3.1, sont presque infiniment plus rapides.

Il doit être clair que tous les SEND et RECEIVE d'un même cycle ne peuvent concerner qu'une seule et même ronde. Un RECEIVE à la ronde t ne peut recevoir d'un voisin qu'un message envoyé par un SEND à la ronde t. Comme les messages émis à la ronde t doivent arriver avant le passage à la ronde $t+1$, le RECEIVE est bloquant jusqu'à la toute fin de la ronde t. Et donc la partie COMPUTE s'exécutera de manière instantanée (vis-à-vis des tops horloge) entre la fin de réception des RECEIVE et le passage à la ronde $t+1$. Bien sûr, avant un SEND, il peut y avoir quelques calculs (comme préparer le message). Mais, dans ce cas, ces calculs ne peuvent pas dépendre des messages qui seront reçus lors de cette même ronde.

Pour résumer, lors de chaque ronde, un processeur échange des messages avec un ou plusieurs de ses voisins (éventuellement avec aucun voisin), le message émis pouvant être différencié selon le voisin, puis effectue un calcul local avec les messages éventuellement reçus lors de cette même ronde.

Il n'est pas conseillé de mélanger dans une même ronde les SEND et les RECEIVE, car le programme résultant ne fera pas nécessairement ce qui est attendu. Le code ci-dessous est donc à éviter car sa sémantique est ambigüe [14].

```
SEND(x,P_0)
y = RECEIVE(P_1)
SEND(x+y,P_2)
```

Il y a deux interprétations possibles. Les trois instructions s'exécutent lors d'une même ronde, et alors le y dans le second SEND$(x+y, P_2)$ n'est celui reçu de P_1, car un message ne peut pas dépendre d'un autre autre message reçu lors d'une même ronde. Le second SEND devrait donc préférablement être écrit juste après le premier. Ou alors le second SEND est réalisé lors d'une autre ronde où le y dépend du RECEIVE(P_1).

Pour clarifier on ajoutera parfois une instruction NEWROUND pour délimiter chacune des rondes, et indiquer leur début. Cette instruction détruit tous les messages reçus à

14. Tout comme le serait une instruction C du type : `i++ += i++;`

la ronde précédente. C'est une histoire de conventions. De manière inhérente, en synchrone, le programme de chaque processeur effectue une suite :

$$.../\textsc{Send}/\textsc{Receive}/\textsc{Compute}/\textsc{Send}/\textsc{Receive}/\textsc{Compute}/...$$
$$\overbrace{\longleftarrow}^{\texttt{ronde}_t}\ \overbrace{\longleftarrow}^{\texttt{ronde}_{t+1}}$$

et donc la convention consiste à se mettre d'accord sur le découpage de cette suite en rondes. Pour certains langages ou simulateurs comme JBotSim, la convention est de commencer une ronde par Receive, et donc de découper en Receive/Compute/Send. Du coup la sémantique est plutôt de dire que le Receive récupère les messages du Send de la ronde précédente. Il devient nécessaire de réécrire en conséquence la définition d'horloges synchrones vues précédemment. De plus, la toute première ronde n'est pas bien définie.

Remarque sur JBotSim. Considérons l'algorithme suivant (à gauche), comprenant $n \geqslant 1$ rondes, où A_i, B_i, $\textsc{Compute}_i$ sont des expressions quelconques éventuellement dépendantes d'un indice de boucle $i \in \{0,\dots,n-1\}$. Pour l'écrire en JBotSim (à gauche), cela se transformerait ainsi (`OnClock()` étant l'équivalent de NewRound et `OnStart()` l'équivalent du premier NewRound) :

```
Pour i = 0 à n−1 :
  NewRound
  Send(A_i)
  Receive(B_i)
  Compute_i
```

```
OnStart(){
  i = 0;
  Send(A_i);
}
OnClock(){
  if(i < n){
    Receive(B_i);
    Compute_i;
    if(++i < n) Send(A_i);
  }
}
```

Il y a donc une propension en JBotSim à répéter les instructions (un doublement en fait) à cause du début et de la fin des rondes qui ne sont pas définies de la même manière.

Quelle que soit la convention on s'attend toujours à ce que les Receive correspondent à des Send écrits précédemment, expliquant l'habitude d'écrire dans les algorithmes plutôt les Send avant les Receive.

Mode asynchrone

> Les tops horloges de processeurs voisins n'ont plus aucune propriété de synchronisation si bien que leur utilisation n'est plus d'aucun recours. Les messages envoyés de P_i vers un voisin P_j arrivent en un temps fini[12] mais imprédictible.

L'exécution de chaque programme est alors dirigée par les évènements correspondant à la réception de message, et ressemble à :

```
P_i :
  M = RECEIVE() // attendre de recevoir un message M
  selon M faire:
    cas 1: Action_1
    cas 2: Action_2
    ...
```

où $\texttt{Action}_k$ comprend des SEND et des COMPUTE (c'est-à-dire des calculs locaux) suivant le message ou le type du message M reçu. Au départ, pour que le système ne reste pas bloqué perpétuellement en réception, on supposera qu'un processeur qui se réveille, spontanément de lui-même ou par un mécanisme extérieur, le fait grâce à l'émission d'un message particulier, disons « START », permettant l'exécution d'une première action.

Une façon unifiée est aussi de dire qu'en mode synchrone les évènements sont déclenchés par les tops horloges, alors qu'en mode asynchrone ils le sont par la réception des messages. Une différence notable aussi est qu'en mode asynchrone, la réception de message (RECEIVE) est bloquante. S'il n'y a pas de panne, la durée d'attente est finie mais imprédictible. Dans le cas synchrone, les RECEIVE ne sont bloquants que pendant la durée d'une ronde. Les messages envoyés (SEND) à une ronde donnée et non traités (c'est-à-dire non consommés par un RECEIVE correspondant) sont simplement détruits.

Si les programmes s'écrivent parfois de manière plus concise en asynchrone qu'en synchrone, en pratique, il est beaucoup plus difficile de mettre au point un programme écrit pour le mode asynchrone, car on ne sait jamais vraiment dans quel ordre vont s'effectuer la réception des messages (et du coup dans quelle ordre vont s'effectuer les calculs correspondants). Évidemment, en mode synchrone, après la ronde t vient nécessairement la ronde $t+1$, et on contrôle de manière beaucoup plus sûre la succession des calculs. En fait, on verra au chapitre 6 qu'il existe des méthodes automatiques, appelées *synchroniseurs*, pour exécuter un code écrit en mode synchrone et l'exécuter sur un système asynchrone.

Comme toujours, la réalité se situe entre ces deux modes. Les réseaux (surtout ceux de grande taille) sont fondamentalement asynchrones. Cependant ils possèdent aussi des horloges permettant l'utilisation de *time-out*. Lorsqu'on attend un accusé de réception par exemple, le *time-out* permet de trancher en prenant une décision avant de connaître la réalité de la situation, quitte à ré-émettre un message inutilement. Malheu-

reusement, dans l'absolu en mode asynchrone, on ne peut pas faire la différence entre un lien très lent et un lien en panne.

Il existe des modèles intermédiaires où l'on suppose connu les délais minimum et maximum de transmission des liens, voir de la dérive des horloges internes. En connaissant le délais maximal (ou un majorant), on peut se ramener au cas synchrone, puisqu'il suffit alors d'attendre ce délais pour démarrer la ronde suivante. Notons qu'une borne sur le délais minimum de transmission est utile dans certaines situations : P_i transmet sa nouvelle table de routage à son voisin P_j, qui est censé lui retransmettre un accusé de réception (voir sa nouvelle table de routage tenant compte de celle de P_i). Si P_i reçoit peu de temps plus tard cet accusé de la part de P_j, s'agit-il du bon accusé, ou est-ce celui d'une réception plus ancienne ? Les notions de passé et de futur sont malheureusement locales à chaque processeur en l'absence d'horloge parfaitement synchrone.

D'ailleurs la théorie de la relativité générale d'Einstein prédit qu'il est impossible de synchroniser deux horloges suffisamment distantes sans que l'observateur nécessite d'avoir de nombreuses informations sur le champ de gravitation et sur son évolution. Cf. cet article sur Wikipédia dont voici un extrait :

> *[...] Vouloir synchroniser les horloges d'un référentiel en relativité générale, sans plus de précision, revient à vouloir synchroniser des horloges réparties dans un référentiel quelconque par la même méthode qu'en relativité restreinte. Dans le cas général, ce n'est réalisable que dans un volume infiniment petit ou le long d'une courbe ouverte, mais pas le long d'une courbe fermée car un tour complet de cette courbe induit une différence de synchronisation de la même manière que le transport parallèle le long d'une boucle fermée induit un déplacement du vecteur tangent initial.*

En particulier, la notion de simultanéité (dans le cadre de la relativité générale) n'est pas transitive : si E_a et E_b sont des événements simultanés, et que E_b et E_c aussi, alors E_a et E_c ne le sont pas forcément. Dès qu'on admet que la vitesse de propagation de l'information est finie (quels que soient les mouvements des référentiels), la notion de simultanéité (et donc de synchronisation) n'a plus de sens (cf. figure 1.10). Le temps est une notion *a priori* locale.

Pour finir, notons qu'Einstein avait définit de manière pragmatique dans [Ein05, p. 894] la vitesse de la lumière dans le vide (en fait $1/c$) comme le temps (supposé constant) mis par la lumière pour faire un aller-retour d'une longueur unité. Il avait précisément écrit :

$$\frac{2AB}{t'_A - t_A} = c$$

Wir setzen noch der Erfahrung gemäß fest, daß die Größe

$$\frac{2\overline{AB}}{t'_A - t_A} = V$$

eine universelle Konstante (die Lichtgeschwindigkeit im leeren Raume) sei.

À cause de l'impossibilité de synchroniser deux horloges distantes, il n'est pas possible de définir la vitesse de la lumière (ou de la transmission de l'information) au-

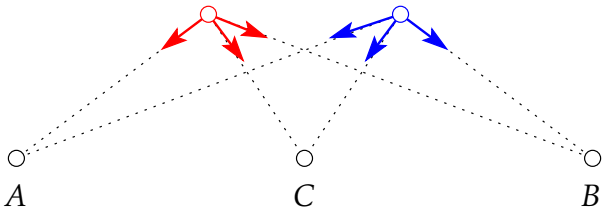


FIGURE 1.10 – Pour C, les flashs lumineux rouge et bleu se produisent simultanément. Pour A le rouge se produit avant le bleu, alors que pour B c'est le contraire. Si l'on remplace les deux sources par deux horloges, on voit qu'on ne peut pas vérifier de manière absolue la synchronie d'horloges distantes sans informations complémentaires. Pire encore : des observateurs en mouvement de A vers B ou de B vers A auraient des conclusions différentes.

trement qu'avec un aller-retour ou plus généralement une boucle [15]. La bonne nouvelle est qu'aucune expérience ne peut distinguer la situation où cette vitesse est la même pour l'aller et le retour de la situation où la vitesse n'est pas homogène sur le parcourt. Ouf ! Ce n'est donc pas grave en pratique si en réalité la lumière n'a pas une vitesse homogène.

1.3.5 Non déterminisme

Un algorithme déterministe a la propriété de fournir toujours le même résultat pour une même entrée. L'exécution est donc toujours la même si les entrées sont les mêmes. Le résultat est ainsi entièrement déterminé par les entrées et l'algorithme. En séquentiel, le non déterministe est lié à la présence d'instruction comme [16] `random()` dont la valeur varie à chaque nouvelle exécution. Cette valeur est en principe non prévisible.

En calcul distribué, le résultat d'une exécution peut dépendre aussi du système sur lequel s'exécute l'algorithme, même en l'absence d'instruction `random()`. En effet, la vitesse les messages circulant sur les liens n'est pas contrôlable par l'algorithme. Évidemment cette source d'indétermination liée au réseau (et plus généralement au médium de communication) n'existe pas en séquentiel (ici le médium de communication est interne à la machine et donc sous contrôle). L'exécution particulière d'un algorithme s'appelle un *scénario*. Bien sûr, dans cette discussion, on ne considère que les systèmes non défaillant, puisque sinon, même en séquentiel, le résultat peut-être non déterminé.

Typiquement, en mode asynchrone, un processeur P_0 pourrait recevoir deux messages différents de deux de ses voisins, disons P_1 et P_2, et décider d'afficher le premier message reçu. On ne pourra prédire si c'est celui de P_1 ou de P_2 (cf. figure 1.11).

Dans le mode synchrone, la situation est différente. Dans la mesure où les SEND

15. Pour plus sur le sujet, voir « *One-way speed of light* » de Wikipédia.
16. Pour des valeurs vraiment aléatoires en `C` il faut utiliser `arc4random()` de la librairie standard.

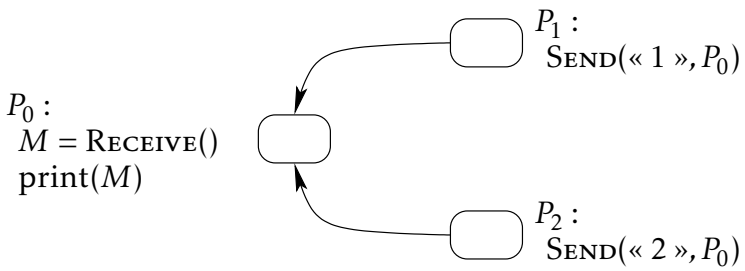


Figure 1.11 – P_0 affiche « 1 » ou « 2 » de manière non déterminée. En mode asynchrone, chaque exécution est potentiellement un nouveau scénario même si les entrées sont les mêmes.

précèdent toujours tous les Receive d'une même ronde, un tel non déterminisme ne peut se produire. Dans l'exemple, on ne sait peut-être pas à l'avance si le message M reçu est de P_1 ou P_2, mais le premier message reçu sera toujours le même, à chaque exécution (synchrone). Les deux messages sont reçus en même temps (il n'y a pas de décalage lié au médium) et traités dans un certain ordre par le Receive(), et ce indépendamment du médium. À vrai dire, en synchrone, on devrait plutôt écrire :

```
P_0 :
  pour chaque voisin P_i, i = 1,2, faire:
    M = Receive(P_i)
    print(M)
```

Et donc le choix d'affichage correspond en fait au choix de faire $i = 1$ ou $i = 2$ en premier dans la boucle `pour`.

Parfois, en mode asynchrone, on souhaite imposer quelques contraintes sur la variabilité des scénarios, notamment celles liées à l'hétérogénéité des vitesses d'exécution des processeurs. On parle de *scheduler*, un élément du modèle qui est virtuellement [17] en charge du séquencement de l'envois des messages, de la vitesse de transmission des messages et de l'exécution des processeurs. Il y a différentes hypothèses, comme l'hypothèse d'un *scheduler* équitable qui ne peut pas bloquer indéfiniment un processeur donné. On en reparlera au chapitre suivant, paragraphe 2.1 page 31.

Bibliographie

[AAB+11] Y. Afek, N. Alon, O. Barad, E. Hornstein, N. Barkai, and Z. Bar-Joseph, *A biological solution to a fundamental distributed computing problem*, Science,

17. Bien sûr en réalité la *scheduler* n'existe pas. C'est une façon de modéliser les aléas du médium et d'exécutiuon des processeurs.

331 (2011), pp. 183–185. DOI : 10.1126/science.1193210.

[ABND95] H. Attiya, A. Bar-Noy, and D. Dolev, *Sharing memory robustly in message-passing systems*, Journal of the ACM, 42 (1995), pp. 124–142. DOI : 10.1145/200836.200869.

[CK10] A. Cornejo and F. Kuhn, *Deploying wireless networks with beeps*, in 24th International Symposium on Distributed Computing (DISC), vol. 6343 of Lecture Notes in Computer Science, Springer, September 2010, pp. 148–162. DOI : 10.1007/978-3-642-15763-9_15.

[DLP+86] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, *Reaching approximate agreement in the presence of faults*, Journal of the ACM, 33 (1986), pp. 499–516. DOI : 10.1145/5925.5931.

[Ein05] A. Einstein, *Zur Elektrodynamik bewegter Körper*, Annalen der Physik, 322 (1905), pp. 891–921. DOI : 10.1002/andp.19053221004.

[FK12] O. Feinerman and A. Korman, *Memory lower bounds for randomized collaborative search and implications for biology*, in 26th International Symposium on Distributed Computing (DISC), vol. 7611 of Lecture Notes in Computer Science, Springer, October 2012, pp. 61–75. DOI : 10.1007/978-3-642-33651-5_5.

[FK13] O. Feinerman and A. Korman, *Theoretical distributed computing meets biology : A review*, in 9th International Conference on Distributed Computing and Internet Technology (ICDCIT), vol. 7753 of Lecture Notes in Computer Science, Springer, February 2013, pp. 1–18. DOI : 10.1007/978-3-642-36071-8_1.

[FLP85] M. J. Fischer, N. A. Lynch, and M. Paterson, *Impossibility of distributed consensus with one faulty process*, Journal of the ACM, 32 (1985), pp. 374–382. DOI : 10.1145/3149.214121.

[GP16] E. Godard and É. Perdereau, *k-set agreement in communication networks with omission faults*, in 20th International Conference on Principles of Distributed Systems (OPODIS), vol. 70 of LIPIcs, December 2016, pp. 8 :1–8 :17. DOI : 10.4230/LIPIcs.OPODIS.2016.8.

[Gra78] J. N. Gray, *Notes on data base operating systems*, in Operating Systems, vol. 60 of Lecture Notes in Computer Science, Springer, January 1978, pp. 393–481. DOI : 10.1007/3-540-08755-9_9.

[HM20] H. Howard and R. Mortier, *Paxos vs Raft : Have we reached consensus on distributed consensus ?*, Tech. Rep. 2004.05074v2 [cs.DC], arXiv, April 2020.

[KLPP11] E. Kantor, Z. Lotker, M. Parter, and D. Peleg, *The topology of wireless communication*, Tech. Rep. 1103.4566v2 [cs.DS], arXiv, March 2011.

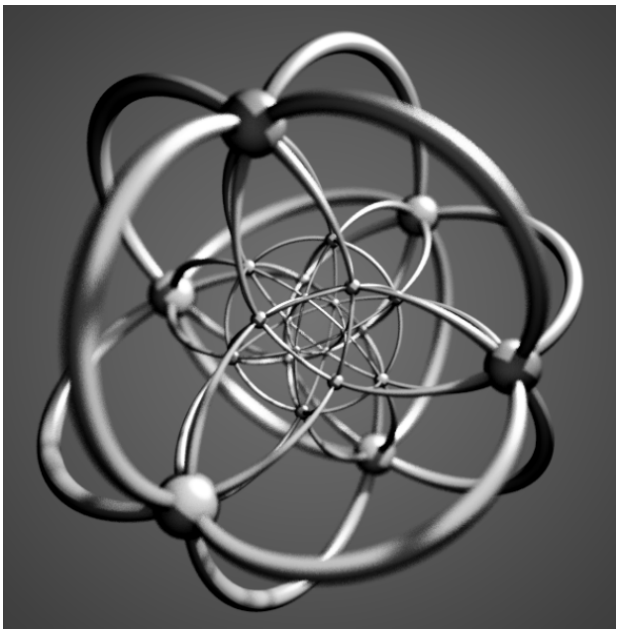
[Ray18] M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems : An Algorithmic Approach*, Springer, 2018. ISBN : 978-3-319-94141-7. DOI : 10.1007/978-3-319-94141-7.

[Tau98] G. Taubenfeld, *Distributed Computing Pearls*, vol. #14 of Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool, May 1998. doi : 10.2200/S00845ED1V01Y201804DCT014.

[WSL19] R. L. Wasserstein, A. L. Schirm, and N. A. Lazar, *Moving to a world beyond "$p < 0.05$"*, The American Statistician, 73 (2019), pp. 1–19. doi : 10.1080/00031305.2019.1583913.

CHAPITRE

2 Complexités et modèles



Sommaire

À partir de maintenant on suppose un mode de communication point-à-point modélisé par un graphe connexe et simple, *a priori* non orienté. On peut communiquer de manière fiable, c'est-à-dire sans panne, directement entre deux entités si elles sont connectées par une arête du graphe. Les messages envoyés finissent tous par arriver

(dans le même ordre d'émission !). Le graphe est statique, c'est-à-dire qu'il n'évolue pas en fonction du temps.

Mots clés et notions abordées dans ce chapitre :

- temps synchrone, temps asynchrone, nombre de messages,
- les modèles LOCAL, CONGEST, ASYNC,
- rappels de base sur les graphes.

2.1 Mesures de complexité

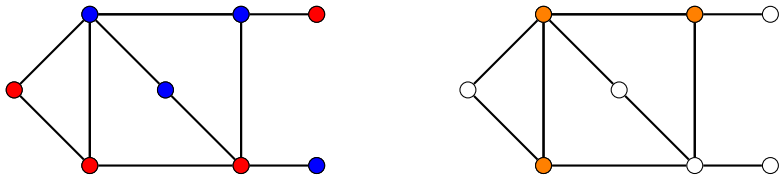
En séquentiel, on utilise surtout la complexité en temps, plus rarement celle en espace bien qu'en pratique l'espace se révèle bien plus limitant que le temps. Pour un programme qui prend un peu trop de temps, il suffit d'être patient. Un programme qui prend un peu trop d'espace doit être réécrit ou alors il faut changer le *hardware*. C'est donc plus contraignant en pratique, dans une certaine mesure bien évidemment.

En distribué, les notions de complexités sont plus subtiles. Les notions de nombre de messages échangés ou de volume de communication font leur apparition. On utilisera essentiellement les complexités en temps et en nombre de messages. Ces complexités dépendent, le plus souvent, du graphe sur lequel s'exécute l'algorithme. Et comme on va le voir, les définitions diffèrent aussi suivant que l'on considère un système synchrone ou asynchrone.

En distribué, les entrées (représentant l'instance du problème) et les sorties d'un algorithme (ou d'un programme) sont stockées sur les sommets du graphe. C'est donc un peu comme en séquentiel sauf qu'entrées et sorties sont distribuées sur le graphe lui-même (cf. figure 2.1).

En séquentiel, la complexité en temps d'un algorithme est lié à sa durée d'exécution dans le pire des cas. Plus précisément, c'est le nombre opérations élémentaires utilisées pendant son exécution. En distribué, il faut revisiter les notions de « durée d'exécution » et « d'opérations élémentaires ». Pour la durée d'exécution, il est tentant de la définir comme la durée entre le démarrage du premier processeur de G exécutant l'algorithme distribué A et l'arrêt du dernier processeur. Et pour les opérations élémentaires, il est tentant de simplement rajouter Send et Receive à la liste. Il y a cependant des cas où A ne termine pas alors que plus aucun message ne circulent et circuleront. C'est le cas où plus aucun message ne sont produit, alors que certains processeurs continuent à faire des calculs locaux. Cependant, les calculs locaux qui ne sont pas suivis d'emission/réception de messages ne sont ni plus ni moins que des calculs séquentiels classiques.

À cause du coût des communications (cf. le paragraphe 1.3.1), on ne retient comme opérations élémentaires seulement Send et Receive. Les autres opérations sont donc considérées comme ayant un coût nul. La *durée de l'algorithme* est alors défini comme la durée entre le moment de la première emission (Send) et la dernière réception



(a) [entrées] (b) [sorties]

FIGURE 2.1 – Exemple de problème d'algorithmique distribuée pour un graphe G.

(a) Entrées ($\forall u \in V(G)$) : une marque rouge ou bleue.

(b) Sorties ($\forall u \in V(G)$) : un état sélectionné ou non (orange ou blanc) de sorte que tout sommet initialement rouge possède au moins un voisin sélectionné.

La question est alors de trouver un algorithme distribué, s'il en existe un, permettant de résoudre ce problème. [*Question*. Donnez des conditions sur G (voir sur les entrées) pour que le problème ait toujours au moins une solution.]

(RECEIVE). Notons au passage qu'il est possible que des messages transitent sur des arêtes après l'arrêt du dernier processeur, la dernière instruction pouvant être un SEND sans RECEIVE correspondant. Il est aussi possible qu'un processeur soit en attente de message (RECEIVE aynchrone) alors que plus aucun message ne soit généré. Dans ces deux cas, on considère l'algorithme comme achévé (vis-à-vis des opérations élémentaires SEND et RECEIVE).

Un message émis puis reçu, c'est-à-dire ayant un RECEIVE correspond au SEND, est dit *consommé*. Et une ronde consommée est une ronde pendant laquelle au moins un message est consommé. Généralement, lorsqu'on conçoit un algorithme distribué, on fait en sorte que sa dernière ronde soit consommée. En effet, un SEND sans RECEIVE correspondant ne pourra pas avoir d'effet exploitable sur l'état futur des sommets (et donc la sortie). Cependant, une ronde non consommée non terminale, c'est-à-dire qui sera suivi de rondes consommées, peut avoir un impact sur la sortie dans la mesure où les tops horloge auront évolué et pourront être exploités par une future ronde consommée.

La définition du temps va dépendre du mode synchrone ou pas. Mais pour les résumer, il s'agit du nombre de rondes ou d'unités de temps jusqu'au dernier message consommé dans le pire des cas.

Définition 2.1 (temps synchrone) *La complexité en* temps *d'un algorithme distribué A sur un graphe G en mode synchrone, noté* TEMPS(A, G)*, est le nombre de rondes écoulées jusqu'au dernier message consommé, durant l'exécution de A sur G dans le pire des cas (c'est-à-dire sur toutes les entrées valides de A).*

Il s'agit donc du nombre de rondes, c'est-à-dire de nombre de cycles SEND/RECEIVE/COMPUTE, la dernière ronde devant être consommée.

Une remarque évidente mais fondamentale est qu'un algorithme de complexité en temps synchrone t implique que pendant son exécution, tout sommet ne peut interagir qu'avec des sommets situés à une distance inférieure ou égale à t.

Définition 2.2 (temps asynchrone) *La complexité en* temps *d'un algorithme distribué A sur un graphe G en mode asynchrone, noté $\textsc{Temps}(A, G)$, est le nombre d'unités de temps écoulées jusqu'au dernier message consommé, durant l'exécution de A sur G dans le pire des cas (c'est-à-dire sur toutes les entrées valides de A et tous les scénarios possibles), l'unité de temps étant définis comme la durée maximum pour qu'un message traverse une arête.*

La remarque précédente à propos de la distance ne s'applique plus en mode asynchrone, puisqu'il est possible que les messages transitent entre voisins bien plus vite que le temps unitaire, et donc en temps $< t$ entre sommets à distance t.

Un scénario possible pour le mode asynchrone est que tous les messages sont émis en même temps et traversent chaque arête en temps 1 exactement, des messages pouvant se croiser sur une même arête. Il s'agit du *scénario synchrone*. Autrement dit, la complexité en temps en mode asynchrone est toujours au moins égale à celle synchrone, puisqu'on maximise le temps sur tous les scénarios possibles. En particulier, si durant l'exécution de A deux sommets à distance t interagissent, c'est que la complexité en temps de A est au moins t, que cela soit en mode synchrone ou pas.

Lors d'un scénario synchrone s'exécutant sur un système asynchrone, tous les messages traversent les arêtes à la vitesse du lien le plus lent, ce qui n'est pas le scénario le plus avantageux. L'enjeu d'un algorithme asynchrone est de pouvoir fonctionner dans n'importe quel scénario et donc surtout dans des scénarios ou des messages traversent beaucoup plus vite les arêtes, par exemple dans certaines régions du graphe ou à certains moments de l'exécution de l'algorithme.

Même si pour beaucoup d'algorithmes asynchrones, le scenario le moins favorable est celui synchrone, ce n'est pas toujours le cas. On peut imaginer, par exemple, un algorithme réalisant une diffusion d'un message M depuis le coin NORD-OUEST d'une grille où chaque sommet u, à la première reception de M, diffuse M à tous ses voisins, puis : (1) s'arrête si M provient d'un voisin SUD ou OUEST ; ou (2) se met à boucler en envoyant des messages en continu, si M provient d'un voisin à l'EST ou au NORD.

Définition 2.3 (nombre de messages) *La complexité en* nombre de messages *d'un algorithme distribué A sur un graphe G, noté $\textsc{Message}(A, G)$, est le nombre de messages envoyés durant l'exécution de A sur G dans le pire des cas (sur toutes les entrées valides de A et tous les scénarios possibles en mode asynchrone).*

Si l'on compte chaque message envoyés, même ceux non-consomés, c'est parce qu'on

considère que chaque envoi de message a un coût (notamment énergétique) pour les sommets.

Résoudre une tâche dans un système distribué prend un certain temps et nécessite un certain volume de communication. Et la plupart du temps, il existe un compromis entre ces deux valeurs. Bien sûr c'est très schématique. Il existe d'autres mesures de complexité, comme la complexité de communication, mesurée en nombre de bits (*bit complexity* en Anglais), donnant le nombre total de bits échangés lors d'une exécution (ou sa variante donnant le nombre maximum de bits échangés le long d'une arête). C'est une mesure plus fine que le nombre de messages puisqu'à nombre de message identique, on pourrait préférer l'algorithme utilisant des messages plus courts économisant ainsi la bande passante. On en reparlera au chapitre 4 sur la complexité de communication.

Comme en séquentiel, on peut aussi définir toute une série de complexités, selon qu'on analyse l'algorithme sur toutes les instances ou pas : analyse dans le pire des cas, dans le cas moyen, selon une distribution particulière, etc.

En asynchrone on peut également limiter l'ensemble des scénarios possibles, par exemple en limitant la puissance de nuisance de l'*adversaire*. On parle aussi de *scheduler*, l'ordonnanceur temporel de tous les messages. Ici l'adversaire est incarné par le réseau avec ses variations dans les transmissions des messages et donc l'exécution de l'algorithme. Une façon de limiter ses scénarios sera par exemple de supposer que le temps de transmission des messages varie suivant les liens mais pas dans le temps si bien qu'un algorithme rusé pourrait apprendre petit à petit les liens rapides et lents et ainsi s'adapter. Aussi, une hypothèse classique surtout utiliser dans le cas d'algorithme probabiliste est que le *scheduler* agit indépendant des choix aléatoires fait par l'algorithme. L'adversaire ne connaît pas à l'avance les choix aléatoires fait par l'algorithme pendant son exécution. Cela revient à supposer que les temps de transmission du *scheduler* sont, certes fixés par un adversaire, mais ils le sont avant le début de l'exécution.

2.2 Modèles représentatifs

Parmis les nombreux modèles qui existent dans la littérature, on en distingue trois qui correspondent à des ensembles d'hypothèses différents. Chaque ensemble d'hypothèses permet d'étudier un aspect particulier du calcul distribué. Un modèle n'a généralement pas vocation à épouser au mieux la réalité. C'est une abstraction permettant de prédire et/ou d'expliquer (parfois via des simulations) le résultat d'expériences bien réelles. Les trois aspects sont : la *localité*, la *congestion*, l'*assynchronisme*.

Bien sûr, d'autres aspets du calcul distribué pourrait être étudiés comme : la tolérance aux pannes, le changement de topologie, la fiabilité (liens/processeurs qui, sans être en panne, sont défaillant), la sécurité. On peut alors parler de modèles de dynamicité du réseau, modèles de mobilité, modèles d'adversaire/d'attaque, etc.

Les hypothèses suivantes sont communes aux trois modèles :

1. **Pas d'erreur** : il n'y a pas de crash lien ou processeur, pas de changement de topologie. Les liens et les processeurs sont fiables. Un message envoyé fini toujours par arriver. Les messages peuvent se croiser sur une arête (pas de collision) mais ils ne peuvent pas se doubler (les messages arrivent dans le même ordre d'émission). On parle de canaux ou de liens FIFO (*First In First Out*).
2. **Calculs locaux gratuits** : en particulier faire un ou plusieurs Send dans le même cycle de calcul a un coût nul. Finalement, seules les communications sont prises en compte dans le coût de l'exécution de l'algorithme.
3. **Identité unique** : les processeurs ont une identité codée par un entier de $O(\log n)$ bits n étant le nombre de processeurs ou le nombre de sommets du graphe. Le réseau n'est pas anonyme. Ces identités font partie de l'instance du problème, l'instance étant codée par l'état initial des sommets du graphe.

La troisième hypothèse (identité unique) ne se révèle pas fondamentale (et donc peut être supprimée) pour certains problèmes. Par exemple, plusieurs algorithmes qu'on verra, dont Flood qui sera vu au chapitre 3 section 3.3, ne l'utilisent pas. Cependant, elle est en fait au cœur des problèmes de brisure de symétrie consistant à casser la symétrie (*symmetry breaking* en Anglais), comme le problème de l'élection d'un *leader* (que nous n'aborderons pas). Sans cette hypothèse, ces problèmes ne sont pas toujours solubles, mêmes en l'absence d'erreur. En effet, deux sommets reliés par une arête par exemple ne pourront jamais faire autrement que de calculer une sortie identique, si leurs entrées sont les mêmes, puisque le cas parfaitement synchrone peut se produire. [*Question.* Peut-on résoudre le consensus pour un tel système ?] Pour la coloration (abordée au chapitre 7), cette hypothèse est nécessaire. Notons cependant qu'en mode synchrone, un algorithme de complexité en temps t n'a besoin d'identités uniques au mieux dans une boule de rayon t puisqu'au cours de l'exécution aucune interaction entre sommets à distance $> t$ n'est possible.

Les trois modèles sont :

LOCAL pour étudier la nature locale d'un problème.

- mode synchrone, tous les processeurs démarrent le calcul au même top
- message de taille illimitée

CONGEST pour étudier l'effet du volume des communications.

- mode synchrone, tous les processeurs démarrent le calcul au même top
- message de taille limitée à $O(\log n)$ bits.

ASYNC pour étudier l'effet de l'assynchronisme.

- mode asynchrone

On pourrait se demander d'où vient cette limite de $O(\log n)$ bits sur la taille des messages dans le modèle CONGEST. C'est un peu l'analogue de l'hypothèse, souvent tacite, qui est faite sur la taille ω des registres (ou mots) du modèle RAM (séquentiel) où

les opérations arithmétiques élémentaires sur des mots de taille ω prennent un temps constant. Par exemple, le problème de calculer la somme des éléments d'un tableau de n entiers a une complexité en temps $O(n)$ car on suppose bien évidemment que les instructions comme `s=0`, `s += T[i]`, `i++` ou `i<n` prennent un temps constant. On prend toujours $\omega \geqslant \log_2 n$ car c'est la taille minimale en bits pour pouvoir représenter un pointeur (ou indice) sur la donnée supposée être de n mots justement.

Pour revenir au modèle CONGEST, on suppose donc que les messages ne peuvent transmettre qu'un nombre constant d'identifiants de sommets ou d'arêtes, ou de valeurs entières qui restent polynomiale en n, typiquement des valeurs dans $[0, |V(G)| + |E(G)|[$. Notez qu'une valeur entière dans l'intervalle $[0, n^4[$ par exemple ne prendra jamais que $\lceil 4 \log_2 n \rceil = O(\log n)$ bits pour être représentée en binaire.

Bien sûr, on peut raffiner et multiplier les modèles (à l'infini ou presque) selon de nouvelles hypothèses, avec par exemple le modèle ASYNC et messages limités, ASYNC et messages illimités, CONGEST et messages de taille b, etc.

2.3 Rappels sur les graphes

Soit $G = (V, E)$ un graphe.

- On note aussi $V(G) = V$, l'ensemble des sommets (*vertices* en Anglais) et $E(G) = E$, l'ensemble des arêtes (*edges* en Anglais) de G. Souvent on note sans le préciser que $n := |V(G)|$ et $m := |E(G)|$.
- Le *degré* du sommet u, noté $\deg(u)$ ou $\deg_G(u)$ s'il y a ambiguité, est le nombre de ses voisins. La somme des degrés des sommets d'un graphe vaut deux fois son nombre d'arêtes : $\sum_{u \in V} \deg(u) = 2m$. En effet, chaque arête contribue au degré de ses deux sommets extrémités.
- G est *connexe* si entre n'importe quelles paires de sommets il existe un chemin les connectant. Si G est connexe alors $m \geqslant n - 1$, et donc $m = \Omega(n)$.
- La *longueur* d'un chemin est le nombre d'arêtes qui le compose. Un *plus court chemin* entre u et v est un chemin de longueur minimum entre u et v. Cette longueur minimum est appelée *distance* et notée $\text{dist}_G(u, v)$.

  Dans le cas de graphes ayant une valuation sur les arêtes (on parle aussi de coût ou de longueur d'arête), la longueur d'un chemin (on parle aussi de coût) est alors la somme des valeurs des arêtes qui le compose.
- L'*inégalité triangulaire* s'applique aux distances dans les graphes, à savoir : pour tous sommets x, y, z de G, $\text{dist}_G(x, y) \leqslant \text{dist}_G(x, z) + \text{dist}_G(z, y)$.
- La *boule de voisinage* de u et de rayon r est l'ensemble des sommets à distance au plus r de u, notée parfois $B_G(u, r) := \{v : \text{dist}_G(u, v) \leqslant r\}$. L'*ensemble des voisins* d'un sommet u dans G sera noté $N_G(u)$ (*Neighbors* en Anglais). Ainsi, $N_G(u) =$

$B_G(u,1) \setminus \{u\}$. On note plus simplement $N(u)$ et $B(u,r)$ si G est clair d'après le contexte.

- Le *diamètre* de G est la valeur $\mathrm{diam}(G) := \max_{u,v \in V(G)} \mathrm{dist}_G(u,v)$. C'est la distance la plus grande dans le graphe.
- L'*excentricité* de u dans G (*eccencricity* en Anglais), notée $\mathrm{ecc}_G(u)$, est la distance séparant u de son plus loin sommet. Plus formellement, $\mathrm{ecc}_G(u) := \max_{v \in V(G)} \mathrm{dist}_G(u,v)$. C'est aussi la hauteur minimum d'un arbre couvrant et de racine u. Notons que $\mathrm{diam}(G) = \max_{u \in V(G)} \mathrm{ecc}_G(u)$. Un *centre* est un sommet d'excentricité minimum. Cette valeur est appelée l'excentricité du graphe et notée $\mathrm{ecc}(G)$.

2.4 Exercices

Exercice 1

Construire un graphe de diamètre trois avec un sommet d'excentricité deux et sans sommet de degré un.

Exercice 2

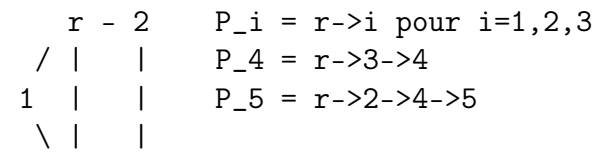
Montrer que pour tout sommet u d'un graphe G, $\frac{1}{2}\mathrm{diam}(G) \leqslant \mathrm{ecc}_G(u) \leqslant \mathrm{diam}(G)$.

Exercice 3

Donner le diamètre, l'excentricité et le nombre de centres des graphes suivants : chemin (P_n), cycle (C_n) et clique ou graphe complet (K_n) à n sommets, ainsi que la grille à $p \times q$ sommets ($M_{p,q}$).

Exercice 4

Dans toute la suite, G représente un graphe connexe de diamètre D avec n sommets dont un distingué noté r, et $P = \{P_i\}$ un ensemble de $n-1$ plus courts chemins prédéterminés de G connectant r à chacun des autres sommets v_i.



```
3 - 4          ronde1 ronde2 ronde3 ronde 4
   /      A1 = 1|2|3  4|5     4|5    5
  5       A2 = 1|4|5  2|3|4|5 5
```

Notez que les chemins P_i ne forment pas nécessairement un arbre.

On considère le problème Π consistant à envoyer un message individuel de $b = O(\log n)$ bits depuis r vers chaque sommets v_i selon le chemins P_i. Ainsi, (G, r, P) représente une instance du problème Π. On note OPT un algorithme résolvant $(G, r, P) \in \Pi$ en temps optimal.

Dans le modèle b-CONGEST prouvez ou réfutez $\forall n, \forall D = D(n)$:

(a) $\forall (G, r, P)$, $\textsc{Temps}(\mathsf{OPT}, G) = O(D)$ [borne supérieure]
(b) $\exists (G, r, P)$, $\textsc{Temps}(\mathsf{OPT}, G) = \Omega(D)$ [borne inférieure]
(c) $\forall (G, r, P)$, $\textsc{Temps}(\mathsf{OPT}, G) = \Omega(D)$ [borne inférieure globale]
(d) $\forall (G, r, P)$, $\textsc{Temps}(\mathsf{OPT}, G) = O(n)$ [borne supérieure]

Même question dans le modèle LOCAL.

Notes bibliographiques pour l'Exercice 4

Le problème Π de l'exercice précédant est un cas particulier d'une forme plus générale appelée Packet Routing. Pour ce problème, le nombre de chemins $|P|$ peut être quelconque, et chaque $P_i \in P$ est un chemin *arête-simple* entre u_i et v_i (ils peuvent s'auto-intersecter mais seulement sur des sommets et ne sont donc pas forcément des plus courts chemins). Contrairement à l'exercice, on n'a pas forcément $u_1 = u_2 = \cdots u_{|P|} = r$. Dans le problème Packet Routing, qui a été très étudié, il s'agit de déterminer un ordonancement minimisant le temps d'acheminement des messages dans le modèle CONGEST (un message à la fois par arête). Dans le domaine de l'ordonnancement, on parle de *makespan*. Il a d'ailleurs été montré NP-difficile de le calculer.

Deux paramètres influencent ce temps : la longueur du plus long chemin, appelée *dilation* de P (notée d), et le nombre de chemins utilisant une même arête, appelé *congestion* de P (notée c). Il est clair que le temps est au moins $\max\{c, d\}$, quel que soit P et le graphe G. [*Question*. Pourquoi ?] Cependant, il a été montré dans [LMR94] qu'il existe toujours un ordonnancement, c'est-à-dire un algorithme distribué, de temps $O(c+d)$ ce qui est asymptotiquement optimal. [*Question*. Pourquoi est-ce asymptotiquement optimal ?] [*Exercice*. Montrez comment déduire de cette borne $O(c+d)$ la question (d) de l'exercice précédent.] [*Exercice*. Calculez les valeurs de c et d dans le cadre de l'exercice précédent lorsque G est un chemin où r est une extrémité. Puis montrez que $\textsc{Temps}(\mathsf{OPT}, G) \not\geq c + d$.] Seulement on ne sait pas construire un tel algorithme, alors qu'on sait qu'il existe ! C'est une application du résultat profond dit « LLL », le Lemme Local de Lovász, qui décrit l'existence d'évènement « rares » évitant toutes une séries de « mauvais » évènements.

Le meilleur algorithme connu pour Packet Routing produit un temps $O(\mu(c + d)(\log^4 \rho)(\log\log\rho))$, où ρ est la longueur totale de tous les chemins de P et μ le nombre d'arêtes distinctes utilisés par un message. On a $\mu \geqslant d$, mais on peut aussi supposer que $\mu \leqslant |E(G)|$ puisque les arêtes non utilisées ne jouent aucun rôle dans G et donc peuvent être supprimées. Cet algorithme est probabiliste [LMR99]. Il existe des variantes où il s'agit de trouver un ensemble de chemins P permettant le temps minimum de routage. Dans ce cas il est possible de construire de tel ordonnancement dont le temps est à un facteur constant de l'optimal [ST01]. Voir aussi [PSW10] pour des développements plus récents et d'autre références.

Bibliographie

[LMR94] F. T. Leighton, B. M. Maggs, and S. B. Rao, *Packet routing and job-shop scheduling in O(congestion + dilation) steps*, Combinatorica, 14 (1994), pp. 167–180. doi : 10.1007/BF01215349.

[LMR99] F. T. Leighton, B. M. Maggs, and A. W. Richa, *Fast algorithms for finding O(congestion + dilation) packet routing schedules*, Combinatorica, 19 (1999), pp. 375–401. doi : 10.1007/s004930050061.

[PSW10] B. Peis, M. Skutella, and A. Wiese, *Packet routing on the grid*, in 10th Latin American Symposium on Theoretical Informatics (LATIN), vol. 6034 of Lecture Notes in Computer Science, April 2010, pp. 120–130. doi : 10.1007/978-3-642-12200-2_12.

[ST01] A. Srinivasan and C.-P. Teo, *A constant-factor approximation algorithm for packet routing and balancing local vs. global criteria*, SIAM Journal on Computing, 30 (2001), pp. 2051–2068. doi : 10.1137/S0097539798335596.

CHAPITRE

3 Diffusion

Sommaire

Dans ce chapitre nous donnons des algorithmes élémentaires pour la diffusion, la concentration, et la construction d'arbre de diffusion.

Mots clés et notions abordées dans ce chapitre :

- diffusion, arbre de diffusion,
- inondation, concentration.

3.1 Diffusion

Il s'agit de transmettre un message M à tous les sommets d'un graphe connexe à n sommets depuis un sommet distingué que l'on notera r_0. Au démarrage, chaque sommet sait s'il est ou pas r_0 via $\mathrm{ID}(r_0)$ supposé connu de tous. C'est la connaissance *a priori*. Bien évidemment, M est une donnée privée de r_0.

On va considérer les modèles synchrones (LOCAL et b-CONGEST) et asynchrones (ASYNC). On supposera que M est suffisamment petit pour tenir dans un message, c'est-à-dire qu'il a une taille d'au plus $b = O(\log n)$ bits si l'on considère le modèle b-CONGEST.

Comme souligné dans la section 1.3.2, la connaissance *a priori* des sommets, c'est-à-dire les informations mise à leur disposition à leur réveil, peut impacter les performances des algorithmes.

Proposition 3.1 *Pour tout graphe G et sommet distingué r_0, tout algorithme distribué de diffusion A sur G, en mode synchrone ou asynchrone et quelle que soit la connaissance a priori des sommets sur G, doit vérifier :*

- $\textsc{Message}(A, G) \geqslant n - 1$.
- $\textsc{Temps}(A, G) \geqslant \mathrm{ecc}_G(r_0)$.

On rappelle (cf. section 2.3) que $\mathrm{ecc}_G(r_0)$, l'*excentricité* de r_0 dans G, n'est ni plus ni moins que la hauteur minimum d'un arbre couvrant G et de racine r_0. C'est donc aussi la profondeur d'un arbre couvrant en largeur d'abord qui a pour racine r_0.

Preuve de la proposition 3.1. Il faut informer $n-1$ sommets distincts du graphe. L'envoi d'un message sur une arête ne peut informer qu'un seul sommet, celui situé à l'extrémité de l'arête. Donc il faut que le message circule sur au moins $n-1$ arêtes différentes de G pour résoudre le problème. Donc $\textsc{Message}(A, G) \geqslant n - 1$.

Dans G il existe un sommet u à distance $t = \mathrm{ecc}_G(r_0)$ de r_0. Dans le scénario synchrone (valable en mode synchrone ou asynchrone), chaque message traversant une arête prend un temps 1. Il faut donc un temps au moins t pour que u soit informé. Donc $\textsc{Temps}(A, G) \geqslant t$. □

Les bornes inférieures données dans la proposition 3.1 s'appliquent pour chaque graphe et quelle que soit la connaissance *a priori* des sommets. En particulier, il n'y a pas de « bons » graphes dans lesquels la diffusion pourrait être résolue plus efficacement que les bornes énoncées.

On verra plus tard dans la proposition 3.5 qu'on peut dire un peu mieux si on limite la connaissance *a priori* des sommets.

3.2 Arbre de diffusion

Une stratégie courante pour réaliser une diffusion est d'utiliser un arbre couvrant enraciné en r_0, disons T. L'intérêt est qu'on ne diffuse M que sur les arêtes de T. Donc, ici on va supposer que chaque sommet u de G connaît T par la donnée de son parent Parent(u) et de l'ensemble de ses fils Fils(u). L'algorithme ci-dessous dépend donc de l'arbre T et de sa racine r_0. Dans la suite, et pour faire court, on écrit « $u \neq r_0$ » à la place de « $\mathrm{ID}(u) \neq \mathrm{ID}(r_0)$ ».

Algorithme $\mathsf{Cast}_T(r_0)$
(code du sommet u)

1. Si $u \neq r_0$, alors $M :=$ Receive()
2. Pour tout $v \in$ Fils(u), Send(M, v)

La connaissance *a priori* des sommets est signalée en couleur. En bleu, il s'agit des paramètres du problème (comme r_0) ou du modèle (comme u).

Si l'algorithme précédent s'exécute dans un environnement synchrone, il faut comprendre la reception comme bloquante, mais seulement pour la ronde courrante, à savoir tant qu'on n'a pas reçu de message on fait un Receive mais il se débloque à la fin de la ronde si nécessaire. [*Exercice.* Écrire l'algorithme avec le formalisme des algorithmes synchrones, notamment avec les instructions NewRound précisant le début de chaque nouvelle ronde.]

Proposition 3.2 *Pour tout graphe G et sommet distingué r_0, en mode synchrone ou asynchrone :*

- $\textsc{Message}(\mathsf{Cast}_T(r_0), G) = |E(T)| = n - 1$.
- $\textsc{Temps}(\mathsf{Cast}_T(r_0), G) = \mathrm{ecc}_T(r_0) \leqslant n - 1$.

Notons que $\mathrm{ecc}_T(r_0)$ n'est ni plus ni moins que la hauteur de l'arbre. Si T est un arbre en largeur d'abord alors $\mathrm{ecc}_T(r_0) = \mathrm{ecc}_G(r_0)$. [*Question.* Est-ce que le contraire est vrai ?] Dans ce cas, l'algorithme Cast_T est optimal en temps et en nombre de messages.

3.3 Inondation

On utilise cette technique (*flooding* en Anglais) en l'absence de structure pré-calculée comme un arbre de diffusion. Tout sommet u ne connaît ni Parent(u) ni Fils(u).

Algorithme $\mathsf{Flood}(r_0)$
(code du sommet u)

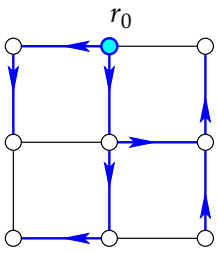


FIGURE 3.1 – Diffusion dans une grille 3×3 selon l'algorithme $\mathsf{Cast}_T(r_0)$, où T en bleu, de racine r_0, est un arbre couvrant de hauteur 3. Il y a autant de messages émis que d'arêtes dans l'arbre ($n-1=8$).

1. Si $u = r_0$, alors SEND(M, v) pour tout $v \in N(u)$.
2. Sinon,
   (a) $M :=$ RECEIVE() et v le voisin qui a envoyé M
   (b) SEND(M, w) pour tout $w \in N(u) \setminus \{v\}$.

On remarque que si, dans l'algorithme Flood ci-dessus, on remplace « $N(u)$ » et « $N(u) \setminus \{v\}$ » par « FILS(u) », alors on retombe sur une autre écriture de l'algorithme Cast_T. Voir la figure 3.2 pour un exemple d'exécution.

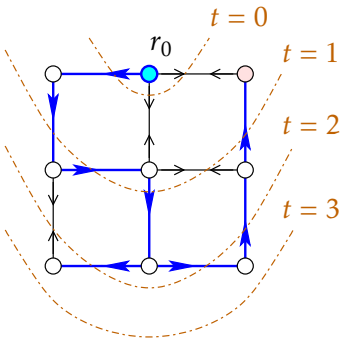


FIGURE 3.2 – Diffusion dans une grille 3×3 selon l'algorithme $\mathsf{Flood}(r_0)$ dans un scenario asynchrone, où 16 messages sont émis. Les messages noirs ($\rightarrow$) arrivent après que le sommet soit informé, tant dis que les messages bleus ($\rightarrow$) sont ceux qui informent le sommet. Quel que soit le scénario, après un temps t, tous les sommets à distance t de r_0 sont informés, même s'ils le sont plus tôt par un chemin plus long que t (comme le voisin coloré de r_0).

Proposition 3.3 *Pour tout graphe G et sommet distingué r_0, en mode synchrone ou asynchrone :*

- $\textsc{Message}(\mathsf{Flood}(r_0), G) = 2m - (n-1)$.

- $\textsc{Temps}(\mathsf{Flood}(r_0), G) = \mathrm{ecc}_G(r_0)$.

Preuve. Le nombre de messages envoyés (instruction Send) est, pour le sommet u, de $\deg(u)$ si $u = r_0$ ou $\deg(u) - 1$ sinon. Donc,

$$\begin{aligned}\textsc{Message}(\mathsf{Flood}(r_0), G) &= \deg(r_0) + \sum_{u \neq r_0} (\deg(u) - 1) = \deg(r_0) + \left(\sum_{u \neq r_0} \deg(u) \right) - (n-1) \\ &= \left(\sum_{u} \deg(u) \right) - (n-1) = 2m - (n-1)\,.\end{aligned}$$

Pour le temps, il est facile de montrer par induction sur la distance t, qu'après un temps t (ou après t rondes en mode synchrone), tous les sommets à distance t de r_0 ont reçu le message M. En effet, c'est vrai pour $t = 0$, et si cela est vrai jusqu'à $t-1$, alors tout sommet à distance t a au moins un voisin à distance $t-1$, qui par induction a reçu le message M. Il est donc certain que ce sommet aura reçu M après une unité de temps (ou une ronde).

Les sommets s'arrêtent donc après un temps au plus t, les éventuellement messages non consommés ne faisant pas partie du décompte du temps (cf. définition 2.1). Or tous les sommets sont à distance au plus $\mathrm{ecc}_G(r_0)$ de r_0. Donc $\textsc{Temps}(\mathsf{Flood}(r_0), G) \leqslant \mathrm{ecc}_G(r_0)$. Il y a égalité à cause de la proposition 3.1 qui s'applique ici. □

Dans le cas trivial d'un cycle à trois sommets, et dans le cas synchrone par exemple, le temps de l'algorithme est bien d'une ronde (car une seule ronde consommée), alors que des messages peuvent continuer à circuler au delà de cette dernière ronde, sans de Receive() correspondant. Voir figure 3.3.

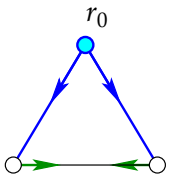


Figure 3.3 – Une ronde consommée pour l'algorithme $\mathsf{Flood}(r_0)$ dans le cas synchrone sur un triangle. Ici 4 messages sont émis, mais les deux derniers messages émis (en vert) ne sont pas consommés. Donc ici Temps = 1 et Message = 4.

3.4 Message *vs.* connaissance

Les deux algorithmes Cast_T et Flood vus précédemment présentent deux compromis différents entre le temps et le nombre de messages. Il est bon de remarquer que dans

l'absolu, aucun n'est meilleur que l'autre. Cela dépend de l'application.

Si M est un fichier de grande taille (par exemple un film à transmettre à un grand ensemble de clients), on a intérêt d'utiliser une diffusion optimale en nombre de messages (on peut imaginer que le coût global de diffusion est grossièrement proportionnel à la taille de M fois le nombre de liens utilisés), quitte à perdre également du temps pour la construction de T. Que la diffusion prenne plus de temps que nécessaire pour certains clients n'est peut-être pas fondamental, à coup sûr du point de vue du diffuseur.

À l'inverse, s'il s'agit de transmettre une alarme, un message court M du style « au feu », Flood sera une meilleure alternative puisqu'il a la propriété de transmettre au plus tôt le message, alors que le nombre de petits messages envoyés est secondaire.

Parenthèse historique. Il est historiquement intéressant de remarquez qu'en France dans les années 1970-80, il a été décidé de garder un système proche de l'algorithme Cast_T pour la transmission d'information. À l'époque, il s'agissait du réseau Transpac [1] (et de la norme X25 popularisée par le Minitel) utilisant la « commutation de circuits » qui établissait en premier lieu une connexion (un circuit) avant de transmettre les données sur ce chemin. L'avantage indéniable de ce choix était la facturation. En effet, il permettait de facturer la transmission à la connexion, en fonction de la distance et de la durée, comme les réseaux téléphoniques. Tout cela au détriement d'autres solutions plus efficaces en terme de débit consistant à envoyer de petits paquets, la « commutation de paquets ». C'est le choix qui sera fait par Arpanet puis Internet grâce à son protocole éponyme IP. Dommage, un peu plus et la France aurait été précurseur d'Internet.

La même philisophie, si l'on peut dire, s'applique au transport aérien. Eurocontrol [2], établissement de gestion du trafic aérien en europe, autorise le décollage d'un avion que s'il est garanti de pouvoir être suivi dans des cellules de contrôle tout au long de son trajet, et en particulier s'il peut attérir. C'est la philosophie consistant à prévoir le trajet entièrement. C'est différent outer-atlantique où les avions décollent avant l'aurotisation d'attérissage (avec d'éventuelles files d'attente en vol).

Peut-on faire mieux que Flood ? Si l'on résume, nous avons vu :

	Temps	Message	Connais.-a-priori
Cast_T	$\mathrm{ecc}_T(r_0)$	$n-1 = O(n)$	T
Flood	$\mathrm{ecc}_G(r_0)$	$2m-(n-1) = O(n^2)$	$\varnothing$

Table 3.1 – Compromis Message/Temps/Connaissance-a-priori pour les algorithmes de diffusion sur les graphes à n sommets et m arêtes.

1. Voir l'article https://www.epi.asso.fr/revue/articles/a2001b.htm.
2. https://www.eurocontrol.int/

Étant donné la simplicité de Flood, on aurait envie de conjecturer que si chaque sommet ne connaît que son identifiant et celui de ses voisins (et donc en l'absence d'une connaissance globale comme un arbre couvrant et de la distinction des voisins en fils, parent et autres), alors on ne peut pas vraiment faire mieux en terme de messages. Et qu'il faut $\Omega(m)$ messages (et pas seulement $n-1$), au moins dans certains graphes si ce n'est tous.

Il se trouve que la réponse à cette question est moins triviale qu'elle n'y paraît (voir l'introduction de [AGPV90]). En effet, si la liste des voisins est connue de chaque sommet, il devient possible de diffuser avec seulement $2(n-1)$ messages (soit l'optimal à un facteur deux près), si des messages de taille $O(n \log n)$ bits sont permis. [*Exercice*. Pourquoi ? Montrez même que des messages de $O(n)$ bits suffisent, si les identifitants sont dans l'intervalle $[0,n[$.] Certes, en faisant ainsi cela fait un volume total de $O(n^2 \log n)$ bits échangés, qui peut se découper en $O(n^2)$ messages de $O(\log n)$ bits. Cela ne remet pas en question la conjecture émise précédemment si on la corrige ainsi : $\Omega(m)$ messages *courts* sont nécessaires dans le pire cas sans connaîssance a priori autre que ses voisins. Mais cet exemple montre que la réponse à cette conjecture ne peut être triviale. La proposition 3.5 que l'on va voir juste après fera également vaciller nos certitudes.

Connaissance à distance ρ. Dans la suite l'expression « connaissance à distance ρ » signifie qu'au réveil chaque sommet u du graphe G connaît tous les chemins composés d'au plus ρ arêtes partant de u. Il s'agit donc du sous-graphe (avec les identifiants des sommets) composé de l'union de ces chemins. On parle parfois de « vue du u à distance ρ ». C'est aussi le sous-graphe induit par $B(u,\rho)$, la boule de rayon ρ centrée en u, mais sans les arêtes entre deux sommets à distance exactement ρ de u. Voir la section 2.3 et la figure 3.4. Il est facile de voir que la connaissance à distance ρ pour u est tout ce que u peut apprendre après ρ rondes synchrones de communications.

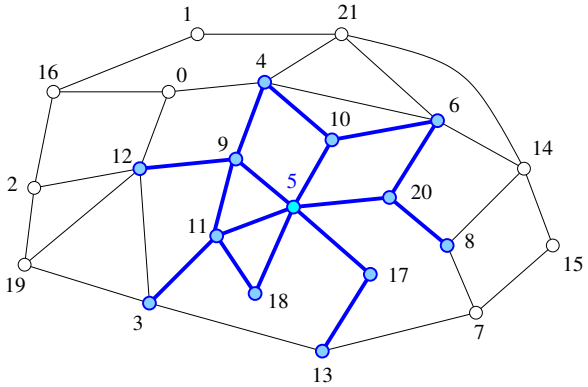


FIGURE 3.4 – Graphe avec $n = 22$ sommets avec des identifiants dans $[0,n[$. En bleu, la « connaissance à distance $\rho = 2$ » pour le sommet d'identifiant 5. Les arêtes entre deux sommets à distance exactement ρ ne font pas partie de cette connaissance.

Donc pour $\rho = 1$, il s'agit nécessairement d'une étoile étiquetée par les identifiants des sommets concernés, les arêtes entre voisins étant inconnues. Le cas $\rho = 0$ correspond au cas où les sommets n'ont aucune information sur le graphe sinon qu'ils connaissent leur identifiant.

L'algorithme Flood correspond au cas $\rho = 0$ alors que Cast_T correspond au cas $\rho = n$. [*Exercice*. Montrez que la connaissance à distance n implique la connaissance d'un arbre commun. Qu'en est-il de $\rho = n-1$? et de $\rho = D$?] Comme on l'a vu, pour une diffusion, $n-1$ messages suffisent si $\rho = n$. Alors que pour $\rho = 0$, il en faut $2m - n + 1 \leqslant n^2$.

La prochaine proposition établit un compromis entre le rayon ρ de la connaissance *a priori*. Rappelons que l'on considère les messages de $b = O(\log n)$ bits.

Proposition 3.4 *Pour tout entier $\rho \geqslant 1$, tout graphe G et sommet r_0, il existe un algorithme de diffusion FloodK_ρ avec une connaissance à distance ρ tel qu'en mode synchrone ou asynchrone $\textsc{Message}(\mathsf{FloodK}_\rho(r_0), G) \leqslant 2\min\{m, n^{1+1/\rho} + n\} - (n-1)$.*

Avant de donner la preuve, explicitons la complexité du nombre de message pour quelques valeurs de ρ :

- $\rho = 1$, $\textsc{Message} \leqslant 2\min\{m, n^2 + n\} = O(n^2)$
- $\rho = 2$, $\textsc{Message} \leqslant 2\min\{m, n^{3/2} + n\} = O(n^{1.5})$
- $\rho = 3$, $\textsc{Message} \leqslant 2\min\{m, n^{4/3} + n\} = O(n^{1.33\ldots})$
- $\rho = \log_2 n$, $\textsc{Message} \leqslant 2\min\{m, n^{1+1/\log_2 n} + n\} = O(n)$

Pour la dernière valeur, il faut remarquer que $\log_2\left(n^{1/\log_2 n}\right) = (1/\log_2 n) \times \log_2 n = 1$, et donc $n^{1/\log_2 n} = 2$.

Preuve. On utilise essentiellement deux idées. La première est que sans communication, chaque sommet peut « désactiver » certaines arêtes de façon à obtenir un sous-graphe connexe $\widetilde{G}$ couvrant G ne possédant que $\widetilde{m} \leqslant n^{1+1/\rho} + n$ arêtes. La deuxième est d'utiliser une simple diffusion à l'aide de Flood dans $\widetilde{G}$, en remarquant bien évidemment que $\widetilde{m} \leqslant m$, d'où le minimum dans l'énoncé.

Pour la désactivation, il faut associer un identifiant absolu à chaque arête. Il faut pouvoir le faire localement et de manière cohérente, c'est-à-dire que tous les sommets doivent pouvoir le faire localement sans communication et tomber d'accord sur ces identifiants. Une façon de faire, pour l'arête uv, est d'utiliser $\mathrm{ID}(uv) := \{\mathrm{ID}(u), \mathrm{ID}(v)\}$ comme identifiant. On utilise ensuite la relation d'ordre lexicographique sur des paires d'entiers, à savoir :

$$\{x,y\} \prec \{x',y'\} \quad \Leftrightarrow \quad \begin{array}{c} \min\{x,y\} \prec \min\{x',y'\} \\ \text{ou} \\ \min\{x,y\} = \min\{x',y'\} \text{ et } \max\{x,y\} < \max\{x',y'\}\ . \end{array}$$

Règle locale de désactivation.

> Un sommet u désactive l'arête incidente uv s'il existe dans G un cycle C de longueur $\leqslant 2\rho$ passant par uv et que $\mathrm{ID}(uv)$ est minimum pour les arêtes de C.

Il est facile de voir que cette règle peut être appliquée par chacun des sommets, sans aucune communication, mais à l'aide de sa connaissance à distance ρ.

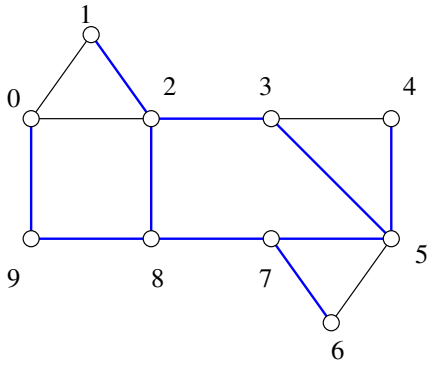


FIGURE 3.5 – Application de la règle de désactivation pour $\rho = 2$ (arêtes actives en bleu) qui aboutit à la suppression de tous les cycles de longueur $\leqslant 2\rho = 4$. Liste des identifiants des arêtes désactivées (ordre croissant) : $\{0,1\}$, $\{0,2\}$, $\{3,4\}$, $\{5,6\}$.

Connexité de $\widetilde{G}$. Comme on ne supprime que des arêtes, $\widetilde{G}$ et nécessairement un sous-graphe couvrant G. Ce qui pourrait empêcher l'algorithme Flood de fonctionner est que $\widetilde{G}$ ne soit pas connexe. Il faut donc montrer que $\widetilde{G}$ est connexe, en supposant bien sûr que G le soit aussi. A priori cela semble évident : lorsqu'un sommet décide de désactiver une arête, c'est qu'un cycle court existe et donc qu'il existe un chemin alternatif évitant l'arête. Malheureusement, ce cycle n'existe que dans G. Si cela est en effet vrai lorsque la première arête est désactivée, cela ne l'est plus forcément pour les suivantes et encore moins dans $\widetilde{G}$ où des désactivations en cascades pourraient se produire. L'exemple d'une échelle circulaire ou une roue avec $\rho = 2$ suffisent pour ce convaincre qu'une preuve s'impose. Chaque sommet a une vision locale du graphe et les désactivations ne se font pas de manière séquentielle en tenant compte des suppressions précédentes. Elles se font de manière simultanées dans la mémoire des sommets qui ne tient compte que de G, pas des décisions des autres sommets.

Soit $(e_1, \ldots, e_p)$ la suite des arêtes désactivées, c'est-à-dire qui étaient dans G et qui ne sont plus dans $\widetilde{G}$, ordonnée par identifiants croissants, soit

$$\mathrm{ID}(e_1) \prec \mathrm{ID}(e_2) \prec \cdots \prec \mathrm{ID}(e_p)\,.$$

Pour tout $i \in \{1, \ldots, p\}$, notons $G_i = G \setminus \{e_1, \ldots, e_i\}$ le graphe G où les i plus petites arêtes de la liste ont été supprimées. Ainsi $G_0 = G$ et $G_p = \widetilde{G}$. On a aussi $G_i = G_{i-1} \setminus \{e_i\}$. Notons

que l'algorithme distribué peut construire G_0, G_p, mais apriori il n'y a aucun moyen de construire un G_i pour $0 < i < p$. Cela n'empêche pas que, pour les besoins de l'analyse, la définition de G_i soit parfaitement valide.

Par induction sur i, montrons que G_i est connexe. C'est bien le cas pour G_0. Supposons que G_{i-1} est connexe. Soit C un cycle dans G ayant provoqué la suppression de e_i, c'est-à-dire passant par e_i qui a le plus petit identifiant de C. Alos C ne contient aucune arête e_j avec $j < i$ puisque les arêtes sont ordonnées par identifiants croissants. Il suit que C est aussi dans G_{i-1}. La suppression de e_i dans G_{i-1} produit un graphe encore connexe, car ses extrémités sont encore connectées via le chemin $C \setminus \{e_i\}$ qui est dans G_i. Donc $G_{i-1} \setminus \{e_i\}$ est connexe, c'est-à-dire G_i, et $G_p = \widetilde{G}$ aussi.

Nombre d'arêtes de $\widetilde{G}$. La règle de désactivation implique qu'il n'existe plus dans $\widetilde{G}$ de cycle de longueur $\leqslant 2\rho$. En effet, s'il en existait un, l'arête de plus petit identifiant sur ce cycle, disant uv, serait désactivée par u et v.

On va utiliser le fait suivant :

> Tout graphe sans cycle de longueur $\leqslant 2k$ possède $m < n^{1+1/k} + n$ arêtes.

Les détails de la preuve de ce fait peut-être trouvée dans [Gav24][Proposition 1.2, Chp. 1]. En deux mots, l'idée est de montrer que tout graphe H à n sommets, m arêtes et sans cycle de longueur $\leqslant 2k$ contient un arbre complet de profondeur k et de degré $\geqslant d$ où $d = m/n$ est le degré moyen. En particulier, cet arbre possède au moins $(d-1)^k$ feuilles, et bien sûr moins de n sommets. Du coup, on obtient l'inégalité $(d-1)^k = (m/n-1)^k < n$, ce qui implique le résultat $m < n^{1+1/k} + n$. Pour trouver cet arbre, il faut extraire un sous-graphe non vide où tous les sommets sont de degré $\geqslant d$. Et pour cela il suffit de « densifier » H en enlevant itérativement tout sommet de degré $< d$, c'est-à-dire inférieur à la moyenne. À la fin de cette densification, il reste un sous-graphe avec au moins un sommet (sinon H possèderait $< dn = m$ arêtes, ce qui est absurde) et tous sont de degré $\geqslant d$ par construction.

Avec des arguments plus complexes, il est possible de montrer deux fois mieux : tout graphe sans cycle de longueur $\leqslant 2k$ possède $m \leqslant \frac{1}{2}\left(n^{1+1/k} + n\right)$ arêtes [AHL02], et tout graphe sans cycle de longueur paire $\leqslant 2k$ possède $m \leqslant \frac{1}{2}n^{1+1/k} + 2^{k^2}n$ arêtes [LV05]. Notons que si l'on interdit seulement les cycles de longueur exactement $2k = 6$, il existe des graphes avec plus de $\left(\frac{1}{2} + 0.0338\right)n^{1+1/k}$ arêtes [FNV06].

Flood dans $\widetilde{G}$. L'abscence de cycle de longueur $\leqslant 2\rho$ implique que $\widetilde{G}$ possède $\widetilde{m} \leqslant n^{1+1/\rho} + n$ arêtes. Comme on a aussi $\widetilde{m} \leqslant m$, il suit que :

$$\widetilde{m} \leqslant \min\left\{m, n^{1+1/\rho} + n\right\}.$$

La connexité de $\widetilde{G}$ garantit une exécution valide de Flood. D'après la proposition 3.3,

$$\textsc{Message}(\mathsf{FloodK}_\rho(r_0), G) = \textsc{Message}(\mathsf{Flood}(r_0), \widetilde{G}) = 2\widetilde{m} - (n-1)$$

ce qui donne bien la formule désirée en remplaçant $\widetilde{m}$ par le majorant précédant. □

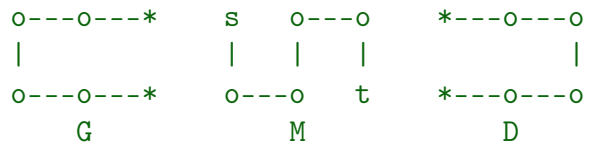
On peut donc compléter la table 3.1 ainsi :

	Message	Connais.-a-priori	
Cast_T	$O(n)$	T	$\rho = \Omega(\log n)$
FloodK_ρ	$O(n^{1+1/\rho})$	connais. à distance	$\rho \geqslant 1$
Flood	$O(n^2)$	$\varnothing$	$\rho = 0$

Table 3.2 – Compromis Message/Connaissance-a-priori pour les algorithmes de diffusion sur les graphes (connexes) à n sommets.

Comme vont le montrer les exercices suivants, on ne peut pas dire grand chose sur la complexité en temps de FloodK_ρ, sinon que $\textsc{Temps}(\mathsf{FloodK}_1(r_0), G) = \mathrm{ecc}_{\widetilde{G}}(r_0)$. Pour $\rho \in \{0,1\}$, $\textsc{Temps}(\mathsf{FloodK}_\rho(r_0), G) = \mathrm{ecc}_G(r_0)$ car $\widetilde{G} = G$ dans ces deux cas, puisqu'aucun sommet ne voit de cycle.

[*Exercice*. Montrer qu'en ajoutant un seul sommet à un graphe G peut suffire à désactiver toutes les arêtes initialement dans G pour tout $\rho \geqslant 2$ et tout G.] [*Exercice*. Soit G une grille $2 \times n$ (deux lignes horizontale de n sommets donc avec $2n$ sommets) et soit P un chemin couvrant de G. Montrez qu'il existe, pour $\rho = 2$, une affectation d'identifiants de G telle que $\widetilde{G}$, c'est-à-dire G sans les arêtes désactivées, soit exactement P. Pour cela vous pourrez supposer que tout chemin P couvrant G peut être formé à partir de trois types de configurations : gauche (G), milieu (M) et droite (D). Les chemins en configuration G (resp. D) ont leur extrémités alignées sur la colonne la plus à droite (resp. gauche) et forme une sorte de U tournée de 90 degré vers à droite (resp. gauche). Les chemins en configurations M ont leur extrémités sur la première et dernière colonne et ondulent entre les deux lignes horizontales. Le chemin P est alors (1) soit de type G, D ou M ; soit (2) la concaténation de G|M|D où les colonnes entre G|M et M|D sont identifiées et où la première et la dernière arête verticale de M sont supprimées. Ci-dessous, un exemple de chemin P en configuration G|M|D (ici disconnectée) allant de s à t, qui part d'abord vers la configuration G et revient dans M, puis ondule dans M, pour terminer dans D et revenir en t.



Proposez une numérotation par induction sur le nombre de cases de chaque configuration basée sur le fait (à prouver) que cela ne change en rien que les identifiants soient

réels ou entier.] [*Exercice*. Construire un graphe G à n sommets, avec un sommet distingué r_0 d'excentricité constante, et tel que $\textsc{Temps}(\mathsf{FloodK}_2(r_0), G) = \Omega(n)$.]

Le compromis établit dans la proposition 3.4 est essentiellement le meilleur possible. On ne démontrera pas les résultats suivants qui, comme dans le modèle CONGEST, supposent des messages de taille $b = O(\log n)$ bits :

Proposition 3.5 ([AGPV90]) *Pour tout $n, m, \rho \in \mathbb{N}$, il existe un graphe G avec n sommets et m arêtes pour lequel tout algorithme de diffusion A avec des messages d'au plus $O(\log n)$ bits, en mode synchrone ou asynchrone et avec une connaissance [3] à distance ρ :*

- *si $\rho \leqslant 1$, $\textsc{Message}(A, G) = \Omega(m)$;*
- *si $\rho \geqslant 2$, $\textsc{Message}(A, G) = \Omega\left((1/\rho) \cdot \min\{m, n^{1+c/\rho}\}\right)$, avec $c = 1$ pour $\rho \in \{2, 3, 5\}$ et $c = 2/3$ sinon [4].*

3.5 Arbre couvrant

Considérons maintenant le problème de construire de manière distribué un arbre couvrant de racine r_0. Des variantes de ce problème seront examinées plus amplement au chapitre 5. Plus précisément on souhaite qu'à la fin de l'exécution de l'algorithme chaque sommet u de G fixe une variable $\textsc{Parent}(u)$ correspondant à son parent ($\perp$ pour [5] $\textsc{Parent}(r_0)$), et un ensemble $\textsc{Fils}(u)$ correspondant à l'ensemble de ses fils.

Proposition 3.6 *À partir de tout algorithme de diffusion depuis r_0 on peut construire un arbre couvrant de racine r_0 avec une complexité en temps et en nombre de messages équivalentes.*

Preuve. Si l'on connaît un algorithme A de diffusion à partir de r_0, il suffit, pendant l'exécution de A, de définir le parent de u comme le voisin v d'où u a reçu M en premier. La flèche du temps interdit la création d'un cycle, et la diffusion à tous les sommets garantit que la forêt ainsi calculée est bien couvrante et connexe.

Si l'on souhaite récupérer les fils de u, il faut émettre vers son parent un message particulier (comme « je suis ton fils ») et collecter ce type de messages. Cela ajoute $n-1$ messages supplémentaires, et une unité à la complexité en temps. Cela ne modifie pas les complexités en temps et en nombre de messages car $n-1$ est une borne inférieure

3. On peut même supposer que tous les sommets connaissent n.

4. En fait, si la Conjecture d'Erdos-Simonovitz est vraie, $c = 1$ pour tout ρ. Elle affirme que pour chaque entier $\rho \geqslant 2$, il existe un graphe à n sommets sans cycle de longueur $\leqslant 2\rho$ avec $\Omega(n^{1+1/\rho})$ arêtes. Elle est vraie pour $\rho \in \{2, 3, 5\}$, et pour les autres valeurs, les constructions les plus denses possèdent $\Omega(n^{1+(2/3)/\rho})$ arêtes, voir [KSV13].

5. On utilise le symbol $\perp$ pour spécifier une valeur non définie.

pour la diffusion d'après la proposition 3.1. □

Notons qu'en asynchrone, le sommet u ne sait pas déterminer à partir de quel moment sa liste de fils est complète ou non. La liste sera complète après un temps $t+1$, où t est la complexité en temps de la diffusion. Mais n'ayant pas d'horloge, u est incapable de déterminer si le temps $t+1$ est révolu ou non. C'est le problème de la détection de la terminaison qui n'est pas exigée ici.

[*Exercice*. Déduire que tout algorithme distribué de calcul d'arbre couvrant nécessite $\Omega(m)$ messages, même si chaque sommet connaît ses voisins.]

On laisse en exercice l'écriture précise d'un algorithme permettant de construire un arbre couvrant de racine r_0. Cependant, en combinant les propositions 3.3 et 3.6, on obtient directement :

Corollaire 3.1 *Il existe un algorithme distribué noté* SpanTree *(basé sur* Flood*) permettant de construire un arbre couvrant G de racine r_0 avec (en mode synchrone ou asynchrone) :*

- $\textsc{Message}(\mathsf{SpanTree}, G) = 2m + O(n)$.
- $\textsc{Temps}(\mathsf{SpanTree}, G) = \mathrm{ecc}_G(r_0) + O(1)$.

Il est a noté que les arbres ainsi construits diffèrent beaucoup s'ils s'exécutent en mode synchrone ou asynchrone. Par exemple, dans le cas où G est une clique (graphe complet), l'arbre généré par SpanTree en synchrone est toujours une étoile, alors qu'il peut s'agir de n'importe quel arbre couvrant en mode asynchrone (voir figure 3.6).

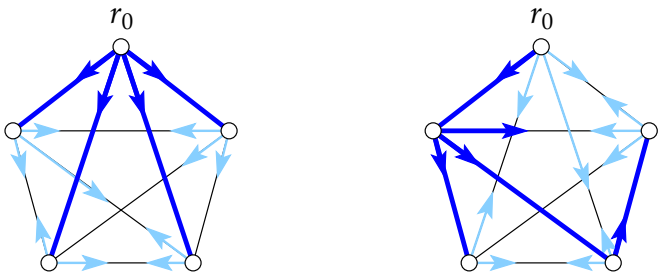


FIGURE 3.6 – Deux exécutions de l'algorithme SpanTree pour une clique à 5 sommets. En bleu clair les messages qui ne donneront pas lieu à des arêtes de l'arbre couvrant final.

3.6 Diffusion avec détection de la terminaison

Un problème potentiel des algorithmes précédents est qu'un sommet ne sait pas si la diffusion est terminée. C'est en fait un problème général en algorithmique distribuée.

Cela peut être très gênant en asynchrone lorsque plusieurs algorithmes, ou plusieurs étapes d'un même algorithme, doivent être enchaînés séquentiellement dans un ordre bien précis. Le risque est que des messages de deux algorithmes ou étapes se mélangent et aboutissent à des erreurs.

On distingue deux types de terminaisons :

Locale. Le processeur a terminé. En particulier, il n'émet ni ne reçoit plus aucun message, même si certains messages peuvent encore circuler sur des arêtes incidentes.

Globale. Tous les processeurs ont terminé et plus aucun message ne circule (ou plus exactement, il n'y a plus de réception de messages).

Bien évidemment, en séquentiel terminaison locale et globale coïncides.

Mais au-delà de la terminaison, il y a aussi la détection de la terminaison globale de l'algorithme. On en distingue trois :

Sans détection. L'algorithme est terminé, mais aucun processeur ne le détecte.

Détection locale. Il existe un processeur qui détecte que l'algorithme est terminé.

Détection globale. Tous les processeurs ont détecté que l'algorithme est terminé.

On va considérer la variante de la diffusion où, en plus de réaliser une diffusion à proprement parlée, l'on souhaite que chaque sommet u possède une variable booléenne $\textsc{Fin}(u)$, qui lorsqu'elle passe à $\textsc{Vrai}$ indique à u que tous les sommets de G ont bien reçu le message M. Il s'agit donc de la détection globale de la terminaison globale de l'algorithme.

La solution consiste à modifier l'algorithme Flood en envoyant un message (« *DONE* ») à son parent lorsque qu'on est sûr que tous les sommets de son propre sous-arbre ont été informés. Puis, le sommet r_0 diffuse un message de fin (« *END* ») vers tous les sommets de G. Pour cela on se sert de l'arbre que l'on construit en même temps que Flood (message « *FILS* »).

Dans l'algorithme ci-dessous, la variable c représente le nombre d'accusés qu'un sommet est supposé recevoir. Notons que l'algorithme suivant nécessite que les sommets connaissent leur nombre de voisins (leur degré).

Algorithme FloodEcho(r_0)
(code du sommet u)

1. $\textsc{Fils}(u) := \varnothing$ et $\textsc{Fin}(u) := \textsc{Faux}$.
2. Si $u = r_0$, alors $\textsc{Send}$(« DATA,M »,v) pour tout $v \in N(u)$, poser $c := \deg(u)$.
3. Sinon,
   (a) $M := \textsc{Receive}()$ et v le voisin qui a envoyé M.
   (b) $\textsc{Send}$(« FILS »,v), $\textsc{Send}$(« DATA,M »,w) pour tout $w \in N(u) \setminus \{v\}$.
   (c) Poser $\textsc{Parent}(u) := v$ et $c := \deg(u) - 1$.
4. Tant que $c > 0$:

(a) M' := Receive() et v le voisin qui a envoyé M'.
(b) Si M' est de type « DATA », Send(« ACK », v).
(c) Si M' est de type « FILS », Fils(u) := Fils(u) $\cup \{v\}$.
(d) Si M' est de type « ACK » ou « *DONE* », poser $c := c - 1$.

5. Si $u \neq r_0$, Send(« *DONE* », Parent(u)) et attendre un message « END ».
6. Fin(u) := Vrai, Send(« *END* », v) pour tout $v \in$ Fils(u).

Dans la suite on notera $D = \text{diam}(G)$ le diamètre de G.

Proposition 3.7 *Pour tout graphe G et sommet distingué r_0 :*
En mode synchrone :

- $\textsc{Message}(\mathit{FloodEcho}(r_0), G) = O(m)$,
- $\textsc{Temps}(\mathit{FloodEcho}(r_0), G) = O(\text{ecc}(r_0)) = O(D)$.

En mode asynchrone :

- $\textsc{Message}(\mathit{FloodEcho}(r_0), G) = O(m)$,
- $\textsc{Temps}(\mathit{FloodEcho}(r_0), G) = O(n)$.

Preuve. [*Cyril*. IL FAUT MONTRER QUE L'ALGO EST VALIDE]

En ce qui concerne le nombre de messages, les étapes 2,3 envoient $\sum_u \deg(u) = 2m$ messages, qui sont reçus à l'étape 4. Les étapes 5,6 concentrent et diffusent le long d'un arbre, donc consomment $2(n-1)$ messages. En tout, cela fait $O(m+n) = O(m)$ messages puisque G est connexe (et donc $m \geqslant n-1$).

En ce qui concerne le temps, les étapes 2,3,4 prennent un temps $O(D)$. La concentration puis la diffusion aux étapes 5,6 prennent un temps $O(D)$ en synchrone. Malheureusement, en mode asynchrone, on ne maîtrise pas la hauteur de l'arbre, qui peut être de $n-1$. La complexité en temps asynchrone est donc $O(D+n) = O(n)$ (car $D < n$) dans le pire des scénarios. □

3.7 Concentration

La concentration est un problème similaire à celui de la diffusion. Il s'agit pour chaque sommet u du graphe d'envoyer un message personnel M_u vers un sommet distingués r_0. C'est en quelque sorte le problème inverse, sauf que les messages ne sont pas tous identiques comme dans le cas d'une diffusion.

Voici une solution, lorsqu'un arbre T de racine r_0 est disponible, c'est-à-dire connu de chaque sommet u par les variables Parent(u) et Fils(u).

[*Cyril*. Il y a en fait deux problèmes : (1) Le ConvergeCast selon un arbre T déjà construit, où chaque sommet doit renvoyer à son parent une certaine fonction f de sa valeur et de celles reçues de tous ces voisins. Cela sert par exemple pour les Dijkstra, Synchroniseurs β, ... f peut être la somme (et cela sert à compter le nombre de sommets de l'arbre), où une fonction constante (qui sert à indiquer à la racine lorsque tout le monde à fini). Et (2), le problème de convergence avec messages individuels et en l'absence d'arbre. Le problème (1) est en fait important puisqu'il est à la base de Map-Reduce, ici f est correspond à Reduce. Illustrer les problèmes Map-Reduce de base comme compter un mot donné sur les milliers de pages web (Reduce f est la somme), ou le calcul du *page-rank* de Google où l'on veut compter le nombre de fois qu'une page web est pointée.]

Algorithme $\mathsf{Converge}_T(r_0)$
(code du sommet u)

1. Poser $c := |\textsc{Fils}(u)|$.
2. Si $c = 0$ et $u \neq r_0$, alors $\textsc{Send}(M_u, \textsc{Parent}(u))$.
3. Tant que $c > 0$:
   (a) $M := \textsc{Receive}()$ et v le voisin qui a envoyé M.
   (b) $c := c - 1$.
   (c) Si $u \neq r_0$, $\textsc{Send}(M, \textsc{Parent}(u))$.

Si aucun arbre n'est disponible, le plus simple est alors d'en calculer un, par exemple grâce à SpanTree, puis d'appliquer $\mathsf{Converge}_T$. La mauvaise solution étant que chacun des sommets réalise indépendamment une diffusion. On est donc sûr que r_0 reçoive une copie de chaque message, mais ce n'est pas efficace en terme de nombre de messages avec $O(nm)$ messages.

Proposition 3.8 *Il existe un algorithme distribué* Converge *qui réalise pour tout graphe G et sommet r_0 la concentration vers r_0 avec :*

En mode synchrone :

- $\textsc{Message}(\mathsf{Converge}, G) = O(m)$,
- $\textsc{Temps}(\mathsf{Converge}, G) = O(D)$.

En mode asynchrone :

- $\textsc{Message}(\mathsf{Converge}, G) = O(m)$,
- $\textsc{Temps}(\mathsf{Converge}, G) = O(n)$.

Bibliographie

[AGPV90] B. Awerbuch, O. Goldreich, D. Peleg, and R. Vainish, *A trade-off between information and communication in broadcast protocols*, Journal of the ACM, 37 (1990), pp. 238–256. doi : 10.1145/77600.77618.

[AHL02] N. Alon, S. Hoory, and N. Linial, *The Moore bound for irregular graphs*, Graphs and Combinatorics, 18 (2002), pp. 53–57. doi : 10.1007/s003730200002.

[FNV06] Z. Füredi, A. Naor, and J. Verstraëte, *On the Turán number for the hexagon*, Advances in Mathematics, 203 (2006), pp. 476–496. doi : 10.1016/j.aim.2005.04.011.

[Gav24] C. Gavoille, *Analyse d'algorithme – Cours d'introduction à la complexité paramétrique et aux algorithmes d'approximation*, 2024. http://dept-info.labri.fr/~gavoille/UE-AA/cours.pdf. Notes de cours.

[KSV13] P. Keevash, B. Sudakov, and J. Verstraëte, *On a conjecture of Erdős and Simonovits : Even cycles*, Combinatorica, 33 (2013), pp. 699–732. doi : 10.1007/s00493-013-2863-8.

[LV05] T. Lam and J. Verstraëte, *A note on graphs without short even cycles*, The Electronic Journal of Combinatorics, 12 (2005), p. #R5. doi : 10.37236/1972.

CHAPITRE

4 Complexité de communication

Sommaire

La taille des messages peut avoir un impact non négligeable sur les performances des l'algorithmes, notamment sur leur temps d'exécution. C'est tout l'intérêt de l'étude du modèle CONGEST. Prouver que résoudre un problème distribué donné ne peut se faire sans transmettre une certaine quantité de messages de taille limitée, c'est-à-dire un certain volume de communication, est difficile. Comme nous allons le voir,

dans certains cas, il est possible de conclure en utilisant la *complexité de communication*. Cette notion a été introduite par Yao en 1979 dans « *Some Complexity Questions Related to Distributive Computing* », article qui a valu le prix Turing en 2000 à son auteur [Yao79].

Mots clés et notions abordées dans ce chapitre :

- simulation
- algorithme de détection
- borne inférieure pour Set-Disjointness
- propriété des rectangles, ensemble discriminant
- protocole sans-préfixe

4.1 Introduction

Les bornes inférieures sur la complexité en temps pour le modèle LOCAL s'appliquent bien évidemment au modèle CONGEST, ce dernier pouvant être vu comme une restriction du modèle LOCAL à des messages de taille $O(\log n)$ bits. Mais les arguments que l'on peut utiliser pour le modèle LOCAL sont souvent bien trop faibles pour être intéressants dans le modèle CONGEST. En effet, certains problèmes sont de manière inhérente locaux, comme la détection de « voisins qui se connaissent » par exemple (détection de cycle de longueur trois). Ce problème peut bien évidemment être réalisée en un nombre constant de rondes dans le modèle LOCAL. Cela n'implique malheureusement pas grand chose dans le modèle CONGEST où l'on est loin de savoir le résoudre en un nombre constant de rondes.

Il y a bien sûr des problèmes qui ne sont pas locaux, c'est-à-dire qui même dans le modèle LOCAL nécessitent un nombre de rondes arbitrairement grand. C'est le cas, par exemple, du calcul d'un arbre couvrant de poids minimum qui sera abordé au paragraphe 5.3 du chapitre 5. Il peut nécessiter $\Omega(D)$ rondes pour des graphes de diamètre D, même dans le modèle LOCAL. [*Exercice**. 1 Précisez les entrées/sorties pour le problème du calcul distribué d'un arbre couvrant de poids minimum. 2 Montrez qu'il faut au moins D rondes dans le cas d'un cycle de longueur $n = 2D + 1$ (donc de diamètre D), en supposant que chaque sommet possède la liste de ses arêtes incidentes et leur poids.] [*Exercice**. Montrez que tout problème distribué décidable sur un graphe G possède un algorithme distribué sur G de complexité en temps $O(D)$ dans le modèle LOCAL, où $D = \mathrm{diam}(G)$ est supposé connu de tous les sommets. Montrez qu'en fait, on peut faire $2r+2$ rondes, où r est le rayon de G, sans que r ou D soit préalablement connu des sommets.] Cependant, pour ce problème dans le modèle CONGEST, il est possible de prouver une borne inférieure sur le temps en $\Omega(D + \sqrt{n}/\mathrm{polylog}(n))$, ce qui est au moins $\sqrt{n}/\mathrm{polylog}(n)$, qui vaut même pour les graphes de diamètre $D \geqslant 4$. Ce résultat, qui utilise la complexité de communication, montre une séparation entre les modèles LOCAL et CONGEST. En fait, il a été montré dans [LPSP01], qu'il fallait un temps au moins $\sqrt{n}/\mathrm{polylog}(n)$ pour des graphes de diamètre $D = 4$, et un temps au moins

$n^{1/4}/\text{polylog}(n)$ pour des graphes de diamètre $D = 3$. Cependant, pour les graphes de diamètre $D = 2$, il a été montré que $O(\log n)$ rondes suffisaient, faisant apparaître une autre séparation entre le cas $D = 2$ et $D > 2$.

Dans ce chapitre, on ne démontrera pas de telles séparations pour des problèmes non locaux. On va plutôt s'intéresser à des problèmes plus locaux. À titre d'exemple, citons deux tâches réalisables en temps constant dans le modèle LOCAL mais pas dans le modèle CONGEST :

Problème 1. Chaque sommet d'un arbre enraciné souhaite collecter les identifiants de tous ces petits-fils, c'est-à-dire les fils de ses fils ou encore ses descendants à distance deux.

Problème 2. On souhaite détecter si dans le graphe un sommet possède deux voisins qui ont un autre voisin en commun, c'est-à-dire détecter si le graphe possède ou pas un C_4, un cycle à quatre sommets.

Dans le contexte du calcul distribué, *détecter* une configuration particulière signifie :

1. au moins un sommet la détecte s'il elle se produit quelque part (et ce sommet passe dans un état permanent [1] YES) ; et
2. aucune détection n'ait lieu si elle ne se produit pas (et tous passent dans l'état permanent NO).

Cela correspond au modèle des alarmes incendie dans un bâtiment : un feu est détecté si au moins une alarme se déclenche, et en l'absence de feu aucune alarme n'est autorisée à se déclencher.

De manière plus compact, on peut dire qu'un algorithme distribué *détecte* une configuration si et seulement si au moins un sommet la détecte.

Dans le modèle LOCAL, une solution en deux rondes au problème 1 pourrait être comme ci-dessous, en supposant que l'arbre est connu via les variables PARENT et FILS :

Algorithme Collecter-ID-Petits-Fils
(code du sommet u)

1. $S := \{\text{ID}(u)\}$
2. Pour $i \in \{1, 2\}$ faire :
   (a) NEWROUND
   (b) SEND$(S, \text{PARENT}(u))$
   (c) $S := \bigcup_{v \in \text{FILS}(u)} \{\text{RECEIVE}(v)\}$

1. On veut dire par là que la sortie de l'algorithme (ici YES) peut être lue dans une certaine variable du sommet.

Il est clair d'après ce code qu'après la première ronde chaque sommet u possède dans sa variable S la liste des IDs de ses fils, et qu'après la seconde, u contient dans S l'ensemble des identifiants de ses petits-fils, comme demandé.

[*Exercice**. On considère le problème 1 dans le modèle CONGEST, en supposant que les n sommets ont des identifiants $\{0,\dots,n-1\}$. 1 Montrez qu'il suffit de transmettre au total au plus $n + O(\log n)$ bits par arête. 2 Montrez qu'il est nécessaire de transmettre $\Omega(n)$ bits sur une arête dans le pire des cas. 3 Que devient cette complexité (nombre de bits au total sur une arête) si l'arbre est de hauteur deux ? 4 Un arbre (enraciné) T est dit α-équilibré si le nombre de descendants de deux fils quelconques de tout sommet u sont proches à un facteur α près, c'est-à-dire si $|T_v|/|T_w| \in [1/\alpha, \alpha]$ si $|T_v|, |T_w|$ représentent le nombre de sommets du sous-arbres issu de u. Montrez que si T est α-équilibré et de degré minimum $\geqslant n^c$ (pour les non-feuilles), où $\alpha \geqslant 1$ et $0 < c < 1$ sont deux constantes fixées, alors il suffit de transmettre au plus $o(n)$ bits par arête. Préciser cette quantité (en bits) lorsque $\alpha = 2$ et $c = 1/3$.]

La variante du problème 1 qui consiste à récupérer le plus petit identifiant de ses petits-fils (ou le XOR des identifiants ou n'importe quelle fonction associative et commutative des identifiants) est réalisable en un temps constant dans le modèle CONGEST [*Exercice*. Comment ?] Notons qu'on peut facilement étendre les résultats concernant le problème 1 aux cas des graphes, c'est-à-dire à la variante consistant à collecter les identifiants des sommets à distance deux dans un graphe quelconque. En particulier, on peut montrer qu'il faut et il suffit de $\Theta(n/\log n)$ rondes dans le modèle CONGEST pour collecter les identifiants à distance deux dans un graphe.

[*Exercice**. Montrez que les bornes en $\Theta(n)$ bits par arête de l'exercice précédant pour le problème 1 concernant les arbres peut être appliquées aux cas des graphes. En déduire le nombre optimal de rondes de $\Theta(n/\log n)$ dans le modèle CONGEST. Que devient ce nombre de rondes dans le modèles b-CONGEST ?]

Par contre, on va établir que le problème 2 de la détection de C_3 ou de C_4 dans le modèle CONGEST nécessite de transmettre au moins $\sqrt{n}/\text{polylog}(n)$ bits sur une arête et donc nécessite $\sqrt{n}/\text{polylog}(n)$ rondes dans le pire des graphes (cf. l'Exercice 2). Ce résultat fait appel à la complexité de communication qu'on définira plus formellement dans la section 4.4. Mais dans un premier temps, on va montrer dans la section 4.2 comment détecter un C_3 ou C_4 en temps $O(\sqrt{n})$, toujours dans le modèle CONGEST.

4.2 Détection de C_3 ou C_4

L'objectif est donc de détecter un C_3 ou un C_4 dans un graphe G qui a n sommets. Plus précisément, à la fin de l'algorithme, chaque sommet u doit écrire son résultat dans une variable booléenne $\textsc{Detect}(u) \in \{\textsc{Vrai}, \textsc{Faux}\}$ avec la propriété :

$$G \text{ contient un } C_3 \text{ ou } C_4 \iff \exists u \in V(G),\ \textsc{Detect}(u) = \textsc{Vrai}\,.$$

L'objectif est d'atteindre une complexité en temps de $O(\sqrt{n})$ dans le modèle CONGEST. On montrera plus tard en section 4.5 l'optimalité de cette complexité. Notons la différence de complexité entre connaître les identifiants à distance deux (une variante du problème 1) et le fait de détecter des intersections dans ce même ensemble d'identifiants (le problème 2).

Pour simplifier, on va présenter l'algorithme dans le modèle b-CONGEST lorsque $b = O(\sqrt{n}\log n)$ bits. On en déduira automatiquement un algorithme dans le modèle CONGEST classique, c'est-à-dire avec $b = O(\log n)$, en multipliant le temps par un facteur $\sqrt{n}$, grâce à la proposition suivante.

Proposition 4.1 *Soit A un algorithme distribué sur un graphe G en mode synchrone utilisant des messages d'au plus b bits. Pour tout entier $b' \in [1, b]$, il existe un algorithme A′ calculant les mêmes sorties que A sur G et utilisant des messages d'au plus b′ bits tel que :*

- $\textsc{Temps}(A', G) = \lceil b/b' \rceil \cdot \textsc{Temps}(A, G)$;
- $\textsc{Message}(A', G) = \lceil b/b' \rceil \cdot \textsc{Message}(A, G)$.

Preuve. On construit A′ à partir de A, connaissant b et b'. Posons $k := \lceil b/b' \rceil$. Chaque ronde i de A est subdivisée en k sous-rondes notées $i_1, \ldots, i_k$ pour A′. Puis, chaque message M à destination d'un voisin v voisin de u est découpé en k sous-messages $M_1, \ldots, M_k$ chacun de b' bits. Les derniers sous-messages peuvent être vides si la taille de M n'est pas assez grande. Le sous-message M_j est alors envoyé à la sous-ronde i_j. La réception en v de M_j s'effectue lors de la sous-ronde i_j. La sous-ronde i_k est chargée de fusionner tous les sous-messages $M_1 \circ \cdots \circ M_k$ reçus afin d'obtenir le message M. La partie Compute de la ronde i est effectuée lors de la dernière sous-ronde i_k, lorsque tous les messages reçus ont été réassemblés.

Il est clair que les calculs (et donc les sorties) de A et de A′ sur G sont identiques (en fait les calculs de A′ aux sous-rondes i_k, en particulier la sortie de l'algorithme). Les complexités en temps et en nombre de messages pour A′ ont été multipliées par k par rapport à A. □

Pour simplifier la présentation, on supposera que $\mathrm{ID}(u) = u$. Les rondes étant clairement distinguées, les NewRound sont superflues. Pour les illustrations, on va associer une couleur à chaque sommet u suivant le résultat de la première ronde : bleu si $b_u = \textsc{Vrai}$ et rouge sinon. Voir la figure 4.1.

Algorithme Detect_C$_{3,4}$
(code du sommet u)

Ronde 1

- SendAll(u)
- $S := \bigcup_{v \in N(u)} \{\textsc{Receive}(v)\}$

- $b := (|S| < \sqrt{n})$ ($b \in \{\text{Vrai}, \text{Faux}\}$, u est bleu ou rouge)

Ronde 2

- SendAll(b) (envoie la couleur de u)
- $B := \bigcup_{v \in N(u)} \{\text{Receive}(v)\}$ ($B = \{b_v : v \in N(u)\}$)
- $R := \{v : b_v \in B \text{ et } b_v = \text{Faux}\}$ (sélectionne les voisins rouges)
- $r := (|R| < \sqrt{n})$ ($r \in \{\text{Vrai}, \text{Faux}\}$)

Ronde 3 (ou Phase 3)

- Si $r = \text{Faux}$, $R := \varnothing$ (si trop de voisins rouges)
- Si $b = \text{Vrai}$, SendAll(S) sinon SendAll(R) (envoie S ou R suivant la couleur)
- $\mathcal{T} := \bigcup_{v \in N(u)} \{\text{Receive}(v)\}$ ($\mathcal{T} = \{T_v : v \in N(u), T_v = S_v \text{ ou } R_v\}$)
- Detect $:= (\neg r) \vee (\exists T_v \neq T_w \in \mathcal{T},\ |(T_v \cup \{v\}) \cap (T_w \cup \{w\})| > 1)$ (fixe Detect(u))

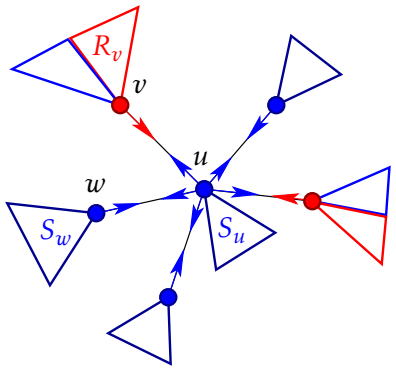


Figure 4.1 – Illustration de la ronde 3 de l'algorithme Detect_C$_{3,4}$. Les échanges de voisinages diffèrent suivant la couleur des sommets (ici un sommet u bleu).

On note qu'on suppose la valeur de n connue de tous les sommets. Mais, comme on va le voir, n'importe quel majorant $M \geqslant \sqrt{n}$ suffit pour garantir de fonctionnement de l'algorithme avec des messages d'au plus $(M-1)\lceil \log n \rceil$ bits, en remplaçant les deux occurrences de « $\sqrt{n}$ » par M.

Le reste de la section est consacré à l'analyse de l'algorithme Detect_C$_{3,4}$.

Tous les messages sont d'au plus $O(\sqrt{n} \log n)$ bits. En effet, ils contiennent :

- pour la ronde 1, un seul identifiant, soit $\lceil \log n \rceil$ bits ;
- pour la ronde 2, un seul bit ; et
- pour la ronde 3, au plus $\lceil \sqrt{n} \rceil - 1$ identifiants, soit $O(\sqrt{n} \log n)$ bits.

Il faut montrer que :

1. Si Detect(u) = Vrai, alors G contient un C_3 ou C_4 ;
2. Si G contient un C_3 ou C_4, alors il existe un sommet u tel que Detect(u) = Vrai.

Pour le premier point, on suppose donc que Detect(u) = Vrai. Il y a deux cas.

Cas 1a. Si r = Vrai. C'est que l'intersection $(T_v \cup \{v\}) \cap (T_w \cup \{w\})$, pour deux voisins v, w de u, contient en plus de u un autre sommet. Cet autre sommet permet de former un C_3 ou C_4 dans G car les ensembles $T_v \in \mathcal{T}$, $T_v \in \{S_v, R_v\}$, correspondent, dans tous les cas (même si $R_v = \varnothing$), à des voisinages (éventuellement partiels) de G.

Cas 1b. Si r = Faux. On est alors en présence d'un sommet u qui possède $|R_u| \geqslant \sqrt{n}$ voisins rouges (en particulier u doit être rouge), chacun ayant $|S_v| \geqslant \sqrt{n}$ voisins. Voir la figure 4.2.

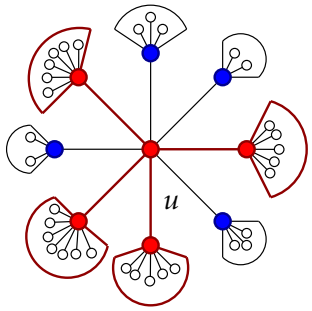


Figure 4.2 – Contradiction sur le nombre de sommets des voisins rouges d'un sommet u ayant r = Faux, alors qu'il n'y a pas de C_3 ou C_4.

Cela implique qu'il existe un C_3 ou C_4 passant par u, sinon les voisinages des sommets rouges voisins de u deviennent deux à deux disjoints (en ignorant u commun à tous les voisinages). Dans ce cas, le nombre de rouge de sommets de G serait au moins de :

$$1 + |R_u| + \sum_{v \in R_u} (|S_v| - 1) \;\geqslant\; 1 + \sqrt{n} + \sqrt{n} \times (\sqrt{n} - 1) \;=\; 1 + n$$

en contradiction avec le fait G possède n sommets. On note que l'argument tient en remplaçant « $\sqrt{n}$ » par n'importe quelle valeur $M \geqslant \sqrt{n}$.

Pour le deuxième point, on suppose donc que G possède un cycle C_t de longueur $t \in \{3, 4\}$. Il y a plusieurs cas à considérer, suivant la couleur des sommets de C_t. On peut supposer r_u = Vrai pour chaque sommet u de C_t, puisque sinon Detect(u) = Vrai à cause du terme « $\neg r$ » dans la dernière ligne de la ronde 3. En particulier, d'après la Ronde 3 de l'algorithme, les sommets bleus ont diffusé à leurs voisins leur voisinage complet et les sommets rouges tout leur voisinage rouge.

Il y a deux cas (voir la figure 4.3).

Cas 2a. Tous les sommets de C_t sont bleus. Dans ce cas, il existe un sommet u voisin de deux sommets bleus v, w. Alors Detect(u) = Vrai puisque les voisinages S_v, S_w complets vont être reçus par u à la ronde 3 qui va donc détecter le sommet z et C_t qui se trouve nécessairement dans leurs intersections.

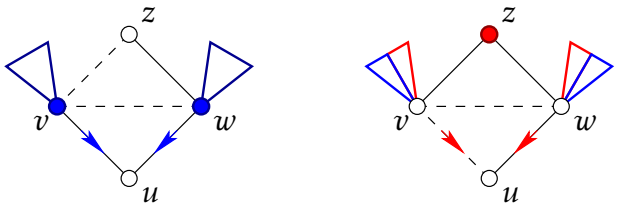


FIGURE 4.3 – Les deux cas de coloration des sommets d'un cycle C_t avec $t \in \{3,4\}$: ils sont tous bleus ou pas. Suivant la valeur de t et le cas de figure, $z = v$ (à gauche) ou $u = v$ (à droite) sont possibles. Dans tous les cas, u va détecter z dans l'intersection des voisinages transmis par v et w.

Cas 2b. Il existe au moins un sommet rouge dans C_t, disons z. Dans ce cas, on considère le sommet u qui est à l'opposé de z si $t = 4$ ou bien le sommet v si $t = 3$ en posant $u = v$. Alors DETECT(u) = VRAI puisque les voisinages R_v et R_w vont être reçus par u à la ronde 3 (rappelons que $r_v = r_w =$ VRAI), et bien sûr $z \in R_v \cap R_w$ puisque z est rouge. En fait, suivant les couleurs de v et w, u peut recevoir les sur-ensembles $S_v \supseteq R_v$ et $S_w \supseteq R_w$. Dans tous les cas, u va détecter C_t qui se trouve nécessairement dans cette intersection car z est rouge.

Ceci démontre la validité de l'algorithme Detect_C$_{3,4}$. Ainsi, nous avons montré :

Proposition 4.2 *Dans le modèle CONGEST, il est possible de détecter un C_3 ou C_4 en temps $O(\sqrt{n})$.*

Notons que le problème problème 1 généralisé aux graphes, c'est-à-dire donnant pour chaque sommet la liste des ID des sommets à distance deux, permettrait de résoudre le problème de détection de C_3 ou C_4 avec $\Theta(n/\log n)$ rondes. Cependant, si cela suffit, un sommet n'a pas forcément besoin de connaître les ID de ses voisins à distance deux, même si cela paraît nécessaire pour déterminer l'intersection (ou son abscence) des voisinages de ses voisins. La sortie des deux problèmes n'est pas exactement la même (liste d'ID pour problème 1 *versus* booléenne pour la détection), ce qui produit possiblement des solutions différentes aux performances différentes.

[*Exercice*. Montrez que, de manière générale, dans le modèle b-CONGEST on peut détecter un C_3 ou C_4 en temps $O(\sqrt{n}/b \cdot \log n)$.] [*Exercice**. Discutez d'une extension de l'algorithme Detect_C$_{3,4}$ permettant de détecter un C_3.]

4.3 Argument de simulation

La complexité de communication est un outil permettant de déterminer le nombre de bits à échanger entre plusieurs individus (traditionnellement Alice et Bob) afin de

résoudre une tâche donnée. C'est la version épurée du calcul distribué : deux sommets connectés par une arête, cf. figure 4.4.

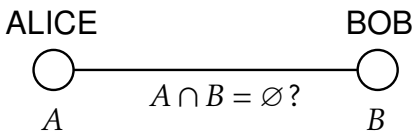


FIGURE 4.4 – Complexité de communication : combien de bits Alice et Bob doivent-ils s'échanger pour déterminer si leur ensemble privé s'intersectent ?

Un exemple de tâche à résoudre est la suivante : Alice et Bob possèdent chacun un sous-ensemble $A, B \subseteq \{0, \dots, n-1\}$. La question est de savoir combien de bits Alice et Bob ont-ils besoin de s'échanger pour savoir si A et B s'intersectent ? (Bien sûr, Alice et Bob ont la liberté de se mettre d'accord sur un protocole avant de recevoir chacun leur sous-ensemble privé.) C'est le problème SET-DISJOINTNESS pour lequel il a été établit que la complexité de communication était de $\Theta(n)$ bits [2]. Ce problème a été beaucoup étudié [She14]. On reviendra sur ce résultat dans la section 4.5.

L'idée générale de la réduction du problème distribué au problème de communication est un argument de simulation. On suppose qu'Alice et Bob disposent de l'algorithme distribué supposé résoudre une certaine tâche dans le modèle CONGEST. Ils simulent alors l'exécution de l'algorithme chacun sur une partie spécifique d'un graphe particulier, dépendant des entrées d'Alice et Bob (voir figure 4.5). De cette simulation, Alice et Bob en déduisent un protocole permettant de résoudre le problème de communication : les messages circulant sur les arêtes entre les parties, les sommets contrôlés par ceux d'Alice et ceux de Bob, donnent les messages du protocole de communication ; et les messages internes et les calculs locaux définissent les calculs privés d'Alice et Bob.

Si le problème de communication nécessite au moins k bits, c'est-à-dire sa complexité de communication est au moins k, alors une des, disons p, arêtes connectant la partie d'Alice à celle de Bob doit transmettre au moins k/p bits durant l'exécution de l'algorithme. Cela implique une borne inférieure de $(k/p)/\text{polylog}(n)$ rondes dans le modèle CONGEST puisqu'en une ronde, le long d'une arête, on ne peut transmettre qu'au plus $O(\log n)$ bits.

Pour l'anecdote, on ne connaissait toujours pas en 2025 la complexité du problème de la détection d'un triangle dans le modèle CONGEST. On pense qu'il n'est pas possible de le faire dans tout graphe en temps constant, mais on en a pas la preuve. Il est possible cependant de montrer qu'il faut transmettre $\Omega(\log n)$ bits sur une arête pour

2. Le résultat reste vrai même pour un protocole de communication probabiliste (= utilisation de bits aléatoires communs et probabilité de succès $> 1/2$), cf. [HW07]. Dans le cas déterministe la borne est n, cf. [AB16, page 273] (on verra que c'est précisément $n+1$). Dans le cas quantique, par contre, la borne est de $\tilde{\Theta}(\sqrt{n})$ qubits. Mais c'est une autre histoire [BGK+18]. En fait, dans le cas quantique la borne est de $\tilde{\Theta}(n/r + r)$ pour r est le nombre autorisé de rondes de communications entre Alice et Bob.

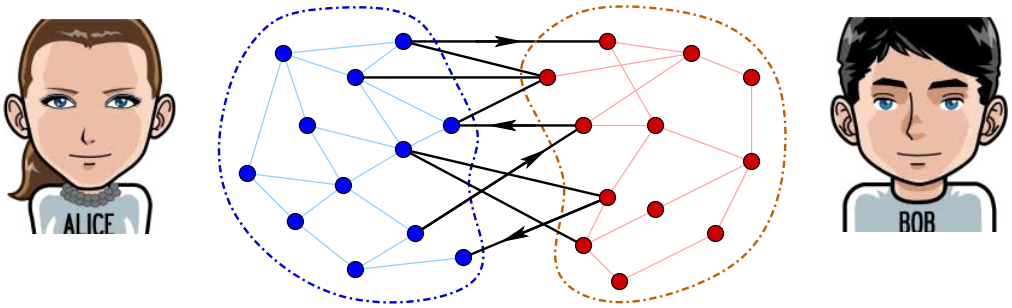


FIGURE 4.5 – Protocole de communication entre Alice et Bob déduit de la simulation d'un algorithme distribué dans le modèle CONGEST.

distinguer un triangle d'un hexagone [3]. Pour des travaux reliés à ce problème, se reporter à [FGKO18][EFF+22][FLMT24][Luc24].

Notons que la complexité du problème de détection d'un triangle dans un graphe par un algorithme séquentiel n'est pas non plus connue : c'est entre n^2 et $n^{2.3\ldots}$.

On cache ici certains détails dans le terme polylog(n), l'équivalent du « tapis » donc. Parmi ces détails, il y a par exemple le fait que dans la simulation, l'algorithme distribué peut utiliser le temps puisqu'il est synchrone, alors qu'Alice et Bob ne le peuvent pas. Pour être exact, ils peuvent le faire seulement entre les nœuds de la partie qu'ils simulent. Ils doivent donc spécifier le numéro de ronde, mais aussi les ID des extrémités des arêtes connectant Alice et Bob. En effet, l'algorithme distribué à simuler pourrait envoyer à la ronde t un premier message de u à v du type : « le laps de temps écoulé jusqu'à mon prochain message déterminera son contenu » et puis un second message au top $t + x$ du type : « voici mon message » ou « ça y est ! ». Le voisin v en déduit x par différence des tops horloge. Pourtant le nombre de bits échangés entre u et v est constant, alors que la valeur x transmise via l'horloge peut être arbitrairement longue et coder n'importe quelle information. Pour que les joueurs puissent réaliser correctement la simulation, il faut donc ajouter le numéro de ronde (de l'algorithme distribué simulé) à chaque message sur les arêtes Alice-Bob, ainsi que l'identifiant des extrémités des arêtes (pour qu'Alice et Bob s'y retrouvent). Bref, si le temps total de l'algorithme est polynomial [4], cette simulation ajoutera $O(\log n)$ bits à chacun des messages.

On va revenir maintenant sur la complexité de communication à proprement parlée. En particulier on va montrer que pour SET-DISJOINTNESS de taille n Alice et Bob doivent s'échanger au moins $n + 1$ bits, ce qui est optimal.

3. Avec l'hypothèse que les identifiants sont dans $\{0, \ldots, n-1\}$, un espace d'identifiants plus important rendant plus grandes les bornes inférieures.

4. Bien évidemment, si le temps est plus grand que polynomial, l'affaire est pliée puisqu'on veut démontrer que ce temps n'est pas trop petit. De manière générale, il faut ajouter $O(\log T)$ bits à chaque message, où T est la borne inférieure que l'on veut rajouter.

4.4 Définition

Pour faire simple, les problèmes de communication auxquels on s'intéresse reviennent à calculer une fonction booléenne $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ à l'aide d'un protocole de communication entre deux parties, traditionnellement Alice et Bob. On peut généraliser à des ensembles plus généraux et des fonctions $f : X \times Y \to Z$, et aussi à des problèmes de communication impliquant plus de deux parties.

Dans ce contexte, Alice connaît l'entrée x, Bob l'entrée y, et bien sûr ils ont comme connaissance commune un protocole qui va leur permettre de calculer $f(x,y)$. Ils disposent d'une puissance de calcul arbitraire, sauf qu'ils ne peuvent pas communiquer en dehors de l'application du protocole. Bien évidemment, chacun ignore l'entrée de l'autre, et l'exécution du protocole doit faire en sorte que le dernier message échangé soit $f(x,y)$ sachant que c'est Alice qui commence. Voir la figure 4.6 pour un exemple.

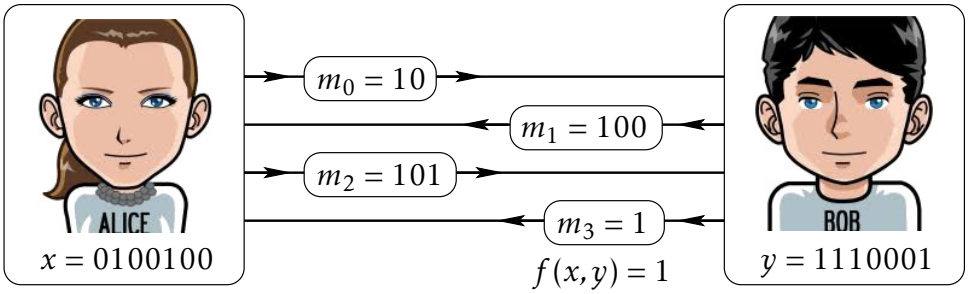


FIGURE 4.6 – Exemple de protocole de communication entre Alice et Bob pour le calcul d'une certaine fonction booléenne f. Ici, Alice et Bob doivent déterminer si leurs entrées respectives ont un « 1 » à une même position. Si c'est le cas $f(x,y) = 1$, sinon $f(x,y) = 0$. Le protocole Π qu'ils appliquent est le suivant : chacun envoie tour à tour la prochaine position de leurs bits à 1 susceptible d'être commun (en partant de 0) jusqu'à soit trouver une position commune (et le premier qui la détecte envoie 1), soit ne plus avoir de position à 1 (et le premier qui s'en aperçoit envoie 0). La suite des messages échangés, $(m_0, m_1, m_2, m_3) = (10, 100, 101, \boxed{1})$, appelée *script*, permet de calculer $f(x,y) = \boxed{1}$ dans l'exemple. [*Exercice.* Déterminez toutes les entrées possibles d'Alice et de Bob qui auraient chacune produit le même script.]

Pour raccourcir un peu la définition de [AB16], à chaque ronde $i = 0, 1, \ldots$, on notera $J(i) \in \{\text{Alice}, \text{Bob}\}$ le joueur censé émettre le message de la ronde i et $I(i) \in \{x, y\}$ l'entrée du joueur $J(i)$. Bien évidemment, on a $J(i) \neq J(i+1)$ pour tout i. D'après notre convention, $J(0) = \text{Alice}$ et $I(0) = x$. Attention ! c'est un peu le mode *talky-walky* où chacun parle à son tour. À la ronde i, c'est $J(i)$ qui parle à $J(i+1)$ qui l'écoute sagement. Il ne s'agit pas de rondes synchrones comme dans le modèle LOCAL où les messages d'une même ronde se croisent.

La définition la plus (trop ?) simple que l'on peut trouver pour un protocole de com-

munication est la suivante, la définition originale de Yao de 1979 étant moins intuitive. On y reviendra cependant dans la section 4.6.

13.1 DEFINITION OF TWO-PARTY COMMUNICATION COMPLEXITY

Now we formalize the informal description of communication complexity given above.

Definition 13.1 *(Two-party communication complexity)* Let $f : \{0,1\}^{2n} \to \{0,1\}$ be a function. A *t-round two-party protocol* Π *for computing* f is a sequence of t functions $P_1, \ldots, P_t : \{0,1\}^* \to \{0,1\}^*$. An execution of Π on inputs x, y involves the following: Player 1 computes $p_1 = P_1(x)$ and sends p_1 to Player 2, Player 2 computes $p_2 = P_2(y, p_1)$ and sends p_2 to Player 1, and so on. Generally, at the ith round, if i is odd, then Player 1 computes $p_i = P_i(x, p_1, \ldots, p_{i-1})$ and sends p_i to Player 2, and similarly if i is even then Player 2 computes $p_i = P_i(y, p_1, \ldots, p_{i-1})$ and sends p_i to Player 1.

The protocol Π is valid if for every pair of inputs x, y, the last message sent (i.e., the message p_t) is equal to the value $f(x, y)$. The *communication complexity* of Π is the maximum number of bits communicated (i.e., maximum of $|p_1| + \ldots + |p_t|$) over all inputs $x, y \in \{0,1\}^n$. The *communication complexity* of f, denoted by $C(f)$ is the minimum communication complexity over all valid protocols Π for f.

For every function, $C(f) \leq n+1$ since the trivial protocol is for first player to communicate his entire input, whereupon the second player computes $f(x, y)$ and communicates that single bit to the first. Can they manage with less communication?

FIGURE 4.7 – Définition comme donnée dans [AB16], extrait de la page 271.

La définition 4.1 ci-après ne fait que paraphraser la définition reproduite sur la figure 4.7.

Définition 4.1 *Un protocole Π pour f est une suite de fonctions $\{P_i\}_{i \in \mathbb{N}}$ telles que pour toutes entrées $x, y \in \{0,1\}^n$ et toute ronde* [5] *$i \in \mathbb{N}$:*

- $P_i : \{0,1\}^n \times (\{0,1\}^*)^i \to \{0,1\}^*$;
- *$J(i)$ envoie à $J(i+1)$ le message $m_i = P_i(I(i), m_0, \ldots, m_{i-1})$; et*
- *le dernier message envoyé est la valeur de $f(x, y)$.*

Les fonctions P_i indiquent quel message doit être transmis à la ronde i par le joueur $J(i)$, message qui dépend de l'entrée du joueur $J(i)$ et de tous les messages déjà échangés. Dans cette définition, il n'y a pas de notion d'horloge et encore moins de délais de transmission des messages. Chaque fonction P_i indique le message qu'il doit être transmis et c'est tout. Le receveur $J(i+1)$ reçoit le message et calcule P_{i+1}. En quelque sorte les messages passent de l'envoyeur au récepteur sans délais ni erreur.

Définition 4.2 *Le* coût de communication *d'un protocole Π pour une fonction booléenne f est la somme des longueurs des messages envoyés (en bits) pour les pires entrées x, y.*

5. Contrairement à la définition de [AB16, page 271], on supposera que i commence à 0.

D'après, ces définitions, il est toujours possible de garantir un coût de communication de $n+1$ pour toute fonction f : il suffit à Alice de transmettre entièrement son entrée x à Bob qui renvoie à Alice $f(x,y)$ comme second et dernier message. C'est le protocole *trivial*.

[*Exercice*. Quel est le coût de communication, en fonction de la taille n des entrées, pour le protocole donné dans l'exemple de la figure 4.6 ?]

Tout comme la définition de [AB16, page 271] (voir figure 4.7), on aimerait bien définir la complexité de communication de f comme le plus petit coût de communication d'un protocole pour f. On la définira précisément plus tard dans la section 4.6, car comme on va le voir, la bonne définition contient une subtilité. En effet, la definition 4.1 n'interdit pas les messages vides (de coût nuls) par exemple, et autorise des protocoles ambigus quant à leur terminaison. Mais d'ici là, on en aura pas besoin.

Il faut retenir que la complexité de communication a été introduit essentiellement pour établir des bornes inférieures sur le nombre de bits à transmettre pour résoudre une tâche. Il n'a pas vraiment de vocation à construire de « vrais » protocoles de communications (bornes supérieures).

4.5 Bornes inférieures pour Set-Disjointness

Le problème Set-Disjointness est un problème de communication qui consiste à déterminer si deux sous-ensembles $A, B \subseteq \{0,\dots,n-1\}$ sont disjoints ou pas. On notera, pour tout $x,y \in \{0,1\}^n$,

$$disj_n(x,y) \;:=\; \neg\left(\bigvee_{i=0}^{n-1} x[i]\wedge y[i]\right) = \bigwedge_{i=0}^{n-1} \neg(x[i]\wedge y[i])$$

la fonction booléenne qui renvoie 1 si et seulement si x et y n'ont pas de 1 en commun, c'est-à-dire s'il n'existe pas de position $i \in \{0,\dots,n-1\}$ telle que $x[i]=y[i]=1$. Résoudre Set-Disjointness ou calculer $disj_n(x,y)$ pour tout x,y est le même problème. En effet, il suffit de coder par $x[i]=1$ le fait que l'entier $i \in A$ et $x[i]=0$ sinon. De même $y[j]=1$ ssi $j \in B$. On dit que x est le *vecteur caractéristique* de A, et y celui de B. Notons que l'exemple du protocole de la figure 4.6 avait pour vocation de calculer $\neg disj_7(x,y)$.

Étant donné un protocole Π pour f, on appelle « script » pour l'entrée (x,y) la suite

$$\text{script}_\Pi(x,y) \;:=\; (m_0, m_1, \dots, m_t)$$

des messages envoyés lors de l'exécution de Π, le message m_i étant envoyé par le joueur $J(i)$. En particulier, $m_t = f(x,y)$ (cf. définition 4.1). Les protocoles de communication induisent des scripts qui vérifient la propriété suivante dite du « rectangle ».

Propriété 4.1 *Si* $\text{script}_\Pi(x,y) = \text{script}_\Pi(x',y')$, *alors* $\text{script}_\Pi(x',y) = \text{script}_\Pi(x,y') = \text{script}_\Pi(x,y)$.

Dit autrement Π produit toujours, via ses scripts, une partition en *rectangles monochromatiques* de la *matrice* de f, soit $((f(x,y)))_{x,y\in[0,2^n[}$. Il s'agit de la matrice booléenne de dimension $2^n \times 2^n$ composée des valeurs $f(x,y)$ pour les $2^n \cdot 2^n = 2^{2n}$ entrées possibles (x,y) de f, cf. la figure 4.8. Ici un *rectangle monochromatique* est une sous-matrice [6] de la matrice de f où f prend la même valeur.

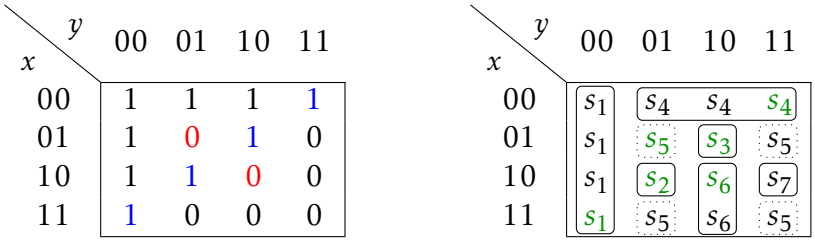


FIGURE 4.8 – Matrice de la fonction $disj_n$ pour $n = 2$. La matrice de gauche représente les valeurs $disj_2(x,y)$, et celle de droite une partition possible en 7 rectangles monochromatiques $s_1, \ldots, s_7$, correspondant à autant de scripts potentiels pour cette même fonction. Cette partition ne dit pas s'il existe un protocole utilisant précisément cette partition, ou si seuls 7 scripts suffisent. Cependant, les 6 entrées colorées (en rouge ou bleu à gauche et en vert à droite) doivent nécessairement correspondre à des scripts deux à deux différents. [*Exercice.* Montrez qu'il faut au moins 3 scripts différents pour partitioner les valeurs 0 en rectangles monochromatiques. En déduire qu'il faut au minimum 7 scripts pour cette fonction.]

Notons qu'avoir des scripts égaux pour des entrées différentes n'est pas si exceptionnel que cela. Par exemple, pour le protocole trivial Π_0 où Alice envoie à Bob son entrée x et Bob renvoie l'égalité $eq_n(x,y)$, on va avoir $\text{script}_{\Pi_0}(x,y) = \text{script}_{\Pi_0}(x,y') = (x,0)$ pour toutes les entrées $y' \neq y$, ce qui est plutôt fréquent (il y a $2^n - 1$ tels y'). Voir aussi l'exemple de la figure 4.6. En fait, l'égalité des scripts est plutôt la règle que l'exception, en particulier lorsque les scripts sont relativement courts par rapport aux entrées. [*Exercice.* Pourquoi dans tout protocole « efficace », c'est-à-dire utilisant moins de scripts que le protocole trivial Π_0, un des ses scripts doit apparaître au moins 2^{n-1} fois dans la matrice de f ?]

Preuve. On le montre par induction sur le nombre de messages échangés. En particulier, au départ, on remarque qu'Alice sachant x envoie un message m_0 que l'entrée de Bob soit y ou y', puisqu'elle ne la connaît pas.

Dit autrement, si le script pour (x,y) commence par m_0, alors cela sera le cas pour (x,y') puisque $m_0 = P_0(x,\epsilon)$ ne dépend pas de l'entrée de Bob. Et si le script pour (x',y')

6. Une sous-matrice est obtenue à partir de la matrice initiale en sélectionnant un sous-ensemble de lignes et de colonnes, pas forcément contigu comme s_5 dans la figure 4.8.

commencent aussi par $m_0 = P_0(x', \epsilon)$, cela sera le cas pour (x', y) puisque m_0 ne dépend pas de l'entrée de Bob. Donc si (x, y) et (x', y') commencent par m_0 comme précisé par l'hypothèse, alors cela sera le cas aussi pour (x', y) et (x, y').

Puis, Bob sachant y et m_0 enverra son message $m_1 = P_1(y, m_0)$ que l'entrée d'Alice soit x ou x', puisque dans les deux cas Alice a envoyé m_0 à cause de l'égalité précédente. Et si le script pour (x', y') continue aussi par $m_1 = P_0(y', m_0)$, cela sera le cas aussi pour (x, y'), puisqu'on l'a vu, m_0 est identique que l'entrée l'Alice soit x ou x'. Et ainsi de suite à chaque ronde.

Montrons le plus formellement. Pour simplifier l'écriture, on notera pour toute entrée (u, v) :

$$\begin{aligned} m_i(u,v) &= \text{script}_\Pi(u,v)[i] \\ s_i(u,v) &= m_0(u,v), m_1(u,v), \ldots, m_i(u,v) \end{aligned}$$

c'est-à-dire le i-ème message ainsi que la suite des $i+1$ premiers messages de $\text{script}_\Pi(u, v)$. Comme avant la ronde 0 aucun message n'a été envoyé, on posera par extension $s_i(u, v) = \epsilon$ si $i = -1$.

Supposons que $\text{script}_\Pi(x, y) = \text{script}_\Pi(x', y')$. On va montrer, par induction sur i, que $s_i(x', y) = s_i(x, y') = s_i(x, y)$ pour tout $i \in \{0, \ldots, t\}$ où $t+1$ est le nombre de messages du script, ce qui suffit à démontrer le résultat souhaité puisque $\text{script}_\Pi(x, y) = (s_t(x, y))$. Bien évidemment, $\text{script}_\Pi(x, y) = \text{script}_\Pi(x', y')$ implique $s_i(x, y) = s_i(x', y')$ pour tout i.

La propriété est vérifiée pour $i = -1$. En effet, $s_i(x', y) = s_i(x, y') = s_i(x, y) = \epsilon$ dans ce cas. Supposons que pour tout $i \geqslant 0$, $s_{i-1}(x', y) = s_{i-1}(x, y') = s_{i-1}(x, y)$. Comme $s_i(u, v) = s_{i-1}(u, v), m_i(u, v)$, il suffit de montrer que $m_i(x', y) = m_i(x, y') = m_i(x, y)$.

Supposons que i soit pair. Alors pour l'entrée (x, y) :

$$\begin{aligned} m_i(x,y) &= P_i(x, s_{i-1}(x,y)) && \text{(par définition)} \\ &= P_i(x, s_{i-1}(x,y')) && \text{(par hypothèse)} \\ &= m_i(x,y') && \text{(par définition)} \end{aligned}$$

On a également pour l'entrée (x', y') :

$$\begin{aligned} m_i(x',y') &= P_i(x', s_{i-1}(x',y')) && \text{(par définition)} \\ &= P_i(x', s_{i-1}(x',y)) && \text{(par hypothèse)} \\ &= m_i(x',y) && \text{(par définition)} \end{aligned}$$

Comme $\text{script}_\Pi(x, y) = \text{script}_\Pi(x', y')$, $m_i(x, y) = m_i(x', y')$. Il suit que $m_i(x', y) = m_i(x, y') = m_i(x, y)$ comme souhaité. Le cas i impair est similaire en remplaçant $P_i(x, \ldots)$ par $P_i(y, \ldots)$ et $P_i(x', \ldots)$ par $P_i(y', \ldots)$. □

Ce qui augmente le coût d'un protocole, c'est le nombre total de scripts qu'il utilise pour calculer f sur toutes les entrées (x, y) possibles. C'est assez clair, au moins intuitivement. Imaginons que pour calculer f, sur toutes les entrées possibles, seuls deux

scripts sont nécessaires, disons s_1 et s_2. Alors, sachant cela, Alice et Bob pourraient se mettre d'accord et s'arranger pour que chacun des scripts soit courts. [*Question.* Pourquoi, dans ce cas précis, ils peuvent s'arranger pour ne communiquer qu'au plus deux bits, en supposant $s_1 = (m_0, ..., m_t)$ et $s_2 = (n_0, ..., n_r)$ et $t \leqslant r$.] Notez bien que ce codage peut être déterminer à l'avance puisqu'Alice et Bob connaissent f et l'ensembles des entrées possibles (et donc la partition en scripts de la matrice).

Et inversement, et toujours intuitivement, un protocole de coût k ne pourra posséder que qu'au plus 2^k scripts différents. Donc si on arrive à montrer que *tout* protocole utilise au moins 2^k scripts différents on aura une borne inférieure sur son coût. C'est en tout cas l'intuition, car, comme on va le voir plus précisément dans la section 4.6, il existe un petit écart entre le « codage » d'un script et son « coût ».

La réciproque de la propriété 4.1 permet de se passer d'un protocole particulier et ainsi d'exhiber une propriété intrinsèque de f, en particulier le nombre minimum de scripts nécessaires pour calculer f. En effet, si $f(x,y) = f(x',y')$ et que $f(x',y) \neq f(x,y)$ ou $f(x,y') \neq f(x,y)$, alors nécessairement $\text{script}_\Pi(x,y) \neq \text{script}_\Pi(x',y')$ pour tout protocole Π. Voir la matrice de gauche sur la figure 4.8.

On en vient naturellement à la notion suivante, dite d'*ensemble discriminant* pour la valeur v (*fooling set* en Anglais). Il s'agit d'un ensemble d'entrées qui deux à deux violent la propriété des rectangles. Plus formellement, pour un certain $v \in \{0,1\}$, c'est un sous-ensemble $F_v \subseteq \{0,1\}^n \times \{0,1\}^n$ d'entrées pour f tel que :

$$\forall (x,y),(x',y') \in F_v, \quad f(x,y) = f(x',y') = v \ \text{ et } \ \bar{v} \in \{f(x',y), f(x,y')\} \ .$$

Il est alors clair que, pour tout $v \in \{0,1\}$, le nombre total de scripts de tout protocole pour f est au moins $|F_v|$ puisque deux à deux les entrées de F_v doivent créer des scripts différents. Bien évidemment, les scripts pour F_v et $F_{\bar{v}}$ doivent être différent. On a donc,

Proposition 4.3 *Si F_0 et F_1 sont des ensembles discriminants pour f pour des valeurs différentes, alors le nombre de scripts de tout protocole pour f est au moins $|F_0| + |F_1|$.*

Notons que des ensembles discriminants permettent de prouver au mieux qu'il existe 2^{n+1} scripts différents, car on ne pourra jamais avoir (x,y) et (x,y') comme éléments d'un même ensemble discriminant. [*Question.* Pourquoi ?].

Par exemple, pour la fonction $disj_2$, on peut vérifier que l'ensemble des entrées

$$\{\,(00,11),\ (01,10),\ (10,01),\ (11,00)\,\}$$

correspondant à la diagonale montante est un ensemble de discriminant pour la valeur 1. C'est lié au fait qu'en dessous de cette diagonale, toutes les valeurs de $disj_2$ sont nulles alors que sur la diagonale elles valent 1. On a également l'ensemble des entrées

$$\{\,(01,01),\ (10,10)\,\}$$

qui est discriminant pour la valeur 0. Plus généralement, montrons que :

Proposition 4.4 *Tout protocole de communication pour $disj_n$ possède au moins 2^n+n scripts différents.*

Preuve. Soient

$$F_1 := \{ (x,\bar{x}) : x \in \{0,1\}^n \} \quad \text{et} \quad F_0 := \left\{ (x,x) : x = 0^i 1 0^{n-i-1}, i \in \{0,\dots,n-1\} \right\}.$$

Clairement, x et $\bar{x}$ sont toujours disjoints, et donc $disj_n$ vaut 1 pour les entrées $(x,y) \in F_1$. De même, x n'est jamais disjoint de lui-même dès que x possède au moins un bit à 1, et donc $disj_n$ vaut 0 pour les entrées $(x,y) \in F_0$. Montrons qu'ils sont discriminants.

Pour F_1. Soient $(x,\bar{x}) \neq (x',\bar{x'}) \in F_1$. Puisque $x \neq x'$, il existe une position i où x et x' diffère, disons par exemple $x[i] = 0$ et $x'[i] = 1$. Il suit que $\bar{x}[i] = 1 = x'[i]$, et donc que x' et $\bar{x}$ ne sont pas disjoints, c'est-à-dire $disj_n(x',\bar{x}) = 0$.

Pour F_0. Si $(x,x) \neq (x',x') \in F_0$, alors x et x' sont disjoints, c'est-à-dire $disj_n(x,x') = 1$, car x et x' n'ont qu'un seul bit à 1 et sont différents.

On vérifie facilement que $|F_1| = 2^n$ et $|F_0| = n$. □

[*Exercice.* Montrez que pour la fonction $disj_n$, tout ensemble discriminant pour la valeur 0 de la forme $F_0 = \{(x,x) : x \in W\}$, pour un certain ensemble W, est de taille au plus n, alors qu'on pourrait s'attendre à $2^{\Omega(n)}$.] [*Exercice.* Construisez la matrice pour la fonction eq_2. Trouver des ensembles discriminants les plus grands possibles pour la fonction eq_n, pour chaque valeur $v \in \{0,1\}$. Quelle est la somme totale des tailles $|F_0|+|F_1|$ des ensembles ?] [*Exercice.* En construisant la matrice pour $n = 2$, trouver des ensembles discriminants les plus grands possibles pour la fonction $gt_n(x,y)$ (*greater than*) qui est vraie si et seulement si $x > y$.]

4.6 Protocole sans-préfixe

Revenons sur la définition de la complexité de communication. La raison pour ne pas la définir comme [AB16, page 271] (voir la figure 4.7) est qu'il est toujours possibles de définir, pour chaque fonction f, un protocole de communication ayant un coût de seulement 2 bits.

En effet, il suffit à Alice de transmettre le mot vide ϵ, tour à tour, pendant $x \in [0, 2^n[$ rondes, puis de transmettre à Bob un 0 pour indiquer la fin. De son coté Bob répond un mot vide ϵ à chaque mot vide reçu jusqu'à recevoir le 0 terminal d'Alice. À ce moment là, Bob peut calculer $f(x,y)$ car il a compté le nombre de messages vides reçus, qui vaut précisément x. Puis Bob transmet la réponse à Alice. Les scripts d'un tel protocole ressemblent donc à ceci

$$\text{script}(x,y) = (\underbrace{\epsilon,\epsilon,\epsilon,\epsilon,\dots,\epsilon,\epsilon}_{2x}, 0, f(x,y)).$$

Ce protocole, parfaitement valide, a pour coût 2 puisque la somme des longueur des messages envoyés est seulement de 2. [*Question*. Combien y'a-t'il de scripts ?] Il y a donc un problème à définir, comme dans la définition [AB16, page 271], la complexité de communication comme le coût minimum d'un protocole calculant f. Plus précisément, il n'y a plus de lien entre le nombre minimum de scripts d'un protocole (ce qu'on c'est compter ou minorer) et complexité de communication (ce qu'on veut établir : le nombre de bits nécessaires à transmettre). En effet, on pourrait penser que le nombre maximum de scripts d'un protocole de coût k est 2^k qui sont pour $k = 2$:

$$(0,0),\ (0,1),\ (1,0),\ (1,1)$$

Mais c'est oublier tous ceux avec le mot vide

$$(\epsilon,0,0),\ (\epsilon,\epsilon,0,0),\ (\epsilon,\epsilon,\epsilon,0,0),\ \ldots$$

Il y en a une infinité (qui dépend cependant du nombre de rondes de communication en tre Alice et Bob).

Bien sûr, on pourrait interdire les protocoles utilisant des messages vides. L'idée étant que lorsqu'on communique le mot vide ϵ, on communique quelque chose. En fait, bien que valide, ce protocole avec mot vide suppose, d'une certaine façon, qu'Alice et Bob possèdent une horloge et utilisent l'astuce de comptage déjà décrite page 4.3. Donc on devrait compter quelque chose pour les mots vides ou interdire les horloges ...

Mais cela ne suffit pas. En effet, même sans mot vide, il y a potentiellement un problème pratique pour le receveur pour déterminer si le message a été complètement reçu. Comment à une ronde donnée Bob pourra distinguer un message « 1 » d'un autre « 11 » si, suivant l'entrée d'Alice, les deux messages sont possibles pour cette ronde ? Et ce problème pourrait concerner le dernier message si bien que le receveur du dernier message, le bit valant $f(x,y)$, pourrait penser que le protocole n'est pas terminé. [*Exercice*. Modifier les entrées pour le protocole décrit sur la figure 4.6 de sorte à obtenir deux scripts, l'un étant préfixe de l'autre, si bien que la terminaison est potentiellement ambiguë.]

Pour pouvoir correctement définir la complexité de communication comme le coût minimum d'un protocole, il faut ajouter une condition importante que doit vérifier le protocole. Il faut que les messages échangés jusqu'à la $i$ème ronde, sur toutes les entrées possibles, forment des ensembles de mots sans-préfixe. Plus précisément,

> Pour tout $i \in \mathbb{N}$, l'ensemble de mots suivant :
>
> $$\mathscr{M}_\Pi(i) := \bigcup_{(x,y)} \left\{ m_0(x,y) \circ m_1(x,y) \circ \cdots \circ m_{\min\{i,t(x,y)\}}(x,y) \right\}$$
>
> est *sans-préfixe*. On dit alors que Π est sans-préfixe.

C'est donc les ensembles de mots $\mathscr{M}_\Pi(0), \mathscr{M}_\Pi(1), \ldots$ qui doivent chacun être sans-préfixe.

Avant de définir « sans-préfixe », notons qu'ici $t(x,y)$ représente le nombre de rondes (-1) pour le calcul de $f(x,y)$ selon Π, les numéros de ronde commençant à 0 d'après la définition 4.1. Le mot binaire $m_0(x,y) \circ \cdots \circ m_{\min\{i,t(x,y)\}}(x,y)$ représente la concaténation des messages successivement échangés jusqu'à la ronde i du protocole Π exécuté sur l'entrée (x,y), étant entendu que les messages s'arrêtent au-delà de la ronde $t(x,y)$ si $i > t(x,y)$ (ce qui explique la présence du $\min\{i,t(x,y)\}$). Dit autrement, c'est la concaténation des messages des scripts tronqués à la ronde $\min\{i,t(x,y)\}$. Étant donné Π, les ensembles $\mathscr{M}_\Pi(i)$ peuvent être calculés en simulant le protocole sur chaque entrées (x,y) possibles d'Alice et Bob.

On dit qu'un ensemble X de mots – on parle de *code* – est *sans-préfixe* si aucun mot de X n'est préfixe d'un autre mot de X. Dit autrement, les mots de X sont deux à deux sans-préfixes. On rappelle qu'un mot w est préfixe d'un mot w' si w' débute par w. Par exemple, le mot "code" est préfixe de "coder" mais pas de "décode" ni de "décoder". Les ensembles de mots binaires $\{0,1\}$, $\{1,00,01\}$, et $\{10,11,001\}$ sont trois exemples d'ensembles sans-préfixes, alors que $\{0,1,01\}$ ne l'est pas. Il est facile de vérifier que la concaténation de mots sans-préfixes produit des mots sans-préfixes. Par exemple, avec le code $\{10,11,001\}$, on peut reconstruire sans ambiguïté la suite de messages à partir de la concaténation 10110011110. Il s'agit de $(10,11,001,11,10)$. En fait, un code X sans-préfixe permet de prédire la longueur de chaque message dans un flux de messages, c'est-à-dire qu'on peut remettre les virgules.

Généralement, pour un ensemble X sans-préfixe, $\epsilon \notin X$. Ou alors c'est que $X = \{\epsilon\}$, puisque ϵ est préfixe de tout autre mot. Mais un code avec $|X| = 1$ est particulièrement inutile pour envoyer efficacement des messages.[*Exercice.* Pour un protocole Π sans-préfixe, que peut-on dire de l'ensemble $\mathscr{M}_\Pi(i) \cup \mathscr{M}_\Pi(j)$ pour $i \neq j$?] [*Exercice.* Montrez que le protocole décrit dans la figure 4.6 n'est pas sans-préfixe.] [*Exercice.* Montrez que pour le protocole Π décrit dans la figure 4.6, il existe un mot $s \in \mathscr{M}_\Pi(t)$ provenant de deux scripts différents, où t est le numéro de ronde maximum du protocole (le nombre maximum de rondes est $t+1$ car on commence par la ronde 0). Dit autrement, montrez que Π peut produire deux scripts différents avec une concaténation des messages identique et égale à s.]

Avec un protocole sans-préfixe, lors de la ronde i, le receveur $J(i+1)$ sera capable de déterminer quand est-ce que le message m_i est terminé, même s'il reçoit les bits de $J(i)$ un à un [7]. Plus utile encore, le receveur pourra déterminer lorsque le protocole est terminé, et donc déterminer le dernier message et la valeur de $f(x,y)$. [*Exercice.* Pourquoi, si le protocole est sans-préfixe, Alice et Bob peuvent déterminer lorsque le protocole sur (x,y) est terminé ?]

Définition 4.3 *La* complexité de communication *de f est le coût minimum d'un protocole de communication sans-préfixe pour f.*

7. Pour donner une sens à la notion de protocole sans-préfixe, on supposera que les bits des messages sont transmis dans l'ordre des bits de poids fort d'abord.

Sans cette propriété de sans-préfixe, même sans utiliser le mot vide, la complexité de communication de $disj_n$ aurait été effectivement $\ll n$ comme on le verra dans l'Exercice 3. Ces protocoles rendent imprédictible la taille des messages, justement, faisant perdre le caractère sans-préfixe du protocole. [*Exercice.* Montrez qu'on peut transmettre une chaîne binaire arbitraire de longueur n connue grâce à une suite de messages $(m_0, m_1, \ldots, m_t)$ sans mot vide telle que $\sum_{i=0}^{t} |m_i| \leqslant 2n/3 + O(1)$.]

Il y a une équivalence entre mot sans-préfixe et les chemins dans un arbre binaire ordonné (fils 0 et fils 1). En fait, dans la définition originale de Yao (cf. figure 4.9), un protocole est justement un arbre « presque » binaire où les chemins jusqu'aux feuilles correspondent aux scripts, et où certains nœuds sont étiquetés par Alice ou Bob, la racine étant Alice. Un message de la ronde i est un chemin entre un nœud $J(i)$ et un nœud $J(i+1)$, sans nœuds Alice ou Bob entre. Il suit que les mots formés des i premières rondes (correspondant à des chemins depuis la racine) sont sans-préfixes. On note qu'avec cette définition, on a presque [8] automatiquement que la longueur des messages est au moins la hauteur de l'arbre soit $\log_2 \ell$ où ℓ est le nombre de feuilles, c'est-à-dire le nombre de scripts.

Il reste à établir la relation entre le nombre de scripts pour un protocole et son coût, commençons par montrer que :

Proposition 4.5 *Tout protocole de communication pour f de coût k et utilisant au plus t rondes peut être transformé en un protocole sans-préfixe pour f de coût $2(k+t)$ et avec un nombre de scripts identique.*

Preuve. Soit Π un protocole quelconque pour f, donc pas forcément sans-préfixe et utilisant potentiellement des messages vides, et soit k son coût. On va transformer Π en un autre protocole $\tau(\Pi)$ sans-préfixe et qui calcule également f. Il aura de plus le même nombre de scripts.

Tout d'abord, pour chaque mot $m \neq \epsilon$, on considère la transformation $\delta(m)$ qui consiste à doubler chaque bit de m sauf le dernier qui est inversé [7]. Par exemple, $\delta(011) = 001110$. Notez que $\delta(\epsilon)$ n'est pas définie, ϵ n'a pas de dernier bit ! Retrouver m à partir de $\delta(m)$ est trivial puisqu'il suffit de lire les bits deux par deux et de s'arrêter lorsque les bits diffèrent.

Vérifions que pour tout ensemble de mots non vides, l'ensemble des mots transformés par δ est sans-préfixe. Soit $m \neq m'$ deux messages. Si $\delta(m)$ est un préfixe de $\delta(m')$, c'est que les deux derniers bits de $\delta(m)$, qui sont « 01 » ou « 10 », se trouvent alignés avec les mêmes deux bits de $\delta(m')$. Si $|m'| > |m|$, ce n'est pas possible puisque tous les bits intermédiaires de m' ont été doublé. Si $|m'| = |m|$, comme $m \neq m'$, $\delta(m)$ et $\delta(m')$ diffèrent

8. Pour cela il faut supposer que l'arbre est binaire et complet (chaque nœud est à 2 ou 0 fils). On peut alors « standardiser » un protocole en remarquant qu'on peut toujours contracter sans ambiguïté toute arête uv où v est l'unique fils de u.

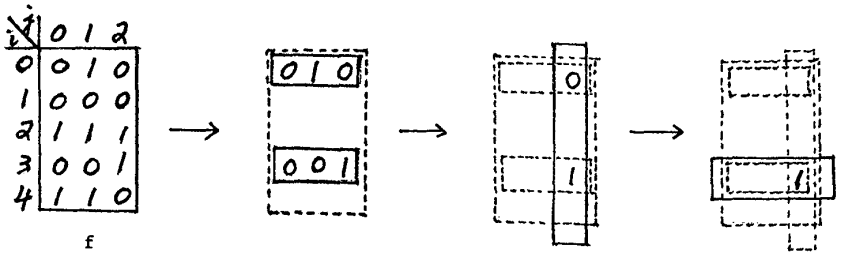


Figure 1 A function f, and the successive steps taken by the algorithm below in finding f(3,2).

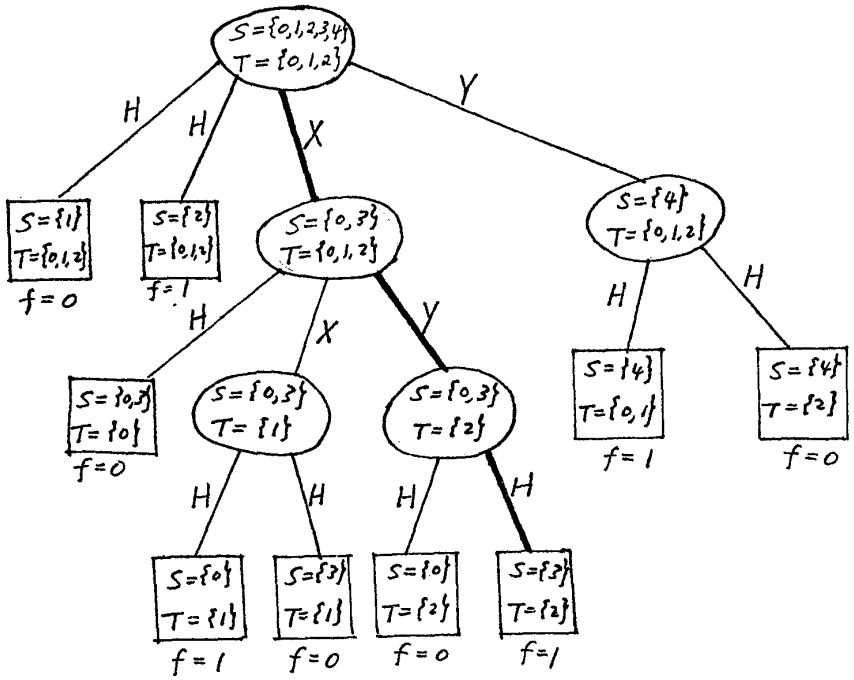


Figure 2. An algorithm for computing f. The sequence of signals exchanged for input i=3, j=2 is XYH.

FIGURE 4.9 – Arbre définissant un protocole de communication selon la définition originale de Yao. Dans ce schéma, les étiquettes X, Y, H sur les arêtes signifient respectivement des messages d'un bit 0, 1 et HALT, le signal particulier pour terminer la transmission. Dans sa définition, les messages sont d'un seul bit et la transmission se termine dès qu'Alice ou Bob a pu déterminer la valeur de $f(i, j)$ où ici i, j sont des entiers.

par une autre paires qui n'est pas la dernière de $\delta(m)$. Et donc dans tous les cas $\delta(m)$ n'est pas un préfixe de $\delta(m')$. Les mots ainsi transformés sont sans-préfixes.

Maintenant, on transforme chaque message $m_i(x,y)$ de Π émis à la ronde i de la manière suivante :

$$\tau(m_i(x,y)) := \begin{cases} 1 \circ m_i(x,y) & \text{si } i = t(x,y) \text{ est la dernière ronde} \\ 01 & \text{si } i < t(x,y) \text{ et } m_i(x,y) = \epsilon \\ 00 \circ \delta(m_i(x,y)) & \text{sinon} \end{cases}$$

Décoder $m_i(x,y)$ depuis $\tau(m_i)$ est facile en remarquant que $X = \{1, 01, 00\}$ est sans-préfixe. Donc les trois cas peuvent être facilement identifié. (Le cas « $1 \circ m_i(x,y)$ » correspond en fait aux messages « 10 » ou « 11 » suivant la valeur de $f(x,y)$ puisque c'est le dernier message.) Pour les trois cas, on a donc $|\tau(m_i(x,y))| \leqslant 2 + 2|m_i(x,y)|$. En utilisant le fait que $\sum_{i=0}^{t(x,y)} |m_i(x,y)| \leqslant k$, le coût de $\tau(\Pi)$ vaut $\max_{(x,y)} \sum_{i=0}^{t(x,y)} |\tau(m_i(x,y))| \leqslant 2t + 2k$.

La transformation τ affecte seulement les messages individuels $m_i(x,y)$ (surtout leur longueur), et pas leur nombres ni leur succession. Donc le nombre de scripts pour Π et $\tau(\Pi)$ est identique.

Il reste à montrer que, pour chaque i, l'ensemble $\mathscr{M}_{\tau(\Pi)}(i)$ est sans-préfixe, et donc que le protocole $\tau(\Pi)$ est sans-préfixe. Pour alléger l'écriture, posons $s_i(x,y)$ les mots de $\mathscr{M}_{\tau(\Pi)}(i)$, si bien que $\mathscr{M}_{\tau(\Pi)}(i) = \bigcup_{(x,y)} \{s_i(x,y)\}$. Considérons $(x,y) \neq (x',y')$ avec $t(x,y) \leqslant t(x',y')$. Il y a deux cas :

Cas 1. $s_i(x,y)$ et $s_i(x',y')$ sont la concaténation du même nombre de messages, c'est-à-dire $t(x,y) = t(x',y')$ ou $i \leqslant t(x,y)$. Alors $s_i(x,y)$ et $s_i(x',y')$ ne sont pas préfixes l'un de l'autre car on vient de montrer que chaque message $\tau(m_j(x,y))$, $\tau(m_j(x',y'))$ de $s_i(x,y)$ et $s_i(x',y')$, ne sont pas préfixes l'un de l'autre, pour tout $j \in [0, \min\{i, t(x,y)\}]$. Et donc leur concaténation non plus.

Cas 2. $s_i(x,y)$ possède strictement moins de messages que $s_i(x',y')$, c'est-à-dire $t(x,y) < i \leqslant t(x',y')$. Il suit que $s_i(x,y)$ possède comme dernier message « 10 » ou « 11 » indiquant la fin du protocole pour (x,y) alors que tous les messages de $s_i(x',y')$ ne peuvent être « 10 » ou « 11 » puisque pour (x',y') le protocole n'est pas fini. Donc $s_i(x,y)$ et $s_i(x',y')$ ne sont pas préfixes l'un de l'autre. □

On définit alors le nombre minimum de scripts pour une fonction f comme :

$$\sigma(f) := \min_{\Pi} \left| \{\text{script}_{\Pi}(x,y) : (x,y) \in \{0,1\}^n \times \{0,1\}^n\} \right|$$

c'est-à-dire le nombre minimum de scripts pour tout protocole Π calculant f et toutes les entrées de f. Il est important de voir que $\sigma(f)$ est un nombre indépendant du fait qu'on considère des protocoles sans-préfixes ou pas, puisque grâce à la proposition 4.5 on peut sans perte de généralité supposer que le protocole minimisant le nombre de scripts peut être est sans-préfixe (et donc sans mot vide).

On a vu que $\sigma(f) \geqslant |F_0| + |F_1|$ pour tout ensembles discriminants (proposition 4.3). Comme le montre l'exemple de la figure 4.8, il arrive que $\sigma(f) > |F_0|+|F_1|$ même pour les ensembles discriminants les plus grands possibles (cf. l'exercice issu de cette figure). Et même pour la meilleure partition de la matrice de f en $r(f)$ rectangles monochromes, où $r(f)$ est le plus petit possible, il est possible que $\sigma(f) > r(f)$. Donc la partition en rectangles et la technique des ensembles discriminants restent des bornes inférieures sur $\sigma(f)$. [*Exercice***. Donnez un exemple de fonction f (qui peut être définie par sa matrice), montrant que $\sigma(f) > r(f) > |F_0|+|F_1|$ pour les plus grands ensembles discriminants possibles.] [*Exercice***. Montrez que si la complexité de communication de f est k, alors la matrice de f peut être partitionnée en $O(2^k)$ rectangles monochromatiques, c'est-à-dire en rectangles ayant tous la même valeur pour f, cf. [Yao79, Th.1].]

Proposition 4.6 *La complexité de communication de f est au moins $\log_2 \sigma(f)$, pour toute f.*

Preuve. On va s'intéresser au *codage binaire* d'un ensemble de mots X (pas forcément binaires). Il s'agit d'associer à chaque mot $w \in X$ un *code*, c'est-à-dire un mot binaire d'une certaine longueur fixée à l'avance (et bien sûr la plus courte possible).

L'inégalité de Kraft affirme qu'un ensemble X de mots possède un codage binaire sans-préfixe si et seulement si

$$\sum_{w \in X} 2^{-\ell(w)} \leqslant 1 \qquad \text{(Kraft)}$$

où $\ell(w)$ représente la longueur désirée (en bits) du mot w. Attention ! Cela ne dit pas si un code donné est sans-préfixe, mais permet de dire si des longueurs de code sont atteignables par un codage sans-préfixe. [*Exercice*. Reprendre les exemples d'ensembles de mots binaires page 73, en particulier $\{10, 11, 001\}$ et $\{0, 1, 01\}$, et vérifier l'inégalité à partir de leur longueur. Construire sur les longueurs $\{2, 2, 3\}$ un code qui n'est pas sans-préfixe.] Cette inégalité implique que tout ensemble M de mots binaires sans-préfixes, chaque mot $w \in M$ étant limité à $\ell(w) \leqslant k$ bits, doit vérifier $|M| \leqslant 2^k$. En effet, on doit avoir $\sum_{w \in M} 2^{-\ell(w)} \leqslant 1$. Cette somme est minimum lorsque chaque $\ell(w)$ est maximum, ce qui revient à dire qu'il faut choisir $\ell(w) = k$ pour tout $w \in M$. Cela donne donc $\sum_{w \in M} 2^{-k} \leqslant 1$ ou encore $|M| \cdot 2^{-k} \leqslant 1$, ce qui implique $|M| \leqslant 2^k$.

Soit Π un protocole sans-préfixe pour f de coût k. On veut montrer que $k \geqslant \log_2 \sigma(f)$. Pour cela, on considère les ensembles $M := \mathscr{M}_\Pi(t)$ et $S := \bigcup_{(x,y)} \{\text{script}_\Pi(x, y)\}$, où t est le nombre maximum de rondes (-1) effectuées par Π. Autrement dit :

$$\begin{aligned} M &= \bigcup_{(x,y)} \Big\{ m_0(x,y) \circ m_1(x,y) \circ \cdots \circ m_{t(x,y)}(x,y) \Big\} \quad \text{et} \\ S &= \bigcup_{(x,y)} \Big\{ (m_0(x,y), m_1(x,y), \ldots, m_{t(x,y)}(x,y)) \Big\} . \end{aligned}$$

On va montrer que $|M| = |S|$.

Clairement, $|M| \leqslant |S|$, puisqu'on peut construire de manière unique chaque mot $m_0(x,y) \circ \cdots \circ m_{t(x,y)}(x,y) \in M$ depuis $\text{script}_\Pi(x,y) \in S$. (Techniquement, cette construction définit une injection de $M \to S$, ce qui montre que $|M| \leqslant |S|$.) Une autre façon de voir ce premier point, est qu'enlever les virgules des scripts de S permet d'obtenir un ensemble de mots M qui est plus petits ou égal.

Chaque ensemble $\mathscr{M}_\Pi(i)$, pour $i \in [0,t]$, est complètement déterminé par Π. Et étant donnés ces ensembles, il est possible de reconstruire $\text{script}_\Pi(x,y) \in S$ à partir de la concaténation $m_0(x,y) \circ \cdots \circ m_{t(x,y)}(x,y) \in M$. En effet, ils sont sans-préfixes. On peut donc identifier sans ambiguïté $m_0(x,y)$ à partir de $\mathscr{M}_\Pi(0)$, puis $m_0(x,y) \circ m_1(x,y)$ depuis $\mathscr{M}_\Pi(1)$, et donc de déduire $m_1(x,y)$, ... et ainsi de suite jusqu'à identifier $m_{t(x,y)}(x,y)$. Grâce au protocole Π, on a ainsi construit de manière unique chaque script de S depuis un mot de M. (Cela définit une injection de $S \to M$.) Il suit que $|S| \leqslant |M|$, et donc que $|M| = |S|$. (Techniquement, on a construit une bijection entre M et S, grâce à Π.)

Par définition de σ, $|S| \geqslant \sigma(f)$ puisque $|S|$ n'est rien de plus que le nombre de scripts de Π. Le coût de f étant k, le plus long mot de M est précisément de k bits (cf. définition 4.2). De plus, $M = \mathscr{M}_\Pi(t)$ est un ensemble de mots sans-préfixes par hypothèse sur Π. Par l'inégalité (Kraft), il suit que $2^k \geqslant |M| = |S| \geqslant \sigma(f)$, soit $k \geqslant \log_2 \sigma(f)$, ce qu'on voulait démontrer. □

[*Exercice**. Montrez que le coût de communication de tout protocole sans message vide pour f est au moins $\frac{1}{3} \cdot \log_2 \sigma(f)$.]

On a vu précédemment (cf. figure 4.8) que $\sigma(\mathit{disj}_2) \geqslant 6$. (C'est même au moins 7 d'après l'exercice.) La complexité de communication est donc au moins $\log_2 6 \sim 2.58$ bits, soit 3 bits, ce qui est optimal (c'est toujours au plus $n+1$ bits pour une fonction booléenne sur des entrées de n bits).

Plus généralement,

Proposition 4.7 *La complexité de communication de disj_n est de $n+1$.*

Preuve. En combinant les propositions 4.4 et 4.6, on obtient donc la borne inférieure de $\lceil \log_2(2^n + n) \rceil$ bits sur le coût de communication de tout protocole sans-préfixe. On vérifie facilement que $\lceil \log_2(2^n + n) \rceil = n+1$ dès que $n \geqslant 1$.

En fait, qu'il est en fait inutile de construire un grand ensemble F_0 comme donné par la proposition 4.4. En effet, on remarque que $\lceil \log_2(2^n + 1) \rceil = n+1$, et donc qu'on a juste besoin de l'ensemble F_1 avec $|F_1| = 2^n$ et de dire qu'il y a au moins une valeur à 0 dans la matrice, c'est-à-dire qu'il existe un ensemble F_0 avec $|F_0| \geqslant 1$. □

La relation entre la complexité de communication d'une fonction f et son nombre de scripts minimum $\sigma(f)$ fait l'objet de nombreuses recherches. Si la proposition 4.6 établit une borne inférieure de $\Omega(\log \sigma(f))$, le contraire est beaucoup moins clair. Depuis assez longtemps (cf. [KN97]) on sait que la complexité de communication est au

plus $O(\log^2 \sigma(f))$ pour toute fonction f. Mais ce n'est que récemment, dans [AKK16], qu'il a été montré l'existence de fonction f dont la complexité de communication est au moins $\log^{2-o(1)} \sigma(f)$.

4.7 Exercices

Exercice 1

On considère le problème de la détection de C_4 évoqué dans la section 4.3 (problème 2). Déterminer le temps nécessaire et suffisant pour le résoudre dans le modèle LOCAL.

Exercice 2

On se propose de démontrer que le problème de détection d'un C_4 nécessite $\Omega(\sqrt{n}/b)$ rondes dans le modèle b-CONGEST, c'est-à-dire le modèle CONGEST où les messages sont de taille au plus b bits.

(0) Montrez qu'on peut résoudre le problème en deux rondes si $b = n$.

Le *produit cartésien* de deux graphes X et Y est le graphe G, noté $X \square Y$, dont l'ensemble des sommets est :

$$V(G) := V(X) \times V(Y) = \{(x, y) : x \in V(X), y \in V(Y)\}$$

et l'ensemble des arêtes est :

$$E(G) := \{((x, y), (x', y')) \in V(G) : ((x = x') \wedge (y, y') \in E(Y)) \vee ((y = y') \wedge (x, x') \in E(X))\} .$$

Autrement dit, deux sommets sont voisins si les sommets dont ils sont issus étaient voisins dans l'un des deux graphes comme illustré par la figure 4.10.

(a) Construire un graphe H à $n = 6$ sommets sans cycle de longueur 4 et avec un nombre maximum m d'arêtes (on peut faire au moins $m = 7$).

(b) Construire le graphe $G = H \square K_2$ où K_r est la clique à r sommets. Dans la suite on notera H_0 et H_1 les deux copies de H, sous-graphes contenus dans G.

Soient $A, B \subseteq E(H)$ deux sous-ensembles d'arêtes de H. On note $G_{A,B}$ le graphe obtenu à partir de G en ne gardant dans la copie H_0 de H que les arêtes de A, et dans la copie H_1 que les arêtes de B. Les arêtes des copies de K_2, celles connectant H_0 et H_1, sont par contre préservées.

(c) Montrer que $G_{A,B}$ possède un C_4 si et seulement si $A \cap B \neq \varnothing$.

(d) Expliquer comment déduire un protocole de communication pour résoudre Set-Disjointness entre Alice et Bob (comme discuté dans la section 4.3) à partir d'un

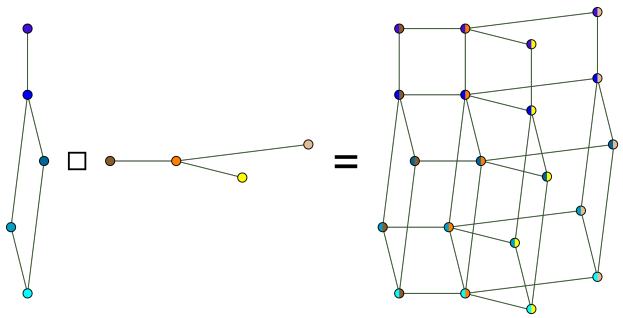


FIGURE 4.10 – Produit cartésien $X \square Y$ de deux graphes X et Y (source Wikipédia).

algorithme distribué résolvant le problème de la détection de C_4. Cet algorithme distribué pourra être exécuté sur un graphe inspiré de $G_{A,B}$.

On admettra sans preuves les deux résultats suivants :

1. La complexité de communication pour le problème SET-DISJOINTNESS entre sous-ensembles de $\{1,\dots,m\}$ est $m+1$ bits [9], cf. la proposition 4.7.
2. Il existe un graphe à n sommets sans C_4 ayant $m > (n/2)^{3/2}$ arêtes (en fait sans cycle de longueur < 6). On pourra trouver dans [Gav24, Chap. 1] une preuve constructive de ce résultat. En gros, on considère un graphe biparti avec q^2+q+1 sommets dans chacune des parts, q étant un nombre premier, et tel que chaque sommet possède $q+1$ sommets dans l'autre part tels que deux voisinages n'ont qu'au plus un sommet en commun. On peut construire la solution lorsque $q = 2$ en considérant le plan de Fano.

(e) Déduire de ce qui précède que le nombre de rondes nécessaires pour résoudre le problème de détection de C_4 est au moins $\Omega(\sqrt{n}/b)$ dans le modèle b-CONGEST.

Exercice 3

L'objectif est de montrer qu'il existe un protocole sans message vide calculant $disj_n$ avec un coût de communication de $2n/3+4$ bits.

Le principe du protocole est basé sur deux idées.

(1) La première est d'échanger chacun une partie (une moitié) de son entrée à l'autre. On découpe $x = (x_1, x_2)$ et $y = (y_1, y_2)$ avec $|x_1| = |y_1| = \lceil n/2 \rceil$ et $|x_2| = |y_2| = \lfloor n/2 \rfloor$. Alice communique x_1 à Bob, et Bob communique y_2 à Alice. Une fois ces phases de communication terminées, chacun calcule en parallèle un résultat partiel. Alice calcule $a = disj_{|x_2|}(x_2, y_2)$ et Bob $b = disj_{|x_1|}(x_1, y_1)$. Finalement, Alice transmet a à Bob qui renvoie $a \wedge b = disj_n(x, y)$ à Alice, ce qui fait deux bits supplémentaires.

9. En fait il existe un résultat plus précis si les ensembles n'ont que k éléments. Il faut alors $\Theta(\log \binom{m}{k}) = \Theta(k \log(m/k))$ bits [DKO14].

(2) La deuxième est de transmettre sa partie avec un protocole qui envoie des messages d'un ou de deux bits à chaque ronde. On ne décrit que le protocole exécuté par Alice. Celui de Bob, exécuté en alternance avec Alice, est similaire (il faut considérer y et non x, et l'indice j devra varier, non pas dans $[0, \lceil n/2 \rceil[$, mais dans $[\lceil n/2 \rceil, n[$) :

Alice :

---

1. $j := 0$
2. Tant que $j < \lceil n/2 \rceil - 1$, faire :
   - Si $x[j] = 0$, envoyer $x[j+1]$, et poser $j := j + 2$;
   - Si $x[j] = 1$, envoyer les deux bits $x[j+1], x[j+2]$, et poser $j := j + 3$.

À chaque ronde, si Bob a déjà reconstruit j bits de l'entrée d'Alice (au départ $j = 0$), et qu'il reçoit un message d'un bit, c'est que le bit reçu est $x[j+1]$ et que $x[j] = 0$. Et si c'est un message de deux bits, c'est qu'il a reçu $x[j+1]$ et $x[j+2]$, et que $x[j] = 1$. Il peut alors localement mettre à jour $x[0] \dots x[j+1]$ voir $x[0] \dots x[j+2]$, et aussi j. À la fin Bob a reconstruit la partie x_1 d'Alice (voir un bit de plus), soit au plus $|x_1| + 1 = \lceil n/2 \rceil + 1$ bits, tandis qu'Alice à reconstruire la partie y_2 de Bob (voir un bit de plus), soit au plus $|y_2| + 1 = \lfloor n/2 \rfloor + 1$ bits.

Ainsi à chaque ronde i d'Alice, Bob reconstruire $b_i \in \{2, 3\}$ bits de la partie d'Alice, alors qu'elle n'en transmet que $b_i - 1 \leqslant 2b_i/3$. (On vérifie que $2 - 1 \leqslant 2 \cdot 2/3$ et que $3 - 1 \leqslant 3 \cdot 2/3$.) Quand le protocole d'Alice se termine, Bob a reconstruit $\sum_i b_i \leqslant (\lceil n/2 \rceil - 2) + 3 = \lceil n/2 \rceil + 1$ bits de la partie d'Alice. Mais le nombre de bits transmis n'est que de :

$$\sum_i (b_i - 1) \;\leqslant\; \sum_i \left(\frac{2}{3} \cdot b_i \right) \;\leqslant\; \frac{2}{3} \cdot (\lceil n/2 \rceil + 1) \;.$$

L'analyse est la même pour le nombre de bits transmis de Bob vers Alice, soit $2/3 \cdot (\lfloor n/2 \rfloor + 1)$. Au total fait, en tenant compte des deux derniers bits finaux pour le calcul de $f(x, y)$, cela fait au plus (utilisant le fait que $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$) :

$$\frac{2}{3} (\lceil n/2 \rceil + 1) + \frac{2}{3} (\lfloor n/2 \rfloor + 1) + 2 \;=\; \frac{2}{3} n + \frac{10}{3} \text{ bits}$$

ce qui fait donc au plus $2n/3 + 4$ bits.

En fait, le même protocole pourra être utilisé pour toute fonction qui peut être calculée par une combinaison de la moitié des entrées, comme par exemple l'égalité « $x = y$ » que l'on peut exprimer aussi par la fonction booléenne :

$$eq_n(x, y) \;:=\; \bigwedge_{i=0}^{n-1} \neg (x[i] \oplus y[i]) \;.$$

On a donc un protocole en $2n/3 + O(1)$ pour toutes ces fonctions là.

Bibliographie

[AB16] S. Arora and B. Barak, *Computational Complexity : A Modern Approach*, Cambridge, 2016. isbn : 9780521424264, 9780511530753, 0521424267.

[AKK16] A. Ambainis, M. Kokainis, and R. Kothari, *Nearly optimal separations between communication (or query) complexity and partitions*, in 31st Conference on Computational Complexity (CCC), R. Raz, ed., vol. 50 of LIPIcs, May 2016, pp. 4 :1–4 :14. doi : 10.4230/LIPIcs.CCC.2016.4.

[BGK+18] M. Braverman, A. Garg, Y. K. Ko, J. Mao, and D. Touchette, *Near-optimal bounds on the bounded-round quantum communication complexity of disjointness*, SIAM Journal on Computing, 47 (2018), pp. 2277–2314. doi : 10.1137/16M106140.

[BJ17] B. Bukh and Z. Jiang, *A bound on the number of edges in graphs without an even cycle*, Combinatorics, Probability and Computing, 26 (2017), pp. 1–15. doi : 10.1017/S0963548316000134.

[DKO14] A. Drucker, F. Kuhn, and R. Oshman, *On the power of the congested clique model*, in 33rd Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, July 2014, pp. 367–376. doi : 10.1145/2611462.2611493.

[EFF+22] T. Eden, N. Fiat, O. Fischer, F. Kuhn, and R. Oshman, *Sublinear-time distributed algorithms for detecting small cliques and even cycles*, Distributed Computing, 35 (2022), pp. 207–234. doi : 10.1007/s00446-021-00409-3.

[FGKO18] O. Fischer, T. Gonen, F. Kuhn, and R. Oshman, *Possibilities and impossibilities for distributed subgraph detection*, in 30th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), ACM Press, July 2018, pp. 153–162. doi : 10.1145/3210377.3210401.

[FLMT24] P. Fraigniaud, M. Luce, F. Magniez, and I. Todinca, *Even-cycle detection in the randomized and quantum CONGEST model*, Tech. Rep. 2402.12018v1 [cs.DC], arXiv, February 2024.

[Gav24] C. Gavoille, *Analyse d'algorithme – Cours d'introduction à la complexité paramétrique et aux algorithmes d'approximation*, 2024. http://dept-info.labri.fr/~gavoille/UE-AA/cours.pdf. Notes de cours.

[HW07] J. Håstad and A. Wigderson, *The randomized communication complexity of set disjointness*, Theory of Computing, 3 (2007), pp. 211–219. doi : 10.4086/toc.2007.v003a011.

[KN97] E. Kushilevitz and N. Nisan, *Communication Complexity*, Cambridge University Press, 1997.

[LPSP01] Z. Lotker, B. Patt-Shamir, and D. Peleg, *Distributed MST for constant diameter graphs*, in 20th Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, 2001, pp. 63–71. doi : 10.1145/383962.383984.

[Luc24] M. Luce, *Distributed Randomized and Quantum Algorithms for Cycle Detection in Networks*, PhD thesis, Université Paris Cité, Paris, France, December 2024.

[She14] A. A. Sherstov, *Communication complexity theory : Thirty-five years of set disjointness*, in 39th International Symposium on Mathematical Foundations of Computer Science (MFCS), vol. 8634 of Lecture Notes in Computer Science, Springer, August 2014, pp. 24–43. doi : 10.1007/978-3-662-44522-8_3.

[Yao79] A. C.-C. Yao, *Some complexity questions related to distributive computing*, in 11th Annual ACM Symposium on Theory of Computing (STOC), ACM Press, April 1979, pp. 209–213. doi : 10.1145/800135.804414.

CHAPITRE 5

Arbres couvrants

Sommaire

Dans ce chapitre on s'intéresse au calcul d'arbres couvrants (BFS, DFS, MST) en mode asynchrone. On suppose donc dans la suite qu'on est en mode asynchrone (model ASYNC). Rappelons qu'un arbre couvrant d'un graphe G est un sous-arbre de G qui est un arbre et qui passe par chacun des sommets de G.

Mots clés et notions abordées dans ce chapitre :

- arbres BFS et DFS,
- Dijkstra distribué, Bellman-Ford distribué,
- algorithme GHS.

5.1 Arbres en largeur d'abord

Il s'agit de calculer un arbre BFS de G (de l'Anglais *Breadth First Search*) depuis un sommet fixé, r_0. Ici le poids des arêtes n'est pas pris en compte. On fera comme si les poids sont unitaires, même si certains algorithmes se généralisent très bien aux valuations quelconques. Les arbres BFS servent souvent de première brique pour les algorithmes plus complexes (comme ceux de synchronisation, voir le chapitre 6) où pour économiser du temps on est amener à réaliser des diffusions et des concentrations (voir le chapitre 3) via des arbres de hauteur la plus petite possible.

Un arbre BFS de racine r_0 est un arbre T couvrant G tel que $d_T(r_0, u) = d_G(r_0, u)$ pour tout sommet u de T. C'est donc un arbre enraciné donnant un plus court chemin de tout sommet à la racine.

Nous avons vu au chapitre 3 comment construire un arbre couvrant à partir de Flood(r_0), voir le corollaire 3.1. En mode synchrone cet arbre est précisément un BFS (cf. la figure 5.1). L'enjeux ici est de le faire en mode asynchrone.

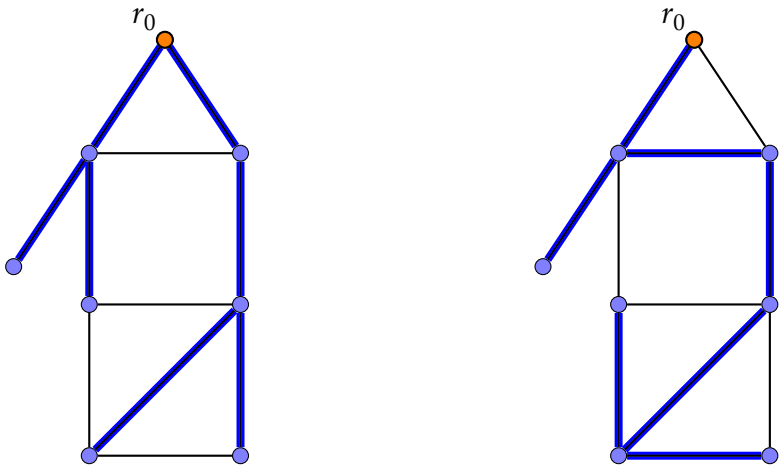


FIGURE 5.1 – Exemple de calcul d'arbre couvrant, grâce à Flood(r_0), dans le cas synchrone et asynchrone. Le cas synchrone produit un arbre BFS.

En combinant ce qui a été vu au chapitre 3, on déduit que :

Proposition 5.1 *Tout algorithme distribué de calcule d'arbre couvrant sur les graphes à m*

arêtes et de diamètre D doit avoir une complexité de $\Omega(D)$ en temps et de $\Omega(m)$ en nombre de messages.

[*Exercice**. Écrivez la preuve.] Notons qu'à cause de la proposition 3.1, on ne peut pas vraiment dire mieux que ces deux bornes. Elles sont atteintes.

5.1.1 Un algorithme *layer-based* (Dijkstra)

Ici on essaye de simuler l'algorithme de Dijkstra qui, rappelons-le, construit progressivement un arbre de plus courts chemins, ajoutant un par un le sommet le plus proche de r_0. En fait, il ne s'agit pas vraiment de l'algorithme de Dijkstra mais plus d'un parcours en largeur d'abord (BFS) car on suppose que les arêtes de G ont un poids unitaire. Donc l'arbre se construit par niveaux. La généralisation aux poids quelconques est discutée en fin de section.

Le principe de la version distribuée, que nous appellerons Dijkstra_Dist, est le suivant : l'algorithme s'exécute en *phases*, chacune des phases étant chargée d'étendre l'arbre courant d'un niveau supplémentaire. Plus précisément, notons par T_p l'arbre couvrant construit à la fin de la phase p. L'arbre que l'on cherche est localement défini par les variables $\textsc{Parent}(u)$ et $\textsc{Fils}(u)$. On indiquera par une variable booléenne $\textsc{Fin}(u)$ le fait que le sommet u a terminé l'algorithme. On dira qu'il est *inactif* si $\textsc{Fin}(u) = \textsc{Vrai}$ et *actif* sinon. On utilisera également deux compteurs, $\textsc{CptFils}(u)$ et $\textsc{CptNew}(u)$.

L'objectif est que T_p soit un BFS couvrant tous les sommets à distance au plus p de r_0. Donc lorsque $p = 0$, T_0 se réduit au sommet r_0.

Initialement $p := 0$, et pour tout sommet u, $\textsc{Fin}(u) := \textsc{Faux}$ et $\textsc{Parent}(u) := \textsc{Fils}(u) := \perp$, sauf que $\textsc{Fils}(r_0) := \varnothing$. Un sommet u va pouvoir ainsi détecter s'il est dans T_p ou non (si $\textsc{Fils}(u) \neq \perp$ ou non) et s'il est une feuille ou non (si $\textsc{Fils}(u) := \varnothing$ ou non).

La construction de T_{p+1} à partir de T_p est réalisée de la manière suivante :

Algorithme Dijkstra_Dist(r_0)

(principe pour un sommet u à la phase p pour construire T_{p+1})

1. La racine r_0 diffuse un message « *START* » à tous les sommets[1] de T_p grâce à $\text{Cast}_{T_p}(r_0)$. Lors de la diffusion, u pose $\textsc{CptFils}(u) := |\textsc{Fils}(u)|$.
2. À la réception d'un message « *START* », chaque feuille $u \in T_p$ qui est active envoie un message « *NEW* » à tous ses voisins sauf à[2] $\textsc{Parent}(u)$, et pose $\textsc{CptNew}(u) := |N(u)| - 1$.
3. À la reception d'un message « *NEW* » émis par v, si $u \notin T_p$ et que c'est la première fois[3] pour cette phase, on pose $\textsc{Parent}(u) := v$ et $\textsc{Fils}(u) := \varnothing$, et u renvoie un message « *FILS* » à v. Sinon (si $u \in T_p$ ou si ce n'est pas la première fois[4]), u renvoie un message « *ACK* » à v.

4. À la reception d'un message « *FILS* » ou « *ACK* » émis par v, u décrémente CptNew(u) et ajoute v à Fils(u) si c'est un message « *ACK* ». Quand CptNew(u) = 0, il envoie à Parent(u) un message « *DONE* », ou bien « *END* » s'il n'a reçu que des messages[5] « *ACK* ». Dans ce dernier cas, u devient inactif en posant Fin(u) := Vrai.
5. À la réception d'un message « *DONE* » ou « *END* », u décrémente CptFils(u). Quand CptFils(u) = 0, il envoie à Parent(u) un message « *DONE* » s'il a reçu au moins un « *DONE* », ou bien « *END* » sinon. Dans ce dernier cas, u devient inactif en posant Fin(u) := Vrai. Si de plus $u = r_0$, alors si Fin(u) = Faux, incrémenter p et recommencer à l'étape 1, et sinon (si Fin(u) = Vrai) l'algorithme est terminé.

Par induction sur p, on peut facilement montrer que :

Proposition 5.2 *À la fin de la phase p, T_{p+1} est un arbre* BFS *du graphe induit par tous les sommets à distance au plus $p+1$ de r_0.*

Par simplicité, posons $p_{\max} = \mathrm{ecc}_G(r_0)$. D'après la proposition 5.2, l'algorithme Dijkstra_Dist(r_0) a calculé un arbre BFS pour le graphe G lorsqu'il est arrivé à la fin de la phase $p_{\max} - 1$ puisque $T_{p_{\max}}$ couvre tous les sommets. Cependant, il y faut quand même exécuter la phase $p_{\max}$ car, à la fin de la phase $p_{\max} - 1$, même s'il est vrai que $T_{p_{\max}}$ est l'arbre recherché, la racine à l'étape 5 va être encore active car elle va recevoir au moins un message « *DONE* ». Donc la racine devient inactive seulement à la fin de l'étape $p_{\max}$. Les phases de l'algorithme sont donc $p = 0, 1, ..., p_{\max}$, soit un total de $p_{\max} + 1 \leqslant D + 1$ phases.

On va maintenant déterminer les complexités Message et Temps. On rappelle que m et D représentent respectivement le nombre d'arêtes et le diamètre du graphe G. Pour cela on définit F_p comme l'ensemble des arêtes incidentes aux feuilles actives de T_p mais qui ne sont pas dans T_p. Ici on parle des feuilles actives au début de la phase p. La remarque est que les arêtes de F_p sont précisément celles où circulent les messages « *NEW* », « *FILS* » et « *ACK* ».

---

1. On pourrait économiser ici quelques messages en ne diffusant qu'aux sommets actifs de T_p. Mais pour cela il faudrait les distingués dans Fils(u) avec un stockage supplémentaire. Sans stockage supplémentaire, un sommet inactif pourrait ne pas continuer la diffusion. Dans tous les cas, cela ne changera pas les complexités pour Temps et Message.
2. Ici u pourrait ne pas envoyer de message « *NEW* » à un voisin v qui lui aurait déjà envoyé un message « *NEW* ». Mais c'est utiliser de l'espace de stockage supplémentaire pour un cas qui peut ne jamais se produire.
3. Il le détecte par Fils(u) = $\perp$.
4. Donc même s'il est devenu inactif pendant la phase précédente ($p-1$), il est possible que u envoie encore des messages « *ACK* » dans l'étape 3 de la phase p s'il est connecté à une feuille de T_p.
5. Le sommet u peut implémenter cette étape en préparant un message M à envoyer à Parent(u) avec M :=« *END* ». Et u pose M :=« *DONE* » dès qu'il reçoit un message « *FILS* ». D'ailleurs u pourrait ne pas attendre d'avoir CptNew(u) = 0 pour envoyer un message « *DONE* » si un message « *FILS* » est reçu avant.

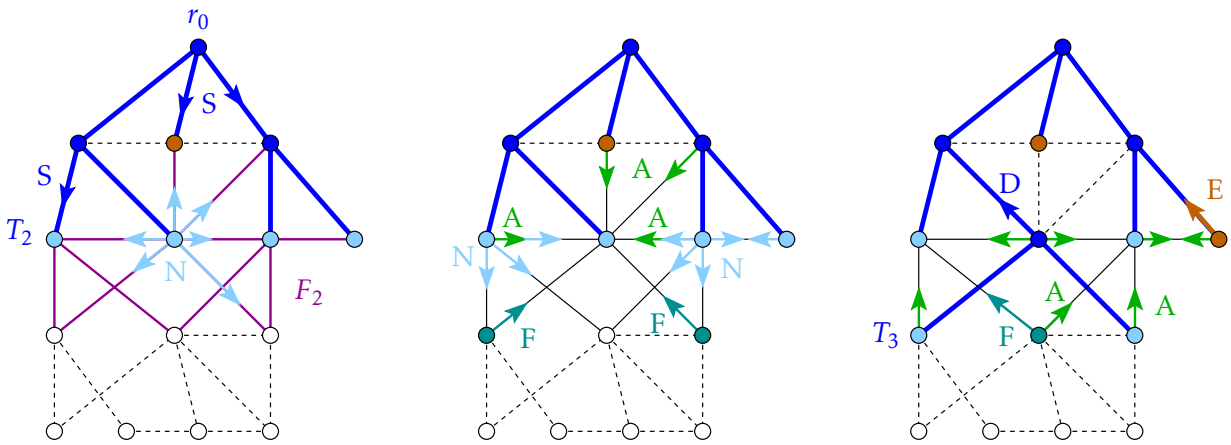


FIGURE 5.2 – Exécution asynchrone de l'algorithme Dijkstra_Dist(r_0) à la phase $p = 2$. Au début, l'arbre T_2 possède une feuille inactive et quatre actives. L'ensemble F_2 contient 11 arêtes sur lesquelles seront envoyés tous les messages « *NEW* ». Les futurs fils des feuilles actives de T_2 seront déterminés à l'étape 4 par les messages « *FILS* ». Les étapes de l'algorithme (notamment 3 et 4) ne sont pas nécessairement exécutées dans le même ordre par tous les sommets. Une feuille active de T_2 peut émettre un message « *ACK* » avant ou après son message « *NEW* ». À la fin de la phase $p = 2$, T_3 sera achevé, et la feuille la plus à droite de T_2 deviendra inactive avec l'émission d'un message « *END* » transmis au parent.

Proposition 5.3 $\textsc{Message}(\textit{Dijkstra\_Dist}(r_0), G) \leqslant \sum_{p=0}^{p_{\max}} \left(2|E(T_p)| + 4|F_p|\right) = O(np_{\max}+m) = O(nD+m)$.

Preuve. Le nombre total de messages est le nombre de messages pour la phase p, sommé sur chacune des phases de l'algorithme. Lors de la phase p, on remarque que l'algorithme n'envoie des messages que sur des arêtes de T_p ou de F_p.

Il réalise une diffusion dans T_p et à la fin une concentration. Cela coûte donc $2|E(T_p)|$ messages. Soit $uv \in F_p$. L'exploration du niveau $p+1$ avec les messages « *NEW* » puis « *FILS* » ou « *ACK* » fait transiter deux messages sur uv si $v \notin T_p$ et quatre si $v \in T_p$. Au total le nombre de messages est donc au plus :

$$\sum_{p=0}^{p_{\max}} \left(2|E(T_p)| + 4|F_p|\right) .$$

Comme chaque T_p est un arbre, $|E(T_p)| < n$, et donc $\sum_{p=0}^{p_{\max}} 2|E(T_p)| = O(np_{\max})$.

Pour borner $\sum_p |F_p|$, la remarque est qu'un sommet u ne peut être une feuille active que pour une seule phase. En effet, à l'étape 4 une feuille active de T_p devient soit un nœud interne dans T_{p+1} soit reste une feuille mais devient inactive pour toujours. Donc en sommant les arêtes incidentes àe tous les sommets de G on obtient un majorant sur les arêtes incidentes des feuilles actives pour toutes les phases, soit $\sum_p |F_p| \leqslant 2m$. En fait on a $\sum_p |F_p| = 2(m-(n-1))$ car les arêtes de T_p sont exclues de F_p, soit une arête vers le parent pour chaque feuille active (sauf r_0). D'où :

$$\textsc{Message}(\textsf{Dijkstra\_Dist}(r_0), G) \;=\; O(np_{\max}) + O(m) = O(np_{\max} + m) .$$

□

On remarque que la diffusion est la concentration à la phase p prennent chacune un temps p, et que l'exploration du nouveau niveau prend un temps 2, tout cela dans le pire des scénarios où tous les messages prennent le temps maximal (temps 1). D'où :

Proposition 5.4 $\textsc{Temps}(\textit{Dijkstra\_Dist}(r_0), G) = \sum_{p=0}^{p_{\max}} (2p+2) = O({p_{\max}}^2) = O(D^2)$.

Comme on le verra au chapitre 6, l'algorithme Dijkstra_Dist est ni plus ni moins que l'algorithme Flood synchronisé par le synchroniseur β.

La généralisation de Dijkstra_Dist à un graphe valué arbitrairement est plutôt laborieuse. Chaque sommet de la couche voisine de T_p doit gérer sa distance à r_0 ainsi que son parent provisoir dans T_p. Et, seul le plus proche peut former la nouvelle feuille de T_{p+1}. Comme le nombre de sommets n'augmente que de un à la fois, le nombre de phases passe de $O(D)$ à $O(n)$. Dans le même temps le nombre de messages par phase augmente jusqu'à $O(m)$ car les feuilles de T_p peuvent rester actives dans de nombreuses phases ultérieures. On ne détaillera pas plus cette approche.

5.1.2 Un algorithme *update-based* (Bellman-Ford)

L'algorithme Dijkstra_Dist maintient un sous-arbre correct en le faisant grandir, niveau après niveau. Ceci nécessite en fait une synchronisation par la racine r_0, ce qui *in fine*, fait prendre beaucoup de temps à l'algorithme : $O(D^2)$ alors qu'on pourrait espérer $O(D)$. Au contraire, la version distribuée de Bellman-Ford calcule très rapidement un arbre, éventuellement incorrect, et essaye progressivement de l'améliorer par mise à jour successives. D'après [Wal07][HSS07], c'est cet algorithme qui a été implémenté pour le routage dans ARPANET en 1969, le premier réseaux internet, cf. la figure 5.3.

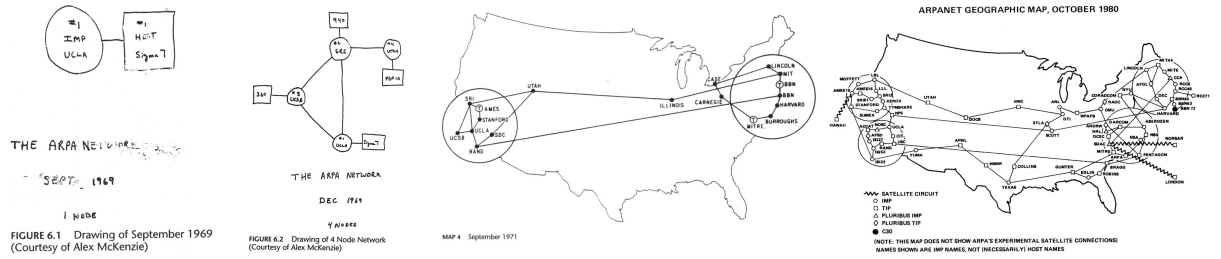


FIGURE 5.3 – Cartographie du premier réseau internet (ARPANET), en septembre 1969 (1 nœud), en décembre 1969 (4 nœuds), en septembre 1971 (18 nœuds) et en octobre 1980 (70 nœuds).

Plus précisément, l'idée de Bellman-Ford_Dist est d'adapter au cas asynchrone l'algorithme Flood qui, on le sait, calcule un arbre BFS dans le cas synchrone. La différence avec Flood est que chaque sommet diffuse la distance à laquelle il pense être de r_0, c'est-à-dire son niveau dans l'arbre courant dans le cas de valuation uniforme. Si l'arbre courant n'est pas un BFS (dans le cas asynchrone donc), c'est qu'une partie du réseau est allé trop vite. Cela signifie que le sommet va prochainement recevoir un message lui indiquant qu'un de ces voisins est en fait plus proche de la racine que le parent qu'il a déjà choisi. Dans ce cas, il peut mettre à jour son parent et son niveau en diffusant à ses voisins son nouveau niveau.

Pour alléger l'écriture du code ci-dessous, on va supposer que la racine r_0 se réveille en recevant le message d'un sommet fictif $v := \perp$. Initialement, pour tout sommet u, on a $\textsc{Layer}(u) := +\infty$, cette dernière variable représentant une estimation (en fait un majorant) de la distance entre u et r_0.

Algorithme Bellman-Ford_Dist(r_0)
(code du sommet u)

$\textsc{Layer}(u) := +\infty$

Répéter :

- $d := \textsc{Receive}()$, et soit v l'émetteur
  (si $u = r_0$, poser $v := \perp$ et $d := 0$)
- Si $d + 1 < \textsc{Layer}(u)$:

1. $\textsc{Layer}(u) := d + 1$
2. $\textsc{Parent}(u) := v$
3. $\textsc{Send}(\textsc{Layer}(u), w)$ pour tout $w \in N(u) \setminus \{v\}$

Comme il est décrit, l'algorithme Bellman-Ford_Dist ne calcule pas la variable Fils, mais il est facile de le modifier pour qu'il le fasse. [*Exercice*. Décrivez les modifications à apporter pour le faire.] Et, à cause du « Répéter », l'algorithme ne termine jamais localement. Cependant, le nombre de messages finit par s'arrêter comme on va le voir dans l'analyse de sa complexité en Temps. On rappelle (cf. la section 2.1 et la définition 2.2) que la fin de l'algorithme est défini comme le moment où le dernier message a été consommé (ou dit autrement le moment où plus aucun message n'est consommé). C'est un exemple d'algorithme distribué sans détection de la terminaison locale (voir le paragraphe 3.6).

Il est assez clair qu'à chaque moment, la relation $u \mapsto \textsc{Parent}(u)$ définit un arbre de racine r_0 (pas forcément couvrant). Mais aussi que la variable $\textsc{Layer}(u)$, lorsqu'elle est mise à jour, représente le coût du chemin de u à r_0 dans l'arbre. Cela implique que $\textsc{Layer}(u) \geqslant \text{dist}_G(r_0, u)$.

Il est aussi facile de montrer par induction sur d que (cf. figure 5.4) :

Proposition 5.5 *Pour chaque entier $d \geqslant 0$, après un temps $d+1$, chaque sommet u à distance $d+1$ de r_0 a reçu un message contenant d. Par conséquent, à cause des instructions 1-3, $\textsc{Layer}(u) = d + 1$ et $\textsc{Parent}(u) = v$ pour un certain voisin v de u.*

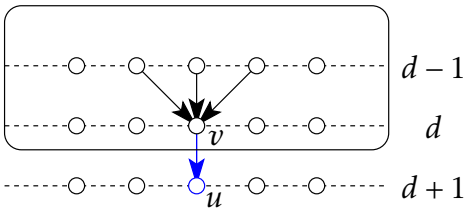


Figure 5.4 – Si v du niveau d a reçu un message $d-1$ après un temps d (par hypothèse), alors son voisin u au niveau $d+1$ aura reçu un message d après un temps au plus $d+1$ d'au moins un sommet du niveau d (en particulier de v).

On déduit facilement de la proposition 5.5 qu'au bout d'un certain temps, tous les sommets du graphe ont leurs variables Layer et Parent qui sont correctes, c'est-à-dire qui valent respectivement la distance à r_0 et le parent dans l'arbre BFS.

Théorème 5.1 *Pour tout graphe G et racine r_0, en synchrone ou asynchrone :*

- $\textsc{Temps}(\textit{Bellman-Ford\_Dist}(r_0), G) = \text{ecc}_G(r_0) = O(D)$.

- $\textsc{Message}(\textsf{Bellman-Ford\_Dist}(r_0), G) = O(m \cdot n)$.

Preuve. La proposition 5.5 implique qu'en temps $\mathrm{ecc}_G(r_0)$ les variables $\textsc{Layer}(u)$ et $\textsc{Parent}(u)$ ne changent plus. Cependant, après la dernière mise à jour, des messages peuvent circuler (et être consommés par un $\textsc{Receive}$) pendant encore une unité de plus. D'où la complexité en temps. Bien que le « répéter » semble indiquer que l'algorithme ne se termine jamais, en fait la définition du temps ne tient en compte que le temps du dernier message consommé. On remarque que ce temps peut être atteint dans le cas d'un scenario synchrone.

On envoie des messages seulement lorsque la variable $\textsc{Layer}(u)$ est mise à jour (instruction 1-3). Et chaque mise à jour coûte au plus $2m$ messages. C'est en fait un peu moins car dans l'instruction 3 chaque sommet $u \neq r_0$ n'envoie que $\deg(u) - 1$ messages. C'est donc au plus $\sum_u (\deg(u) - 1) + 1 = 2m - n + 2$, le terme « $+1$ » étant pour la correction de « $\deg(u) - 1$ » dans la somme $\sum_u$ lorsque $u = r_0$.

Maintenant, le nombre de fois où $\textsc{Layer}(u)$ peut être mise à jour est au plus $n - 1$ car $n - 1$ est la valeur maximum et cette valeur n'est mise à jour que si elle diminue d'au moins 1 (l'inférieur strict « $<$ » est très important ici). En fait, le nombre de fois que $\textsc{Layer}(u)$ peut changer est au plus $n - \mathrm{dist}_G(r_0, u)$ puisqu'une fois que $\textsc{Layer}(u) = \mathrm{dist}_G(r_0, u)$, $\textsc{Layer}(u)$ ne peut plus changer. Il suit que le nombre de messages est au plus $2m \cdot (n-1) = O(m \cdot n)$. □

[*Exercice*. Décrivez un graphe et un scenario catastrophe où Bellman-Ford_Dist comme décrit ci-dessus produit $\Omega(n^3)$ messages.]

Il n'est pas difficile de voir que le nombre de messages produits par Bellman-Ford_Dist dépend directement du nombre de fois où la variable $\textsc{Layer}(u)$ est mise à jour. Appelons $\mu(u)$ ce nombre. Du coup le nombre de messages vaut précisément :

$$\textsc{Message}(\textsf{Bellman-Ford\_Dist}(r_0), G) \;=\; \deg(r_0) + \sum_{u \neq r_0} \mu(u)(\deg(u) - 1) \;<\; 2m \cdot \mu \qquad (5.1)$$

où $\mu = \max_{u \in V(G)} \mu(u)$, soit le nombre maximum de fois qu'un sommet change sa variable $\textsc{Layer}(u)$. Suivant la valeur de μ, le nombre de messages peut être ainsi être moindre que celui avancé dans le théorème 5.1, en particulier si $\mu \ll n - 1$. C'est le cas si le réseau est faiblement asynchrone. [*Exercice*. Donnez un majorant du nombre de messages de Bellman-Ford_Dist dans un scenario synchrone.] [*Exercice**. Pour tout réel $\alpha \in [0, 1]$, on dira qu'un scenario est α-synchrone si chaque message prend un temps au moins α pour traverser une arête. Dit autrement, le temps de traversée prend un temps imprévisible de $[\alpha, 1]$. Le cas 1-synchrone correspond au cas synchrone et 0-synchrone au cas asynchrone. Donnez un majorant du nombre de messages Bellman-Ford_Dist lors d'un scenario α-synchrone, en fonction de n, m, D et bien sûr α.]

C'est en fait cet algorithme là qui est à la base du protocole principal d'Internet, BGP (*Border Gateway Protocol*). Malheureusement, dans le cas valué où les arêtes ont des poids arbitraires [6] (et pas seulement unitaire), Bellman-Ford_Dist peut consommer un nombre exponentiel de messages pour converger vers un arbre de plus courts chemins, comme l'explique la proposition 5.6. [*Exercice*. Comment faut-il modifier l'algorithme Bellman-Ford_Dist pour qu'il marche avec des poids arbitraires, en supposant que u connaît les poids $\omega(uv)$ de chacun de ses voisins v ?]

[*Cyril*. Que devient le temps dans le cas valué ? La proposition 5.5 ne tient plus.]

Proposition 5.6 *Il existe un graphe valué à n sommets et un scenario asynchrone où l'algorithme Bellman-Ford_Dist (version valuée) produit $2^{\Omega(n)}$ messages.*

Preuve. L'exemple est inspiré de [ABNG94, p. 2516]. On considère le graphe composé d'une chaîne de k triangles comme représenté sur la figure 5.5 qui possède au total $n = 2k+1$ sommets dont deux distingués $s = a_{k-1}$ et $t = a_0$. L'arête $a_i - b_i$ du triangle i, pour $i \in \{1,\dots,k\}$, est valuée 2^{i-1}, toutes les autres étant valuées 1.

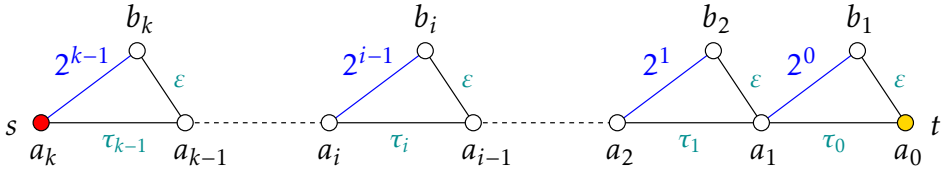


FIGURE 5.5 – Graphe valué à $2k+1$ sommets où Bellman-Ford_Dist(s) peut utiliser plus de 2^k messages dans un scenario asynchrone. Le temps de traversée des messages est fixé par τ_i et ε. Les arêtes bleutées ont des poids de la forme 2^i, tous les autres étant unitaires.

L'objectif est de montrer qu'il existe un scenario où tous les chemins entre s et t vont être construits par ordre strictement décroissant de coût.

La première remarque est que pour chaque entier $x \in [0, 2^k[$, il existe un chemin de s à t de coût $x+k$. Cela fait donc 2^k chemins différents. Pour le voir on considère le chemin

$$s = a_k - a_{k-1} - \cdots - a_1 - a_0 = t$$

de coût k de s à t utilisant les arêtes de base des triangles. Puis, si le bit $i-1$ dans l'écriture binaire de x vaut $x[i-1] = 1$, $i \in \{1,\dots,k\}$, on modifie localement le chemin passant par le triangle i en utilisant le détour par b_i. L'allongement est précisément de 2^{i-1} à cause de l'arête $a_i - b_i$, et ce indépendamment des autres bits de x. Le coût total de ce chemin est ainsi de $x+k$ car $x = \sum_{i=1}^{k} x[i-1] \cdot 2^{i-1}$. Voir un exemple figure 5.6.

La deuxième remarque est que chacun des 2^k chemins entre s et t, s'il est produit par l'algorithme, générera au moins un nouveau message puisqu'il induit un changement

6. Il est sous-entendu que les « poids » sont non nuls.

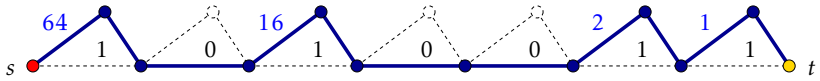


FIGURE 5.6 – Exemple de chemin de s à t pour $k = 7$ et $x = 1010011_2 = 64 + 16 + 2 + 1 = 83$. Son coût est $x + k = 90$.

de parent dans l'instruction 2 qui est suivit d'au moins [7] un SEND en 3. Il en résulte un total d'au moins $2^k = 2^{(n-1)/2} = 2^{\Omega(n)}$ messages comme désiré. Notons que l'algorithme construit toujours des chemins dans l'ordre décroissant de longueur.

Reste à montrer qu'un scenario peut effectivement construire tous ces chemins. On considère un scenario où le temps de transmission des messages circulant sur l'arête $a_i - a_{i-1}$ du triangle i vaut un certain $\tau_i \in]0,1]$, valeurs décroissantes avec i et qui seront définies précisément plus tard. Le temps de transmission des deux arêtes incidentes à b_i vaut $\varepsilon/2$ pour un certain ε suffisamment petit par rapport aux τ_i. Le temps de traversée des deux arêtes d'un chemin $a_i - b_i - a_{i-1}$ prend donc un temps ε, indépendamment de i.

On suppose qu'on lance Bellman-Ford_Dist depuis $r_0 = s$. L'intuition est la suivante. Au départ, le premier chemin construit de s à t passe par chaque b_i car le temps de traversée des $2k$ arêtes est $k\varepsilon$, valeur choisie $< \tau_i$ pour tout i, si bien qu'aucun message n'a encore pu traverser d'arête $a_i - a_{i-1}$. Le coût de ce premier chemin est $x + k$ avec $x = 2^k - 1 = 11\cdots11_2$. Puis progressivement, le chemin évolue et son coût diminue. Après un temps τ_i le sommet a_i informe le sommet a_{i-1} qu'un raccourci via $a_i - a_{i-1}$ existe, diminuant strictement le coût de 2^{i-1} sur la partie du chemin de s à a_{i-1}. L'impact sur la valeur courante de x est de passer $x[i-1]$ de 1 à 0. Cependant, dès que a_{i-1} est informé de ce changement, la diffusion à ses voisins, notamment vers b_{i-1} par les liens rapides, fait que le chemin se met à jour jusqu'à t en passant par tous les b_j avec $j < i$. Cela correspond alors à poser $x[j-1] = 1$ pour tous les $j < i$. La valeur x est alors la plus grande possible telle que $x[i-1] = 0$. Bien sûr, pour ne rater aucun chemin, il va falloir montrer qu'au moment où $x[i-1]$ repasse à 1, on avait bien $x[j] = 0$ pour tous les $j < i$.

Plus formellement (hypothèse d'induction), on suppose que lorsqu'un sommet a_i est informé d'un changement de distance à s, alors après une certaine durée d_i les 2^i chemins entre a_i et t auront été construits. On veut montrer que cela est vrai pour $i = k$, $d_i = (2^{i+1} - 2) \cdot \varepsilon$ et $\tau_i = 2\varepsilon + d_{i-1} = 2^i\varepsilon$. On peut choisir $\varepsilon = 1/2^k < 1$ si bien que le temps de transmission le plus long sera $\tau_k = 2^k \cdot \varepsilon = 1 \in]0,1]$ comme requis. Au passage, les 2^k chemins vont être construits en temps $d_k < 2$.

Pour $i = 0$, lorsque a_0 est informé, l'unique chemin réduit au sommet $a_0 = t$ est construit sans aucune communication. On en déduit donc qu'après que a_0 ait été informé, et après un temps $d_0 = (2^{0+1} - 2) \cdot \varepsilon = 0$, tous les chemins de a_0 à t ont bel et bien été construits. C'est donc vrai pour $i = 0$.

7. C'est en fait deux changements de PARENT : celui de a_{i-1} puis de b_i. Et comme c'est une diffusion à ses voisins moins le parent, cela génère en fait $(\deg(a_{i-1}) - 1) + (\deg(b_i) - 1) = 5$ messages.

Supposons la propriété vraie pour $i-1$, et montrons qu'elle est encore vraie pour i. Lorsque a_i est informé d'un nouveau chemin de s à a_i (notons T cet instant comme sur la figure 5.7), le chemin via $a_i - b_i - a_{i-1}$ est construit en temps ε. Pendant ce temps, un message transite sur l'arête $a_i - a_{i-1}$. Au moment $T+\varepsilon$, le sommet a_{i-1} est informé. Alors les 2^{i-1} chemins entre a_{i-1} et t se construisent (induction) ce qui dure d_{i-1}. Le message sur $a_i - a_{i-1}$ n'est toujours pas arrivé car on remarque que $\tau_i > \varepsilon + d_{i-1}$. Au temps $T+\tau_i$ le sommet a_{i-1} devient de nouveau informé d'un chemin de s à a_{i-1} encore plus court passant par $a_i - a_{i-1}$. À ce moment là, les 2^{i-1} chemins entre a_{i-1} et t se construisent de nouveau en temps d_{i-1}.

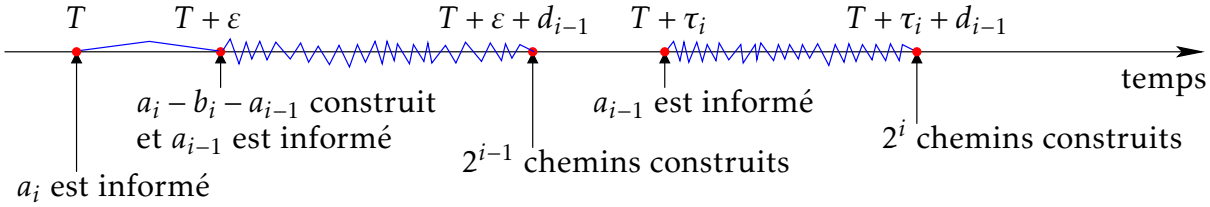


FIGURE 5.7 – Chronologie de la construction des 2^i chemins depuis a_i.

Entre a_i et t, lorsque a_i a été informé par s d'un nouveau chemin, $2 \cdot 2^{i-1} = 2^i$ chemins différents se sont donc construits. Il a fallut pour cela une durée de

$$\tau_i + d_{i-1} = (2\varepsilon + d_{i-1}) + d_{i-1} = 2\varepsilon + 2 \cdot (2^i - 2) \cdot \varepsilon = (2^{i+1} - 2) \cdot \varepsilon = d_i$$

ce qui prouve la propriété pour i et termine la preuve. □

[*Exercice*. Montrez comment modifier un seul sommet du contre-exemple de la proposition 5.6 pour que le nombre de messages produits par Bellman-Ford_Dist (version valuée) soit d'au moins $(\Delta-1) \cdot 2^{k-1} = 2^{\Omega(n \log \Delta)}$ dans le pire des cas d'un graphe valué à n sommets de degré maximum $\Delta \geqslant 4$.] [*Exercice*. En supposant que les poids des arêtes du graphe sont des entiers de $\{1, \ldots, W\}$. Donnez alors un majorant sur le nombre de messages produits par Bellman-Ford_Dist (version valuée). Même question si, de plus, le scenario est α-synchrone (cf. exercice précédent).]

On remarquera pour que le scenario catastrophe évoqué dans la proposition 5.6 se produise, il faut que : (1) l'*aspect ratio* du graphe, c'est-à-dire le ratio entre la longueur la plus grande et la plus petite dans le graphe, soit exponentiel en n ; et que (2) le scenario soit fortement asynchrone, typiquement que les temps de transmission le long des arêtes soient inversement proportionnels aux poids des arêtes.

Ces conditions sont rarement réunies en pratique. Cela signifie aussi que la simulation d'un algorithme distribué peut très bien passer à coté de scenarios catastrophes, en terme de graphes, de valuation et d'assynchronisme. Il faut donc se méfier. Une simulation, généralement, ne prouve absolument rien, malheureusement. Si elle est correctement programmée, au mieux, elle peut montrer des failles éventuelles de l'algorithme en exhibant un contre exemple (graphes et/ou scenarios catastrophes).

[*Exercice**. Montrez que le nombre de chemins simples (auto-évitants) dans un graphe à n sommets et de degré maximum Δ depuis un sommet fixé est au plus $P(n,\Delta) = 1+\Delta\sum_{i=0}^{n-2}(\Delta-1)^i$. En admettant que $P(n,\Delta) = (\Delta-1)^{n-1}\cdot(1+O(1/n))$, en déduire que Bellman-Ford_Dist (version valuée) produit au plus $2^{O(n\log\Delta)}$ messages. Pensez à les compter en fonction du nombre d'arêtes. D'après un exercice précédent, le nombre de messages produits peut atteindre $2^{\Omega(n\log\Delta)}$.] [*Exercice*. Soit $\Gamma(G,r_0)$ l'ensemble de tous les chemins simples issus de r_0 dans le graphe valué G. On notera $\omega(P)$ le coût d'un chemin P (somme des poids). Soit $C = |\{\omega(P) : P \in \Gamma(G,r_0)\}|$, c'est-à-dire le nombre de coûts différents pour les chemins de $\Gamma(G,r_0)$. Notez que $C \leqslant |\Gamma(G,r_0)|$. Montrez des exemples où $C = O(1)$ et $|\Gamma(G,r_0)| = \Omega(n^2)$. Pensez à des graphes de petit diamètre. Majorez le nombre de messages pour Bellman-Ford_Dist (version valuée) appliqué à (G,r_0), en fonction de C.]

5.1.3 Résumé

Il existe d'autres algorithmes de calcul d'arbres BFS en asynchrones, mais aucun n'est optimal à la fois en temps et en nombre de messages. Les plus sophistiqués sont basés sur des algorithmes synchrones avec l'utilisation de synchroniseurs que l'on verra au chapitre 6. La table 5.1 donne une synthèse.

	Message	Temps
Borne inférieure	$\Omega(m)$	$\Omega(D)$
Dijkstra_Dist	$O(m+nD)$	$O(D^2)$
Bellman-Ford_Dist	$O(mn)$	$O(D)$
[Asp05][BDLP08]	$O(mD)$	$O(D)$
Meilleur connu [AP90]	$O(m+n\log^3 n)$	$O(D\log^3 n)$

Table 5.1 – Compromis Temps/Message pour les algorithmes distribués en mode asynchrone pour le calcul d'un arbre BFS.

[*Cyril*. À revoir, en particulier si pour [AP90] et [AR93] il faut que le synchroniseur γ soit déjà construit ou pas.]

L'algorithmes [AP90][AR93] se généralisent aux graphes valués avec les mêmes complexités. L'algorithme Aspnes [Asp05], qu'on va brièvement présenter ci-après, utilise sans le dire le *synchroniseur* α sur l'algorithme Flood, notion que sera abordée au chapitre 6. L'algorithme [BDLP08] n'est qu'une implémentation particulière de l'algorithme Aspnes qui utilise moins de ressources mémoire et des messages de taille constante au lieu de $O(\log n)$ bits.

Dans l'algorithme Aspnes(r_0) un sommet u peut envoyer deux types de messages paramétrés par un entier d, exact(d) ou plus-que(d), pour dire qu'il sait qu'il est à distance

exactement d ou qu'il est à distance $> d$ de la racine r_0. Par exemple, on pourrait poser exact$(d) := (d, 0)$ et plus-que$(d) := (d, 1)$. Au départ on fait comme si tous les sommets avaient déjà envoyé et reçu un message plus-que(-1) de tous leurs voisins.

[*Cyril*. Il faudrait prendre un exemple concret avec disons 4 sommets et détailler l'implémentation. Il faut détailler le stockage pour chaque voisin d'une émission et réception d'une certaine valeur d. Disons, pour chaque sommet u, des mots binaires $R(d)[i]$ et $S(d)[i]$ indiquant si d a été reçu ou envoyé au i-ème voisin de u. Sans doute que seules les valeurs $d-2$, $d-1$ et d suffisent ?]

Algorithme Aspnes(r_0)
(code du sommet u)

La racine r_0 envoie exact(0) à tous ses voisins et termine.
Chaque $u \neq r_0$ envoie à tous ses voisins un message :

1. plus-que(d) dès qu'il a reçu et envoyé plus-que$(d-1)$ de tous ses voisins.
2. exact(d) dès qu'il reçoit un message exact$(d-1)$ (et il choisit cet émetteur comme parent), et qu'il a reçu plus-que$(d-2)$ et envoyé plus-que$(d-1)$ de tous ses voisins.

Notons que les sommets doivent stocker des messages pour pouvoir déterminer le type de message à envoyer. On se réfère à [Asp05] pour la validité de cet algorithme.

5.2 Arbres en profondeur d'abord

Il s'agit de réaliser un parcours DFS du graphe (de l'anglais *Depth First Search*). Plus précisément, on souhaite visiter tous les sommets d'un graphe en suivant les arêtes. En séquentiel on utilise un parcours en profondeur d'abord (DFS). Le principe est de démarrer depuis un sommet r_0 et pour chaque sommet v faire : si v a des voisins non encore visités, alors on en visite un. Sinon, on revient vers le sommet qui a visité v en premier. S'il aucun de ces sommets n'existe, la recherche est terminée. En séquentiel, cela prend un temps $\Theta(m)$, tous les voisins de tous les sommets pouvant être testés. L'Exercice 1 propose de détailler la version distribué de cet algorithme.

5.3 Arbres de poids minimum

Il s'agit de calculer un MST du graphe (de l'Anglais *Minimum Spanning Tree*), soit un arbre couvrant de poids minimum. C'est à ne pas confondre avec un arbre couvrant de plus courts chemins (*Shortest-Path Spanning Tree* ou SPST) qui se calcule en séquentiel avec l'algorithme de Dijsktra par exemple.

Chaque arête e du graphe G est supposé avoir un *poids*, c'est-à-dire un réel $\geqslant 0$, noté $\omega(e)$. Le problème de l'arbre de poids minimum consiste à déterminer l'arbre T couvrant G dont la somme des poids de ses arêtes est la plus petite posibles. Dans la version distribuée du problème, on supposera que chaque sommet possède une table des poids des arêtes de ses voisins.

[*Exercice*. En considérant un cycle à 5 sommets dont les arêtes ont des poids dans $\{1,\dots,5\}$, montrer que le SPST peut avoir un poids strictement supérieur au poids du MST.]

En séquentiel, il y a principalement deux algorithmes : Prim et Kruskal (voir la figure 5.8).

Pour Prim (1957), on maintient un arbre T de racine r_0 qui grossit en ajoutant à chaque fois l'arête sortante de poids minimum, c'est-à-dire l'arête incidente dont une seule extrémité est dans T et qui est de poids le plus petit possible. L'arbre est initialisé au sommet seul r_0. Sa complexité (en séquentiel) est de $O(m+n\log n)$ si on utilise un tas de Fibonacci.

Pour Kruskal (1956, puis redécouvert par Dijkstra en 1959), on trie les arêtes, puis on ajoute à la forêt l'arête la plus petite ne créant pas de cycle. Au départ $T=(V(G),\varnothing)$ ne contient aucune arrête. Sa complexité (en séquentiel) est de $O(m\log n)$, ce qui est plus lent mais plus simple à implémenter que Prim.

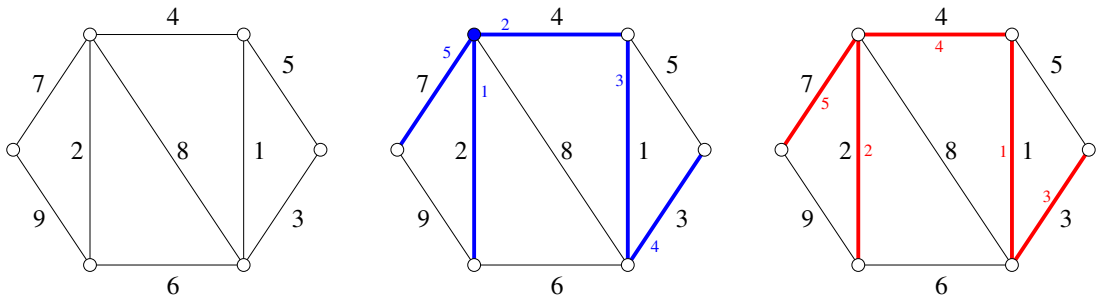


FIGURE 5.8 – Calcul d'un MST pour un graphe valué : en bleu avec l'algorithme de Prim, en rouge avec celui de Kruskal. Les numéros colorés sont l'ordre d'apparition des arêtes.

L'inconvénient de ces approches, très efficaces en séquentiel, est qu'elles proposent un ajout progressif d'arêtes, une par une, pour arriver à la solution. Ceci n'est pas facile à paralléliser et encore moins à réaliser par un algorithme distribué. De manière générale, plus il y a de dépendances entres les étapes, plus cela sera difficile à gérer en asynchrone (et nécessitera d'étapes de synchronisation). Pour être efficace, on cherche donc des algorithmes avec aussi peu de dépendances que possible, ce qui revient à découpe l'algorithme en peu d'étapes chacune avec ayant un maximum de parallélisme.

Remarque de culture générale.

Le problème du MST peut être généralisé de la manière suivante, appelé « arbre de Steiner » : on fixe un sous-ensemble de sommets, disons $U \subseteq V(G)$, et on demande de construire un arbre couvrant U et de poids minimum. Le problème du MST est un problème de Steiner particulier avec $U = V(G)$. Motivation : un groupe d'utilisateurs doit recevoir le même film vidéo et il faut optimiser le coût de diffusion.

La version euclidienne consiste à prendre pour G l'ensemble des points du plan, chaque sommet étant connecté à tous les autres (c'est donc un graphe complet infini) par une arête dont le poids représente la distance euclidienne entre ses extrémités. Les sommets de $G \setminus U$ utilisés dans la solution de coût minimum sont appelés points de Steiner (points noircis de la figure 5.9). Dans le plan, le gain en terme de coût entre l'arbre de Steiner et le MST est au mieux de $1 - \sqrt{3}/2 \approx 13\%$. L'arbre MST est donc une $2/\sqrt{3}$-approximation pour l'arbre de Steiner dans le plan.

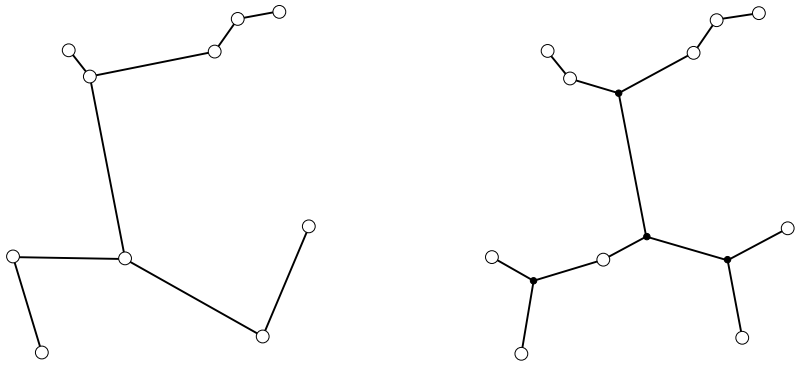


FIGURE 5.9 – MST dans le cas euclidien (à gauche) comparé avec l'arbre de Steiner (à droite) ici plus court d'environ 6%.

Calculer un arbre de Steiner est un problème est NP-complet (on ne connaît pas encore d'algorithme polynomial en n et on pense qu'il n'y en a pas), et il le reste même si tous les poids sont égaux ou dans le cas euclidien. Cependant, si $|U| = k$, il existe des algorithmes en $2^k \cdot n^{O(1)}$. Il est considéré comme un problème classique car c'est l'uns des 21 premiers problèmes qui ont été montrés NP-complets par Richard Karp en 1972 [Kar72] par réduction à la NP-complétude de SAT prouvé indépendamment par Leonid Levin et Stephen Cook en 1971 [Coo71], date considérée comme la naissance de la théorie de la NP-complétude.

5.3.1 Algorithme GHS

Il s'agit de l'algorithme de Gallager, Humblet et de Spira [GHS83]. Le principe et simple, mais pas les détails de l'implémentation. Bien qu'il fonctionne dans le mode

asynchrone, on ne donnera les détails de l'algorithme (simplifié) seulement dans le mode synchrone. L'algorithme original est optimal en nombre de messages, contrairement à la version présentée ici.

Il va être important que les arêtes aient des poids tous différents (sinon des cycles peuvent apparaître). On peut se passer de cette hypothèse soit en modifiant la relation d'ordre « < » sur les poids, soit en modifiant légèrement les poids eux mêmes.

Pour la première façon de faire, on peut départager les égalités de poids de manière consistante : l'arête ayant une extrémité avec le plus petit identifiant sera considérée comme la plus petite pour toutes celles de même poids. S'il y a encore égalité (les arêtes de même poids étaient incidentes au même sommet d'identifiant minimum) alors on les départages avec l'identifiant maximum, qui lui doit être différent pour chaque arête incidente au même sommet [8]. Dit autrement, pour une arête uv, il s'agit d'un ordre lexicographique sur les triplets

$$(\omega(uv), \min\{\mathrm{ID}(u), \mathrm{ID}(v)\}, \max\{\mathrm{ID}(u), \mathrm{ID}(v)\}) .$$

L'autre façon de faire [*Cyril. À passer en exercice ?*], si les poids sont des entiers [9] et qu'une borne stricte [10] B sur l'identifiant maximum est connue, est d'augmenter légèrement les poids pour les rendre uniques. Inutile alors de toucher à la relation « < » sur les poids. Par exemple, on pourra poser [11] :

$$\omega'(uv) := \omega(uv) + \mathrm{ID}(u)/B^2 + \mathrm{ID}(v)/B^3 \geqslant \omega(uv)$$

ce qui s'interprète plus facilement en représentant la partie décimale en chiffres de base B. Rappelons que multiplier ou diviser par B^i revient à décaler à droite ou à gauche le point décimale. En base B, on a :

$$\omega'(uv) = \omega(uv).\boxed{0}\boxed{\mathrm{ID}(u)}\boxed{\mathrm{ID}(v)} .$$

Il devient clair que la partie décimale $\mathrm{ID}(u)/B^2 + \mathrm{ID}(v)/B^3$ identifie l'arête uv. D'autre part la somme sur $n-1 < B$ termes quelconques $\omega'(uv)$, sur $n-1$ les arêtes d'un arbre couvrant par exemple, ne pourra pas affecter la partie entière de cette somme à cause de la présence du 0 (en base B) juste après le point décimal.

Dit autrement, pour tout arbre couvrant T :

$$\omega(T) \overset{(1)}{\leqslant} \omega'(T) \overset{(2)}{<} 1 + \omega(T) . \tag{5.2}$$

8. C'est le même procédé qu'on avait utilisé dans la proposition 3.4 pour associer un identifiant unique à chaque arête.

9. Si les poids ne sont pas entiers, on peut encore s'en sortir en tenant compte de l'écart minimum entre deux poids différents.

10. On veut dire par là que tous les identifiants sont dans $[0, B[$.

11. En pratique on multipliera tout par B^3 pour ne manipuler que des entiers.

Ces deux inégalités s'appliquent en particulier aux arbres couvrant de poids minimum T_ω et $T_{\omega'}$, calculés respectivement pour les poids ω et ω'. Notons que $T_{\omega'}$ est unique contrairement à T_ω. Comme T_ω est minimum pour ω et $T_{\omega'}$ est minimum pour ω', on a :

$$\omega(T_\omega) \overset{(1)}{\leqslant} \omega(T_{\omega'}) \quad \text{et} \quad \omega'(T_{\omega'}) \overset{(2)}{\leqslant} \omega'(T_\omega)\,. \tag{5.3}$$

En combinant toutes ces inégalités, plus précisément (5.3)(1), (5.2)(1) avec $T = T_{\omega'}$, (5.3)(2) et enfin (5.2)(2) avec $T = T_\omega$, on déduit que :

$$\omega(T_\omega) \overset{(1)}{\leqslant} \omega(T_{\omega'}) < 1 + \omega(T_\omega)\,. \tag{5.4}$$

Les poids $\omega(\cdot)$ étant entiers, il est clair que (5.4)(1) est en fait une égalité. Donc la somme des poids $\omega(\cdot)$ sur les arbres T_ω et $T_{\omega'}$ est la même. L'arbre $T_{\omega'}$ est donc bien un MST.

Cette solution cependant suppose de connaître une borne B sur l'identifiant maximum.

Au passage, on pourrait se demander comment faire si les sommets n'ont pas d'identifiant ? En fait on peut montrer, si le graphe est symétrique comme un cycle $u_0 - u_1 - \cdots - u_{n-1}$ de longueur paire où toutes les arêtes ont le même poids, qu'aucun algorithme distribué déterministe ne peut calculer de MST. Car, si $u_i - u_{i+1}$ décide d'ajouter l'arête à la solution, alors $u_{i+2} - u_{i+3}$ pourrait faire de même [12]. Du coup, dans un scénario parfaitement synchrone, soit toutes les arêtes sont dans la solution (créant un cycle), soit aucune, soit seule une sur deux (non connexe).

L'algorithme GHS est une variante de l'algorithme d'Otakar Borůvka (publié en 1926 sous le titre : *O jistém problému minimálním* — Sur un certain problème minimal), lui-même proche de celui de Kruskal. L'algorithme procède par phases successives, les sommets du graphe étant partitionés en sous-graphes connexes appelés *fragments*. Au départ, chaque sommet constitue un fragment.

Algorithme d'Otakar Borůvka

(principe de la phase p, voir l'exemple figure 5.10)

1. Chaque fragment détermine en parallèle son arête sortante de poids minimum.
2. Les fragments fusionnent grâce à ces arêtes.
3. On passe à la phase suivante s'il reste plus qu'un fragment.

Comme on va le voir dans la preuve de la proposition 5.7, l'algorithme s'arête après seulement $p = \lfloor \log_2 n \rfloor$ phases, ce qui donne un algorithme séquentiel de complexité $O(mp) = O(m \log n)$.

12. Notez que l'argument, si appliqué entre $u_i - u_{i+1}$ et $u_{i+1} - u_{i+2}$, ne marche pas. Car u_{i+1} peut distinguer ses deux arêtes en décidant, par exemple, que sa première arête est celle par qui il a reçu le premier message.

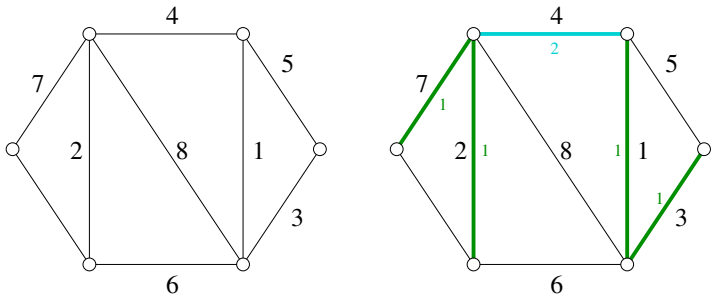


FIGURE 5.10 – Calcul d'un MST avec l'algorithme de Borůvka. Les numéros colorés sont les numéros de phase d'apparition des arêtes. Dans cet exemple, il y a que $2 = \lfloor \log_2 6 \rfloor$ phases.

Pour être plus précis, on pose T^* un MST pour le graphe G. Comme les poids sont uniques, la solution est également unique. On appelle *fragment* de G un sous-arbre de T^* (ou encore un sous-graphe connexe de T^*. Si F est un fragment qui ne couvre pas G, c'est-à-dire si $F \neq T^*$, on notera $\textsc{Out}(F)$ l'arête sortante de F de poids minimum, c'est-à-dire ayant une extrémité dans F et l'autre en dehors.

Propriété 5.1 *Si un fragment $F \neq T^*$, alors $F \cup \{\textsc{Out}(F)\}$ est aussi un fragment.*

Preuve. Si cela n'était pas le cas, il y aurait une autre arête sortante de F, disons $uv \in T^*$ avec $u \in F$, connectant F à $\overline{F}$, la composante connexe de $T^* \setminus F$ contenant v (cf. la figure 5.11). Notons que F et $\overline{F}$ sont des sous-arbres de T^*. Cependant l'arête $\textsc{Out}(F)$ connecte aussi F à $\overline{F}$. Par minimalité de $\omega(T^*)$ on doit avoir $\omega(uv) = \omega(\textsc{Out}(F))$ ce qui implique $uv = \textsc{Out}(F)$ à cause de l'unicité des poids.

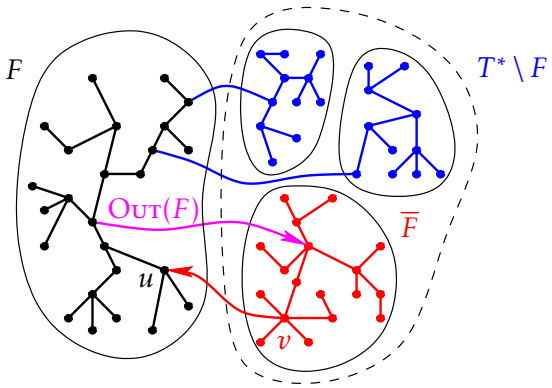


FIGURE 5.11 – Illustration de la preuve de la proposition 5.1 affirmant que l'arête $\textsc{Out}(F)$ doit être dans T^*.

□

Il découle de la propriété 5.1 que si F et F' sont deux fragments disjoints tels que $\textsc{Out}(F) = uv$ avec $u \in F$ et $v \in F'$, alors $F \cup \{uv\} \cup F'$ est aussi un fragment : c'est un sous-graphe connexe sans cycle (F et F' étant disjoints) et toutes les arêtes sont dans T^*. Notez bien qu'il est possible d'avoir $\textsc{Out}(F') \neq uv'$. Dans ce cas on peut former un plus grand fragment encore (comme dans l'exemple de la figure 5.12).

Le principe de l'algorithme GHS est donc de maintenir les fragments (qui sont des arbres enracinés), de calculer en phase parallèle pour chaque fragment F l'arête $\textsc{Out}(F)$ et de fusionner les fragments jusqu'en avoir plus qu'un. Le principe de chaque phase est détaillé ci-après.

On remarque qu'une suite de k fusions ne peut former de cycle comme $F_1 \cup \{e_1\} \cup F_2 \cup \cdots \cup F_k \cup \{e_k\} \cup F_1$ où $e_i = \textsc{Out}(F_i)$, puisqu'on aurait $\omega(e_1) > \cdots > \omega(e_k)$. En effet, si le fragment F_i a choisit $e_i = \textsc{Out}(F_i)$ c'est que $\omega(e_{i-1}) > \omega(e_i)$ car e_{i-1} et e_i sont toutes deux sortantes de F_i. Il suit que le fragment F_1 aurait une arête sortante e_k de poids strictement inférieur à $e_1 = \textsc{Out}(F_1)$: contradiction. Remarquez qu'il est crucial que la relation « < » sur le poids des arêtes soit un ordre strict, d'où l'hypothèse initiale des poids distincts.

De manière générale, la relation sur les fragments induit par Out (c'est-à-dire on met un arc $T \to T'$ si $\textsc{Out}(T)$ connecte T à T') possède une structure pseudo-acyclique où chaque composante connexe contient exactement deux fragments sont mutuellement en relation (cf. figure 5.12). On parle parfois de graphes fonctionnels (*functional graphs* ou *graphs of functions* en Anglais). [*Question.* Pourquoi il n'y a qu'une seule paire de fragments par composante mutuellement en relation ?] Dans le cas d'un choix mutuel, on peut garder comme racine de la nouvelle composante le fragment de plus petit ID (sur cette arête) et diffuser ce choix à toute les fragments de cette composante.

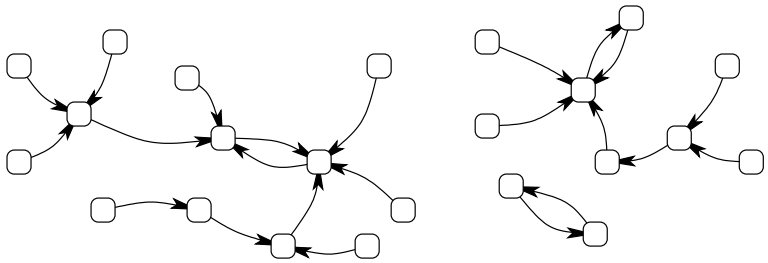


FIGURE 5.12 – Structure entre les fragments (=sommets) induite par la relation Out (=arcs). Chaque nouvelle composante connexe (ici trois) contient exactement deux fragments mutuellement en relation.

L'algorithme GHS fonctionne donc en phases synchronisées, chaque phase consistant à déterminer en parallèle pour chaque fragment T l'arête $\textsc{Out}(T)$. Puis les fragments sont fusionnés en mettant à jour les relations de parents et de fils, la racine du fragment, ainsi qu'une table décrivant pour chaque sommet la liste des arêtes sortantes du nouveau fragment (pour calculer la future arête Out).

On va maintenant donner quelques détails de l'implémentation dans le cas synchrone. Dans le cas asynchrone, même si certaines étapes restent valides, il faut faire en particulier attention aux communications impliquant des fragments de taille différente et au changement de phase.

Une des difficultés, qui arrivent même dans le cas synchrone, est que les fragments peuvent grandir à des vitesses assez différentes. Même si les plus petits doublent au moins à chaque étapes, il peut arriver que certains fragments deviennent rapidement très grands si bien que la synchronisation des phases entre fragments devient nécessaire afin de garantir un nombre d'étapes faible.

Pour simplifier, on supposera une synchronisation par phase que l'on va réaliser par le calcule préalable d'un arbre couvrant T_0, grâce à Flood, depuis un sommet distingué r_0. Cela prend un temps $O(D)$ et $O(m)$ messages, mais pré-suppose évidemment que le graphe soit connexe. On pourra se référer aussi à la figure 5.13. S'il y avait plusieurs composante connexe, il faudrait déterminer un sommet r_0 dans chacune d'elles en résolvant, par exemple, le problème de Leader Election.

Chaque sommet u possède tout au long de l'algorithme un premier ensembles de variables qui sont initialisées lors du calcul de T_0 :

- $\textsc{Frag}(u)$ identifiant de la racine de son fragment. Initialement $\textsc{Frag}(u) := u$;
- $\textsc{Parent}(u)$ et $\textsc{Fils}(u)$ donnant respectivement le parent et les fils dans l'arbre couvrant le fragment de u. Initialement $\textsc{Parent}(u) := \perp$ et $\textsc{Fils}(u) := \varnothing$.

Il possède également les variables suivantes qui sont initialisées au début de chaque nouvelle phase par l'étape 1 de synchronisation de l'algorithme :

- $\textsc{State}(u)$ indiquant l'état d'un sommet. Initialement $\textsc{State}(u) :=$ DONE.
- $\textsc{Border}(u)$ contenant la liste des voisins de u qui ne sont pas dans le fragment de u. Initialement $\textsc{Border}(u) := \varnothing$.
- $\textsc{Best}(u) = (u'v', w')$ étant l'arête sortante minimum et son poids. Dans un premier temps il va s'agir de l'arête minimum sortante de u, puis la minimum sortante parmi tous les descendants de u dans son fragment, u compris. Initialement $\textsc{Best}(u) := (uu, +\infty)$.
- $\mathrm{F}(u) \in \textsc{Fils}(u)$ étant le fils menant au descendant u' de u tel que $\textsc{Best}(u) = (u'v', w')$. Initialement $\mathrm{F}(u) := \perp$.
- $\mathrm{R}(u)$ étant une liste des messages « *READY* » reçus des voisins de $\textsc{Border}(u)$. Initialement $\mathrm{R}(u) := \varnothing$.

Algorithme GHS

(principe de la phase p en mode synchrone)

1. La racine r_0 diffuse dans T_0 un message « SYNC » indiquant à chaque racine de commencer une nouvelle phase.

2. Chaque sommet u qui est racine et qui reçoit un message « SYNC » diffuse dans son fragment un message « PULSE ».
3. Lors de cette diffusion, à la réception d'un message « PULSE », chaque sommet u (y compris la racine) :
   - Envoie un message « FRAG » à tous ses voisins, hors son parent et ses fils, pour déterminer localement l'arête sortante minimum. À la reception d'un message « FRAG » un sommet v doit renvoyer un message « $\textsc{Frag}(v)$ ».
   - Après avoir récupérer toutes les racines des fragments qu'il compare à $\textsc{Frag}(u)$, il met à jour ses variables $\textsc{Border}(u)$ et $\textsc{Best}(u)$.
   - Après avoir diffuser le message « PULSE » à tous ses voisins de $\textsc{Fils}(u)$, il passe dans l'état $\textsc{State}(u) :=$ PULSE.
4. Si u est une feuille de son fragment dans l'état PULSE, c'est-à-dire avec $\textsc{Fils}(u) = \varnothing$ et $\textsc{State}(u) =$ PULSE, il initie une concentration vers $\textsc{Frag}(u)$ avec un message « MIN, $\textsc{Best}(u)$ ».
5. Lors de cette concentration, chaque sommet u (y compris les feuilles) :
   - Attend de recevoir un message « MIN, $(u_i v_i, w_i)$ » de chacun de ses fils.
   - Détermine l'arête sortante minimum parmi celles reçues de ses fils et la compare à $\textsc{Best}(u)$. S'il en trouve une plus petite, il met à jour $\textsc{Best}(u)$ avec ce minimum et le fils $\mathrm{F}(u)$ dont est issu cette arête minimum [13].
   - Envoie un message « MIN, $\textsc{Best}(u)$ » à $\textsc{Parent}(u)$.
   - Il passe dans l'état $\textsc{State}(u) :=$ MIN.
6. Lorsque la racine du fragment passe dans l'état MIN, c'est-à-dire $u = \textsc{Frag}(u)$ et $\textsc{State}(u) =$ MIN, elle diffuse un message « READY, $u^*v^*, \mathrm{F}(u)$ » dans son fragment où $(u^*, v^*, w^*) = \textsc{Best}(u)$, tout en changeant partiellement le fragment comme ci-après.
7. Lors de cette diffusion, à la réception un message « READY, u^*v^*, f », chaque sommet u (y compris la racine) :
   - Diffuse aux sommets de $\textsc{Fils}(u) \cup \textsc{Border}(u)$ le message « READY, u^*v^*, f' » où $f' := \mathrm{F}(u)$ si $f = u$ and $f' := f$ sinon.
   - Si $f = u$, échange [14] $\textsc{Parent}(u)$ avec son fils $\mathrm{F}(u)$.
   - Pose $\textsc{Frag}(u) := u^*$, $\textsc{Best}(u) := (u^*v^*, 0)$, et passe dans l'état $\textsc{State}(u) :=$ READY indiquant que son fragment est prêt à fusionner via l'arête minimum u^*v^*.
8. À la reception d'un message « READY, ab, f » venant d'un sommet $u \in \textsc{Border}(v)$, un sommet v ajoute (ab, u) à sa liste $\mathrm{R}(v)$.
9. Lorsque qu'un sommet u est dans l'état READY, $\textsc{Best}(u) = (uv, w)$ et que $(uv, v) \in \mathrm{R}(u)$, alors u est la racine d'une arête double de fusion. Si $u < v$, il diffuse alors un

13. Notez que si $\textsc{Best}(u)$ n'a pas changé, on ne change pas $\mathrm{F}(u)$ qui vaut $\perp$.
14. Notez que si $u = u^*$, alors l'échange produit $\textsc{Parent}(u^*) := \perp$.

message « CAST, Frag(u) » dans son fragment et tout ceux qui veulent fusionner. Il applique pour cela l'étape suivante [15] :

10. Lors de cette diffusion un sommet u dans l'état READY et qui reçoit un message « CAST, r » (y compris l'initiateur pour lequel $r = \text{Frag}(u)$) :
    - $\text{Frag}(u) := r$.
    - Attend de recevoir dans $R(u)$ autant de messages que $|\text{Border}(u)|$.
    - Ajoute v à $\text{Fils}(u)$ pour tous les messages [16] tels que $(uv, v) \in R(u)$.
    - Diffuse à tous ses sommets de $\text{Fils}(u)$ le message « CAST, r ».
    - Passe dans l'état $\text{State}(u) :=$ CAST.
11. Un sommet u qui ne rediffuse aucun message « CAST, r » dans l'étape précédente concentre un message « DONE, f_1, f_2 » vers la racine r_0 de T_0, où f_1, f_2 sont respectivement les plus petits et le plus grands identifiants des fragments rencontrés pendant la concentration. Il faut ici attendre de recevoir l'information de ses fils dans T_0, et pour les feuilles $f_1 := f_2 := \text{Frag}(u)$. Lorsque r_0 a reçu un message « DONE, f_1, f_2 » de tous les sommets du graphe on recommence l'étape 1 de synchronisation sauf si $f_1 = f_2$ signifiant qu'il n'y qu'un seul fragment, et l'algorithme s'arrête.

Proposition 5.7 *Pour tout graphe G, en mode synchrone ou asynchrone :*

- $\text{Temps}(GHS, G) = O(n \log n)$.
- $\text{Message}(GHS, G) = O(m \log n)$.

Preuve. L'observation principale est qu'à chaque étape, le fragment de plus petite taille, en nombre de sommets, double au moins à l'étape suivante. En effet, par définition, le plus petit fragment fusionne avec un fragment de taille au moins aussi grande. Il en résulte un nouveau fragment de taille au moins le double.

Donc, par induction, après $p \geqslant 0$ étapes, le plus petit fragment est de taille au moins 2^p, étant de taille $2^0 = 1$ au départ. L'algorithme s'arrête lorsqu'il ne reste qu'un seul fragment, ce qui arrive dès que le plus petit atteint une taille $> n/2$. [*Question. Pourquoi ?*]. On cherche donc le plus petit nombre d'étapes p tel que $2^p > n/2 \Leftrightarrow p > \log_2(n/2)$. On vérifie facilement que :

$$\lfloor \log_2(n/2) \rfloor \leqslant \log_2(n/2) < \lfloor \log_2(n/2) \rfloor + 1$$

et donc l'entier cherché est $p = \lfloor \log_2(n/2) \rfloor + 1 = \lfloor \log_2 n \rfloor$.

Le temps d'une phase est proportionnel à la profondeur des fragments fusionnés, soit $O(n)$. Donc $\text{Temps}(GHS, G) = O(np) = O(n \log n)$.

15. Un peu comme s'il s'était diffusé à lui même le message « CAST, Frag(u) ».
16. On peut aussi diffuser à ces fils là sans atteindre le message « CAST ».

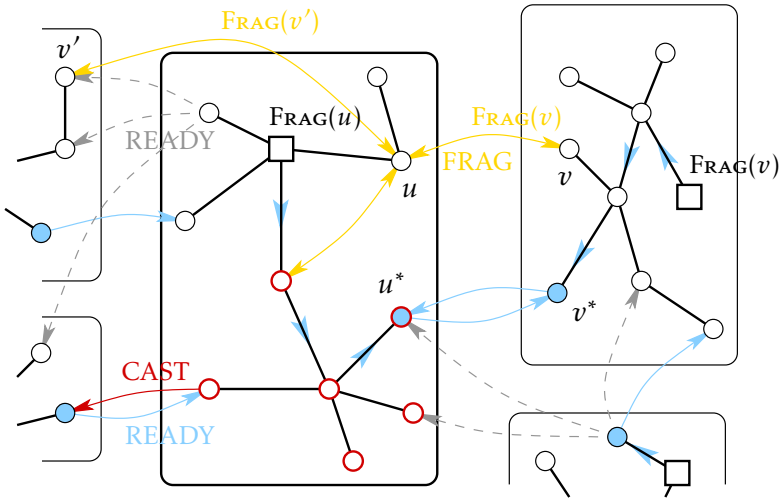


Figure 5.13 – Les différents messages envoyés pendant l'exécution d'une phase de l'algorithme GHS. Après une réorientation du fragment, le sommet u^* devient la nouvelle racine de son fragment, tout comme chacun des sommets cyan. Ici $u^* < v^*$ et donc le sommet u^* devient le premier sommet à passer dans l'état CAST, état qui se diffuse ensuite dans son fragment et au-delà. Le sommet u^* sera la racine du nouveau fragment résultant de toutes les fusions.

Le nombre de messages d'une phase est $O(n)$ pour déterminer Out. Mais la mise à jour des voisins qui sont dans le nouveau fragment, nécessite un total de $O(m)$ messages. Cette mise à jour est nécessaire pour calculer Out puisqu'on doit distinguer à un moment ou à un autre les arêtes externes des arêtes internes d'un fragment. D'où $\text{Message}(\text{GHS}, G) = O(mp) = O(m \log n)$. □

5.3.2 Notes bibliographiques

En combinant l'algorithme de Borůvka et de Prim on peut obtenir un algorithme séquentiel de meilleure complexité. En exécutant d'abord Borůvka pendant $t = \log\log n$ étapes, et en contractant les fragments obtenus, on obtient un graphe avec au plus $n' = n/2^t = n/\log n$ sommets. On peut alors terminer la construction du MST avec Prim en $O(m + n' \log n') = O(m + n)$. Au total cela donne un algorithme en $O(mt + m + n) = O(m \log\log n + n)$.

Dans le cas séquentiel, il existe un algorithme probabiliste linéaire en moyenne [KKT95] et un algorithme déterministe quasi-linéaire en $O(m \cdot \alpha(m,n))$ où $\alpha(m,n)$ est la fonction inverse d'Ackermann [17] [Cha00][Pet99]. Ce dernier algorithme est réputé être extrêmement complexe. En fait [PR02] ont construit un algorithme qui s'exécute en un temps proportionnel au nombre optimal de comparaisons pour déterminer la solution. Ce nombre n'est pas connu mais qui est au plus $O(m \cdot \alpha(m,n))$. Il est également connu que si les poids sont des entiers représentés en binaires, alors le MST peut être calculé en $O(m+n)$ opérations entières [FW94]. De nombreuses explications et notes bibliographiques sur le calcul séquentiel peuvent être trouvés ici.

Des progrès sur le problème du MST en distribué ont été réalisés depuis [GHS83]. En particulier il a été démontré que tout algorithme distribué nécessitait un temps $\Omega(\sqrt{n}/b + D)$ dans le modèle b-CONGEST [PR00] (donc en synchrone mais avec des messages de b bits), et ceci même pour des graphes de diamètre aussi faible que $D = \Omega(\log n)$. Le meilleur algorithme connu, toujours dans le modèle b-CONGEST avec $b = O(\log n)$, se rapproche de cette borne avec une complexité en temps [18] de $O(\sqrt{n}\log^* n + D)$ [GKP98][KP98], mais la complexité en nombre de messages est loin de l'optimal. La borne inférieure sur le nombre de message est $\Omega(m + n\log n)$. L'algorithme réalisant le meilleur compromis entre le temps et le nombre de messages, dans le modèle CONGEST, est de $O((\sqrt{n} + D)\log n)$ pour la complexité en temps et de $O(m \log n + n \log n \log^* n)$ pour la complexité en messages [Elk17].

Pour certaines classes de graphes il existe des algorithmes plus performants encore.

17. Plus précisément $\alpha(m,n) = \min_{i \geqslant 1}\{A(i, 4\lceil m/n\rceil) > \log n\}$ avec : $A(0,j) = 2j$ pour $j \geqslant 0$; $A(i,0) = 0$ et $A(i,1) = 2$ pour $i \geqslant 1$; $A(i,j) = A(i-1, A(i,j-1))$ pour $i \geqslant 1$ et $j \geqslant 2$.

18. La fonction $\log^* n$ sera abordée dans le chapitre suivant dans la section 7.4.3. Elle est très peu croissante, beaucoup moins que $\log\log n$ par exemple. Il faut juste retenir que pour toute valeur pratique de n, $\log^* n \leqslant 5$.

Par exemple, pour les graphes planaires, il existe des algorithmes en temps $O(D \log D \cdot \log n)$ toujours dans le modèle CONGEST [GH15][GH16]. De manière plus générale, on peut calculer un MST en temps $O(\sqrt{g} \cdot D \log D \cdot \log n)$ pour tout graphes dessinables sur une surface de genre g [HIZ16] et au-delà [GH21].

5.4 Exercices

Exercice 1

Écrire un algorithme de paramètre r_0 implémentant le DFS décrit dans le cours section 5.2 qui optimise le temps, c'est-à-dire que le jeton doit passer exactement deux fois par chaque arête d'un arbre couvrant, et passer par tous les sommets.

Bibliographie

[ABNG94] B. Awerbuch, A. Bar-Noy, and M. Gopal, *Approximate distributed Bellman-Ford algorithms*, IEEE Transactions on Communications, 42 (1994), pp. 2515–2517. doi : 10.1109/26.310604.

[AP90] B. Awerbuch and D. Peleg, *Network synchronization with polylogarithmic overhead*, in 31st Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, October 1990, pp. 514–522. doi : 10.1109/FSCS.1990.89572.

[AR93] Y. Afek and M. Ricklin, *Sparser : A paradigm for running distributed algorithms*, Journal of Algorithms, 14 (1993), pp. 316–328. doi : 10.1006/jagm.1993.1016.

[Asp05] J. Aspnes, *Distributed breadth-first search*, in Notes on Theory of Distributed Systems, Yale University, 2005, ch. 5.

[BDLP08] C. Boulinier, A. K. Datta, L. L. Larmore, and F. Petit, *Space efficient and time optimal distributed BFS tree construction*, Information Processing Letters, 108 (2008), pp. 273–278. doi : 10.1016/j.ipl.2008.05.016.

[Cha00] B. Chazelle, *A minimum spanning tree algorithm with inverse-Ackermann type complexity*, Journal of the ACM, 46 (2000), pp. 1028–1047. doi : 10.1145/355541.355562.

[Coo71] S. A. Cook, *The complexity of theorem proving procedures*, in 3rd Annual ACM Symposium on Theory of Computing (STOC), ACM Press, May 1971, pp. 151–158. doi : 10.1145/800157.805047.

[Elk17] M. Elkin, *A simple deterministic distributed MST algorithm, with near-optimal time and message complexities*, in 36th Annual ACM Symposium on Prin-

ciples of Distributed Computing (PODC), ACM Press, July 2017, pp. 157–163. DOI : 10.1145/3087801.3087823.

[FW94] M. L. Fredman and D. E. Willard, *Trans-dichotomous algorithms for minimum spanning trees and shortest paths*, Journal of Computer and System Sciences, 48 (1994), pp. 533–551. DOI : 10.1016/S0022-0000(05)80064-9.

[GH15] M. Ghaffari and B. Haeupler, *Distributed algorithms for planar networks I : Planar embedding*, July 2015.

[GH16] M. Ghaffari and B. Haeupler, *Distributed algorithms for planar networks II : Low-congestion shortcuts, MST, and min-cut*, in 27th Symposium on Discrete Algorithms (SODA), ACM-SIAM, January 2016, pp. 202–219. DOI : 10.1137/1.9781611974331.ch16.

[GH21] M. Ghaffari and B. Haeupler, *Low-congestion shortcuts for graphs excluding dense minors*, in 40th Annual ACM Symposium on Principles of Distributed Computing (PODC), A. Miller, K. Censor-Hillel, and J. H. Korhonen, eds., ACM Press, July 2021, pp. 213–221. DOI : 10.1145/3465084.3467935.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira, *A distributed algorithm for minimum-weight spanning trees*, ACM Transactions on Programming Languages and Systems, 5 (1983), pp. 66–77. DOI : 10.1145/357195.357200.

[GKP98] J. A. Garay, S. Kutten, and D. Peleg, *A sublinear time distributed algorithm for minimum-weight spanning trees*, SIAM Journal on Computing, 27 (1998), pp. 302–316. DOI : 10.1137/S0097539794261118.

[HIZ16] B. Haeupler, T. Izumi, and G. Zuzic, *Near-optimal low-congestion shortcuts on bounded parameter graphs*, in 30th International Symposium on Distributed Computing (DISC), vol. 9888 of Lecture Notes in Computer Science (ARCoSS), Springer, September 2016, pp. 158–172. DOI : 10.1007/978-3-662-53426-7_12.

[HSS07] K. R. Hutson, T. L. Schlosser, and D. R. Shier, *On the distributed Bellman-Ford algorithm and the looping problem*, INFORMS Journal on Computing, 19 (2007), pp. 542–551. DOI : 10.1287/ijoc.1060.0195.

[Kar72] R. M. Karp, *Reducibility among combinatorial problems*, in The IBM Research Symposia Series, R. Miller and J. Thatcher, eds., NY : Plenum Press, 1972, pp. 85–103.

[KKT95] D. R. Karger, P. N. Klein, and R. E. Tarjan, *A randomized linear-time algorithm to find minimum spanning trees*, Journal of the ACM, 42 (1995), pp. 321–328. DOI : 10.1145/201019.201022.

[KP98] S. Kutten and D. Peleg, *Fast distributed construction of small k-dominating sets and applications*, Journal of Algorithms, 28 (1998), pp. 40–66. DOI : 10.1006/jagm.1998.0929.

[Pet99] S. Pettie, *Finding minimum spanning trees in $O(m\alpha(m,n))$*, Tech. Rep. TR-99-23, University of Texas, October 1999.

[PR00] D. Peleg and V. Rubinovich, *A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction*, SIAM Journal on Computing, 30 (2000), pp. 1427–1442. doi : 10.1137/S0097539700369740.

[PR02] S. Pettie and V. Ramachandran, *An optimal minimum spanning tree algorithm*, Journal of the ACM, 49 (2002), pp. 16–34. doi : 10.1145/505241.505243.

[Wal07] D. Walden, *The bellman-ford algorithm and "distributed bellman-ford"*, May 2007. https://www.walden-family.com/public/bf-history.pdf.

CHAPITRE

6 Synchroniseurs

Sommaire

La conception et la mise au point d'un algorithme distribué pour un système asynchrone est bien plus difficile que pour un système synchrone. L'analyse de sa validité doit prendre en compte tous les scénarios possibles, alors qu'un synchrone il n'y

a, le plus souvent, qu'un seul scénario possible. Pour s'en convaincre, il suffit de considérer le problème de la construire d'un arbre BFS (vu au chapitre 5) et de mettre en balance les analyses des algorithmes Flood et Dijkstra_Dist.

L'idée, introduite en 1982 par Gallager [Gal82] puis généralisée par Awerbuch en 1985 [Awe85], est de transformer automatiquement un algorithme synchrone pour qu'il puisse fonctionner sur un système asynchrone. Le synchroniseur simule donc des envois de messages synchrones, au top d'horloge t, sur un système asynchrone, sans horloge. Ceci se fait en payant un surcoût en nombre de messages dits de synchronisation et des pénalités sur le temps. Il faut également prendre en compte de temps de construction éventuelle du synchroniseur. Plusieurs synchroniseurs sont possibles, chaque fois avec des compromis différents sur ces surcoûts en messages et en temps. Les synchroniseurs que l'on présentera, une fois construits, utiliseront des messages de taille constante.

Mots clés et notions abordées dans ce chapitre :

- synchroniseurs α, β, γ,
- *spanners*, synchroniseurs δ_k.

6.1 Méthodologie

Soit S un algorithme synchrone écrit pour un système synchrone et σ un synchroniseur. On souhaite construire un nouvel algorithme $A = \sigma(S)$, obtenu en combinant σ et S, qui puisse être exécuté sur un système asynchrone. Pour que la simulation soit correcte, il faut que l'exécution de A sur un système asynchrone soit « similaire » à l'exécution de S sur un système synchrone (on donnera plus tard un sense plus précis à « exécutions similaires »).

Dans chacun des sommets, l'algorithme A a deux composantes :

1. La composante originale représentant l'algorithme S que l'on veut simuler, c'est-à-dire toutes les variables locales, les routines et les types de message utilisés par S.
2. La composante de synchronisation, c'est-à-dire le synchroniseur σ, qui peut comprendre ces propres variables locales, routines et types de message utilisés.

Ces deux composantes doivent être bien distinctes puisque pendant l'exécution de A, par exemple, des messages des deux composantes vont être amener à coexister. Les messages « ACK » originaux de S ne doivent pas être confondus avec ceux ajoutés par le synchroniseurs.

L'idée de base est que chaque processeur u possède un générateur de *pulse* (= top horloge), $\textsc{Pulse}(u) = 0, 1, 2, \ldots$ variable qui intuitivement simule les tops horloges d'une horloge globale. Dans l'algorithme A on va exécuter la ronde i de l'algorithme S (grâce à la composante originale) uniquement lorsque $\textsc{Pulse}(u) = i$.

Malheureusement, après avoir exécuté la ronde i de S, u ne peut pas se contenter d'incrémenter $\textsc{Pulse}(u)$ et de recommencer avec la ronde $i+1$. Il faut attendre ses voisins

puisque sinon un voisin v pourrait recevoir des messages de u de la ronde $i+1$ alors que v est toujours dans la ronde i. La situation peut être encore moins favorable, avec un décalage pire encore. Il pourrait se produire par exemple qu'aux 5 rondes suivantes, $i+1, i+2, \ldots, i+5$ de S, il ne soit pas prévu que u envoye et reçoive des messages. Si bien que u pourrait effectuer tous ses calculs locaux de la ronde $i+1$ à la ronde $i+5$ et passer directement à $\textsc{Pulse}(u) = i+6$. Si à la ronde $i+6$ il envoie un message à v, v qui est dans la ronde i aurait légitimement l'impression de recevoir des messages du futur.

Pour résoudre le problème, on pourrait penser à première vue qu'il suffit de marquer chaque message par le numéro de ronde de l'émetteur, et ainsi retarder les messages venant du futur. La question qui se pose est alors de savoir combien de temps il faut les retarder. Malheureusement, un processeur v ne peut pas savoir *a priori* combien de messages de la ronde i il est censé recevoir. Sans cette information v pourrait attendre de recevoir un message de la ronde i, alors qu'à la ronde i de S il est peut être prévu qu'aucun voisin ne communique avec v. Ainsi l'envoi de messages spécifiques, dit de synchronisation, est nécessaire.

En asynchrone, lorsqu'un sommet attend de recevoir un message il attend potentiellement pour toujours. La réception étant bloquante et il ne peut pas avoir de *timeout* puisqu'il n'y a pas d'horloge. En synchrone, les réceptions se débloquent à la fin de chaque ronde.

Comme on va le voir dans la proposition 6.1, pour que la simulation fonctionne, il suffit de garantir ce qu'on appelle la « compatibilité de pulse ».

Définition 6.1 (Compatibilité de pulse) *Si un processeur u envoie à un voisin v un message M de la composante originale à la ronde i lorsque $\textsc{Pulse}(u) = i$, alors M est reçu en v lorsque $\textsc{Pulse}(v) = i$.*

C'est la définition du synchronisme donnée au paragraphe 1.3.4 qui disait que les messages émis lors de la ronde i sont reçus à la ronde i de leurs voisins.

Bien sûr, les exécutions de A et de S ne peuvent pas être identiques, car d'une part sur un système asynchrone, les exécutions ne sont pas déterministes (deux exécutions sur les mêmes entrées ne donne pas forcément le même résultat !), et que d'autre part les messages de σ dans l'exécution de A n'existent évidemment pas dans l'exécution de S.

On donne maintenant un sens plus précis à la notion de similarité entre les exécutions de S et de A.

Définition 6.2 (Exécutions similaires) *Les exécutions de S sur un système synchrone et de $A = \sigma(S)$ sur un système asynchrone sont* similaires *si pour tout graphe G (l'instance) sur lesquels s'exécutent S et A, toute variable X, toute ronde i et toute arête uv de G :*

- *Les messages de la composante originale envoyés (ou reçus) par u sur l'arête uv lorsque* $\textsc{Pulse}(u) = i$ *sont les mêmes que ceux de S à la ronde i envoyés (ou reçus) par u sur l'arête uv.*
- *La valeur de X en u au début de l'exécution de A lorsque* $\textsc{Pulse}(u) = i$ *est la même que celle de X en u dans l'exécution du début de la ronde i de S. La valeur de X en u lorsque A est terminé est la même que celle de X en u lorsque S est terminé.*

Pour illustrer la définition précédente supposons que $S = \mathsf{SpanTree}$, qui calcule un arbre couvrant BFS (en largeur d'abord) sur un système synchrone. Si $A = \sigma(S)$ et S ont des exécutions similaires, alors en particulier les variables $\textsc{Parent}(u)$ et $\textsc{Fils}(u)$ seront identiques à la fin des exécutions de A et de S. Elles définiront alors les mêmes arbres. Il en résulte que l'algorithme A calcule un arbre couvrant BFS sur un système asynchrone.

Proposition 6.1 *Si le synchroniseur σ garantit la compatibilité de pulse pour tout algorithme synchrone S, toute ronde, tout graphe G et tout sommet, alors les exécutions de S et de $\sigma(S)$ sur G sont similaires.*

La question principale est donc : « comment implémenter la compatibilité de pulse ? c'est-à-dire quand est-ce que u doit incrémenter $\textsc{Pulse}(u)$? »

Pour cela nous avons encore besoin de deux notions.

- Le sommet u est $\textsc{Safe}$ pour la ronde i, et on note $\textsc{Safe}(u,i)$ le prédicat correspondant, si tous les messages originaux à la ronde i envoyés de u à ses voisins ont été reçus. (En gros, u a terminé la ronde i.)
- Le sommet u est $\textsc{Ready}$ pour la ronde $i+1$, et on note $\textsc{Ready}(u,i+1)$ le prédicat correspondant, si tous ses voisins sont $\textsc{Safe}$ pour la ronde i. (En gros, u est prêt pour la ronde suivante.)

La proposition suivante nous indique donc quand $\textsc{Pulse}(u)$ peut être incrémenté.

Proposition 6.2 *Si $i = \textsc{Pulse}(u)$, si $\textsc{Safe}(u,i)$ et $\textsc{Ready}(u,i+1)$ sont vraies, alors la compatibilité de pulse de la ronde i est garantie.*

[*Cyril*. À revoir. Il faut introduire la règle READY (incrémenter si $\textsc{Safe}(u,i)$ et $\textsc{Ready}(u,i+1)$) et la règle DELAY (ralentir les messages du futur). Et dire : « Si les règles READY et DELAY sont appliquées, alors la compatibilité de pulse pour la ronde i est grantie. »]

On peut montrer qu'une seule des deux conditions ne suffit pas à garantir la compatibilité de pulse.

Preuve. Il faut montrer qu'un message M envoyé de u vers un voisin v lorsque $\textsc{Pulse}(u) = i$ est bien reçu lorsque $\textsc{Pulse}(v) = i$. Comme $\textsc{Ready}(u,i+1)$ est vraie, $\textsc{Safe}(v,i)$ est vraie.

[*Cyril*. À FINIR] □

6.2 Mesures de complexité

Grâce à la proposition 6.2 il suffit d'implémenter les propriétés Safe(u,i) et Ready$(u,i+1)$ pour savoir lorsqu'il faut incrémenter Pulse(u). On a donc deux étapes à réaliser, illustrées sur la figure 6.1.

Étape Safe. À chaque message M de la composante originale reçu on renvoie un message « ACK ». Ainsi l'émetteur u peut savoir lorsque tous ses messages de la ronde $i =$ Pulse(u) ont été reçus.

Étape Ready. Lorsqu'un sommet v devient Safe pour la ronde $i =$ Pulse(v) il diffuse cette information à ses voisins via un message « SAFE ». Il indique ainsi à son voisin u la possibilité d'incrémenter Pulse(u).

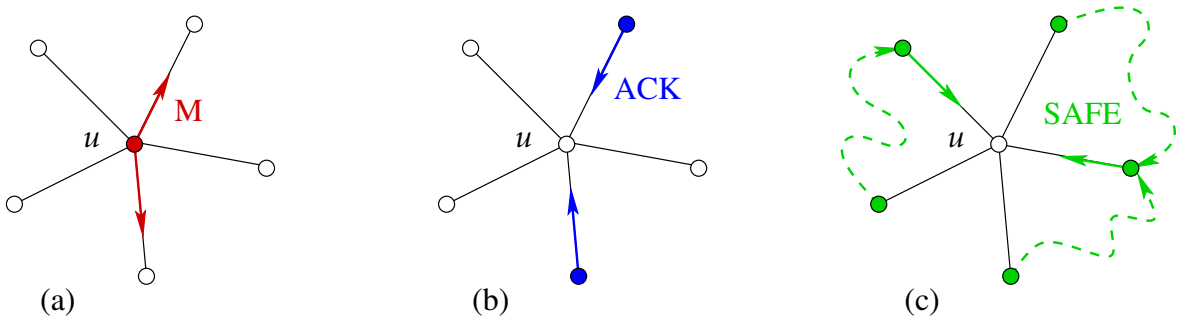


Figure 6.1 – (a) Messages envoyés à la ronde i depuis u à certains de ses voisins par la composante d'origine. (b) Messages « ACK » de l'étape Safe. (c) Messages « SAFE » de l'étape Ready issus des voisins de u mais transitant seulement par certains voisins.

L'étape Safe n'est pas très couteuse. Le temps et le nombre de messages sont au plus doublés. L'étape Ready est généralement plus coûteuse puisque u peut, à une ronde donnée, recevoir potentiellement plus de messages « SAFE » qu'il n'a émis de messages originaux. Par exemple, u pourrait n'avoir émis aucun messages originaux à la ronde i, et pourtant u devra recevoir un message « SAFE » de tous ses voisins pour pouvoir incrémenter Pulse(u).

Notons cependant qu'il n'y a aucune raison pour que v envoie son message « SAFE » à u directement sur l'arête vu. Certes c'est la solution la plus simple et rapide de communiquer l'information. Mais si le temps n'est pas le critère principal on peut faire différemment et économiser des messages. Ce qui importe à u c'est de savoir quand tous ses voisins sont Safe. Un autre sommet pourrait jouer le rôle de collecteur des messages « SAFE ». Et, dans un deuxième temps, ce sommet pourrait informer u par un message différent, disons « PULSE », que tous ses voisins sont Safe pour la ronde i (cf. figure 6.2).

Notons qu'à un moment donné, il pourrait avoir $2m$ messages « SAFE » émis si tous les sommets sont Safe en même temps et décident d'émettre à tous leurs voisins, alors

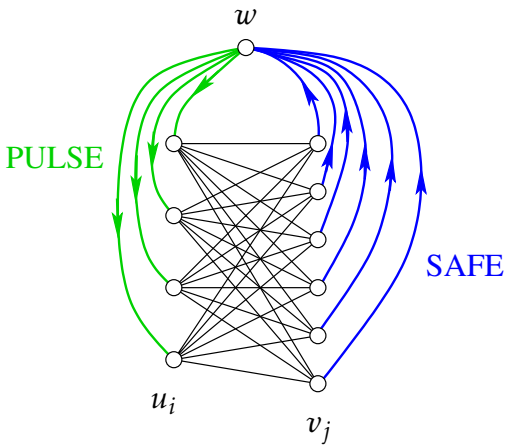


FIGURE 6.2 – Un sommet w chargé de collecter les messages « SAFE » et de transmettre un message « PULSE » aux sommets u_i lorsque tous leurs voisins v_j sont SAFE. Dans cet exemple il y a $6+4$ messages échangés contre 6×4 si l'on avait utilisé les arêtes $v_i u_j$.

qu'il n'y a jamais que n sommets intéressés de recevoir un message « PULSE ». Il serait donc envisageable de n'utiliser que n messages « PULSE » et de ne pas diffuser « SAFE » à tous ses voisins.

Les synchroniseurs diffèrent sur l'étape READY, l'étape SAFE étant commune à tous les synchroniseurs que l'on va présenter.

On note respectivement $T_{\text{init}}(\sigma)$ et $M_{\text{init}}(\sigma)$ le temps et le nombre de messages pour initialiser la composante de synchronisation.

Notons toute de suite que l'initialisation de σ est toujours au moins aussi couteuse qu'une diffusion car il faut au minimum que chaque sommet u exécute PULSE$(u) := 0$ et que tous les sommets aient l'impression de commencer S en même temps la première ronde. Donc généralement, $T_{\text{init}}(\sigma) = \Omega(D)$ et $M_{\text{init}}(\sigma) = \Omega(m)$. On applique donc l'outil général des synchroniseurs aux algorithmes distribués de complexité en temps $\Omega(D)$ et en nombre de messages $\Omega(m)$.

Comme l'algorithme original S fonctionne par rondes, pour mesurer l'efficacité d'un synchroniseur on s'intéresse aux surcoûts en temps et en nombre de messages générés par la simulation entre deux rondes consécutives.

On note $M_{\text{pulse}}(\sigma)$ le nombre maximum de messages de la composante de synchronisation générés dans tout le graphe entre deux changements de pulse consécutifs, hormis les messages « ACK » de l'étape SAFE. Si on ne compte pas ces messages dans $M_{\text{pulse}}(\sigma)$ c'est parce qu'ils correspondent toujours au nombre de messages envoyés pendant deux changements de pulse et que ce nombre peut être très variable d'une ronde à une autre. Il dépend de S, pas vraiment de σ.

Enfin, on note $T_{\text{pulse}}(\sigma)$ la durée (en unité de temps) entre l'instant où le dernier

sommet passe sa pulse[1] à i et l'instant où le dernier sommet passe à sa pulse à $i+1$, maximisée sur tous les i. Plus formellement, en notant par $t_{\text{last}}(i)$ l'instant où le dernier sommet passe sa pulse à i, on a :

$$T_{\text{pulse}}(\sigma) := \max_{i \geqslant 0} \{t_{\text{last}}(i+1) - t_{\text{last}}(i)\} .$$

Donc cette fois-ci, contrairement à $M_{\text{pulse}}(\sigma)$, $T_{\text{pulse}}(\sigma)$ inclus le temps du message original de S, mais cette contribution est toujours d'une unité.

Attention ! On n'a pas dit que pour un sommet u donné le temps de changement de pulse est borné par $T_{\text{pulse}}(\sigma)$. Cela pourrait être plus ! Car, même si v est le dernier sommet à passer sa pulse à $i+1$, il est possible que la pulse de v soit passée à i un peu avant le dernier, disons u, comme illustré sur la figure 6.3. [*Exercice*. Quelle est la durée maximale entre deux pulses pour un sommet donné ?]

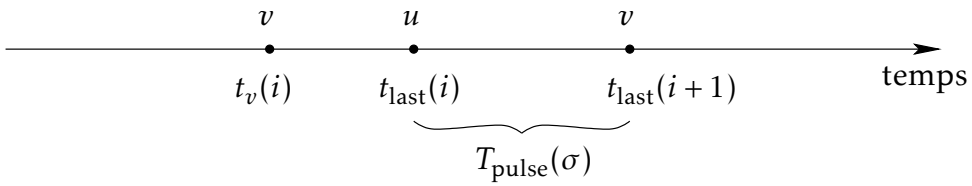


FIGURE 6.3 – Pour un sommet donné (ici v), la durée entre deux pulses peut être plus grande que $T_{\text{pulse}}(\sigma) = t_{\text{last}}(i+1) - t_{\text{last}}(i)$. Ici $t_v(i)$ représente l'instant où la pulse de v passe à i.

Cependant, si l'algorithme S s'exécute en $\tau = \text{TEMPS}(S, G)$ rondes synchrones, alors, après le temps d'initialisation de σ, $\sigma(S)$ s'exécutera en temps $\tau \cdot T_{\text{pulse}}(\sigma)$ puisque la durée maximale de l'algorithme $\sigma(S)$ est

$$\begin{aligned} t_{\text{last}}(\tau) - t_{\text{last}}(0) = \sum_{i=0}^{\tau-1} (t_{\text{last}}(i+1) - t_{\text{last}}(i)) &\leqslant \tau \cdot \max_{0 \leqslant i < \tau} \{t_{\text{last}}(i+1) - t_{\text{last}}(i)\} \\ &= \tau \cdot T_{\text{pulse}}(\sigma) . \end{aligned}$$

Plus précisément,

Proposition 6.3 *Soit σ un synchroniseur implémentant les étapes* SAFE *et* READY*, et S un algorithme synchrone sur un graphe G. Alors,*

- $\text{TEMPS}(\sigma(S), G) \leqslant T_{\text{init}}(\sigma) + \text{TEMPS}(S, G) \cdot T_{\text{pulse}}(\sigma)$.
- $\text{MESSAGE}(\sigma(S), G) \leqslant M_{\text{init}}(\sigma) + 2 \cdot \text{MESSAGE}(S, G) + \text{TEMPS}(S, G) \cdot M_{\text{pulse}}(\sigma)$.

Preuve. Les termes additifs $T_{\text{init}}(\sigma)$ et $M_{\text{init}}(\sigma)$ sont clairs d'après leurs définitions. Chaque ronde coûte un temps $T_{\text{pulse}}(\sigma)$ et $M_{\text{pulse}}(\sigma)$ messages hors les messages originaux de S et de l'étape SAFE qui sont comptabilisés dans le terme $\boxed{2} \cdot \text{MESSAGE}(S, G)$.

1. C'est une façon de dire que la variable PULSE(u) passe à i.

Donc sur l'exécution totale de S, le surcoût est de $\textsc{Temps}(S,G)\cdot T_{\text{pulse}}(\sigma)$ pour le temps et $\textsc{Temps}(S,G)\cdot M_{\text{pulse}}(\sigma)$ pour le nombre de messages. □

La meilleure situation possible est lorsque la simulation d'une ronde prend un temps constant et que le nombre de messages est constant en moyenne pour chaque sommet. On dira qu'un synchroniseur σ est *optimal* dès lors que $T_{\text{pulse}}(\sigma) = O(1)$ et $M_{\text{pulse}}(\sigma) = O(n)$.

6.3 Quelques synchroniseurs

Les deux premiers synchroniseurs, α et β, sont élémentaires et servent de base pour le troisième, le synchroniseur γ. Nous présenterons brièvement un quatrième synchroniseur δ_k, paramétré par un entier $k > 0$ et basé sur la notion de *spanners*, qui propose un compromis entre le surcoût en temps et en nombre de messages. Il permet aussi de retrouver les performances des synchroniseurs α, β, γ.

6.3.1 Synchroniseur α

C'est le plus simple. L'étape Ready est simplement implémentée en envoyant à chacun de ses voisins un message « SAFE » aussitôt que le sommet apprend qu'il est SAFE. Cela représente un coût de $2m$ messages par ronde.

Le surcoût $T_{\text{pulse}}(\alpha)$ en temps est borné 3 en prêtant attention de bien considérer le dernier sommet v à passer à la pulse $i+1$, ce qui se produit à l'instant $t_{\text{last}}(i+1)$, sachant que chaque voisin u de v est déjà passé à la pulse i à un moment antérieur à $t_{\text{last}}(i)$. [*Cyril*. À finir.]

Encore une fois, pour un sommet donné particulier, le passage de la pulse i à $i+1$ peut-être bien plus long ! L'intuition est que certes l'armada navigue à la vitesse du plus lent bateau, mais le plus lent bateau, lui, n'est ralenti par aucun autre.

Pour un exemple concret, considérons une chaîne $v_1 - \cdots - v_t$ de t sommets, et un scenario où $\textsc{Pulse}(v_i) = i$. C'est possible, car deux sommets voisins peuvent avoir des pulses qui diffèrent d'une unité : $\textsc{Pulse}(v_i) = i$, $\textsc{Safe}(v_i, i) = \textsc{Vrai}$ mais $\textsc{Ready}(v_i, i) = \textsc{Faux}$, alors que $\textsc{Ready}(v_{i+1}, i+1) = \textsc{Vrai}$ signifiant que v_{i+1} peut passer à la pulse $i+1$ un peu avant que v_i ne le fasse. Dans ces conditions il est possible que pour avoir $\textsc{Ready}(v_t, t+1) = \textsc{Vrai}$ et que v_t passe sa pulse à $t+1$, le sommet v_t ait à attendre un temps $t-1$. C'est le temps nécessaire à v_1 pour qu'il passe sa pulse de 1 à t.

En fait, on peut factoriser l'étape Safe et Ready en supprimant les messages « ACK » pour chaque message de la composante originale. Le message « SAFE » sert alors d'accusé de reception. Du coup $T_{\text{pulse}}(\alpha) = 2$ au lieu de 3, et dans la proposition 6.3, on peut enlever le facteur 2 devant $\textsc{Message}(S,G)$. Pour l'initialisation du synchroniseur,

consistant à poser $\textsc{Pulse}(u) := 0$ pour tout u, on utilise une simple diffusion grâce à Flood.

Pour résumer :

Proposition 6.4 *Pour tout graphe G,*

- $T_{\text{init}}(\alpha) = O(D)$ *et* $M_{\text{init}}(\alpha) = O(m)$*;*
- $T_{\text{pulse}}(\alpha) = O(1)$ *et* $M_{\text{pulse}}(\alpha) = O(m)$.

Le synchroniseur α, qui optimise le temps, est optimal pour les graphes avec $m = O(n)$ arêtes, comme les graphes de degré bornés, les graphes planaires, etc. Il est par contre très consommateur de messages lorsque le graphe est dense, puisque le nombre de messages augmente alors d'un facteur $m \sim n^2$ par rondes.

6.3.2 Synchroniseur β

On utilise un arbre couvrant T de racine r_0 qui est chargé de collecter tous les messages « SAFE », grâce à un Converge_T. Notez qu'il s'agit du même message pour tous les sommets. Lorsque la racine r_0 les a tous reçus, chaque sommet interne attendant de recevoir tous ceux de leurs fils, elle diffuse dans tout T un message « PULSE » grâce à $\text{Cast}_T(r_0)$. Il est alors clair qu'un sommet u recevant un message « PULSE » est certain que tous les messages « SAFE » de ces voisins on été émis (car tous reçus par r_0) et qui sont donc Safe.

En considérant la construction d'un arbre BFS selon Bellman-Ford_Dist décrit au paragraphe 5.1.2), on en déduit :

Proposition 6.5 *Pour tout graphe G,*

- $T_{\text{init}}(\beta) = O(D)$ *et* $M_{\text{init}}(\beta) = O(nm)$*;*
- $T_{\text{pulse}}(\beta) = O(D)$ *et* $M_{\text{pulse}}(\beta) = O(n)$

Le synchroniseur β, qui optimise le nombre de messages, est optimal pour les graphes de diamètre constant.

6.3.3 Synchroniseur γ

Ce synchroniseur est une sorte de compromis entre les deux synchroniseurs précédents.

Il est basé sur une partition en sous-graphes connexes (ou *clusters*) du graphe. Chaque *cluster* possède un arbre couvrant ainsi qu'une arête « privilégiée » avec chacun de ses *clusters* voisins, deux *clusters* étant voisins s'il existe au moins une arête du graphe les connectant (voir la figure 6.4).

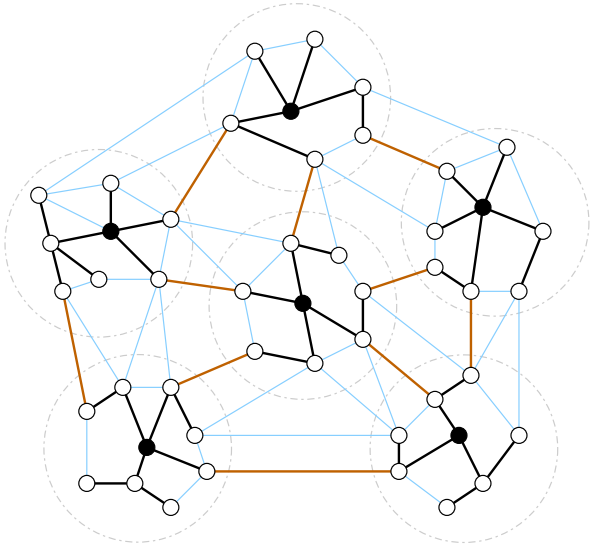


FIGURE 6.4 – Partition en *clusters* pour le synchroniseur γ. La profondeur des arbres est $r_\gamma = 2$ et il y a $m_\gamma = 10$ arêtes privilégiées (en brun). Les arêtes en bleu clair ne sont pas utilisées pour l'étape READY.

L'étape READY, pour la ronde i, est implémentée en trois sous-étapes par un mixte des synchroniseurs α (entre les *clusters* voisins), et β (à l'intérieur de chaque *cluster*). Au départ, chaque *cluster* se synchronise grâce à un synchroniseur β définit sur son arbre couvrant. À l'issue de cette sous-étape, chaque *cluster* devient SAFE, c'est-à-dire chaque sommet du *cluster* est SAFE. Dans la seconde sous-étape, chaque sommet incident à une arête privilégiée envoie un message au *cluster* voisin pour lui signifier que son *cluster* est SAFE. Enfin, dans une troisième sous-étape, chaque *cluster* diffuse dans son arbre l'information que tous leurs *clusters* voisins sont SAFE. Plus précisément, lorsque la racine devient informée, elle diffuse l'information dans son *cluster* pour indiquer à chaque sommet qu'il est READY pour la ronde $i+1$ et qu'il donc peut incrémenter sa variable PULSE.

[*Cyril*. Figure avec deux clusters.]

FIGURE 6.5 – Les trois sous-étapes pour l'étape READY pour le synchroniseur γ.

Les complexités en temps et nombre de messages du synchroniseur γ pour l'étape READY sont contrôlées par la profondeur maximum des arbres couvrant les *clusters* (r_γ), et le nombre d'arêtes privilégiées (m_γ). En effet, on vérifie facilement que [2] :

- $T_{\text{pulse}} \leqslant 4r_\gamma + 1 = O(r_\gamma)$; et
- $M_{\text{pulse}} \leqslant 2m_\gamma + (n - \#\text{clusters}) = O(m_\gamma + n)$.

2. On rappelle que le nombre d'arêtes d'une forêt couvrante composée de k arbres est $n - k$.

On retrouve le synchroniseur α en choisissant n *clusters* réduit à un seul sommet, soit $r_\gamma = 0$ et $m_\gamma = m$. Et on retrouve le synchroniseur β en choisissant un seul *cluster* comportant tout le graphe, soit $r_\gamma = D$ et $m_\gamma = 0$.

Dans ce cours, on ne détaillera pas plus la construction distribuée des clusters et arêtes privilégiées, mais on peut montrer qu'en temps et nombre de messages $O(m + n\log^4 n)$ on peut garantir des valeurs poly-logarithmiques pour r_γ et m_γ/n. Plus précisément, [*Cyril.* [AR93]? et amélioré depuis par [Elk05]?]

Proposition 6.6 *Pour tout graphe G,*

- $T_{\text{init}}(\gamma) = O(m + n\log^4 n)$ *et* $M_{\text{init}}(\gamma) = O(m + n\log^4 n)$*;*
- $T_{\text{pulse}}(\gamma) = O(\log^3 n)$ *et* $M_{\text{pulse}}(\gamma) = O(n\log^3 n)$

Avec ce synchroniseur, une fois construit, on en déduit le calcul d'un BFS en temps $O(D\log^3 n)$ et $O(m\log^3 n)$ messages, car en synchrone avec SpanTree on a vu qu'il était possible de calculer un BFS en temps $O(D)$ et avec $O(m)$ messages.

6.3.4 Synchroniseur δ_k

C'est une approche générale basée sur la notion de « sous-graphe couvrant peu dense », *sparse spanner* en Anglais. C'est une généralisation des synchroniseurs α et β. Ici $k \geqslant 1$ est un paramètre entier, et en gros $\alpha = \delta_1$ et $\beta = \delta_{+\infty}$.

L'idée est la suivante : supposons que l'on ait déjà construit un sous-graphe couvrant H de G. Le synchroniseur, noté σ_H, va implémenter l'étape Ready en utilisant seulement les arêtes de H. Pour informer tous ses voisins qu'il devient Safe, un sommet v va diffuser des messages « SAFE » seulement à ses voisins de H plutôt que ses voisins de G. Bien sûr pour atteindre tous ses voisins de G (notamment ceux qui ne sont pas directement voisins dans H) il va falloir que ces messages soient retransmis suffisamment longtemps afin de progresser suffisamment loin dans H.

Plus précisément, notons $s(H)$ la distance maximum dans H entre deux sommets voisins de G (voir la figure 6.6). Plus formellement,

$$s(H) := \max_{uv \in E(G)} \text{dist}_H(u,v)\,.$$

Chaque sommet v effectue s mini-rondes comme ceci. Dès qu'il devient Safe, v initialise un compteur local $c(v) := 0$ et envoie un message « SAFE » à tous ces voisins de H. Dans le même temps, il attend de recevoir les messages « SAFE » émis par ses voisins de H. Une fois qu'il les a tous reçus, il incrémente $c(v)$ et recommence une nouvelle mini-ronde. Lorsque $c(v) = s$, le sommet v arrête ces mini-rondes, passe dans l'état Ready et peut incrémenter Pulse(v).

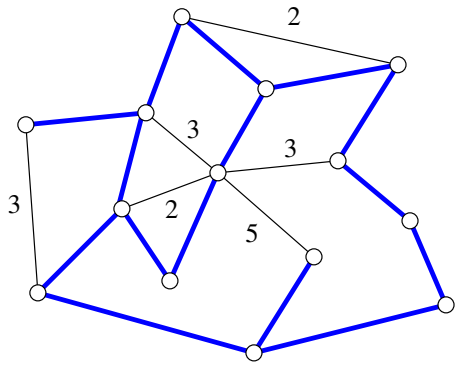


FIGURE 6.6 – Sous-graphe couvrant H (en bleu) pour G. Les étiquettes sur les arêtes représente la distance $\mathrm{dist}_H(u,v)$ pour les arêtes $uv \in E(G)\setminus E(H)$ (pour les arêtes $uv \in E(H)$, $\mathrm{dist}_H(u,v) = 1$ bien évidemment). La plus grande de ces valeurs est $s(H) = 5$.

Proposition 6.7 *Lorsque $c(v) = s$, tout sommet u à distance au plus s de v dans H est* SAFE. *Par conséquent, tout voisin u de v dans G est* SAFE.

Preuve. Il s'agit d'une simple induction sur la distance $d = c(u)$ où tous les sommets sont SAFE. C'est trivialement vrai pour $d = 0$. Ensuite, par définition de s, les sommets voisins de G sont à distance au plus s dans H. [*Cyril*. À revoir.] □

À la lumière de cette implémentation, un paramètre important est donc la valeur $s(H)$ qu'on appelle *facteur d'étirement* de H. Il représente le facteur par lequel les distances de G sont modifiées. Plus $s(H)$ est petit plus H « ressemble » à G. C'est une mesure de distortion des distances. En effet, il est clair que pour toute paire de sommets u, v (et pas seulement les voisins) on a :

$$\mathrm{dist}_G(u,v) \leqslant \mathrm{dist}_H(u,v) \leqslant s(H)\cdot\mathrm{dist}_G(u,v)$$

car on peut construire un chemin de u à v dans H en partant d'un plus court chemin de u à v dans G, donc de longueur $\mathrm{dist}_G(u,v)$, et en « étirant » chacune de ses arêtes d'un facteur $s(H)$. Notons que si G est connexe, alors H l'est aussi et dans ce cas $s(H) < +\infty$ est fini.

Proposition 6.8 *Soit H un sous-graphe couvrant de G. Alors G possède un synchroniseur σ_H tel que :*

- $T_{\mathrm{pulse}}(\sigma_H) \leqslant s(H) + 2$ *; et*
- $M_{\mathrm{pulse}}(\sigma_H) \leqslant 2s(H)\cdot|E(H)|$.

Preuve. Entre deux rondes consécutives, l'étape READY du synchroniseur σ_H prend un temps $\leqslant s(H)$ et génère $\sum_u \deg_H(u) = 2|E(H)|$ messages par mini-rondes, soit au plus un total $s(H)\cdot(2|E(H)|)$ messages entre deux rondes.

L'étape Safe prend un temps au plus unitaire tout comme les messages de la composante originale, ce qui rajoute deux unités au temps. □

On retrouve donc le synchroniseur α en prenant $H = G$. En effet, $s(H) = 1$ dans ce cas. Et donc $T_{\text{pulse}}(\sigma_H) = O(1)$ et $M_{\text{pulse}}(\sigma_H) = O(s(H)\cdot|E(H)|) = O(m)$. Quant au synchroniseur β, on pourrait choisir H comme un arbre BFS. Il vient alors $T_{\text{pulse}}(\sigma_H) = O(D)$ et $M_{\text{pulse}}(\sigma_H) = O(Dn)$. En fait, le synchroniseur β fait un peu mieux car pour l'étape Ready chaque sommet n'envoie qu'un seul message et pas $O(D)$.

Il reste à montrer que pour tout graphe, il existe de « bons » sous-graphes couvrant (*spanners*).

Proposition 6.9 *Soit $k \geqslant 1$ un entier. Alors, tout graphe à n sommets possède un sous-graphe couvrant avec moins de $n^{1+1/k} + n$ arêtes et un facteur d'étirement $\leqslant 2k-1$.*

L'énoncé est évident pour $k = 1$, puisqu'il affirme que tout graphe possède un sous-graphe d'étirement 1 ayant $O(n^2)$ arêtes : c'est le graphe lui-même ! Il l'est déjà beaucoup moins pour $k = 2$, avec l'existence de sous-graphes d'étirement 3 pour seulement $O(n^{1.5})$ arêtes.

Bien qu'*a priori* non trivial, la proposition se démontre par l'analyse d'un simple algorithme glouton, malheureusement séquentiel, donné ci-dessous. (Voir la figure 6.7 pour un exemple d'exécution.)

Algorithme $\mathsf{Spanner}_k(G)$

1. $H := (V(G), \varnothing)$
2. Pour chaque $uv \in E(G)$, si $\text{dist}_H(u,v) > 2k-1$, alors $H := H \cup \{uv\}$

Figure 6.7 – Exemple d'exécution de l'algorithme $\mathsf{Spanner}_k(G)$ avec $k = 2$, en prenant les arêtes dans l'ordre de leur numéro. Chaque arête de G a été remplacé par un chemin de longueur au plus $2k-1$ dans H, et H ne contient plus de cycles de longueur $2k = 4$.

Les détails de la preuve peuvent être trouvés dans [Gav24][Chapitre 1, section 4], mais ils sont basés sur les observations suivantes qui découlent de l'exécution de l'algorithme :

- $s(H) \leqslant 2k-1$; et
- H n'a pas de cycles de longueur $2k$ ou moins.

Or tout graphe sans cycle de longueur $\leqslant 2k$ possède moins de $n^{1+1/k}+n$ arêtes [Gav24], d'ou le résultat.

Le résultat de la proposition 6.9 reste vrai pour les graphes valués, c'est-à-dire lorsque l'étirement est défini par rapport aux coûts des plus courts chemins. Il suffit, dans l'algorithme Spanner_k, de considérer les arêtes de G par ordre croissant de longueur. Remarquons que pour $k = +\infty$ ou disons $k = n$, l'algorithme Spanner_k est le même que celui de Kruskal. En particulier H est un arbre (ou une forêt si G n'est pas connexe).

Cependant, il existe un algorithme distribué qui construit en temps ... et ... messages un sous-graphe H ayant $O(k \cdot n^{1+1/k})$ arêtes et un facteur d'étirement $O(k)$. [*Cyril.* À finir.]

En combinant les propositions 6.8 et 6.9, on en déduit :

Proposition 6.10 *Pour tout graphe à n sommets et entier $k \geqslant 1$, il existe un synchroniseur δ_k vérifiant :*

- $T_{\text{init}}(\delta_k) = \ldots$ *et* $M_{\text{init}}(\delta_k) = \ldots$;
- $T_{\text{pulse}}(\delta_k) = O(k)$ *et* $M_{\text{pulse}}(\delta_k) = O(k^2 \cdot n^{1+1/k})$.

Preuve. [*Cyril.* À FINIR] □

À quelques facteurs logarithmiques près on retrouve aussi le synchroniseur γ en prenant [3] $k = \lceil \log_2 n \rceil$.

6.3.5 Borne inférieure

La proposition 6.8 montre un lien entre synchroniseur efficace et l'existence de sous-graphes couvrant peu denses et de faible facteur d'étirement. La proposition 6.9 montre qu'on peut aussi les construire. Le lien est en fait beaucoup plus fort qu'il n'y paraît et cela va permettre d'établir une limite absolue sur le meilleur compromis possible entre le surcoût en temps et nombre de messages d'un synchroniseur.

La suite va montrer que la contraposée de la proposition 6.8 est essentiellement vraie (à des facteurs constants près), à savoir que si G possède un synchroniseur σ avec $T_{\text{pulse}}(\sigma) \leqslant s$ et $M_{\text{pulse}}(\sigma) \leqslant m$, alors forcément il possède un sous-graphe couvrant à m arêtes et de facteur d'étirement s.

Plus précisément,

3. En effet, on a $n^{1/\log_2 n} = 2$ car $\log_b(n^{1/\log_b n}) = (1/\log_b n) \cdot \log_b n = 1$. Et donc $n^{1/\log_b n} = b$.

Proposition 6.11 *Si G n'a pas de sous-graphe couvrant ayant au plus m arêtes et un facteur d'étirement au plus s, alors tout synchroniseur σ pour G doit vérifier $T_{\text{pulse}}(\sigma) > s$ ou $M_{\text{pulse}}(\sigma) > m$.*

Par exemple, le graphe biparti complet $K_{p,q}$ qui a pq arêtes, ne possède pas de sous-graphe couvrant ayant $< pq$ arêtes et un facteur d'étirement < 3. [*Question*. Pourquoi ?] Par conséquent, ou bien $T_{\text{pulse}}(\sigma) \geqslant 3$ ou bien $M_{\text{pulse}}(\sigma) \geqslant pq$, et ceci quel que soit le synchroniseur σ.

Preuve. Soit G un tel graphe et supposons qu'il possède pourtant un synchroniseur σ avec $M_{\text{pulse}}(\sigma) \leqslant m$. (Bien évidemment si $M_{\text{pulse}}(\sigma) > m$ pour tout σ, la proposition est démontrée.) Le synchroniseur doit garantir qu'un sommet ne va pas passer à une nouvelle ronde avant que tous les messages de la ronde précédente lui étant destinés ne soient arrivés. Donc entre deux rondes consécutives de l'algorithme synchrone S, il doit avoir dans $A = \sigma(S)$ un transfert d'informations (dites de synchronisation) entre chaque paire de sommets voisins du graphe. Ce transfert, qui n'ont aucune raison de se produire le long de chaque arête de G, doit prendre un temps au plus $T_{\text{pulse}}(\sigma)$ et utiliser au plus $M_{\text{pulse}}(\sigma)$ messages.

Soit E l'ensemble des arêtes utilisées par σ entre deux rondes consécutives pour le transfert des informations de synchronisation. Puisque $M_{\text{pulse}}(\sigma) \leqslant m$, on a $|E| \leqslant m$. Par hypothèse sur G, le facteur d'étirement de tout sous-graphe H induit par les arêtes E vérifie $s(H) > s$. Cela implique qu'il existe deux voisins u, v de G qui sont à distance $> s$ dans H. Le transfert des informations de synchronisation peut donc prendre entre u et v, dans un mauvais scénario, au moins $s(H) > s$ unités de temps. Et donc, $T_{\text{pulse}}(\sigma) > s$. □

Ce résultat doit être rapproché du suivant qui implique une borne inférieure sur le meilleur compris possible d'un synchroniseur, à des constantes multiplicatives prêtes.

Proposition 6.12 ([Awe85]) *Pour chaque entier $k \geqslant 1$, il existe un graphe à n sommets où tout sous-graphe couvrant de facteur d'étirement au plus k possède au moins $\frac{1}{4}n^{1+1/k}$ arêtes.*

Donc le synchroniseur optimal (surcoût en temps $O(1)$ et en messages $O(n)$) n'existe pas. [*Exercice*. Pourquoi ?]

En fait, il a été conjecturé (Erdős-Simonovits [Erd64][ES82]) que pour tout entier $k \geqslant 1$, qu'il existe des graphes où tout sous-graphe couvrant de facteur d'étirement $< 2k+1$ possède $\Omega(n^{1+1/k})$ arêtes. La conjecture a été prouvée pour $k = 1, 2, 3$ et 5. Autrement dit, le compromis établi dans la proposition 6.9 est le meilleure possible.

Au chapitre 9 on verra comment construire rapidement ces sous-graphes à la base de la construction de ces synchroniseurs.

6.4 Exercices

Exercice 1

Montrer qu'aucun des synchroniseurs α, β, et γ n'est optimal (en temps et en nombre de messages) pour l'Hypercube H_d de dimension d. On rappelle que H_d est un graphe dont les sommets sont les mots binaires de longueur d et dont les arêtes sont les paires de mots qui diffèrent d'exactement un bit. Il possède $n = 2^d$ sommets et $m = d \cdot 2^{d-1}$ arêtes.

En utilisant le fait que tout H_d possède un sous-graphe couvrant de facteur d'étirement 3 ayant au plus $4n$ arêtes [4], montrer qu'il possède un synchroniseur optimal.

Bibliographie

[AR93] Y. Afek and M. Ricklin, *Sparser : A paradigm for running distributed algorithms*, Journal of Algorithms, 14 (1993), pp. 316–328. doi : 10.1006/jagm.1993.1016.

[Awe85] B. Awerbuch, *Complexity of network synchronization*, Journal of the ACM, 32 (1985), pp. 804–823. doi : 10.1145/4221.4227.

[DZ00] W. Duckworth and M. Zito, *Note : Sparse hypercube 3-spanners*, Discrete Mathematics, 103 (2000), pp. 289–295. doi : 10.1016/S0166-218X(99)00246-2.

[Elk05] M. Elkin, *Computing almost shortest paths*, ACM Transactions on Algorithms, 1 (2005), pp. 283–323. doi : 10.1145/1103963.1103968.

[Erd64] P. Erdős, *Extremal problems in graph theory*, in Public House Cszechoslovak Academy of Sciences, Prague, 1964, pp. 29–36.

[ES82] P. Erdős and M. Simonovits, *Compactness results in extremal graph theory*, Combinatorica, 2 (1982), pp. 275–288. doi : 10.1007/BF02579234.

[Gal82] R. G. Gallager, *Distributed minimum hop algorithms*, Tech. Rep. LIDS-P-1175, M.I.T., Cambridge, MA, January 1982.

[Gav24] C. Gavoille, *Analyse d'algorithme – Cours d'introduction à la complexité paramétrique et aux algorithmes d'approximation*, 2024. http://dept-info.labri.fr/~gavoille/UE-AA/cours.pdf. Notes de cours.

[PU89] D. Peleg and J. D. Ullman, *An optimal synchronizer for the hypercube*, SIAM Journal on Computing, 18 (1989), pp. 740–747. doi : 10.1137/0218050.

4. En utilisant la construction d'origine [PU89] $7n$ arêtes suffisent, mais en utilisant le raffinement de [DZ00] $4n$ arêtes suffisent.

CHAPITRE 7

Coloration

Sommaire

Dans ce chapitre nous allons voir comment partitioner les sommets d'un graphe en k ensembles indépendants, ce qui revient à calculer une k-coloration du graphe.

L'enjeu ici est de réaliser cette tâche le plus rapidement possible, et d'étudier la nature locale de ce problème. Seule la complexité en temps importe. On se place dans les hypothèses du modèle LOCAL (voir la section 2.2).

Mots clés et notions abordées dans ce chapitre :

- coloration des arbres et cycles (1-orientations),
- algorithme en $\log^* n$,
- borne inférieure pour le cycle,
- $(\Delta+1)$-coloration,
- problèmes localement vérifiables (LCL).

7.1 Introduction

Casser la symétrie joue un rôle majeur en calcul distribué. Beaucoup de tâches nécessitent la collaboration d'un grand nombre de processeurs (parallélisme) mais interdissent à des processeurs trop proches d'effectuer la même action et ainsi de prendre la même décision. Dans certains cas, il peut être interdit que des processeurs voisins agissent en concurrence. Lorsqu'on construit un algorithme distribué on souhaite idéalement créer un maximum de parallélisme (pour aller vite) tout en interdisant les processeurs voisins de faire la même chose (brisure de symétrie). Un moyen d'atteindre cet objectif, dans une phase préliminaire de l'algorithme, est de partitioner les processeurs en ensembles indépendants [1], c'est-à-dire sans aucune arête entre processeurs d'un même ensemble. Cela revient à répartir les processeurs voisins dans des classes (de couleurs) différentes. Pour créer un maximum de parallélisme, il faudrait un maximum de processeurs dans chaque classe de couleurs, soit des ensembles indépendants les plus grands possibles, ce qui *in fine* revient à diminuer le nombre d'ensembles ou de couleurs.

Par exemple, si l'on voit les pixels d'un écran de télévision comme une grille de processeurs élémentaires, le mode 1080i de la norme *HD Ready* impose que les lignes paires et les lignes impaires fonctionnement en parallèles mais différemment (1920×1080 entrelacé) afin d'optimiser le flux vidéo et la vitesse d'affichage.

Un autre exemple typique est la construction et l'optimisation des tables de routage. On est souvent amené à partitioner le réseau en régions (ou *cluster*) et à choisir un processeur particulier par région jouant le rôle de centres ou de serveur.

Dans certaines situations on impose même qu'il n'y ait qu'un seul processeur actif en même temps, ce qui revient à résoudre le problème d'élection d'un *leader* évoqué page 7. Dans le cadre de notre cours, cependant, nous supposons que nous avons des identités (voir plus en détail le paragraphe 2.2 du chapitre 2 concernant le modèle LOCAL). Donc, dans le modèle LOCAL on pourrait faire quelque chose comme « Si $\mathrm{ID}(u) = 0$,

1. On parle parfois de « stables ».

alors Leader(u) := Vrai ... » et ainsi distinguer un processeur explicitement de tous les autres. Enfin, il peut aussi s'agir du problème de l'exclusion mutuelle où une ressource ne doit être accédé que par une entité à la fois.

Nous allons donc nous intéresser à la nature local du problème de la coloration qui revient à casser la symétrie entre voisins. Un autre problème très lié à la brisure de symétrie est le calcul d'ensemble indépendant maximaux. Il sera étudié au chapitre 8 suivant.

7.2 Définition du problème

Une *coloration propre* d'un graphe G est une fonction $c\colon V(G) \to \mathbb{N}$ telle que $c(u) \neq c(v)$ pour toute arête $\{u,v\} \in E(G)$. On dit que c est une *k-coloration* pour G si $c(u) \in \{0,\dots,k-1\}$ pour tout sommet u de G. Sans précision particulière, les colorations seront toujours considérées comme propres.

Colorier un graphe revient à partitioner les sommets en ensembles indépendants. Les ensembles indépendants permettent de créer du parallélisme puisque tous les sommets d'un même ensemble peuvent interagir librement avec leurs voisins, qui par définition ne sont pas dans le même ensemble. Moins il y a de couleurs, plus grand est le parallélisme.

Le *nombre chromatique* de G, noté $\chi(G)$, est le plus petit k tel que G possède une k-coloration. Déterminer si $\chi(G) \leqslant 2$, prend un temps linéaire (graphes bipartis), alors que savoir si $\chi(G) = 3$ est un problème notoirement NP-complet même si G est cubic ou si G est un graphe planaire 4-régulier (et pourtant on sait que $\chi(G) \leqslant 4$ dans ces cas là). C'est même NP-difficile d'approximer $\chi(G)$ à un facteur $n^{1-\varepsilon}$ près. Dans la suite on notera par $\Delta = \Delta(G) = \max\{\deg(u) : u \in V(G)\}$ le degré maximum de G.

Dans ce chapitre, on ne va pas s'intéresser au calcul du nombre chromatique d'un graphe, mais plutôt essayer de déterminer le plus « rapidement » possible et de manière distribuée des colorations (propres donc) avec « raisonnablement peu » de couleurs. Par « rapidement » on veut dire en temps significativement plus petit que le diamètre du graphe (on veut un algorithme local !), et par « raisonnablement peu » on veut dire $O(\Delta)$ couleurs, un nombre *a priori* indépendant de n.

En effet, non seulement $\chi(G) \leqslant \Delta(G)+1$ pour tout graphe, ce qu'on peut montrer par une simple induction comme schématisé sur la figure 7.1). Mais en plus cela peut être réalisé très simplement en séquentiel par un algorithme glouton GreedyColoring présenté et analysé ci-après dans la proposition 7.1.

On notera aussi qu'on ne peut pas, en général, remplacer le terme « $\Delta(G)+1$ » par quelque chose de strictement plus petit. Les cliques à n sommets sont de nombre chromatique n et de degré maximum $n-1$. C'est le pire des cas, car le théorème de Brooks

(1941) établit que si G est ni une clique ni un cycle impair [2], alors $\chi(G) \leqslant \Delta(G)$. Mais c'est une autre histoire.

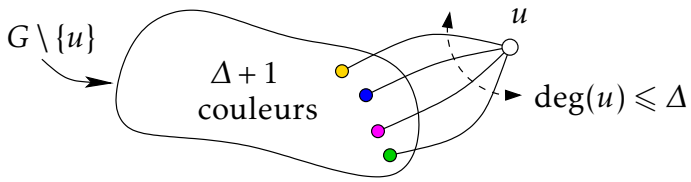


FIGURE 7.1 – Si $\Delta + 1$ couleurs suffisent à colorier $G \setminus \{u\}$, $\Delta + 1$ suffiront pour G. Bien sûr, si G a $\Delta + 1$ sommets ou moins, il est $\Delta + 1$ coloriable.

L'enjeu est donc produire un algorithme distribué capable de « paralléliser » l'algorithme GreedyColoring, un algorithme séquentiel simplissime de complexité linéaire. Comme on va le voir, c'est pas si facile que cela. Donc autant dire qu'on ne cherche pas [3] la parallélisation d'un algorithme séquentiel qui produirait une coloration optimale ou quasi-optimale.

Les meilleurs algorithmes distribués de coloration calculent une $(\Delta+1)$-coloration où Δ est le degré maximum du graphe. Pendant très longtemps (30 ans), le meilleur d'entre eux [PS92] avait une complexité en temps de $2^{O(\sqrt{\log n})}$, ce qui est sous-polynomial [4] et super-polylogarithmique [5]. C'est donc potentiellement bien plus petit que le diamètre du graphe. Si $\Delta < 2^{O(\sqrt{\log n})}$, il est possible de faire mieux que cela. Le dernier en date a une complexité en temps [6] de $O(\sqrt{\Delta}\log^{5/2}\Delta) + \log^* n$. Ce deuxième algorithme nécessite de connaître Δ, en plus de n. Ce n'est que très récemment qu'un algorithme polylogarithmique en n a été trouvé [GGR21][GK22], même si Δ dépend de n et est très grand. Un bref historique des algorithmes distribués de coloration est donné dans le paragraphe 7.8.4.

7.3 Réduction de palette

Parmi les hypothèses du modèle LOCAL, chaque sommet u possède une identité unique, $\mathrm{ID}(u)$. Donc $c(u) := \mathrm{ID}(u)$ est trivialement une coloration propre, certes qui utilise potentiellement beaucoup de couleurs. Le but va être de réduire ce nombre, par des interactions entre voisins, tout en maintenant une coloration propre.

En séquentiel, il y a une technique très simple pour réduire le nombre de couleurs.

2. Un cycle impair est un cycle de longueur impair.
3. Pas encore, mais peut-être qu'un jour ...
4. Plus petit que n^c pour toute constante $c > 0$.
5. Plus grand que $\log^c n$ pour toute constante $c > 0$.
6. On définira plus tard dans la section 7.4.3 la fonction $\log^*$ qui est une fonction très peu croissante, beaucoup moins que $\log\log n$ par exemple.

Elle est basée sur une stratégie gloutonne. Initialement, $c(u) := \mathrm{ID}(u)$. Et pour chaque sommet u, pris dans un ordre quelconque, on modifie la couleur de u par la plus petite couleur non utilisée par un voisin de u. Voir la figure 7.2 pour une illustration.

Plus formellement, on définit pour tout ensemble $X \subseteq \mathbb{N}$,

$$\mathsf{FirstFree}(X) \ := \ \min(\mathbb{N} \setminus X)$$

le plus petit entier naturel qui n'est pas dans X. Par exemple, $\mathsf{FirstFree}(\{0,1,2,4,8\}) = 3$. Il est clair que $\mathsf{FirstFree}(X) \in \{0,\dots,|X|\}$ et donc $\mathsf{FirstFree}(X) \leqslant |X|$ (ça vient du fait qu'on numérote à partir de 0).

Dans la suite on notera $c(N(u))$ l'ensemble des couleurs des voisins de u selon la coloration c. Plus formellement, $c(N(u)) := \{c(v) : v \in N(u)\}$. Bien sûr $\mathsf{FirstFree}(c(N(u))) \leqslant |c(N(u))| \leqslant \deg(u)$. Et, comme $c(u) \notin c(N(u))$, on en déduit que :

$$\mathsf{FirstFree}(c(N(u))) \ \leqslant \ \min\{c(u), \deg(u)\} \ . \tag{7.1}$$

Ainsi, en appliquant FirstFree à un sommet on ne peut que diminuer sa couleur. En séquentiel, l'algorithme glouton est donc (on appelle parfois cet algorithme FirstFit) :

Algorithme GreedyColoring(G) (séquentiel)

1. Pour tout sommet u, poser $c(u) := \mathrm{ID}(u)$.
2. Successivement pour chaque sommet u, poser $c(u) := \mathsf{FirstFree}(c(N(u)))$.

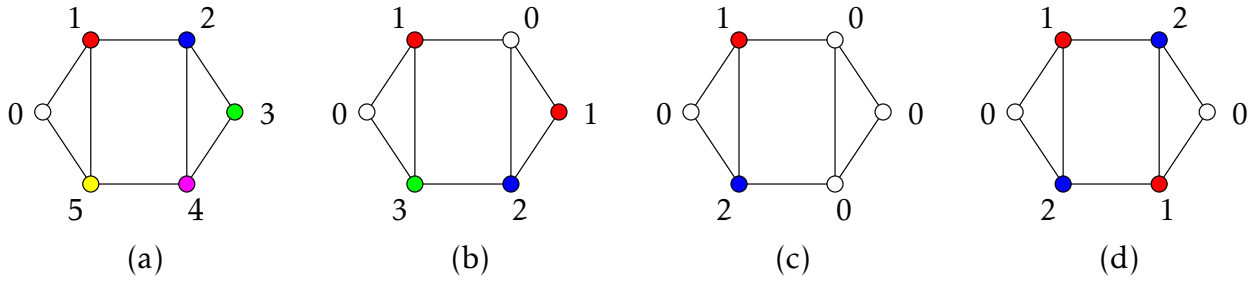


FIGURE 7.2 – Exemple de coloration suivant la règle FirstFree. En (a), une 6-coloration initiale. En (b) résultat en appliquant l'algorithme séquentiel avec l'ordre $0,1,\dots,5$. En (c) résultat en appliquant $\mathsf{FirstFree}(c(N(u)))$ en parallèle. Comme le montre (d) la coloration (b) n'est pas optimale. Cette 3-coloration optimale peut être obtenue avec GreedyColoring(G) et l'ordre des sommets $0,3,1,4,2,5$.

Proposition 7.1 *Pour tout graphe G, $\chi(G) \leqslant \Delta(G)+1$.*

Preuve. Il suffit d'analyser l'algorithme séquentiel GreedyColoring(G). Après l'étape 1, c est une coloration propre. Si, avant l'application de $c(u) :=$ FirstFree($c(N(u))$), c est une coloration propre, alors après c est encore une coloration propre. Donc, à la fin de l'algorithme, c est une coloration propre de G. Et on a vu que FirstFree($c(N(u))) \in [0, \deg(u)]$. Donc $c(u) \leqslant \deg(u) \leqslant \Delta(G)$ ce qui fait $\Delta + 1$ couleurs au plus. □

Remarque. Il existe toujours un ordre des sommets tel que GreedyColoring(G) donne le nombre optimal $\chi(G)$ de couleurs. Pour le voir, il faut : (1) rendre « gloutonne » une coloration optimale, c'est-à-dire s'arranger pour tout u que $c(u)$ soit la plus petite couleur disponible parmi ses voisins – ainsi l'application de FirstFree($c(N(u))$) donnera précisément $c(u)$. Et, (2) considérer les sommets par ordre croissant de leur numéro de couleur. Maintenant, pour rendre gloutonne une coloration il suffit d'appliquer FirstFree sur les sommets par ordre croissant des couleurs ! en observant que l'application de FirstFree ne peut jamais augmenter le numéro de couleur d'un sommet (d'après l'équation 7.1). Dans l'exemple de la figure 7.2, il suffit de considérer la 3-coloration (d) qui est déjà gloutonne, et de prendre par exemple l'ordre $0, 3, 1, 4, 2, 5$.

L'objectif est de trouver un algorithme distribué « rapide » pour calculer une $(\Delta(G)+1)$-coloration de G, ce qui est toujours possible grâce à la proposition 7.1. En terme de degré maximum, c'est la meilleure borne possible puisque pour la clique K_n à n sommets par exemple, on a $\chi(K_n) = n$ alors que $\Delta(K_n) = n - 1$. Autrement dit, $\chi(K_n) = \Delta(K_n) + 1$. En fait, $\chi(G) = \Delta(G) + 1$ si et seulement si G est une clique ou un cycle impair (théorème de Brooks déjà évoqué). Dit autrement, si G est ni une clique ni un cycle impair, alors $\chi(G) \leqslant \Delta(G)$.

L'algorithme GreedyColoring est intrinsèquement séquentiel. En distribué il y a danger à appliquer la règle FirstFree à tous les sommets, tout pouvant se dérouler en parallèle (cf. les sommets 2,3,4 de la figure 7.2(c)). Le problème est qu'en appliquant FirstFree à des sommets voisins il peut se produire que la couleur résultante soit la même. Il faut donc trouver une règle locale pour éviter ceci. On voit ici que l'idéal serait d'avoir justement une coloration et d'appliquer FirstFree aux sommets d'une même couleur pour éviter les collisions ... On tourne visiblement en rond.

En général on procède en essayant de réduire la palette de couleur en appliquant la règle FirstFree à certains sommets seulement. Initialement on utilise comme palette l'intervalle $[0, n[$, qui va se réduire ronde après ronde jusqu'à $[0, \Delta(G)]$. À chaque ronde tous les sommets d'une couleur $k \in]\Delta, n[$ peuvent éliminer cette couleur par l'usage de FirstFree qui garantit que la coloration reste propre.

[*Cyril*. Plus simplement, on pourrait plutôt définir un ReduceColor(c) permettant de tenter de diminuer d'un la palette des couleurs utilisées, ici celle de plus grand numéro. Pour cela, en une ronde, on applique FirstFree si le sommet à la couleur localement maximum. Il n'est pas possible d'avoir deux sommets voisins qui appliquent cette

règles. Et si pour tout sommet u, $|c(N(u))| < k$, alors il est possible d'avoir seulement k couleurs. Disons que s'il existe un sommet u avec $|c(N(u))| \geqslant k$, alors il ne sera pas possible d'avoir moins de k couleurs.]

L'algorithme distribué synchrone suivant permet plus généralement de réduire le nombre de couleurs de k_0 à k_1. On va pouvoir l'appliquer avec $k_0 = n$ et $k_1 = \Delta(G) + 1$. Mais plus tard on verra des exemples où c'est différent. On suppose donc qu'au départ c est une k_0-coloration et que les valeurs k_0, k_1 sont connues de tous les processeurs.

Algorithme ReducePalette(k_0, k_1, c)
(code du sommet u)

Pour chaque entier $k \in [k_1, k_0[$ faire :

1. NewRound
2. Send$(c(u), v)$ pour tout $v \in N(u)$
3. Si $c(u) = k$, alors
   (a) $X := \bigcup_{v \in N(u)} \{\text{Receive}(v)\}$
   (b) $c(u) := \text{FirstFree}(X)$

L'ordre de parcours des entiers de l'intervalle $[k_1, k_0[$ n'a pas d'importance. L'instruction NewRound permet en mode synchrone de démarrer une nouvelle ronde, c'est-à-dire un nouveau cycle Send/Receive/Compute (voir les explications du chapitre 1). Dans un système asynchrone, cette instruction peut être vue comme une synchronisation, un appel à une routine du synchroniseur par exemple (voir le chapitre 6).

On rappelle que dans le mode synchrone, la complexité en temps d'un algorithme A peut se mesurer en nombre de rondes, chaque message se transmettant en temps 1 entre voisins. Autrement dit, si A s'exécute en t rondes sur le graphe G, alors Temps$(A, G) = t$.

Proposition 7.2 *L'algorithme* ReducePalette(k_0, k_1, c) *construit en* $\max\{k_0 - k_1, 0\}$ *rondes une* k_1*-coloration* c *qui était initialement une* k_0*-coloration telle que, pour tout sommet* u, $|c(N(u))| < k_1$.

Dans l'énoncé ci-dessus, $|c(N(u))|$ représente le nombre de couleurs différentes parmi les voisins de u. Évidemment, $|c(N(u))| \leqslant \deg(u) \leqslant \Delta(G)$. Et donc prendre $k_1 = \Delta(G) + 1$ a pour effet de toujours vérifier que $|c(N(u))| < k_1$. [*Cyril*. À revoir/reformuler cette dernière condition qui doit en fait être vraie pendant tout le temps de la réduction. Or, cela peut être vrai au départ, puis devenir faux ensuite. C'est le cas de ShiftDown dans l'arbre. Mais cela marche, car lorsqu'on appliquera plus tard la fonction, c'est pour une seule ronde. Sinon, la condition doit être $|N(u)| < k_1$.]

Preuve. La proposition est trivialement vraie si $k_1 > k_0$, puisqu'aucune boucle est exécutée ($[k_1, k_0[= \varnothing$), et k_0-coloration est alors également une k_1-coloration. On suppose donc dans la suite que $k_1 \leqslant k_0$.

Il faut exactement $k_0 - k_1$ rondes pour exécuter ReducePalette(k_0, k_1, c) car k prend les $k_0 - k_1$ valeurs de l'intervalle $[k_1, k_0[$.

On remarque que la règle FirstFree (instruction 3) est appliquée en parallèle que sur des sommets non adjacents, ceux de couleurs k. Donc après chacune de ces étapes parallèles, la coloration de G reste propre.

Soit $m = \max_u |c(N(u))|$ est le nombre maximum de couleurs dans le voisinage d'un sommet selon la k_0-coloration c initiale de G. Par hypothèse, $m \in [0, k_1[$.

Après l'application de FirstFree$(c(N(u)))$, la couleur de u est dans l'intervalle $[0, m]$ contenant $m+1$ valeurs différents. Comme tous les sommets de G sont examinés par l'algorithme [*Cyril*. Oui mais lorsqu'ils sont examinés $|c(N(u))| < k_1$ peut ne plus être vrai car c a changé entre temps.], il suit que ReducePalette(k_0, k_1, c) construit en temps $k_0 - k_1$ une $(m+1)$-coloration pour G, ce qui est aussi une k_1-coloration puisque $m < k_1$. □

En utilisant le fait qu'au départ les identifiants constituent une n-coloration, on peut donc construire une $(\Delta(G)+1)$-coloration en $n-(\Delta(G)+1)$ rondes, en réduisant la palette de $k_0 = n$ à $k_1 = \Delta(G) + 1$ couleurs. Cela revient à traiter les sommets un par un en sautant les sommets de couleurs $\leqslant \Delta(G)$. On va voir comment faire beaucoup mieux, dans certains cas tout au moins.

Remarquons que ReducePalette$(k+1, k, c)$ a pour effet de supprimer en une seule ronde la couleur k de la coloration c, à condition bien sûr que chaque sommet de couleur k possède strictement moins de k couleurs différentes dans son voisinage.

7.4 Coloration des arbres

Nous allons présenter dans cette partie un algorithme très rapide pour calculer une 3-coloration pour les arbres enracinés. Une 2-coloration serait envisageable pour les arbres : les sommets de niveau pair reçoivent la couleur 0 les autres la couleur 1. Mais en distribué on peut montrer (cf. le théorème 7.6) que cela nécessite dans le pire des cas un temps proportionnel à la profondeur [7] de l'arbre pour la calculer. On va voir qu'on peut aller beaucoup plus vite si l'on admet de perdre une couleur. Cet algorithme sera aussi la base d'un algorithme applicable à tous les graphes.

En fait, le même algorithme s'applique aussi aux cycles, et plus généralement aux graphes 1-orientés, qu'on appelle aussi « pseudo-arbres ». On verra même qu'on peut facilement le généraliser aux graphes k-orientés dont voici la définition.

7. L'intuition est la suivante. Les 2-colorations possibles pour un arbre sont limitées. (En fait, à une permutation des couleurs près, il n'y en a qu'une.) Produire une 2-coloration est équivalent à connaître la parité de sa distance à la racine. Or cette information ne peut être déduite sans communication à distance la profondeur justement.

Définition 7.1 (*k*-orientation) *Une* k-orientation *d'un graphe est une orientation de ses arêtes telle que tout sommet possède au plus* k *arcs sortants.*

Un graphe *k-orientable* est un graphe qui possède au moins une k-orientation. Un graphe *k-orienté* est un graphe muni d'une k-orientation.

Les arbres (et les forêts) sont 1-orientables : il suffit de fixer un sommet comme racine et la relation $\textsc{Parent}(u)$ définit alors cette 1-orientation, chaque sommet possédant au plus un parent. Les cycles ont aussi 1-orientables (définie par la relation successeur). Les graphes 1-orientables capturent à la fois les cycles et les arbres ou chemins.

Les graphes 1-orientables sont les graphes dont chaque composante connexe possède au plus un cycle. Ils ressemblent donc à un cycle (éventuellement de longueur nulle) où à ses sommets pendent des arbres (cf. figure 7.3).

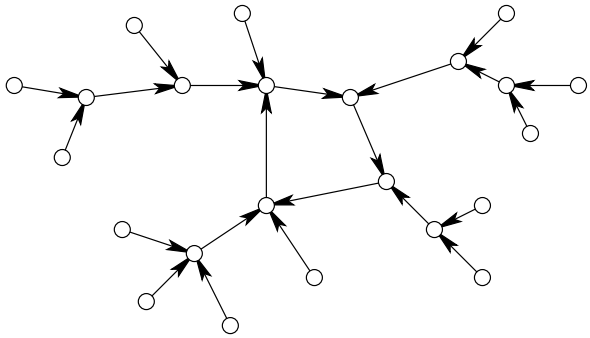


FIGURE 7.3 – Une 1-orientation d'un graphe 1-orientable.

Les graphes k-orientables peuvent être vu comme l'union de graphes 1-orientables. Les graphes planaires sont 3-orientables, car leurs arêtes peuvent être partitionées en trois forêts. Plus généralement, les graphes dont les arêtes peuvent être partitionées en k forêts sont dits d'*arboricité* bornée par k. Les graphes d'arboricité k sont bien évidemment k-orientables, mais le contraire est faux en général. Les cycles, par exemple, sont 1-orientables mais d'arboricité 2. Il existe des algorithmes distribués efficaces pour calculer des $O(k)$-orientations pour un graphe d'arboricité k (voir la section 7.6).

7.4.1 Algorithme pour les 1-orientations

On va supposer que le graphe G est 1-orienté, et que chaque sommet u connaît son orientation via la variable $\textsc{Parent}(u)$. Si u n'a pas d'arc sortant (pas de parent, ce qui arrive inévitablement si G est un arbre), alors $\textsc{Parent}(u) = \perp$.

L'idée de l'algorithme est de répéter un certain nombre de fois un calcul (via la fonction PosDiff décrite ci-après) entre la couleur de u (variable x) et celle de son parent (variable y). À chaque étape le résultat de ce calcul devient la nouvelle couleur de u. Cette couleur définit à chaque étape une coloration propre de G. De plus, la plus grande

couleur diminue fortement à chaque étape. Pour un sommet u sans parent, on lui associe un parent virtuel de couleur 1 ou 0 suivant que la couleur initiale de u (son $\mathrm{ID}(u)$) vaut 0 ou pas.

On suppose que les identités des sommets sont des entiers de $[0, n[$. La couleur finale du sommet u est stockée dans la variable $\textsc{Color}(u)$. Comme on va le voir, la variable ℓ représente le nombre de bits dans l'écriture binaire de la plus grande couleur d'un sommet de G. On rappelle que pour écrire en binaire un entier quelconque de $[0, n[$ il faut au plus $\lceil \log n \rceil$ bits. En effet, il n'y a que 2^k mots binaires de taille k, et $k = \lceil \log n \rceil$ est le plus petit entier tel que $2^k \geqslant n$.

Cet algorithme est la version distribuée de l'algorithme de Cole et Vishkin [CV86] présenté originalement dans le modèle PRAM, un modèle du calcul parallèle. On remarque que dans l'algorithme suivant, la valeur de $\lceil \log n \rceil$ doit être connue [8] de tous les sommets, ainsi que la valeur $\textsc{Parent}(u)$ donc.

Algorithme Color6
(code du sommet u)

1. Poser $x := \mathrm{ID}(u)$ et $\ell := \lceil \log n \rceil$
2. Si $\textsc{Parent}(u) = \perp$, alors $y := \mathsf{FirstFree}(\{x\})$
3. Répéter :
   (a) NewRound
   (b) $\textsc{Send}(x, v)$ pour tout $v \in N(u) \setminus \textsc{Parent}(u)$
   (c) Si $\textsc{Parent}(u) \neq \perp$, alors $y := \textsc{Receive}(\textsc{Parent}(u))$
   (d) $x := \textsc{PosDiff}(x, y)$
   (e) $\ell' := \ell$ et $\ell := 1 + \lceil \log \ell \rceil$

   Jusqu'à ce que $\ell = \ell'$
4. $\textsc{Color}(u) := x$

On note $\log x$ le logarithme en base deux de x, et $\lceil x \rceil$ la partie entière supérieure de x.

7.4.2 La fonction PosDiff

Dans la suite on notera par $\mathrm{bin}(x)$ l'écriture binaire $x \in \mathbb{N}$, et $|\mathrm{bin}(x)|$ sa longueur. Le bit de position p de $\mathrm{bin}(x)$, noté $\mathrm{bin}(x)[p]$ pour un entier $p \in [0, |\mathrm{bin}(x)|[$, est le $(p+1)$-ième bit à partir de la droite dans l'écriture binaire standard de x. Ainsi, $\mathrm{bin}(x)[0]$ est le bit de poids faible, le bit le plus à droite. Par convention on note $\mathrm{bin}(x)[p] = 0$ pour $p \geqslant |\mathrm{bin}(x)|$ comme si $\mathrm{bin}(x)$ était complétée à gauche par des zéros, si bien que $\mathrm{bin}(x)[p] = \lfloor x/2^p \rfloor \bmod 2$ pour tout $p \in \mathbb{N}$.

8. En fait un majorant suffit.

Par exemple, $\text{bin}(6) = 110$, $|\text{bin}(6)| = 3$ et $\text{bin}(6)[0] = 0$, $\text{bin}(6)[1] = \text{bin}(6)[2] = 1$.

Pour tout $x \neq y \in \mathbb{N}$ on pose :

$$\text{PosDiff}(x,y) \;:=\; 2p + \text{bin}(x)[p]$$

où p est *une* position telle que $\text{bin}(x)[p] \neq \text{bin}(y)[p]$. Dit autrement, l'écriture binaire de $\text{PosDiff}(x,y)$ vaut $\text{bin}(p) \circ \text{bin}(x)[p]$, soit p écrit en binaire suivit d'un bit 0 ou 1 issu de $\text{bin}(x)$. Ce qui en particulier implique

$$|\text{bin}(\text{PosDiff}(x,y))| \;=\; |\text{bin}(p)| + 1\ . \tag{7.2}$$

On a également la propriété de « contraction » suivante, qui montre que PosDiff est beaucoup plus petit que ses arguments :

Proposition 7.3 *Si $x,y \in [0,m[$, alors $\text{PosDiff}(x,y) \in [0, 2\lceil \log m \rceil[$.*

Preuve. Soit (p,b) tel que $\text{PosDiff}(x,y) = 2p + b$. On a $p \in [0, \max\{|\text{bin}(x)|, |\text{bin}(y)|\}[$ et $b \in \{0,1\}$. D'après la remarque sur l'écriture binaire des entiers page 138, $x,y \in [0,m[$ implique $|\text{bin}(x)|, |\text{bin}(y)| \leqslant \lceil \log m \rceil$, et donc $p \in [0, \lceil \log m \rceil[$. Donc $\text{PosDiff}(x,y) < 2(p+1) \leqslant 2\lceil \log m \rceil$. □

Notons qu'il n'y a aucune raison pour que deux sommets (voisins ou pas) appliquent la même convention dans le choix de la position p du bit où ils diffèrent avec leurs parents. Cependant, dans toute la suite on choisira la plus petite position p possible. On peut facilement calculer cette valeur en `C`, par exemple en faisant [9] `p = ffs(x^y)-1` qui extrait la position du bit le moins significatif (*Find the First bit Set*) du ou-exclusif bit à bit entre x et y. Le résultat escompté est alors donné par l'expression `(p<<1)|((x>>p)&1)` ou `2*p+(x>>p)%2` si on préfère. Évidemment, prendre le plus petit p possible minimise le résultat de $\text{PosDiff}(x,y)$ (et donc le nombre de couleurs), mais on verra aussi un autre intérêt à ce choix. [*Cyril.* À détailler.]

Exemples :

- $\text{PosDiff}(10,4) = \text{PosDiff}(1010_2, 100_2) = 11_2 = 3$ car $p = 1$ et $b = 1$.
- $\text{PosDiff}(6,22) = \text{PosDiff}(110_2, 10110_2) = 1000_2 = 8$ car $p = 4 = 100_2$ et $b = 0$.
- $\text{PosDiff}(4,0) = \text{PosDiff}(100_2, 0_2) = 101_2 = 5$ car $p = 2 = 10_2$ et $b = 1$.

Notons qu'il est possible que $\text{PosDiff}(x,y) = y$ ou que $\text{PosDiff}(x,y) = x$. Lorsque cela se produit pour une paire de couleurs voisines il est donc impératif que l'autre sommet change de couleur. Il ne peut pas être inactif ! Par exemple, pour tout entier $i \geqslant 0$:

Exemples où $(x,y) \mapsto y$:

9. Disponible dans `#include <strings.h>` de la bibliothèque standard. Généralement cette fonction est implémentée pour s'exécuter en temps constant.

- $\text{PosDiff}(4i, 2) = \text{PosDiff}(\text{bin}(i) \circ 00_2, 10_2) = 10_2 = 2$.
- $\text{PosDiff}(8i, 4) = \text{PosDiff}(\text{bin}(i) \circ 000_2, 100_2) = 100_2 = 4$.
- $\text{PosDiff}(2^i + 2i + 1, 2i + 1) = \text{PosDiff}(1 \circ 0^{i-|\text{bin}(i)|-1} \circ \text{bin}(i) \circ 1, \text{bin}(i) \circ 1) = \text{bin}(i) \circ 1 = 2i + 1$ si $i > |\text{bin}(i)|$, c'est-à-dire si $i \geqslant 3$.

Exemples où $(x, y) \mapsto x$:

- $\text{PosDiff}(0, 2i + 1) = \text{PosDiff}(0_2, \text{bin}(i) \circ 1_2) = 0_2 = 0$.
- $\text{PosDiff}(1, 2i) = \text{PosDiff}(1_2, \text{bin}(i) \circ 0_2) = 1_2 = 1$.
- $\text{PosDiff}(3, 4i + 1) = \text{PosDiff}(11_2, \text{bin}(i) \circ 01_2) = 11_2 = 3$.
- $\text{PosDiff}(5, 8i + 1) = \text{PosDiff}(101_2, \text{bin}(i) \circ 001_2) = 101_2 = 5$.

La propriété essentielle de la fonction PosDiff est la suivante (elle ne dépend pas de la politique du choix de p) :

Propriété 7.1 *Soient $x, y, z \in \mathbb{N}$. Si $x \neq y$ et $y \neq z$, alors $\text{PosDiff}(x, y) \neq \text{PosDiff}(y, z)$.*

Preuve. Supposons que $x \neq y$ et $y \neq z$. Soit (p, b) la position et le bit obtenus dans le calcul de $\text{PosDiff}(x, y)$ et (p', b') dans celui de $\text{PosDiff}(y, z)$. Supposons $\text{PosDiff}(x, y) = \text{PosDiff}(y, z) = v$. Alors $p = p'$ et $b = b'$ car la décomposition de $\text{PosDiff}(x, y)$ en (p, b) est unique, en remarquant que $b = (v \bmod 2)$ et $p = \lfloor v/2 \rfloor$.

Comme $p = p'$, alors $b = \text{bin}(x)[p]$ et $b' = \text{bin}(y)[p'] = \text{bin}(y)[p]$. Mais x et y diffèrent à la position p, donc $\text{bin}(x)[p] \neq \text{bin}(y)[p]$, et donc $b \neq b'$: contradiction.

On a donc bien $\text{PosDiff}(x, y) \neq \text{PosDiff}(y, z)$. □

Cette propriété garantit que la coloration reste propre tant que chaque sommet u de couleur x utilise $\text{PosDiff}(x, y)$ avec un voisin de couleur $y \neq x$. En particulier si y est la couleur du parent de u. C'est un point crucial car si u ne change pas sa couleur x en faisant $\text{PosDiff}(x, y)$, alors ses fils calculant disons leur couleur avec $\text{PosDiff}(w, x)$ pourraient très bien choisir la couleur x. Encore une fois, on a la garantie sur la coloration que si *tous les sommets* calculent PosDiff.

Si u n'a pas de parent, on peut fixer une couleur y comme celle d'un parent virtuel pour u, lui permettant de calculer comme les autres sommets $\text{PosDiff}(x, y)$. Cela produira une coloration propre en choisissant n'importe quelle couleur [10] $y \neq x$. Malheureusement, ce n'est pas exactement ce qui est fait dans l'algorithme Color6. L'instruction 2 qui fixe la couleur y est en dehors de la boucle « Répéter » qui fait varier x. Elle est différente, *a priori*, seulement pour la valeur initiale de x.

On aurait pu supprimer l'instruction 2 et écrire :

3(c) Si $\text{Parent}(u) \neq \perp$, alors $y := \text{Receive}(\text{Parent}(u))$, sinon $y := \text{FirstFree}(\{x\})$

---

10. Il est également important que PosDiff s'applique sur des arguments distincts, la fonction n'étant pas définie sinon.

Mais il se trouve que le « sinon » est en fait inutile [11]. Pour montrer que l'algorithme est bien correct comme il est écrit, on utilise la propriété suivante sur PosDiff qui ne dépend pas non plus de la politique de choix de p.

Propriété 7.2 *Soient $x \neq y \in \mathbb{N}$. Alors, pour tout $i \in \mathbb{N}$, $\textsc{PosDiff}(x,y) \neq 2i + y_i$ où $y_i = \left\lfloor y/2^i \right\rfloor \bmod 2$.*

Preuve. Soit $v = \textsc{PosDiff}(x,y) = 2p + b$ avec p la position où les écritures binaires de x et de y diffèrent et $b = \mathrm{bin}(x)[p]$. On a $v \in \{2p, 2p+1\}$. Si $i \neq p$, alors $v \notin \{2i, 2i+1\}$ et donc $v \neq 2i + y_i$. Si $i = p$, alors $y_i = y_p = \lfloor y/2^p \rfloor \bmod 2 = \mathrm{bin}(y)[p]$, et donc $y_p \neq b = \mathrm{bin}(x)[p]$ par définition de p. Donc $v = 2i + b \neq 2i + y_i$. □

La propriété 7.2 implique que si $x \neq 0$, alors $\textsc{PosDiff}(x,0) \neq 0$. De même si $x \neq 1$, $\textsc{PosDiff}(x,1) \neq 1$. Donc si l'on initialise $y := 0$ si $x \neq 0$ et $y := 1$ si $x = 0$, ou de manière plus concise si l'on pose $y := \mathsf{FirstFree}(\{x\})$, alors $x \neq y$ et $\textsc{PosDiff}(x,y) \neq y$. [*Question.* Montrez que $y := 1 - (x \bmod 2)$ marche aussi.] Cela justifie l'initialisation de y dans l'algorithme Color6 si u n'a pas de parent, car l'invariant $x \leftarrow \textsc{PosDiff}(x,y) \neq y$ restera vrai tout au long du calcul.

7.4.3 Analyse de l'algorithme

Dans la suite on supposera que $n \geqslant 2$, c'est-à-dire que le graphe possède au moins une arête. Pour l'analyse du temps, on définit $x_i(u)$ et ℓ_i comme les valeurs de x du sommet u, et de ℓ *à la fin* de la i-ème boucle « répéter ». (La valeur de ℓ ne dépend pas du sommet u, elle est commune à tous les sommets.) On a :

- $x_0(u) = \mathrm{ID}(u)$.
- $\ell_0 = \lceil \log n \rceil$.
- $\ell_1 = 1 + \lceil \log \lceil \log n \rceil \rceil = O(\log \log n)$.
- $\ell_2 = 1 + \lceil \log (1 + \lceil \log \lceil \log n \rceil \rceil) \rceil = O(\log \log \log n)$.
- $\ell_3 = 1 + \lceil \log (1 + \lceil \log (1 + \lceil \log \lceil \log n \rceil \rceil) \rceil) \rceil = O(\log \log \log \log n)$.
  etc.

Notons que ℓ_i décroît extrêmement rapidement avec i. Pour que $\ell_0 > 3$, il faut avoir $n > 8$, pour que $\ell_1 > 3$, il faut $n > 16$, pour que $\ell_2 > 3$, il faut $n > 256$, et pour que $\ell_3 > 3$, il faut $n > 2^{128}$.

La proposition suivante va nous montrer que l'algorithme termine toujours.

Proposition 7.4 *Pour tout $i > 0$, si $\ell_{i-1} > 3$, alors $\ell_i < \ell_{i-1}$, et $\ell_i = \ell_{i-1}$ sinon.*

11. Et donc c'est plus efficace de faire 3(c) comme la version d'origine, l'efficacité n'ayant que faire de la longueur de sa preuve de validité.

Preuve. On a $\ell_i = 1 + \lceil \log \ell_{i-1} \rceil < 2 + \log \ell_{i-1}$ (car $\lceil x \rceil < 1 + x$). Donc, pour avoir $\ell_i < \ell_{i-1}$, il suffit d'avoir $2 + \log \ell_{i-1} \leqslant \ell_{i-1}$, soit $4\ell_{i-1} \leqslant 2^{\ell_{i-1}}$. C'est vrai dès que $\ell_{i-1} \geqslant 4$. Donc si $\ell_{i-1} > 3$, alors $\ell_i < \ell_{i-1}$.

Si $\ell_{i-1} = 3$, alors $\ell_i = 1 + \lceil \log 3 \rceil = 3$. Si $\ell_{i-1} = 2$, alors $\ell_i = 1 + \lceil \log 2 \rceil = 2$. Et si $\ell_{i-1} = 1$, alors $\ell_i = 1 + \lceil \log 1 \rceil = 1$. Il n'est pas possible d'avoir $\ell_{i-1} = 0$, car $\ell_0 = \lceil \log n \rceil \geqslant 1$ puisque $n \geqslant 2$. □

On déduit de la proposition 7.4 que l'algorithme Color6 termine toujours puisque soit $\ell_0 > 3$ et alors la valeur de ℓ diminue strictement. Soit $\ell_{i-1} \leqslant 3$, et alors la valeur suivante ℓ_i reste aussi à ℓ_{i-1}. Donc on sort toujours de la boucle « Répéter », le test « $\ell = \ell'$ » étant vrai au bout d'un certain nombre d'itérations.

Montrons maintenant qu'il calcule une 6-coloration. Pour ceci, montrons d'abord que ℓ_i borne supérieurement la longueur de l'écriture binaire de toutes les couleurs du graphes. Dit autrement :

Proposition 7.5 *Pour tout $i \geqslant 0$ et tout sommet u, $|\text{bin}(x_i(u))| \leqslant \ell_i$.*

Preuve. Par induction sur i, pour un sommet arbitraire u. On pose $v = \textsc{Parent}(u)$. Si u n'a pas de parent, on pose v comme un sommet virtuel dont la couleur, immuable, est initialisée comme dans l'algorithme grâce à $x_i(v) = \text{FirstFree}(\{x\}) \in \{0,1\}$. Ainsi la couleur $x_i(v)$ est définie pour tout u et tout i. Notons que si u n'a pas de parent (réel), alors $x_i(v) \in \{0,1\}$, et donc $|\text{bin}(x_i(v))| = 1 \leqslant \ell_i$ pour tout i.

Si $i = 0$, on a $x_0(u) \in [0, n[$, et donc $|\text{bin}(x_0(u))| \leqslant \lceil \log n \rceil$, en se rappelant de la remarque sur l'écriture binaire des entiers page 138. Comme $\ell_0 = \lceil \log n \rceil$, on a bien $|\text{bin}(x_0(u))| \leqslant \ell_0$.

Supposons $i > 0$. On a $x_i(u) = \textsc{PosDiff}(x_{i-1}(u), x_{i-1}(v)) = 2p + b$ avec $p \in [0, \max\{|\text{bin}(x_i(u))|, |\text{bin}(x_i(v))|\}[$ et $b \in \{0,1\}$. Par induction sur u et v, que v soit réel ou pas, $|\text{bin}(x_{i-1}(u))| \leqslant \ell_{i-1}$ et $|\text{bin}(x_{i-1}(v))| \leqslant \ell_{i-1}$. Il suit que $p \in [0, \ell_{i-1}[$, ce qui implique que $|\text{bin}(p)| \leqslant \lceil \log \ell_{i-1} \rceil$, en se rappelant de la remarque sur l'écriture binaire des entiers page 138. Dans l'équation 7.2, on a vu que $|\text{bin}(\textsc{PosDiff}(x_{i-1}(u), x_{i-1}(v)))| = |\text{bin}(p)| + 1$, ce qui implique que $|\text{bin}(x_i(u))| \leqslant \lceil \log \ell_{i-1} \rceil + 1 = \ell_i$ (par définition de ℓ_i). □

Proposition 7.6 *À la fin de la dernière étape i de l'algorithme, $x_i(u) \in \{0, \ldots, 5\}$ pour tout sommet u.*

Preuve. D'après la proposition 7.4, l'algorithme se termine avec $\ell_i = \ell_{i-1} \leqslant 3$. Or $x_i(u) = \text{PosDiff}(x_{i-1}(u), x_{i-1}(v)) = 2p + b$ avec $p \in [0, \max\{|\text{bin}(x_{i-1}(u))|, |\text{bin}(x_{i-1}(v))|\}[$ et $b \in \{0,1\}$. D'après la proposition 7.5, $|\text{bin}(x_{i-1}(w))| \leqslant \ell_{i-1}$ pour tout sommet [12] w, et

12. Et de même pour le parent virtuel v si u n'a pas de parent comme noté dans la preuve de la proposition 7.5.

donc $p \in [0, \ell_{i-1}[\subseteq \{0,1,2\}$. Il suit que $x_i(u) \leqslant 2 \times 2 + 1 = 5$. □

Faire l'Exercice 1.

On a vu, grâce aux propriétés 7.1 et 7.2, que l'algorithme maintient avec $x_i(u)$ une coloration propre. D'après la proposition 7.6 il s'agit d'une 6-coloration. On va maintenant borner le nombre de rondes de Color6.

On note

$$\log^{(i)} n = \overbrace{\log\log\log\cdots\log}^{i \text{ fois}} n$$

la fonction « log » itérée i fois. On a $\log^{(0)} n = n$, $\log^{(1)} n = \log n$, $\log^{(2)} n = \log\log n$, etc.

Si la mise à jour de ℓ dans l'instruction 3(e) avait été simplement « $\ell := \log \ell$ », le calcul de ℓ_i en fonction de i aurait beaucoup plus simple. On aurait eut $\ell_i = \log^{(i)} \ell_0 = \log^{(i)} \lceil \log n \rceil = \log^{(i)} \log n = \log^{(i+1)} n$, en supposant n une puissance de deux. À la sortie de l'algorithme après i rondes, on aurait eut $\ell_i = \ell_{i-1} \leqslant 3$ (proposition 7.4), permettant d'extraire facilement i en fonction de n : $\log^{(i)} n = \ell_{i-1} \leqslant 3$ ce qui implique $\log^{(i+2)} n < \log\log 3 = 0.66444...$. Et de l'inégalité $\log^{(i+2)} n \leqslant 1$, on aurait pu conclure un nombre de rondes $i = (\log^* n) - 2$. Malheureusement, les parties entières et le « +1 » dans l'instruction 3(e), nécessaire pour avoir les propositions 7.5 (et donc la proposition 7.6), complexifient l'analyse.

Proposition 7.7 *Pour tout $i > 0$, si $\ell_{i-1} > 3$, alors $\ell_i \leqslant 2 + \lceil \log^{(i+1)} n \rceil$.*

Preuve. On va utiliser le fait suivant (voir aussi [GKP94, p. 71]) :

Fait 7.1 *Soit $f : \mathbb{R} \to \mathbb{R}$ une fonction continue croissante telle que $f^{-1}(n) \in \mathbb{Z}$ pour tout $n \in \mathbb{Z}$. Alors, pour tout $x \in \mathbb{R}$, $\lceil f(\lceil x \rceil) \rceil = \lceil f(x) \rceil$ et $\lfloor f(\lfloor x \rfloor) \rfloor = \lfloor f(x) \rfloor$.*

Une application souvent utile du fait 7.1 est la simplification des récurrences avec des parties entières, afin de montrer, par exemple, que $\lfloor \lfloor n/2 \rfloor / 2 \rfloor = \lfloor n/4 \rfloor$ en prenant $f : n \mapsto n/2$ et $x = f(n)$. Dit autrement on voudrait vérifier que $\lfloor f(\lfloor f(n) \rfloor) \rfloor = \lfloor f(f(n)) \rfloor$. Ce dernier résultat est en fait évident si l'on interprète l'opération $n \mapsto \lfloor f(n) \rfloor = \lfloor n/2 \rfloor$ comme la suppression du bit de poids faible dans l'écriture binaire de n. Il est clair que la suppression du dernier bit, puis de nouveau la suppression du dernier bit, soit l'opération $\lfloor f(\lfloor f(n) \rfloor) \rfloor = \lfloor \lfloor n/2 \rfloor / 2 \rfloor$, a le même effet que la suppression simultanée des deux derniers bits, soit l'opération $\lfloor f(f(n)) \rfloor = \lfloor n/4 \rfloor$. On peut également s'en convaincre en utilisant la base dix et le décalage à gauche du point décimal, au lieu de la base deux.

Démontrons le fait 7.1 dans sa généralité. Soit $x \in \mathbb{R}$ et $n = \lceil f(x) \rceil \in \mathbb{Z}$. Par la croissance de f et de $\lceil \cdot \rceil$, $f(x) \leqslant n \leqslant \lceil f(\lceil x \rceil) \rceil$. Montrons que $\lceil f(\lceil x \rceil) \rceil \leqslant n$. La fonction

f^{-1} est croissante [13] donc $x \leqslant f^{-1}(n)$ puisque $f(x) \leqslant n$. D'où $\lceil x \rceil \leqslant \left\lceil f^{-1}(n) \right\rceil = f^{-1}(n)$ puisque que ce dernier est un entier. En ré-appliquant f, il suit que $f(\lceil x \rceil) \leqslant n$. D'où, $\lceil f(\lceil x \rceil) \rceil \leqslant \lceil n \rceil = n$. Par conséquent $n = \lceil f(\lceil x \rceil) \rceil$.

Pour la partie entière inférieure, c'est-à-dire $\lfloor f(\lfloor x \rfloor) \rfloor = \lfloor f(x) \rfloor$, le raisonnement est similaire en posant $n = \lfloor f(x) \rfloor$ et en inversant toutes les inégalités.

Le fait 7.1 s'applique par exemple aux fonctions $x \mapsto x/i$, $x \mapsto x^{1/i}$, $x \mapsto \log x$ pour chaque $i \in \mathbb{N}^*$. En particulier, $\lceil \log \lceil \log n \rceil \rceil = \lceil \log \log n \rceil$.

On va montrer la proposition 7.7 par induction, et donc soit

$$P(i) := \text{« si } \ell_{i-1} > 3, \text{ alors } \ell_i \leqslant 2 + \lceil \log^{(i+1)} n \rceil. \text{ »}$$

L'induction commence à $i = 1$, car il faut $i > 0$. On a $\ell_1 = 1 + \lceil \log \ell_0 \rceil = 1 + \lceil \log \lceil \log n \rceil \rceil = 1 + \lceil \log \log n \rceil \leqslant 2 + \lceil \log^{(2)} n \rceil$ en appliquant le fait 7.1. Donc $P(1)$ est vraie.

Supposons que $P(i-1)$ soit vraie. On veut prouver $P(i)$ pour $i \geqslant 2$. Montrons dans un premier temps que si $\ell_{i-1} > 3$, alors $\ell_{i-1} \leqslant 2 \cdot \lceil \log^{(i)} n \rceil$.

En effet, si $\ell_{i-1} > 3$ et $i \geqslant 2$, alors $\ell_{i-2} > 3$ (par la proposition 7.4). Donc, par $P(i-1)$, $\ell_{i-1} \leqslant 2 + \lceil \log^{(i)} n \rceil$. En particulier $3 < 2 + \lceil \log^{(i)} n \rceil$ puisque $\ell_{i-1} > 3$, ce qui implique $\lceil \log^{(i)} n \rceil \geqslant 2$. On en déduit que $2 + \lceil \log^{(i)} n \rceil \leqslant 2 \cdot \lceil \log^{(i)} n \rceil$, car $2 + x \leqslant 2x$ dès que $x \geqslant 2$. Au final, $\ell_{i-1} \leqslant 2 \cdot \lceil \log^{(i)} n \rceil$.

D'après l'algorithme, si $\ell_{i-1} > 3$,

$$\begin{aligned}
\ell_i \;=\; 1 + \lceil \log \ell_{i-1} \rceil \;&\leqslant\; 1 + \left\lceil \log\left(2 \cdot \left\lceil \log^{(i)} n \right\rceil\right) \right\rceil \\
&\leqslant\; 1 + \left\lceil \log(2) + \log\left\lceil \log^{(i)} n \right\rceil \right\rceil \\
&\leqslant\; 2 + \left\lceil \log\left\lceil \log^{(i)} n \right\rceil \right\rceil \\
&\leqslant\; 2 + \left\lceil \log^{(i+1)} n \right\rceil \quad \text{(par le fait 7.1)}
\end{aligned}$$

ce qui montre que $P(i)$ est vraie. □

Dans la suite on note [14] $\log^* n = \min\{i : \log^{(i)} n \leqslant 1\}$. Dit autrement c'est le plus petit entier i tel que

$$n \leqslant 2^{\left.2^{\cdot^{\cdot^{2}}}\right\} i}.$$

13. Si on avait f croissante et f^{-1} strictement décroissante, alors on aurait $x \leqslant y \Rightarrow f(x) \leqslant f(y) \Rightarrow f^{-1}(f(x)) > f^{-1}(f(y)) \Rightarrow x > y$: contradiction.

14. Parfois $\log^* n$ est défini comme le plus petit entier i tel que $\log^{(i)} n \leqslant 2$. On peut aussi définir la fonction par $\log^* n = 1 + \log^*(\log n)$ si $n > 1$, et 0 sinon.

Autant dire que la fonction $\log^* n$ croît extrêmement lentement (cf. figure 7.4). Par exemple,

$$\log^*\left(2^{65536}\right) = \log^*\left(2^{2^{16}}\right) = \log^*\left(2^{2^{2^2}}\right) = 5\,.$$

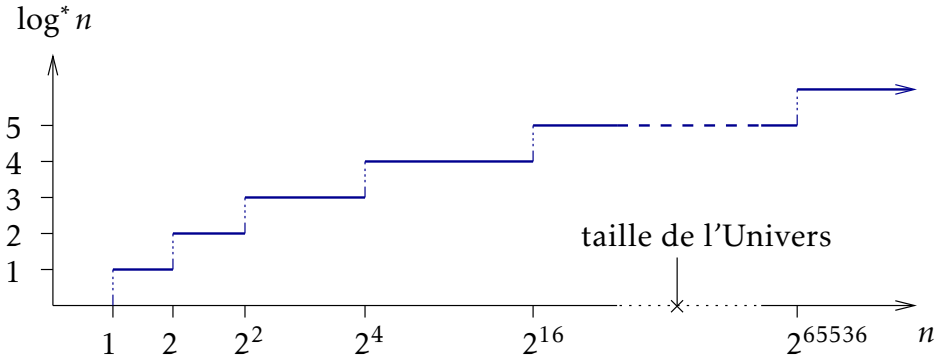


FIGURE 7.4 – La fonction $\log^*$, qui n'est pas bornée, a une croissance extrêmement lente. Lorsque n correspond à un nombre d'entités physiques, $\log^* n \leqslant 5$.

On rappelle que le nombre de particules de l'univers est estimé à seulement $10^{80} \approx 2^{266}$. Raisonnablement, le $\log^*$ de n'importe quel nombre de processeurs ne dépassera jamais 5.

De manière générale, pour toute fonction f, on peut définir[15] $f^*(n) = \inf\{i : f^{(i)}(n) \leqslant 1\}$ où $f^{(i)}(n) = f(f(\cdots f(f(n))\cdots)$ est i fois l'itération de f sur n. Par exemple, si $f(n) = \lfloor n/2 \rfloor$, alors $f^*(n) = \lfloor \log n \rfloor$, car $f^{(i)}(n) = \lfloor n/2^i \rfloor$ est $\leqslant 1$ dès que $n/2^i < 2$, soit $i > (\log n) - 1$ ce qui implique[16] que $i = \lfloor \log n \rfloor$. Et si $f(n) = \left\lfloor \sqrt{n} \right\rfloor$, alors $f^*(n) = 1 + \lfloor \log\log n \rfloor$, car $f^{(i)}(n) = \lfloor n^{1/2^i} \rfloor$ est $\leqslant 1$ dès que $n^{1/2^i} < 2$, soit $\frac{1}{2^i}\log n < 1$ ou encore $2^i > \log n$ ce qui implique que $i = 1 + \lfloor \log\log n \rfloor$. Notez bien que tous les log sont en base deux.

7.4.4 Résumé

Lemme 7.1 *L'algorithme Color6 calcule une 6-coloration en temps au plus* $\log^* n$ *pour tout graphe 1-orienté à* $n > 1$ *sommets.*

Preuve. On a vu dans la proposition 7.6 que Color6 calculait une 6-coloration. Soit t le nombre de boucles « Répéter » réalisées par l'algorithme. C'est le temps pris par l'algorithme, car il correspond au nombre de rondes de communication.

15. Notez que $\min\{i : f^{(i)}(n) \leqslant 1\}$ pourrait ne pas exister.
16. Le plus petit entier $i > x$, avec $x \in \mathbb{R}$, est $i = 1 + \lfloor x \rfloor$.

Si $\ell_0 \leqslant 3$, alors d'après la proposition 7.4, $\ell_1 = \ell_0$, et donc l'algorithme ne prend alors qu'une seule ronde. Puisque $\log^* n \geqslant 1$ dès que $n > 1$, on a donc dans ce cas que $t = 1 \leqslant \log^* n$.

Supposons que $\ell_0 > 3$. Posons $i = (\log^* n) - 1$. Alors on a $2 + \left\lceil \log^{(i+1)} n \right\rceil \leqslant 3$. En effet,

$$\begin{aligned} 2 + \left\lceil \log^{(i+1)} n \right\rceil \leqslant 3 \quad &\Leftrightarrow \quad \left\lceil \log^{(i+1)} n \right\rceil \leqslant 1 \\ &\Leftrightarrow \quad \log^{(i+1)} n \leqslant 1 \quad \Leftrightarrow \quad i+1 = \log^* n\,. \end{aligned}$$

On considère alors la fin de l'étape $i-1$. Notons que $i > 0$ car $\log^* n > 1$ dès que $n > 2$, ce qui est bien notre cas puisqu'on a supposé $\ell_0 = \lceil \log n \rceil > 3$: c'est donc en fait que $n > 8$. On a alors deux sous-cas.

Soit $\ell_{i-1} \leqslant 3$, et alors d'après la proposition 7.4, $\ell_i = \ell_{i-1}$. L'algorithme s'arrête alors à la fin de l'étape $i = (\log^* n) - 1$. Donc $t = i = (\log^* n) - 1$.

Soit $\ell_{i-1} > 3$, et la proposition 7.7 s'applique ($i > 0$). Donc $\ell_i \leqslant 2 + \left\lceil \log^{(i+1)} n \right\rceil \leqslant 3$. D'après la proposition 7.4, $\ell_{i+1} = \ell_i$ et donc l'algorithme s'arrête alors à la fin de l'étape $i+1$. Donc $t = i + 1 = \log^* n$.

Dans tous les cas $t \leqslant \log^* n$. □

L'analyse de la validité de l'algorithme Color6 utilise le fait que les identités des sommets de G définissent initialement une coloration (propre). Tout reste valable si l'on part d'une m-coloration avec m quelconque, et pas seulement $m = n$. Il suffit de poser $\ell := \lceil \log m \rceil$ dans l'instruction 1 de l'algorithme Color6 et que $x \in [0, m[$ soit la couleur initiale de u. Le temps d'exécution est alors au plus $\log^* m$, que $m < n$ ou $m > n$.

Une autre remarque concerne la variable $\textsc{Parent}(u)$. En y regardant de plus près, ce qu'on utilise dans la preuve est que :

- $\textsc{Parent}(u) \in N(u)$; et
- si $v \in N(u)$, alors $u = \textsc{Parent}(v)$ ou $u = \textsc{Parent}(v)$.

Mais en soit, rien n'interdit d'avoir le cas $v = \textsc{Parent}(u)$ et $u = \textsc{Parent}(v)$, c'est-à-dire une double orientation de l'arête uv. Les propositions 7.1 et 7.2 s'appliquent à tout triplet d'entiers (x, y, z) tels que $x \neq y$ et $y \neq z$. On pourrait avoir $x = z$, par exemple si c'est la couleur d'un même sommet. Le seul problème cependant, si on a $v = \textsc{Parent}(u)$ et $u = \textsc{Parent}(v)$, est qu'il faudrait s'assurer que v reçoive bien la couleurs x de u. Pour capturer ce cas, l'instruction 3b de l'algorithme Color6 devrait donc être :

3(b) $\textsc{Send}(x, v)$ pour tout $v \in N(u)$

Cela produit un message en excès si on n'a pas cette double orientation. En fait il faudrait envoyer le message à tout voisin v dont le parent est u. Malheureusement, l'ensemble $F(u) := \{v \in N(u) : \textsc{Parent}(v) = u\}$ n'est pas connu *a priori*. On peut cependant le

calculer en une ronde au départ. [*Question*. Comment ?] Et alors l'instruction 3b deviendrait :

3(b) SEND(x, v) pour tout $v \in F(u)$

7.4.5 De six à trois couleurs

Pour obtenir la 3-coloration finale on élimine simplement les couleurs de numéro $\geqslant 3$. Idéalement, on aimerait appliquer ReducePalette$(6, 3, \text{COLOR})$ qui prend trois rondes. Mais la réduction ne marche *a priori* seulement si pour tout sommet u, $|\text{COLOR}(N(u))| < 3$ (voir l'énoncé de la proposition 7.2). Malheureusement, ce n'est plus forcément le cas avec Color6 si $\deg(u) > 2$, ce qui peut arriver dans le cas d'un arbre. On se ramène au cas favorable et on élimine les couleurs $3, 4, 5$ « manuellement » en appliquant ReducePalette$(k+1, k, \text{COLOR})$ qui supprime la couleur k. Plus précisément, pour chaque couleur $k \in \{3, 4, 5\}$:

1. On modifie localement (en une ronde) la couleur de chaque sommet u de sorte qu'il ne « voit » que deux couleurs, pour avoir $|\text{COLOR}(N(u))| < 3$. Pour cela on fait en sorte que les sommets ayant le même parent (les fratries donc) obtiennent la même couleur.
2. On applique ReducePalette$(k+1, k, \text{COLOR})$ pour éliminer la couleur k, ce qui revient à faire FirstFree sur tous les sommets de couleur k.

Pour le point 1, on utilise la procédure ShiftDown qui consiste simplement à recolorier chacun des sommets par la couleur de son parent. Si le sommet n'a pas de parent, il se colorie avec la plus petite couleur différente de lui-même (soit la couleur 0 ou 1). Pour implémenter la procédure ShiftDown on envoie donc sa couleur à tous ses voisins dont on est leur parent (à tous ses fils donc), puis on se re-colorie avec la couleur reçue. Visuellement, c'est comme si les couleurs se décalaient d'un étage vers les fils (d'où le nom de ShiftDown si les fils sont dessinés en dessous de leur parent). Notons que ShiftDown ne prend qu'une seule ronde.

Algorithme ShiftDown
(code du sommet u)

1. NEWROUND
2. $x :=$ COLOR(u)
3. SEND(x, v) pour tout $v \in N(u) \setminus \text{PARENT}(u)$
4. Si PARENT$(u) \neq \perp$, alors $y :=$ RECEIVE(PARENT(u)) sinon $y :=$ FirstFree$(\{x\})$
5. COLOR$(u) := y$

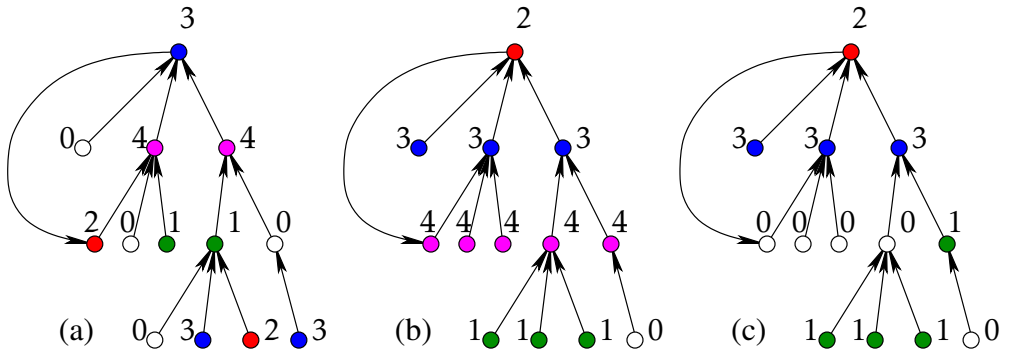


FIGURE 7.5 – Illustration de la procédure ShiftDown appliquée à une 1-orientation ayant une 5-coloration. En (a), on voit que la couleur 4 ne peut pas être totalement supprimée par un simple FirstFree. En (b), après application de ShiftDown. En (c), après la suppression de la couleur 4 par réduction de palette grâce à ReducePalette(5, 4, COLOR), ce qui revient à faire FirstFree à tous les sommets de couleur 4. On voit qu'il faudra effectuer de nouveau un ShiftDown avant de pouvoir supprimer la couleur 3.

Après un ShiftDown, chaque sommet u ne « voit » plus que deux couleurs parmi tous ses voisins : celle de son parent et celle de ses fils (puisqu'ils ont tous la même). On peut donc obtenir trois couleurs avec trois appels à ShiftDown (suivit d'un ReducePalette). Au total cela rajoute 6 rondes.

Algorithme Color6to3

Pour chaque $k \in \{3, 4, 5\}$ faire :

1. ShiftDown
2. ReducePalette$(k+1, k, \text{COLOR})$

L'ordre d'élimination des couleurs n'est pas important. Cependant, il est très important que tous les processeurs terminent en même temps l'algorithme précédent, Color6. C'est bien le cas, la variable ℓ étant identique au cours du temps sur tous les processeurs.

D'après la proposition 7.2, ReducePalette$(k+1, k, \text{COLOR})$ prend une ronde et élimine la couleur k à condition que $|\text{COLOR}(N(u))| < k$ ce qui est bien le cas puisque $k \geqslant 3$ et que ShiftDown garantit précisément que $|\text{COLOR}(N(u))| \leqslant 2$.

D'où le résultat final suivant :

Théorème 7.1 *On peut calculer une* 3*-coloration en temps* $\log^* m + 6$ *pour tout graphe* 1*-orienté et possédant une* m*-coloration* [17].

17. Pour être tout à fait formel, il faudrait supposer $m > 1$ à cause du Lemma 7.1. Cela revient à supposer que le graphe possède au moins une arête, une hypothèse plus que raisonnable dans le cadre du calcul distribué.

Si pour toute valeur raisonnable de m, $\log^* m \leqslant 5$, c'est donc le passage de 6 à 3 couleurs qui est le plus coûteux, le passage de n à 6 couleurs nécessitant lui que 5 étapes ! [*Exercice.* En utilisant l'astuce de la page 7.7.1, montrez qu'on peut améliorer le théorème 7.1 de deux rondes, et donc obtenir un temps d'au plus $\log^* m + 4$.]

Dans [Ryb11] il est démontré, dans le cas plus restrictif d'un arbre 1-orienté à n sommets, que le nombre optimal de rondes est au plus $\log^* n + 3$ et au moins $\log^* n + 1$.

Deux questions restent en suspens :

1. Comment s'affranchir de la connaissance de n dans l'algorithme Color6 ? On pourrait initialiser $\ell := \text{ID}(u)$ par exemple. Mais avec des valeurs différentes de ℓ, les sommets s'arrêteront à des rondes différentes. Il faut alors modifier la fonction PosDiff, car dans sa forme actuelle il est possible que $\text{PosDiff}(x, y) = y$ (cf. les exemples de la page 7.4.2). Cela pose un problème pour re-colorier dans ce cas x en y si jamais y ne change plus. Ceci dit, cela se produit dans un nombre limité de cas. On a vu que cela ne se produit pas lorsque $y \in \{0, 1\}$ (absence de parent).
2. Comment enchaîner avec l'algorithme Color6to3 si tous les sommets ne terminent pas en même temps ? Il faut alors détecter que certains de ses voisins n'ont pas terminer (il faut ajouter au minimum une variable) et attendre le cas échéant.

Les algorithmes distribués qui s'affranchissent de toute information globale sur le graphe (nombre de sommets, degré maximum, ...) sont dits *uniformes*. Dans certaines conditions il est possible de rendre uniforme un algorithme qui initialement ne l'est pas [KSV11]. Voir aussi l'excellent survol [Suo13] sur les algorithmes locaux et le modèle LOCAL.

7.5 Algorithme uniforme pour les 1-orientations

[*Cyril.* Cette section a un problème. La solution est en fait donnée dans `http://dcg.ethz.ch/lectures/podc_allstars` (voir Chapitre 1, Exercises/Solution). Il existe aussi un papier récent [MP23] qui donne une autre solution, mais pour le cycle (ou la ligne).]

On suppose que chaque sommet possède une coloration initiale, c'est-à-dire que $\text{Color}(u)$ est définie pour chaque sommet u et différent de ses voisins. Par exemple, $\text{Color}(u)$ pourrait être $\text{ID}(u)$, mais pas forcément. Cependant, l'algorithme n'a aucune information à propos de $\max_{v \in V(G)} \text{Color}(v)$ la couleur maximum des couleurs initiales des sommets du graphe ou du nombre de sommets, contrairement à Color6 où la valeur de n était codée en dur dans l'algorithme.

Algorithme UnifColor6
(code du sommet u)

1. Poser $x := \text{Color}(u)$
2. Si $\text{Parent}(u) = \perp$, alors $y := \text{FirstFree}(\{x\})$

3. Répéter :
   (a) NewRound
   (b) Send(x, v) pour tout $v \in N(u) \setminus$ Parent(u)
   (c) Si Parent$(u) \neq \perp$, alors $y :=$ Receive(Parent(u))
   (d) Si $x \geqslant 6$ et $y \geqslant 6$, alors $x :=$ PosDiff(x, y)
   (e) Sinon choisir $x \in \{x_0, 2 + x_1\} \setminus \{y\}$ où $x_i = \left\lfloor x/2^i \right\rfloor \bmod 2$

   Tant que $y \geqslant 6$
4. Color$(u) := x$

6-coloration.

Terminaison.

Complexité. Soit $t > 0$ le nombre de rondes exécutées par le sommet u.

On remarque que si, dans la boucle « répéter », l'instruction (e) est exécutée, alors $x < 4$. Notons aussi qu'on aurait pu écrire l'instruction (e) sous la forme d'un simple FirstFree d'un ensemble à trois éléments, ce qui montre aussi que $x < 4$:

(e) Sinon, $x :=$ FirstFree$(\{y, 1 - x_0, 3 - x_1\})$, où $x_i = \left\lfloor x/2^i \right\rfloor \bmod 2$

[*Cyril*. À FINIR]

Donc s'il s'agit d'une coloration, c'est une 6-coloration.

Il est aussi simple de vérifier que le temps d'exécution de l'algorithme UnifColor6 est d'au plus $\log^* m + 1$, si Color$(u) < m$. En effet, l'analyse est similaire à celle de Color6 puisqu'on a vu que lorsque la longueur binaire de la couleur de u (soit $|\text{bin}(x)|$) est $\leqslant 3$ l'algorithme s'arrête avec 6 couleurs. Et cela se produit après au plus $\log^* m$ rondes et applications de PosDiff(x, y) où m est un majorant strict de la couleur initiale de u. Il y a cependant deux différences. La première est que lorsque la couleur x devient < 6, l'algorithme ne s'arrête pas immédiatement comme dans Color6, mais effectue une ronde supplémentaire. La deuxième est que lorsque $y < 6$, alors la nouvelle valeur de x n'est plus forcément égale à PosDiff(x, y) comme dans Color6. Cependant, dans ce cas l'algorithme UnifColor6 fait que $x < 4$ et l'algorithme s'arrête à la ronde suivante. Donc le nombre de rondes de UnifColor6 est au plus celui de Color6 plus un.

Il reste à vérifier que Color est bien une coloration. Pour cela, on considère le début d'une certaine ronde i, c'est-à-dire juste avant l'exécution parallèle, pour la i-ème fois, de l'instruction NewRound par les processeurs encore actifs. On note x_i la couleur du sommet u au début de la ronde i, et y_i la couleur du parent v (éventuellement virtuel) de u. Notons qu'à la ronde i, il est possible que le sommet u soit inactif, c'est-à-dire qu'il

ait terminé d'exécuter son programme. Cependant, x_i est bien défini tant qu'au moins un sommet du graphe reste actif.

On notera x'_i et y'_i les couleurs de u et de son parent v à la fin de la ronde i. On veut montrer que $x'_i \neq y'_i$, en supposant que $x_i \neq y_i$. Observons que si $y_i \geqslant 6$, alors $\textsc{Parent}(u) \neq \perp$ et que v est actif.

Cas 1 : $x_i < 6$ et $y_i < 6$. Alors d'après l'algorithme, $x'_i = x_i$ et $y'_i = y_i$. Donc $x'_i \neq y'_i$.

Cas 2 : $x_i \geqslant 6$ et $y_i \geqslant 6$. Alors v possède un parent (éventuellement virtuel) de couleurs z_i, et $x'_i = \text{PosDiff}(x_i, y_i) \neq \text{PosDiff}(y_i, z_i) = y'_i$ grâce à la propriété 7.1 de PosDiff.

Il n'est pas difficile de voir que si $x_i \geqslant 6$, alors $x'_i < x_i$ (la longueur binaire de la couleur ne fait que de diminuer en appliquant $\text{PosDiff}(x_i, \cdot)$ si $x_i \geqslant 6$). Et donc après le cas 1 on ne peut pas avoir ... [*Cyril*. À FINIR] On note $c_i(u)$ la valeur de la condition c lors de l'exécution de l'instruction 3(d) par le sommet u lors de la ronde i.

Cas 3 : $x_i \geqslant 6$ et $y_i < 6$. On a $y'_i = y_i$. On va supposer que i est le plus petit possible. (On verra que ce cas ne peut se produire plusieurs fois.) Vérifions que la couleur y_i est bien transmise à u lors de la ronde i.

En effet, soit $i = 1$ et alors v exécute à la première ronde un Send vers ses fils (et en particulier vers u). Soit $i > 1$ et alors par minimalité de i la couleur de v est passée de $\geqslant 6$ à < 6 à la ronde $i-1$, c'est-à-dire $y_{i-1} \geqslant 6$ et $y'_{i-1} < 6$. Dans ce cas la condition $c_{i-1}(v) = (y_{i-1} \geqslant 6)$ est Vrai et donc $y_i = y'_{i-1}$ est envoyé par v à ses fils lors de la ronde i.

Par l'instruction 3(e) ($c = \text{Vrai}$ et $y < 6$), on a $x'_i \neq y_i$ et donc $x'_i \neq y'_i$ puisque $y'_i = y_i$. De plus, on remarque que $x'_i < 4$, et donc le cas 3 ne peut se produire qu'une seule fois, ce qui justifie le choix de i.

[*Cyril*. Le cas (4) suivant ne marche pas. En fait, lorsque x s'arrête (devient < 6) alors que son parent continue, rien ne dit que y ne s'arrêtera pas sur la même valeur x dans le futur si y ne dépend que de y et de son parent z. Cependant, le contraire est bon, si y s'arrête avant x. Il faut donc en tenir compte en forçant x à ne pas s'arrêter (on continue les PosDiff dans $\{0, \ldots, 5\}$) si un de ses fils est < 6. Peut-être se restreindre aux cycles avec une fonction $f(x, y, z)$ tenant compte simultanément des deux voisins ?]

Cas 4 : $x_i < 6$ et $y_i \geqslant 6$. On a $x'_i = x_i$. On va supposer que i est le plus petit possible. (On verra que ce cas ne peut se produire plusieurs fois.) Alors v possède un parent (éventuellement virtuel) de couleurs z_i, et donc $y'_i = \text{PosDiff}(y_i, z_i)$. Si $i = 1$, alors $x'_i = \ldots$. Si $i > 1$, alors par minimalité de i, $x'_{i-1} \geqslant 6$. ... Par le choix de x'_i dans l'instruction 3(e), soit $x'_i = w_0$ soit $x'_i = 2 + w_1$ où $w_j = \left\lfloor y_i / 2^j \right\rfloor \bmod 2$. Or par la propriété 7.2, $\text{PosDiff}(y_i, z_i) \notin \{w_0, 2 + w_1\}$.

...

D'où le résultat, en ajoutant les six rondes de Color6to3 pour passer à une 3-coloration [*Cyril*. À FINIR. Il faut expliquer comment appliquer exactement le ShiftDown alors que tous le monde n'a pas encore terminé...] :

Théorème 7.2 *On peut calculer une* 3*-coloration en temps* $\log^* m + 7$ *pour tout graphe* 1*-orienté et possédant une* m*-coloration, la valeur de* m *étant inconnue des sommets.*

7.6 Coloration des k-orientations et au-delà

On peut montrer que les graphes de degré maximum Δ sont $\lceil \Delta/2 \rceil$-orientables. Les graphes planaires sont 3-orientables, et plus généralement, les graphes k-dégénérés sont k-orientables. On rappelle que les graphes k-dégénérés ont la propriété que tout sous-graphe induit possède un sommet de degré au plus k. En enlevant ce sommet et en itérant, on crée au plus k relations de parentés.

Bien sûr la question est de pouvoir calculer rapidement une k-orientation d'un graphe k-orientable *from scratch*. C'est un autre problème en soit. Il existe des algorithmes distribués sophistiqués pour cela. Le meilleur d'entre eux prend un temps $O(\log n)$ [BE08].

Pour les graphes k-orientables, on s'intéresse *a priori* à calculer une $(2k+1)$-coloration : une 3-coloration pour les 1-orientables, une 5-coloration pour les 2-orientables, etc. En effet, il n'est pas trop difficile de voir que :

Fait 7.2 *Tout graphe* k*-orientable est* $(2k+1)$*-coloriable.*

Preuve. Remarquons que dans tout graphe G il existe un sommet de degré au plus le degré moyen, c'est-à-dire un sommet u tel que $\deg(u) \leqslant \frac{1}{|V(G)|}\sum_{v\in V(G)} \deg(v) = 2|E(G)|/|V(G)|$. Or, par définition, si G est k-orientable alors $|E(G)| \leqslant k|V(G)|$, et donc il possède un sommet $\deg(u) \leqslant 2k$. Donc, si l'on peut colorier $G \setminus \{u\}$ avec une $(2k+1)$-coloration [18] c, on pourra colorier G avec $2k+1$ couleurs en coloriant u par $c(u) := \mathsf{FirstFree}(c(N(u))) \in \{0,\dots,\deg(u)\} \subseteq \{0,\dots,2k\}$. Et bien sûr $G \setminus \{u\}$ est k-orientable si G l'est. □

Le nombre de couleurs $2k+1$ est, de manière générale, la meilleure borne possible car il y a des graphes k-orientables sans $2k$-coloration. Par exemple, le graphe complet à $2k+1$ sommets, K_{2k+1}, est k-orientable et de nombre chromatique $2k+1$. En effet, la relation qui à tout sommet $u \in \{0,\dots,2k\}$ associe $\textsc{Parent}_i(u) = (u+i) \bmod (2k+1)$ pour chaque $i \in \{1,\dots,k\}$ définit une k-orientation. Voir l'exemple [19] de la figure 7.6 pour $k = 2$.

18. Ce qui est vrai lorsque $G \setminus \{u\}$ possède $2k+1$ sommets ou moins.

19. En fait, il existe un résultat plus fort : on peut toujours décomposer K_{2k+1} en k cycles hamiltoniens, ce qui a été démontré par Walecki dans les années 1890 (cf. [Als08]). Lorsque $2k+1$ est un nombre premier, alors la relation qui à tout sommet $u \in \{0,\dots,2k\}$ associe $\textsc{Parent}_i(u) = (u+i) \bmod (2k+1)$ pour chaque $i \in \{1,\dots,k\}$ définit effectivement k cycles hamiltoniens. Pour les autres valeurs cela peut être plus complexe, cette relation ne définissant plus forcément des cycles de longueur $2k+1$.

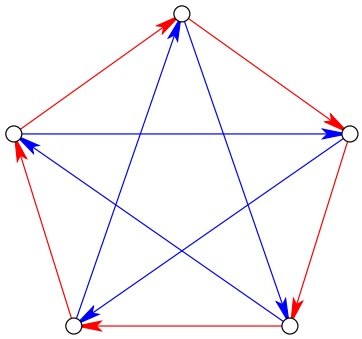


FIGURE 7.6 – Graphe 2-orientable nécessitant une 5-coloration.

En fait pour $k > 1$, on visera plutôt une $(\Delta+1)$-coloration car la réduction parallèle de couleur avec ReducePalette$(3^k, 2k+1, c)$ ne marche que si $|c(N(u))| < 2k+1$ pour tout sommet u. Ici ShiftDown ne peut plus garantir une telle propriété.

[*Cyril.* À revoir ce passage, pointer un exercice (examen janvier 2021) sur les (k', t')-décompositions. On peut faire une $(2k, t')$-décomposition en prenant $k' = 2k$ et $b = 2k+1$. On peut faire une $(2k+1)$-coloration car $m \leqslant (k'+1)(1-1/b)n/2 = (2k+1)(2k/(2k+1))n/2 = kn$ ce qui est bien le cas pour les k-orientations. On obtient un temps de $O(kt') + \log^* n + 2^{O(k)}$ avec $t' = O(\log n/\log(2k+1)/(2k)) = O(\log n/\log(1+O(1/k))) = O(k\log n)$. Au final le temps est de $O(k^2 \log n) + 2^{O(k)}$. C'est lent, mais il y a moins de couleurs et surtout il n'y a pas besoin de connaître l'orientation ! Mais pour cela il faut l'algo en $\log^* n + 3^k$.] Notons quand même que dans un graphe k-orientable G au moins la moitié des sommets ont un degré au plus $4k$. Puisque si la moitié d'eux avaient un degré $\geqslant 4k+1$ on aurait $\sum_u \deg(u) = 2|E(G)| \geqslant (4k+1) \cdot |V(G)|/2 > 2k|V(G)|$ ce qui implique $|E(G)| > k|V(G)|$: contradiction, on a vu précédemment que $|E(G)| \leqslant k|V(G)|$. Donc on pourrait imaginer plusieurs phases où seuls les sommets u avec $|c(N(u))| < 4k+1$ appliquent la réduction de palette. Après $\log n$ phases, tous les sommets se seraient re-coloriés en $4k+1$ couleurs. Cependant cette approche qui produit une $(4k+1)$-coloration en approximativement $(3^k + \log^* n) \cdot \log n$ rondes est bien plus lente puisque $\log^* n$ est extrêmement petit en comparaison à $\log n$.

Théorème 7.3 *On peut calculer une $(\Delta+1)$-coloration en temps $\log^* m + 2^{O(k)}$ pour tout graphe k-orienté de degré maximum Δ possédant une m-coloration.*

Preuve. On suppose ici que chaque sommet u possède k variables $\text{PARENT}_1(u), \ldots, \text{PARENT}_k(u)$ représentant les (au plus) k parents du sommets u. Comme précédemment on pose $\text{PARENT}_i(u) = \perp$ si le i-ème parent de u n'existe pas.

Les arcs liant u à $\text{PARENT}_i(u)$ sont dit de type i. Notons que pour chaque i, les arcs de type i induisent une 1-orientation qu'on notera F_i. L'idée est donc d'appliquer en parallèle l'algorithme Color6 puis Color6to3 pour chaque F_i.

Comme un sommet appartient à plusieurs F_i, la couleur d'un sommet u est alors

représenté par un vecteur $(x_1,\dots,x_k)$ de k composantes, x_i étant la couleur pour F_i. Plus précisément, si la couleur courante du voisin $\textsc{Parent}_i(u)$ de u est le vecteur $(y_1,\dots,y_k)$, alors u, qui reçoit y_i de $\textsc{Parent}_i(u)$, calcule

$$x_i := \text{PosDiff}(x_i, y_i)\ .$$

Bien sûr, l'algorithme se termine après $(\log^* m) + 6$ rondes sur un vecteur où chaque composante est dans $\{0,1,2\}$. Potentiellement, cela fait 3^k vecteurs différents possibles et autant de couleurs possibles pour les sommets du graphe. La réduction ReducePalette prend donc malheureusement un temps [20] $\max\left\{3^k-(\Delta+1),0\right\} = 2^{O(k)}$ pour réduire à $\Delta+1$ couleurs. □

Remarquons qu'il est trivial de calculer une Δ-orientation à partir d'un graphe sans orientation particulière initiale. D'où :

Corollaire 7.1 *On peut calculer une $(\Delta+1)$-coloration en temps $\log^* m + 2^{O(\Delta)}$ pour tout graphe de degré maximum Δ possédant une m-coloration.*

Notons que Δ et m doivent être connus de tous les processeurs. Il a été conjecturé par [SV93] que tout algorithme procédant itérativement par réduction de couleur (c'est-à-dire qui applique successivement la même boucle de calcul à une coloration propre – comme la fonction PosDiff dans Color6) ne peut pas produire une $O(\Delta)$-coloration plus rapidement que $\Omega(\Delta\log\Delta)$ rondes.

Les meilleurs algorithmes connus [BE09][Bar15][FHK16][MT20] pour les graphes de degré maximum Δ permet de calculer une $(\Delta+1)$-coloration en temps $o(\Delta)+\log^* m$. Ces algorithmes ne procèdent pas, bien sûr, par réduction itérative de couleurs. Ils utilisent des colorations qui ne sont pas toutes à fait propres où les sous-graphes induits par chaque classe de couleurs sont de degré maximum borné. (Dans une coloration propre ces sous-graphes doivent être des stables, donc de degré maximum 0.)

7.7 Coloration des cycles et borne inférieure

7.7.1 Coloration des cycles

Nous avons vu que les cycles sont 1-orientables, et à ce titre on peut leur appliquer le résultat de la proposition 7.1. On peut ainsi calculer une 3-coloration en temps $\log^* n + O(1)$. On va présenter une amélioration de l'algorithme dans le cas des cycles permettant d'aller deux fois plus vite. On va supposer que les cycles sont *orientés*,

20. On rappelle que $3^k = (2^{\log 3})^k = 2^{k\log 3} = 2^{O(k)}$.

c'est-à-dire que les sommets de droit et de gauche sont connus de chaque sommet u par les relations $\textsc{Succ}(u)$ et $\textsc{Pred}(u)$. Par rapport à la terminologie précédente, un cycle orienté est équivalent à un cycle 1-orienté c'est-à-dire muni d'une 1-orientation : $\textsc{Succ}(u) = \textsc{Parent}(u)$ et $\textsc{Pred}(u) = N(u) \setminus \textsc{Parent}(u)$.

L'idée est d'imaginer que l'on calcule PosDiff en alternance avec son successeur et son prédécesseur. À chaque ronde, si tous les sommets décident de calculer PosDiff avec leur successeur, alors la coloration reste propre, car c'est comme si l'on avait fixé $\textsc{Parent}(u) = \textsc{Succ}(u)$ dans l'algorithme Color6. De même, si tous décident de faire PosDiff avec leur prédécesseur. Donc la coloration reste propre même si l'on alterne les rondes où le parent est le successeur avec celles où le parent est le prédécesseur. De manière générale, à chaque ronde on peut fixer arbitrairement le parent de chaque sommet, du moment que cela définisse une 1-orientation et que PosDiff est appliquée entre la couleur de u et celle de sa relation. Par exemple, on pourrait l'appliquée les rondes paires avec son successeur, les rondes impaires avec son prédécesseur. On va voir que la deuxième application peut être simulée par u sans avoir à communiquer.

Considérons trois sommets consécutifs de couleurs $z - x - y$. En appliquant PosDiff aux successeurs $z \to x \to y$, on obtient les couleurs $z' - x' - y'$ avec $z' = \textsc{PosDiff}(z, x)$ et $x' = \textsc{PosDiff}(x, y)$. C'est une coloration propre. Si maintenant on applique PosDiff aux prédécesseurs $z' \leftarrow x' \leftarrow y'$, on obtient les nouvelles couleurs $z'' - x'' - y''$ avec $x'' = \textsc{PosDiff}(z', x')$. On a donc pour le sommet central la couleur :

$$x'' \;=\; \textsc{PosDiff}(\textsc{PosDiff}(z, x), \textsc{PosDiff}(x, y)) \,.$$

C'est finalement une fonction qui ne dépend que de x, y, z. Ces valeurs sont connues dès la première communication. La seconde étape de communication (avec les prédécesseurs) est en fait inutile pour calculer x''. On peut donc économiser la moitié des rondes de communications.

Pour passer d'une 6-coloration à une 3-coloration en appliquant cette fois-ci directement ReducePalette$(6, 3, \textsc{Color})$ car la condition $|\textsc{Color}(N(u))| < 3$ est vérifiée pour tout sommet u (voir la proposition 7.2). On va donc aussi deux fois-plus vite pour le passage de 6 à 3 couleurs.

Algorithme ColorRing
(code du sommet u)

1. Poser $x := \mathrm{ID}(u)$ et $\ell := \lceil \log n \rceil$
2. Répéter :
   (a) NewRound
   (b) $\textsc{Send}(x, \textsc{Succ}(u))$ et $\textsc{Send}(x, \textsc{Pred}(u))$)
   (c) $y := \textsc{Receive}(\textsc{Succ}(u))$ et $z := \textsc{Receive}(\textsc{Pred}(u))$)
   (d) $x := \textsc{PosDiff}(\textsc{PosDiff}(z, x), \textsc{PosDiff}(x, y))$
   (e) $\ell' := \ell$ et $\ell := 1 + \lceil \log(1 + \lceil \log \ell \rceil) \rceil$

Jusqu'à ce que $\ell = \ell'$

3. Poser $\textsc{Color}(u) := x$
4. ReducePalette$(6, 3, \textsc{Color})$

D'où le résultat qu'on ne démontrera pas :

Proposition 7.8 *L'algorithme ColorRing produit une 3-coloration en temps* $\lceil \frac{1}{2} \log^* m \rceil + 3$ *sur les cycles orientés possédant une m-coloration.*

En fait, on peut gagner une ronde dans le passage de 6 à 3 couleurs avec une solution *ad hoc* passant de 6 à 4 couleurs en une seule ronde au lieu de deux. On passe ensuite de 4 à 3 avec un ReducePalette$(4, 3, \textsc{Color})$ d'une ronde. Pour cela, les sommets de couleur 4 appliquent FirstFree$(\textsc{Color}(N(u)))$ tandis que ceux de couleurs 5 appliquent FirstFree$(\textsc{Color}(N(u)) \cup \{0, 1\})$. (En fait, ils peuvent se contenter de le faire que lorsqu'ils sont voisins d'une couleur 4.) [*Exercice.* Démontrez que cette astuce permet effectivement de passer de 6 à 4 couleurs dans le cas d'un graphe 6-colorié général (donc plus forcément de degré maximum deux) à condition que chaque sommet u de couleur 4 ou 5 ne voit que deux couleurs, formellement $|\textsc{Color}(N(u))| < 2$.]

La borne la plus précise connue sur le temps pour un cycle orienté muni d'une m-coloration est de $\lceil \frac{1}{2} \log^* m \rceil + 1$ [Ryb11, Théorème 8.3, p. 71]. Il a été démontré depuis dans [RS15] que pour une infinité de valeur de n, tout cycle orienté peut être 3-colorié en exactement $\frac{1}{2} \log^* n$ rondes.

Il est possible de faire seulement 3 rondes si les identifiants sont sur des adresses d'au plus 2048 bits ($m = 2^{2048}$). Il faut faire une ronde classique, puis résoudre le problème en deux rondes par une méthode *ad hoc* sur les 24 couleurs qui restent [21]. Notons que d'après la proposition 7.8, il aurait fallut 6 rondes au lieu de 3 pour $m = 2^{2048}$ (car $\log^*(2^{2048}) = 5$).

Notons qu'en pratique, les adresses sont plutôt sur 128 bits (16 octets pour IPv6), et non 2048. De plus, si les voisins ont des adresses assignées de manière aléatoires uniformes dans $[0, m[$, la probabilité d'avoir au moins un de ses Δ voisins avec la même adresse n'est jamais que $1 - (1 - 1/m)^\Delta \approx 1 - e^{-\Delta/m}$ ce qui pour $m = 2^{128}$ et $\Delta = 2$ comme pour un cycle est d'environ $5.87 \cdot 10^{-39} \approx 0$ (d'après Mapple).

Dans [Ryb11, p. 67] il est démontré qu'il n'est pas possible de passer de 5 à 3 couleurs en une seule ronde alors qu'il est possible en une ronde de passer de 24 à 4 couleurs.

21. D'après la proposition 7.3, s'il y a m couleurs, après l'application d'un PosDiff, il restera $2\lceil \log m \rceil = 2 \cdot 2048 = 2^{12}$ couleurs. Et donc après le deuxième PosDiff (de la première ronde) il restera $2\lceil \log(2^{12}) \rceil = 24$ couleurs.

7.7.2 Réduction rapide de palette

Faisons une petite parenthèse, et généralisons l'idée de réduire plus rapidement une palette de couleurs c comme on l'a vu pour passer de 6 à 4 couleurs en une seule ronde. Ici on va supposer que d est un majorant du nombre de couleurs différentes dans un voisinage, c'est-à-dire $|c(N(u))| \leqslant d$ pour tout sommet u. En une seule ronde on espère supprimer en parallèle les t couleurs de numéro $td, td+1, \ldots, td+t-1$. Dans l'exemple précédent nous avions $d = t = 2$, et nous voulions supprimer les couleurs 4 et 5 en une ronde.

Algorithme FastReduction(d, t, c)
(code du sommet u)

1. NewRound
2. Send$(c(u), v)$ pour tout $v \in N(u)$
3. Si $c(u) \in [td, td+t[$, alors
   (a) $X := \bigcup_{v \in N(u)} \{\text{Receive}(v)\}$
   (b) Si $X \cap [td, td+t[\neq \varnothing$, alors $X := X \cup [0, (c(u) - td)d[$
   (c) Poser $c(u) :=$ FirstFree(X)

Proposition 7.9 *Pour tout graphe ayant une coloration c telle que $|c(N(u))| \leqslant d$ et tout entier $t > 1$, l'algorithme* FastReduction(d, t, c) *supprime en une seule ronde toutes les couleurs de l'intervalle $[td, td+t[$.*

Preuve. Pour tout sommet u, on note $c'(u)$ la couleur de u après l'exécution de FastReduction(d, t, c). Pour démontrer la proposition, il suffit de montrer que :

1) $c'(u) \neq c'(v)$ pour tout voisin v de u ; et
2) si $c(u) \in [td, td+t[$, alors $c'(u) < td$.

Autrement dit, que c' est une coloration et que les couleurs de l'intervalle $[td, td+t[$ ont toutes été supprimées. On distingue trois cas.

Cas 1 : $c(u)$ et $c(v) \notin [td, td+t[$. D'après l'algorithme les couleurs de u et v ne sont pas modifiées : $c'(u) = c(u)$ et $c'(v) = c(v)$. Puisque c est une coloration, $c'(u) \neq c'(v)$ et le point 1) est prouvé. Le point 2) aussi car $c'(u) \notin [td, td+t[$.

Cas 2 : $c(u) \in [td, td+t[$ et $c(v) \notin [td, td+t[$. On a donc que $c'(v) = c(v)$ et $X = c(N(u))$. Comme $c'(u) \notin X$ et $c(v) \in X$, on a donc $c'(u) \neq c'(v)$. On a aussi $c'(u) \leqslant |X| \leqslant d < td$ puisque $t > 1$. On a donc prouver les points 1) et 2) dans ces cas.

Cas 3 : $c(u)$ et $c(v) \in [td, td+t[$. On pose $c(u) = td + i$ et $c(v) = td + j$ avec $0 \leqslant i, j < t$. On a $X = c(N(u)) \cup [0, id[$. Montrons dans un premier temps que $c'(u) \in [id, id + d[$.

Par construction de X, $c'(u)$ est la plus petite couleur qui n'est ni dans $c(N(u))$ ni dans $[0, id[$. Il y a au plus d valeurs dans $c(N(u))$, donc $id + d$ pourrait être *a priori* la première couleur disponible. Cependant les d valeurs de $[id, id+d[$ pour $c(N(u))$ ne sont pas possibles car u a un voisin v qui par hypothèse a une couleur $c(v) \notin [id + d[$ puisque $c(v) = td + j > id$ car $i < t$ et $j \geqslant 0$. Ainsi une couleur de $[id, id + d[$ manque dans $c(N(u))$ et donc la plus petite qui n'est ni dans $[0, id[$ ni dans $c(N(u))$ est dans $[id, id + d[$. Donc $c'(u) \in [id, id + d[$.

Par symétrie, $c'(v) \in [jd, jd+d[$. Les intervalles $[id, id+d[$ et $[jd, jd+d[$ étant disjoints (car $c(u) \neq c(v)$ implique $i \neq j$), on a ainsi montré que $c'(u) \neq c'(v)$. Notons aussi que $c'(u) \in [id, id + d[$ implique que $c'(u) < id + d = (i + 1)d$ et donc $c'(u) < td$ puisque $i < t$. Ceci prouve les points 1) et 2) dans ce dernier cas ce qui termine la preuve. □

[En partant de l'algorithme de Linial de $O(\Delta^2)$-coloration en $\log^* n$ rondes, en déduire un algorithme en $O(\Delta \log \Delta) + \log^* n$ rondes.]

7.7.3 Borne inférieure

Dans cette partie on se pose la question de l'optimalité des algorithmes précédents. Peut-on colorier un arbre ou un cycle plus rapidement encore, en $o(\log^* n)$ rondes ? La réponse est non pour les cycles et certains arbres.

Attention ! cela signifie que pour tout algorithme de 3-coloration A, il existe des instances de cycles (ou d'arbres) pour lesquels A prends $\Omega(\log^* n)$ rondes. Ce type de résultat ne dit pas que A ne peut pas aller plus vite dans certains cas. Par exemple, si l'arbre a un diamètre plus petit que $\log^* n$ (comme une étoile), on peut évidemment faire mieux. De même, si les identités des sommets du cycles ont une distribution particulière on peut faire mieux [22]. On peut même imaginer un ensemble d'algorithmes $A_1, A_2, \ldots$ chacun permettant de faire mieux pour certaines instances. Malheureusement, pour une instance donnée, il n'y a pas vraiment de moyens de déterminer rapidement et de manière distribuée le bon algorithme A_i à appliquer.

On va démontrer précisément une borne inférieure pour les cycles, l'idée étant que pour un chemin (qui est un arbre particulier) on ne peut pas aller vraiment plus vite que pour un cycle. On réduit le problème de coloration des cycles à celui des chemins.

Proposition 7.10 *Soient $P(n)$ et $C(n)$ les complexités respectives en temps pour la 3-coloration des chemins et des cycles orientés à n sommets. Alors $C(n) - 1 \leqslant P(n) \leqslant C(n)$.*

22. Par exemple si les identités sont $0 - 1 - 2 - \cdots$, la coloration $c(u) = u \bmod 3$ sera propre et ne nécessitera aucune communication.

Preuve. L'inégalité $P(n) \leqslant C(n)$ est évidente, car l'algorithme des cycles peut toujours être appliqué à un chemin. La coloration sera propre, en modifiant éventuellement le code des deux processeurs extrémités pour qu'ils simulent la communication avec leur successeur ou prédécesseur qui n'existe pas.

Montrons maintenant que $C(n) - 1 \leqslant P(n)$. Pour cela, on imagine qu'on exécute sur un cycle un algorithme de 3-coloration d'un chemin, donc en temps $P(n)$. On considère la portion du cycle $z - x - y - t$ où l'arête xy est celle du cycle qui n'existe pas dans le chemin. On note $c(u)$ la couleur du sommet u à la fin de l'algorithme de coloration du chemin.

On effectue alors une communication supplémentaire, chaque sommet échangeant sa couleur et son identité avec ses voisins. Si un sommet u n'a pas de conflit avec ses voisins, il s'arête et ne change plus sa couleur. On remarque que les seuls sommets voisins pouvant être en conflits sont x et y. Si $c(x) = c(y)$, alors le sommet de plus petite identité (disons que c'est x) applique $\mathsf{FirstFree}(c(N(x)))$: il choisit une couleur parmi l'ensemble $\{0,1,2\} \setminus \{c(z), c(y)\}$ ($c(z)$ et $c(y)$ sont connues de x). Le seul conflit possible est ainsi éliminé. Donc $C(n) \leqslant P(n) + 1$. □

Donc, une borne inférieure sur $C(n)$ implique une borne inférieure sur celle de $P(n)$. Voici un résultat, du à Linial [Lin87][Lin92] à propos de $C(n)$. Dans la preuve originale, il s'agissait d'une 3-coloration.

Théorème 7.4 *Tout algorithme distribué de 4-coloration des cycles orientés à n sommets nécessite au moins $\frac{1}{2}(\log^* n) - 1$ rondes.*

Puisqu'une 3-coloration est aussi une 4-coloration, il suit que la borne inférieure sur le temps est également valable pour tout algorithme de 3-coloration. Notons que la borne du théorème 7.4 est très proche de celle de la proposition 7.8 en prenant $m = n$. En fait, il a été montré dans [RS15] que pour une infinité de valeurs de n la borne exacte (borne supérieure et inférieure) pour la 3-coloration des cycles orientés à n sommets était $\frac{1}{2}\log^* n$. Dans le théorème 7.4, on peut aussi considérer des cycles orientés m-coloriés et remplacer la borne sur le temps $\frac{1}{2}(\log^* n) - 1$ par $\frac{1}{2}(\log^* m) - 1$.

Le reste de ce paragraphe est consacré à la preuve du théorème 7.4 qui se déroule en trois temps : **Exécution standard**, **Graphe de voisinage**, et **Nombre chromatique**.

Exécution standard. Soit A un algorithme distribué qui dans le modèle LOCAL produit en temps t une coloration propre pour tout cycle orienté à n sommets. On note k le nombre maximum de couleurs produit par A.

On va supposer que l'exécution de A, pour chaque sommet u, se découpe en deux phases (on parle d'*exécution standard*) :

1. Une phase de communication de t rondes, où A collecte toutes les informations disponibles dans le graphe à distance $\leqslant t$ de u.
2. Une phase de calcul, sans plus aucune communication, où A doit produire le résultat final dans la mémoire locale de u.

Il n'est pas difficile de voir que tout algorithme distribué qui s'exécute en t rondes dans le modèle LOCAL possède une exécution standard, et peut en effet être découpé en une phase de communication de t rondes suivi d'une de calcul. Plus précisément, pour tout algorithme A il en existe un autre qui a une exécution standard et qui produit la même sortie après t rondes.

En effet, un sommet u ne peut recevoir d'information d'un sommet v que s'il se situe à une distance $\leqslant t$. Ensuite, tous les sommets exécutent le même algorithme, éventuellement sur une vue locale différente à cause des identifiants ou du graphe qui n'est pas forcément symétrique. Donc si v est à une distance $d \leqslant t$ de u, alors les calculs de v pendant les $p = t - d$ premières rondes de A peuvent être simulées par u, une fois toutes les informations à distance $\leqslant t$ collectées. Notons encore une fois que les calculs de v au delà des p premières étapes ne peuvent pas être communiquées à u, et donc ne sont pas utilisés par A en u.

Bien sûr, dans une exécution standard, le résultat est le même, mais la taille des messages produits pendant l'exécution n'est pas forcément la même. Des calculs intermédiaires peuvent réduire la taille des messages, alors que la collecte des informations peut produire des messages très grands. Par exemple, un sommet peut avoir à communiquer les identifiants de ses nombreux voisins à son parent. Dans le modèle LOCAL la taille des messages n'a pas d'importance.

On résume parfois le modèle LOCAL comme ceci :

> Ce qu'on peut calculer en temps t sur un graphe dans le modèle LOCAL est ce qu'on peut calculer sans communication si tous les sommets connaissent leur voisinage [23] à distance t.

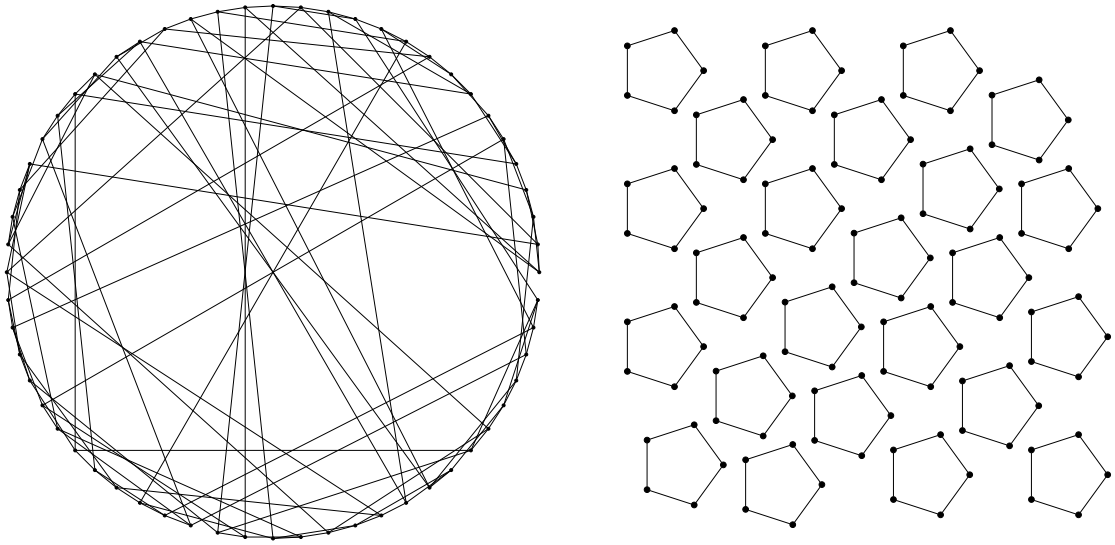
Dans une exécution standard sur des cycles orientés à n sommets, l'algorithme A en u fait donc deux choses :

1. Collecte, pendant la phase de communication, un vecteur appelé *vue* du sommet u et noté $\mathsf{v}(u)$. Il s'agit d'une suite de $2t+1$ identifiants $\mathsf{v}(u) = (x_1, \ldots, x_{t+1}, \ldots, x_{2t+1})$ où $x_{t+1} = \mathrm{ID}(u)$. Les entiers $x_1, \ldots, x_t$ sont les identifiants des t voisins de gauche de u et $x_{t+2}, \ldots, x_{2t+1}$ ceux des t voisins de droite. x_1 et x_{2t+1} sont les identifiants à distance exactement t de u.
2. Produit, pendant la phase de calcul, une couleur pour le sommet u, notée $c_A(u)$. Cette couleur ne dépend donc que de la vue de u. Autrement dit, $c_A(u) = f_A(\mathsf{v}(u))$.

23. C'est-à-dire tous les sommets et leurs entrées (généralement l'identifiants), et les l'arêtes de la boule de rayon t, exceptée les arêtes entre sommets à distance exactement t.

Graphe de voisinage. On définit le graphe $N_{t,n}$, pour t, n entiers tels que $2t+1 \leqslant n$, comme suit :

- Les sommets sont $(x_1, \ldots, x_{2t+1})$ avec $x_i \in \{0, \ldots, n-1\}$ et $x_i \neq x_j$ pour $i \neq j$.
- Les arêtes sont $(x_1, \ldots, x_{2t+1}) - (x_2, \ldots, x_{2t+1}, y)$ avec $y \neq x_1$ si $2t+1 \neq n$.



(a) (b)

FIGURE 7.7 – (a) Le graphe $N_{1,5}$ possède $60 = n \cdot (n-1) \cdots (n-2t+1)$ sommets de degré $4 = 2 \cdot (n-2t-1)$. Par exemple, dans ce graphe, les voisins du sommet 012 sont : 123, 124 et par symétrie 301, 401. On peut démontrer que $\chi(N_{1,5}) = 3$. Donc d'après la proposition 7.11, on peut théoriquement produire une 3-coloration pour un cycle à $n = 5$ sommets en une ($t = 1$) ronde. En revanche, (b) il est beaucoup plus facile de voir que le graphe $N_{2,5}$, qui a 120 sommets, vérifie clairement $\chi(N_{2,5}) = 3$.

Ce graphe est aussi appelé le *graphe de voisinage* (*neighborhood graph* en Anglais). De manière informelle, les sommets de $N_{t,n}$ sont les vues possibles d'un sommet d'un cycle collectées après la phase de communication de t rondes. On met une arête dans entre deux vues, s'il existe un cycle où les deux vues peuvent apparaître simultanément sur deux sommets voisins de ce cycle.

Ce n'est pas utile pour la suite, mais remarquons que le graphe $N_{t,n}$ est un sous-graphe induit d'un graphe de De Bruijn de paramètre n et $2t+1$. Dans un graphe de De Bruijn les lettres x_i des sommets ne sont pas forcément différentes deux à deux. Il y a donc plus de sommets, mais la règle d'adjacence est la même.

La stratégie de la preuve est la suivante. Pour borner inférieurement le temps t de A, on va chercher à borner inférieurement le nombre chromatique du graphe $N_{t,n}$. Par construction, on a :

Proposition 7.11 $\chi(N_{t,n}) \leqslant k$.

Preuve. On va colorier les sommets de $N_{t,n}$ en utilisant l'algorithme A, et donc montrer que k couleurs suffisent pour $N_{t,n}$. La couleur d'un sommet $(x_1,\dots,x_{t+1},\dots,x_{2t+1})$ de $N_{2t+1,n}$, est tout simplement $f_A(x_1,\dots,x_{t+1},\dots,x_{2t+1}) \in \{0,\dots,k-1\}$.

Il reste à montrer qu'il s'agit alors d'une coloration propre de $N_{t,n}$. Supposons qu'il existe deux sommets voisins de $N_{t,n}$ ayant la même couleur. Soient $v_1 = (x_1,\dots,x_{t+1},\dots,x_{2t+1})$ et $v_2 = (x_2,\dots,x_{t+2},\dots,x_{2t+2})$ deux sommets voisins de $N_{t,n}$ coloriés avec A et de même couleur, donc avec $f_A(v_1) = f_A(v_2)$. On considère le cycle C orienté à n sommets (voir la figure ci-dessous) où les $2t+2$ entiers des sommets v_1 et v_2 apparaissent dans l'ordre $(x_1,\dots,x_{2t+2})$ comme identifiants successifs de sommets de C. Notons qu'il est possible que $x_1 = x_{2t+1}$, si $n = 2t+1$.

$$\cdots - x_1 - x_2 - \cdots - x_t - x_{t+1} - \cdots - x_{2t+1} - x_{2t+2} - \cdots$$

Le sommet d'identité x_{t+1} dans C, disons le sommet u_1, a pour vue $\mathsf{v}(u_1) = v_1$. Le sommet d'identité x_{t+2} dans C, disons le sommet u_2, a pour vue $\mathsf{v}(u_2) = v_2$. Donc les couleurs de u_1 et u_2 dans le cycle C sont $c_A(u_1) = f_A(v_1)$ et $c_A(u_2) = f_A(v_2)$. Cela implique que $c_A(u_1) = c_A(u_2)$ ce qui contredit le fait que A calcule une coloration propre pour C. □

Comme le suggère la figure 7.7, et comme on va le voir ci-dessous, la densité [24] et le nombre chromatique de $N_{t,n}$ ont tendance à décroître lorsque t augmente. Grâce à la proposition 7.11, on cherche donc la plus petite valeur de t qui satisfait $\chi(N_{t,n}) \leqslant k$.

Une façon de paraphraser la proposition 7.11 est de dire que s'il existe un algorithme A de k-coloration en temps t pour tout cycle orienté à n sommets, alors $\chi(N_{t,n}) \leqslant k$. La contraposée nous dit donc que : si $\chi(N_{t,n}) > k$, alors il n'est existe pas d'algorithme de k-coloration en temps t pour tous les cycles orientés à n sommets.

En fait, la réciproque de la proposition 7.11 est également vraie. À savoir, si l'on peut colorier $N_{t,n}$ avec k couleurs, alors il existe un algorithme distribué qui calcule en t rondes une k-coloration pour tout cycle orienté à n sommets. On pourrait donc se servir de cet argument pour concevoir des algorithmes distribués de coloration optimaux comme étudiés dans [Ryb11].

Nombre chromatique. Calculer le nombre chromatique de $N_{t,n}$ est un problème difficile en soit. Pour conclure notre preuve on a en fait besoin que d'une minoration. On définit le graphe orienté $B_{t,n}$ comme :

- Les sommets sont $\{x_1,\dots,x_{2t+1}\}$ avec $0 \leqslant x_1 < \cdots < x_{2t+1} < n$.
- Les arcs sont $\{x_1,\dots,x_{2t+1}\} \to \{x_2,\dots,x_{2t+1},x_{2t+2}\}$ avec $x_1 < x_2$.

24. C'est-à-dire le nombre d'arêtes par rapport au nombre de sommets, soit la moitié du degré moyen.

Le graphe $B_{t,n}$ est donc l'orientation d'un sous-graphe induit de $N_{t,n}$: on ne garde dans $N_{t,n}$ que les sommets $(x_1,\dots,x_{2t+1})$ avec $x_1 < \cdots < x_{2t+1}$. Il y a beaucoup moins de sommets dans $B_{t,n}$ que dans $N_{t,n}$. En effet, $B_{t,n}$ a $\binom{n}{2t+1} \leqslant 2^{2t+1}$ sommets, le nombre de sous-ensembles $\{x_1,\dots,x_{2t+1}\}$ de taille $2t+1$ de $\{0,\dots,n-1\}$, alors que $N_{t,n}$ en possède de l'ordre de $n^{2t+1} \gg 2^{2t+1}$.

Comme on le voit sur la figure 7.8(a), le graphe $B_{1,5}$ est nettement plus petit que le graphe $N_{1,5}$ (5 sommets pour $B_{1,5}$ contre 60 pour $N_{1,5}$. Voir aussi la figure 7.7(a)). Bien évidemment, $\chi(B_{t,n}) \leqslant \chi(N_{t,n})$, l'orientation ne change en rien le nombre chromatique des graphes.

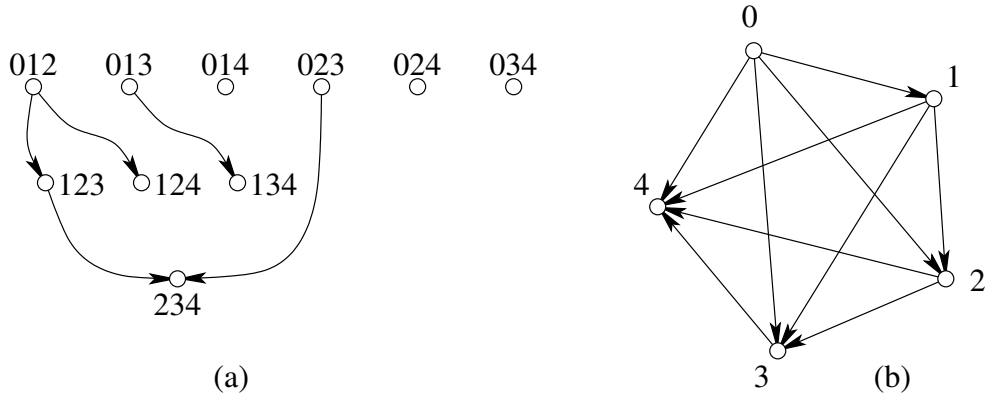
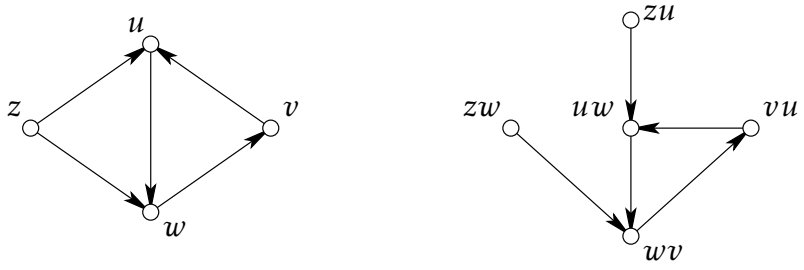


FIGURE 7.8 – (a) Le graphe $B_{1,5}$, et (b) le graphe $B_{0,5}$. On a $B_{1,5} = \mathscr{L}(\mathscr{L}(B_{0,5}))$.

Avant de démontrer la prochaine proposition, nous avons besoin de la définition suivante. Le *line graph* d'un graphe orienté G, noté $\mathscr{L}(G)$, est le graphe orienté dont les sommets sont les arcs de G, et les arcs de $\mathscr{L}(G)$ sont les paires $((u,v),(v,w))$ où (u,v) et (v,w) sont des arcs de G (voir l'exemple de la figure 7.9). Dans la suite on notera plus simplement uv un arc allant du sommet u au sommet v.



ı graphe orienté G à 4 sommets (à gauche) et son *line* ,

FIGURE 7.9 – Un graphe orienté G à 4 sommets (à gauche) et son *line graph* $\mathcal{L}(G)$ (à droite).

Proposition 7.12

i. $B_{0,n}$ *est une clique orientée à* n *sommets.*
ii. $B_{t,n}$ *est isomorphe à* $\mathscr{L}(\mathscr{L}(B_{t-1,n}))$.

Preuve.

i. Les sommets de $B_{0,n}$ sont les n entiers de $\{0,\dots,n-1\}$, et on met un arc de x à y si $x < y$. Il s'agit donc d'une clique orientée (voir la figure 7.8(a) pour un exemple avec $n = 5$).

ii. On remarque que les arcs de $B_{t-1,n}$, à savoir $\{x_1,\dots,x_{2t-1}\} \to \{x_2,\dots,x_{2t}\}$, sont en bijection avec les ensembles $\{x_1,\dots,x_{2t}\}$ qui sont les sommets de $\mathscr{L}(B_{t-1,n})$. Par exemple [25], on peut voir l'arc $013 \to 136$ comme le sommet 0136, et le sommet 1278 comme l'arc $127 \to 278$.

Ainsi, en appliquant trois fois cette bijection sur un chemin de trois arcs et quatre sommets de $B_{t-1,n}$ (voir ci-dessous), on obtient un chemin de deux arcs et trois sommets de $\mathscr{L}(B_{t-1,n})$. Et en ré-appliquant la bijection deux autres fois, on obtient un arcs et deux sommets de $\mathscr{L}(\mathscr{L}(B_{t-1,n}))$.

$$\{x_1,\dots,x_{2t-1}\} \to \{x_2,\dots,x_{2t}\} \to \{x_3,\dots,x_{2t+1}\} \to \{x_4,\dots,x_{2t+2}\}$$
$$\{x_1,\dots,x_{2t}\} \to \{x_2,\dots,x_{2t+1}\} \to \{x_3,\dots,x_{2t+2}\}$$
$$\{x_1,\dots,x_{2t+1}\} \to \{x_2,\dots,x_{2t+2}\}$$

Cette dernière bijection n'est ni plus ni moins que la définition des sommets et des arcs de $B_{t,n}$. Donc $B_{t,n}$ et $\mathscr{L}(\mathscr{L}(B_{t-1,n}))$ sont isomorphes. □

Proposition 7.13 *Pour tout graphe orienté* G, $\chi(\mathscr{L}(G)) \geqslant \log \chi(G)$.

La notation $\mathscr{L}(G)$ pour le *line graph* de G avait donc un nom prédestiné. C'est le premier endroit dans la preuve de la borne inférieure où la fonction log fait son apparition.

Preuve. On considère une k-coloration c optimale de $\mathscr{L}(G)$, c'est-à-dire avec $k = \chi(\mathscr{L}(G))$. On va montrer comment obtenir une 2^k-coloration propre de G. Cela suffit pour démontrer la proposition puisqu'alors $\chi(G) \leqslant 2^k = 2^{\chi(\mathscr{L}(G))}$ ce qui implique bien le résultat souhaité.

25. Cette correspondance ne marche pas pour $N_{t-1,n}$, car par exemple l'arête 4032 – 0324 est peut-être en bijection avec 40324, mais 40324 ne pourra jamais faire parti d'un sommet valide d'un $N_{t,n}$ (présence de deux fois la même valeur ! ce qui ne peut pas arriver si les x_i sont croissants avec i).

Notons que la k-coloration des sommets de $\mathscr{L}(G)$ revient à colorier les arcs uv de G. Pour tout sommet u de G, on pose $f(u) = \{c(vu) : vu \in E(G)\}$ c'est-à-dire l'ensemble des couleurs dans $\mathscr{L}(G)$ des arcs entrant de u. On va voir que f définit une coloration propre de G.

Montrons d'abord que $f(u)$ possède au plus 2^k valeurs distinctes. En effet, $f(u) \subseteq \{0,\dots,k-1\}$ est un sous-ensemble de couleurs de $\mathscr{L}(G)$. Et il existe au plus 2^k sous-ensemble distincts.

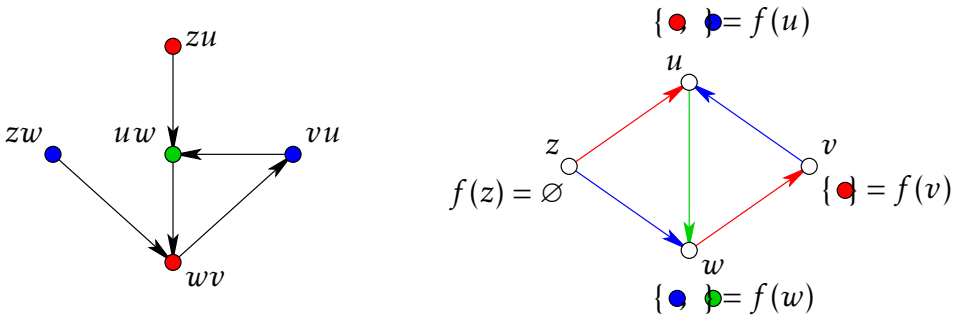


FIGURE 7.10 – Illustration de la coloration de G (à droite) par celle de $\mathscr{L}(G)$ (à gauche).

Montrons enfin que f est une coloration propre de G. Soit vu un arc de G et $c = c(vu)$ la couleur de cet arc dans $\mathscr{L}(G)$. On a $c \in f(u)$. Supposons que $c \in f(v)$. Alors il existe au moins un arc wv entrant de v de couleur $c(wv) = c = c(vu)$. Or dans $\mathscr{L}(G)$ il existe un arc entre wv et vu impliquant $c(wv) \neq c(vu)$: contradiction. Donc $c \notin f(v)$, et donc $f(u) \neq f(v)$. □

Proposition 7.14 $\chi(B_{t,n}) \geqslant \log^{(2t)} n$.

Preuve. D'après la proposition 7.12(*i*), $\chi(B_{0,n}) = n$. En combinant plusieurs fois la proposition 7.12(*ii*) et la proposition 7.13, on obtient que :

$$\begin{aligned}
\chi(B_{t,n}) &= \chi(\mathscr{L}(\mathscr{L}(B_{t-1,n}))) \geqslant \log \chi(\mathscr{L}(B_{t-1,n})) \geqslant \log\log \chi(B_{t-1,n}) \\
&\geqslant \log^{(2)} \chi(B_{t-1,n}) \\
&\geqslant \log\log\left(\log^{(2)} \chi(B_{t-2,n})\right) = \log^{(4)} \chi(B_{t-2,n}) \\
&\dots \\
&\geqslant \log^{(2t)} \chi(B_{0,n}) = \log^{(2t)} n\,.
\end{aligned}$$

□

En combinant la proposition 7.14 et la proposition 7.11, on conclut que

$$\log^{(2t)} n \leqslant \chi(B_{t,n}) \leqslant \chi(N_{t,n}) \leqslant k\,.$$

Pour $k = 4$, on déduit ainsi que le nombre t de rondes de A doit vérifier :

$$\begin{aligned} \log^{(2t)} n \leqslant 4 &\Leftrightarrow \log^{(2t)} n \leqslant 2^{2^1} \\ &\Leftrightarrow \log\log\left(\log^{(2t)} n\right) \leqslant 1 \\ &\Leftrightarrow \log^{(2t+2)} n \leqslant 1 \end{aligned}$$

ce qui implique que $2t+2 \geqslant \log^* n$ par définition de $\log^* n$. Donc le temps de l'algorithme de 4-coloration A est $t \geqslant \frac{1}{2}(\log^* n) - 1$. Cela termine la preuve du théorème 7.4.

7.7.4 Un détour par la théorie de Ramsey

Comme on l'a vu à la fin de la première étape de la borne inférieure (« exécution standard »), l'existence d'un algorithme distribué A de k-coloration en t rondes est liée à la k-coloration de chaque vue $\mathsf{v}(u) = (x_1, \ldots, x_{2t+1})$: si u et v sont adjacents dans un même cycle à n sommets, alors if faut que $f_A(\mathsf{v}(u)) \neq f_A(\mathsf{v}(v))$ où f_A est une fonction ne dépendant que de la vue (les entiers x_i).

La théorie des nombres de Ramsey peut permettre de conclure directement que le nombre de rondes nécessaires n'est pas borné. Il n'est pas seulement une fonction de k mais de n. Si n est vraiment très grand, la fonction f_A calculée par A ne permet pas de garantir une coloration propre en k couleurs. Cependant, extraire la borne inférieure en $\Omega(\log^* n)$ nécessite le calcul du nombre n en fonction de k et de t. C'est un nombre de Ramsey justement comme on va le voir. Voici l'idée générale.

Dans cette théorie, on s'intéresse à k-colorier les p-ensembles de $\{1, \ldots, n\}$, c'est-à-dire aux fonctions $f : 2^{\binom{n}{p}} \to \{1, \ldots, k\}$ associant une couleur à chaque sous-ensemble de taille p. Ici il n'y a pas de notion de voisinage ni de coloration propre, seulement la notion d'ensemble monochromatique. Un q-ensemble de $\{1, \ldots, n\}$ est *monochromatique* selon f si tous ses p-ensembles ont la même couleur. On parle aussi de multi-coloriage (k) d'hyper-arêtes (p) d'hyper-cliques (q).

Le théorème de Ramsey, la version multi-coloriée d'hyper-arêtes, établit que :

Théorème 7.5 (Ramsey (1930)) *Pour tous $k, p, q \in \mathbb{N}$ avec $p \leqslant q$, il existe un entier n, et on note $R_k(p, q)$ le plus petit d'entre eux, tel que pour toute k-coloration des p-ensembles de $\{1, \ldots, n\}$ il existe un q-ensemble de $\{1, \ldots, n\}$ monochromatique.*

Par exemple il est bien connu que $R_2(2, 3) = 6$, et on peut interpréter le résultat ainsi. Si l'on colorie les arêtes ($p = 2$) d'une clique en deux couleurs ($k = 2$), alors à partir de $n = 6$ sommets apparaît nécessairement un triangle ($q = 3$) monochromatique (cf. figure 7.11). Ou encore, tout graphe d'au moins six sommets possède une clique ou

un ensemble indépendant à trois sommets [26].

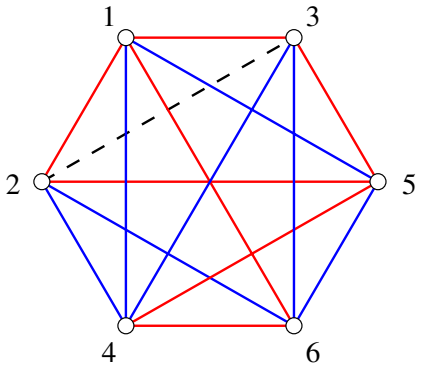


FIGURE 7.11 – Toute 2-coloration des arêtes de K_6 contient un triangle monochromatique qui est ici soit $\{1,2,3\}$ si l'arête en pointillé est rouge ou $\{2,3,4\}$ si elle est bleue, ce qui montre en passant que $K_6 \setminus \{e\}$ peut être coloré sans triangle monochromatique.

Les seules valeurs non triviales connues des nombres de Ramsey pour $k > 2$ sont $R_3(2,3) = 17$ et $R_3(3,4) = 13$. (Voir [Rad21][§7] pour un survol récent.) Traditionnellement, pour les cas $k = p = 2$, soit la 2-coloration des arêtes d'un graphe, on note les nombres de Ramsey plutôt par $R(s,t)$ représentant la taille du plus petit graphe possédant une clique à s sommets ou un stable à t sommets. La valeur exacte de $R_2(2,5) = R(5,5)$ n'est pas connue (elle est entre 43 et 48). De grands mathématiciens pensent que le calcul de $R_2(2,6) = R(6,6)$, nombre compris entre 102 et 165, est hors de portée pour l'humanité. Paul Erdős, l'un d'entre eux, pensait d'ailleurs (source Wikipédia) :

> Erdős a illustré la difficulté d'une détermination exacte en demandant d'imaginer qu'une armée extra-terrestre bien plus puissante que nous débarque et demande la valeur de $R(5,5)$, faute de quoi ils détruiront notre planète. Dans ce cas, dit-il, nous devrions mobiliser tous nos ordinateurs et tous nos mathématiciens dans l'espoir de déterminer ce nombre. Mais s'ils nous demandaient $R(6,6)$, nous ferions mieux d'essayer de détruire les extra-terrestres.

Une avancée majeure a été réalisée récemment [CGMS23] en montrant que $R(s,s) \leqslant \left(4-2^{-7}\right)^s < 3.993^s$, pour s assez grand. Cela améliore exponentiellement, et pour la première fois, la borne de 4^s d'Erdős et Szemerédi de 1935, soit après 88 ans d'efforts. La constante a d'ailleurs été optimisée par [GNNW24] en montrant que $R(s,s) \leqslant 3.7992...^{s+o(s)} < 3.8^s$ for s assez grand. Des progrès ont été aussi réalisés récemment

26. Voici une preuve que $R(3,3) = 6$. Dans toute 2-coloration des arêtes d'une clique à 6 sommets, un sommet aura au moins trois arêtes incidentes, sur les 5, de la même couleur. Mais alors, on ne pourra pas colorier les 3 arêtes entre ces 3 voisins sans créer de triangle monochrome.

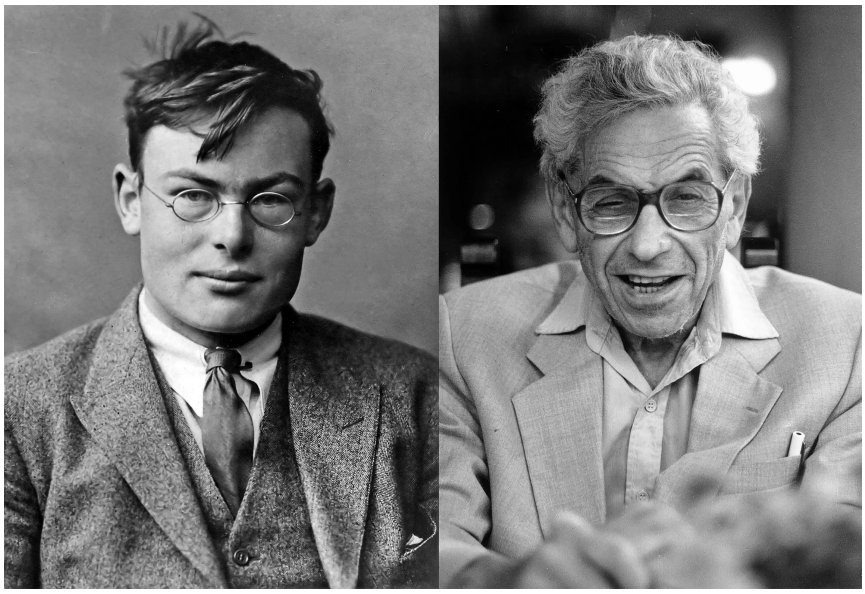


FIGURE 7.12 – Frank P. Ramsey en 1921 et Paul Erdős en 1989. Source © quantamagazine.

sur $R(4,t)$, montrant que $R(4,t) = \Omega(t^3/\log^4 t)$ (voir [MV24]) montrant finalement que $R(4,t) = t^3/\log^{O(1)} t$. Selon une conjecture d'Erdős de 1947, $R(s,t) = t^{s-1}/\log^{O(1)} t$ pour t assez grand devant $s \geqslant 2$, ce qui est donc prouvé maintenant pour $s \in \{2,3,4\}$.

On dit que les nombres de Ramsey généralisent le lemme des trous de pigeons (*Pigeon Holes Principle*). En effet, pour $p = 1$ on peut dire que si on a $n = R_k(1,q)$ pigeons dans un pigeonnier à k trous, alors q d'entre eux passeront par le même trous ($p = 1$). Ainsi on a $R_k(1,q) = k(q-1)+1$, et en particulier, $R_k(1,2) = k+1$ (voir figure 7.13 pour une illustration de $k = 9$).

FIGURE 7.13 – $R_9(1,2) = 10$: si on a 10 pigeons et neuf trous ($k = 9$), alors au moins deux pigeons ($q = 2$) sont dans le même trou ($p = 1$), ici le trou en haut à gauche. (Source Wikipédia).

Quand $p > 1$, les nombres de Ramsey ont tendance à être très grands en fonction q et k. Par exemple, les meilleures bornes supérieures ressemblent à ceci (cf. [GPS12]).

$$\begin{aligned} R_k(2,q) &\leqslant 2^{\binom{R_{k-1}(2,q-1)}{k-1}} \\ R_k(3,q) &\leqslant k \uparrow\uparrow (k(q-1)+1) = k^{\overset{\overset{k}{\cdot^{\cdot^{\cdot}}}}{k}\Big\}\, k(q-1)+1} \\ R_k(p,q) &\leqslant k \uparrow^{p-1} (k(q-1)+1) \end{aligned}$$

où $a \uparrow^n b = a \uparrow^{n-1} (a \uparrow^n (b-1))$ avec $a \uparrow^1 b = a^b$ et $a \uparrow^n 0 = 1$. C'est la notation « flèche » de Knuth. On a par exemple que $a \uparrow^2 b = a \uparrow\uparrow b$ est une pile d'exponentielles de a de hauteur b, et $a \uparrow^3 b = a \uparrow\uparrow\uparrow b = a \uparrow\uparrow (a \uparrow\uparrow (a \uparrow\uparrow (\cdots)))$ où le nombre d'opérateurs $\uparrow\uparrow$ est b. En fait, on sait depuis peu [MV24] que $R_k(4,q) = q^{2k-1}/\log^{O(1)} q$ pour $k \geqslant 2$.

Revenons à notre problème de k-coloration distribué. On peut établir une borne inférieure sur le nombre t de rondes en posant $n = R_k(2t+1, 2t+2)$. Tout algorithme distribué A de k-coloration sur les cycles à n sommets produit en particulier une k-coloration f_A de tous les p-ensembles $\{x_1, \ldots, x_{2t+1}\}$ de $\{1, \ldots, n\}$, les vues où les x_i sont rangés dans l'ordre croissant par exemple. Le théorème 7.5 nous dit donc qu'il doit exister un ensemble $\{x_1, \ldots, x_{2t+2}\}$ monochromatique selon f_A. En particulier, les ensembles $\{x_1, \ldots, x_{2t+1}\}$ et $\{x_2, \ldots, x_{2t+2}\}$ seront de la même couleur ce qui contredit le fait que A est une k-coloration du cycle où ces deux vues seraient voisines. On a donc (voir aussi la figure 7.14 pour une illustration) :

Proposition 7.15 *Aucun algorithme distribué de k-coloration ne peut colorier en t rondes tous les cycles orientés à n sommets si $n \geqslant R_k(2t+1, 2t+2)$.*

Remarquons que dans la preuve ci-dessus, on montre en fait que de nombreuses vues vont être coloriées de la même couleur (pour être précis, $\binom{q}{p} = 2t+2$), alors qu'il suffit de montrer que deux vues adjacentes le sont.

[*Exercice*. Une (a,b)-coloration pour un cycle est une coloration des sommets à l'aide de deux couleurs, disons en noir ou en blanc, de sorte qu'il y ait au plus a sommets noirs et b sommets blancs consécutifs sur le cycle. Notez qu'une 2-coloration classique n'est qu'une $(1,1)$-coloration, et qu'un ensemble indépendant maximal une $(1,2)$-coloration. Utilisez le même argument pour montrer qu'il n'est pas possible en temps t de calculer une (a,b)-coloration tous les cycles orientés à n sommets, si n est suffisamment grand (par rapport à t, a et b).]

7.7.5 En utilisant la réduction de ronde

Pour démontrer encore différemment (et plus simplement) la borne inférieure en $\log^* n$, l'idée ici est de supposer qu'il existe un algorithme A_0 de k-coloration qui utilise

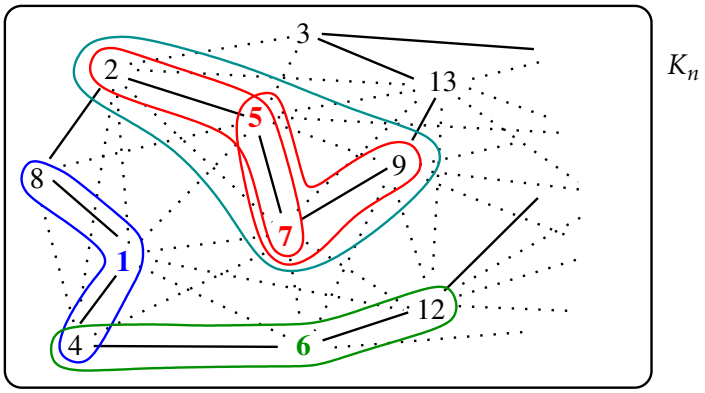


FIGURE 7.14 – Tout algorithme distribué A de 3-coloration sur les cycles à n sommets de complexité t produit une 3-coloration f_A des $(2t+1)$-uplets de $V(K_n) = \{1, \dots, n\}$. Le théorème de Ramsey indique que si $n \geqslant R_3(2t+1, 2t+2)$, alors il doit exister un $(2t+2)$-ensemble monochromatique, c'est-à-dire où chaque $(2t+1)$-ensemble a reçu la même couleur selon f_A. Dans l'exemple, $t = 1$ et le 4-ensemble monochromatique est $\{2,5,7,9\}$. Il contient les triplets $\{2,5,7\}$ et $\{5,7,9\}$ de même couleur. Cela montre qu'un cycle contenant la séquence d'identifiants $\cdots - 2 - 5 - 7 - 9 - \cdots$ sera incorrectement colorié par l'algorithme A. Malheureusement, $R_3(3,4)$ n'est pas connu. D'après [Rad21], c'est au moins 79 et au plus $3^{3^{\cdot^{\cdot^{\cdot}}}}$ (une tour de hauteur 10) d'après les majorations vues précédemment.

seulement t rondes. Puis, de montrer qu'à partir de A_0 on peut construire un nouvel algorithme A_1 qui en temps $t-1$ produit une 2^k-coloration. On en déduit donc un autre algorithme A_2 qui en temps $t-2$ produit une 2^{2^k}-coloration. Et ainsi de suite. On en déduit donc un algorithme A_t qui en temps 0 construit une c-coloration, avec $\log^{(t)} c = k$. Or ce dernier algorithme A_t ne peut exister si $n > 2c+1$, ce qui implique une borne inférieure en $\log^* n$ pour t.

À priori l'idée pour construire A_1 est de se baser sur la vue comprenant $2t-1$ ID's (au lieu de $2t+1$ pour A_0). Puis on génère toutes les vues possibles avec deux ID's de plus et s'arranger pour que les sommets d'à coté, quelque soit leur choix auront une couleur différentes. Il faut a priori utiliser la coloration du Line Graph, colorier la vue de taille $2t+1$ avec les couleurs incidentes possibles, ce qui en fait 2^k. [*Cyril*. À FINIR]

7.8 Cycles et arbres non orientés

7.8.1 Cycles non orientés

L'idée est de se ramener autant que possible à la coloration d'une 1-orientation. Pour cela chaque sommet u choisit un parent $\textsc{Parent}(u)$ parmi ses voisins, par exemple celui ayant le plus petit identifiant. Cela prend une ronde. Cependant, on n'obtient pas forcément une 1-orientation car il peut exister certaines arêtes $u-v$ qui :

(1) sont doublement orientées (car $\textsc{Parent}(u) = v$ et $\textsc{Parent}(v) = u$) ; ou

(2) ne sont pas orientées du tout (car $\textsc{Parent}(u) \neq v$ et $\textsc{Parent}(v) \neq u$).

Pour les autres arêtes, comme par exemple $u \to v \to w$, le sommet v en déduit successeur $\textsc{Succ}(v)$ (disons le voisin w) et un prédécesseur $\textsc{Pred}(v)$ comme étant l'autre voisin (ici u) et peut appliquer le double PosDiff comme dans l'algorithme ColorRing vu au paragraphe 7.7.1.

Le cas (1) n'est pas gênant, puisque les propriétés sur PosDiff, à savoir les propositions 7.1 et 7.2, restent valables dans le cas d'une double orientation (voir aussi page 146). Bien que cela ne soit donc pas nécéssaire, on peut quand même supprimer cette double orientation avec une ronde supplémentaire en décidant par exemple que le sommet de plus petit identifiant d'une arête doublement orientée perd son parent et devient une racine.

Le cas (2) ressemble typiquement à

$$\cdots u' \leftarrow u - v \to w \cdots$$

et peut être résolu par exemple avec un traitement final spécifique pour l'arête $u-v$. Il faut en effet faire en sorte que $u \neq v$ à la fin de l'algorithme (on confond ici les sommets avec leur couleur). Il suffit, par exemple, que seul le sommet de plus petit identifiant, disons v mais pas u, se re-colorie en $\mathsf{FirstFree}(\{u,w\})$. Il faut cependant être vigilant au

cas où w est lui aussi voisin d'un sommet w' de même couleur à cause d'une arête $w-w'$ non orientée. Il ne faut pas que w applique aussi FirstFree qui pourrait donner la même couleur que le FirstFree de v. Plus concrètement on pourrait avoir

$$\cdots u' \leftarrow u - v \rightarrow w - w' \rightarrow \cdots$$
$$\cdots 2 \leftarrow 1 - 1 \rightarrow 2 - 2 \rightarrow \cdots$$

où v et w voudraient se re-colorier en FirstFree$(\{1,2\}) = 0$ car tous les deux peuvent avoir un identifiant plus petit que leur voisin de même couleur qui sont respectivement u et w'. Cependant, en itérant par couleur (car finalement $v \neq w$) on peut forcer à ce que v et w n'appliquent pas FirstFree à la même ronde.

Pour résumer,

Proposition 7.16 *On peut calculer une 3-coloration en temps* $\lceil \frac{1}{2} \log^* m \rceil + 6$ *sur les cycles non orientés possédant une m-coloration.*

Preuve. Il faut une ronde pour fixer une orientation, certes incomplète, où chaque sommet u identifie Parent(u), et donc ses voisins Succ(u) et Pred(u).

On applique ensuite le double PosDiff comme dans ColorRing sans la dernière étape ReducePalette. Cela prend $\lceil \frac{1}{2} \log^* m \rceil$ rondes pour obtenir au plus 6 couleurs. Avec deux rondes supplémentaires, en utilisant FastReduction$(2, 2,$Color$)$, on obtient alors au plus 3 couleurs (cf. le paragraphe 7.7.2). Mais cela n'est pas forcément une coloration propre à cause des arêtes qui n'ont pas été orientées par la relation Parent lors de la première ronde.

Successivement pour chaque couleur $k \in \{0,1,2\}$, chaque sommet u de couleur k se re-colorie en FirstFree$(N(u))$ s'il a un voisin de même couleur ayant un identifiant plus grand. Cette règle garantit que deux voisins de même couleur k ne peuvent appliquer à la même ronde un tel FirstFree. Ils seront donc de couleurs différentes après cette ronde.

Cela prend trois rondes supplémentaires pour traiter toutes les arêtes non orientées monochromatiques $k-k$, une ronde pour chacune des trois couleurs possibles pour k. Au total l'algorithme prend $1 + \lceil \frac{1}{2} \log^* m \rceil + 2 + 3$ rondes comme annoncé. □

7.8.2 Arbres non enracinés

Dans le cas des arbres, on parle plutôt d'arbres non enracinés que d'arbres non orientés.

Pour les arbres non enracinés, il est tentant d'essayer d'appliquer la même technique que pour les cycles non orientés développée au paragraphe 7.8.1, à savoir : (1) calculer en une ronde une orientation ; (2) appliquer l'algorithme de 3-coloration pour les parties 1-orientées ; et (3) corriger les arêtes non orientées monochromatiques. La raison

fondamentale pour laquelle cela marche plutôt bien est que, lors de l'étape (3), le sous-graphe induit par une couleur k donnée forme un ensemble d'arêtes indépendantes [27] qu'il est trivial de re-colorier proprement (en l'occurence en une seule ronde grâce à un FirstFree).

Malheureusement, la situation est beaucoup moins favorable pour les arbres non enracinés car les sommets d'une couleur k donnée peuvent induire un sous-graphe quasiment aussi complexe que l'arbre de départ. [*Exercice.* Montrez que pour tout arbre T, il est possible de construire un arbre T' avec le double de sommets de sorte qu'après les étapes (1) et (2) de la méthode utilisée pour les cycles non orientés, le sous-graphe induit par une certaine couleur $k \in \{0,1,2\}$ corresponde à l'arbre T.]

Si pour le cycle la différence entre orienté ou pas n'était pas significative, il en va tout autrement pour le cas des arbres, que cela soit en termes de nombre de couleurs et de nombre de rondes.

Théorème 7.6 *Tout algorithme distribué de $\lfloor \frac{1}{2}\sqrt{d} \rfloor$-coloration pour les arbres non enracinés d-régulier [28] et de hauteur h nécessite $2h/3$ rondes.*

On ne prouvera pas ce résultat du à Linial [Lin92] qui implique qui y a une grande différence de performances pour la 3-coloration entre les graphes 1-orientés (où l'orientation est donnée et connue des sommets, cf. le théorème 7.1) et les graphes 1-orientables (où l'orientation est possible mais reste à calculer).

Corollaire 7.2 *Tout algorithme distribué de 3-coloration pour les graphes 1-orientables à n sommets nécessite $\Omega(\log n)$ rondes.*

Preuve. On considère un arbre d-régulier de hauteur h avec $d = 36$. Cet arbre, qui est 1-orientable, possède $n < 35^{h+1}$ sommets [*Exercice.* Pourquoi ?], ce qui implique $h + 1 > \log_{35} n$. Notons que $\lfloor \frac{1}{2}\sqrt{d} \rfloor = 3$. D'après le théorème 7.6, il faut au moins $2h/3 > (2/3)(\ln n / \ln 35 - 1) = \Omega(\log n)$ rondes pour tout algorithme de 3-coloration. □

Cependant, un résultat de [BE11] implique l'on peut produire, pour tous les graphes d'arboricité bornée (dont les arbres), une $(\Delta+1)$-coloration en temps $O(\log n)$.

7.8.3 Complexités dans les cycles et grilles

Des progrès récents ont été réalisés pour montrer que les bornes inférieures en $\Omega(\log^* n)$ restent vraies pour toute une classe de problèmes. Plus précisément les problèmes « localement vérifiables » ou LCL (= *Locally Checkable Labeling* en Anglais). Il

27. Dit autrement c'est un sous-graphe de degré 1, appelé aussi 1-facteur.
28. Chaque sommet qui n'est pas une feuille a exactement d voisins, et pas d fils comme dans le cas orienté. Sa hauteur est la distance maximum entre une feuille et sa racine.

s'agit de problèmes dont la sortie peut être représentée de manière distribuée par une étiquette associée à chacun des sommets (*labeling*) de sorte que la validité d'une solution peut être vérifiée en chaque sommet en examinant les étiquettes de son voisinage (sommets à une distance bornée). C'est un peu l'équivalent de la classe NP pour le distribué.

La coloration ou encore le problème de l'ensemble indépendant maximal sont typiquement des problèmes localement vérifiables. Pour la $(\Delta+1)$-coloration, l'étiquette est la couleur, et il suffit de vérifier que chaque sommet a une couleur dans $[0,\Delta]$ qui est différente de celles de ses voisins.

Le résultat assez surprenant est le suivant (cf. [BHK+16][CP17]) :

> Tout problème localement vérifiable dans un cycle à n sommets ne peut avoir que l'une des trois complexités suivantes : $\Theta(1)$, $\Theta(\log^* n)$ ou $\Theta(n)$.

En fait, le résultat reste vrai pour des graphes plus généraux, comme les grilles $n \times n$ par exemple. Pour le cycle, il est de plus possible de décider, pour chaque problème, laquelle des trois complexités est la bonne, ce qui n'est plus vrai pour une grille de dimension deux ou plus — c'est indécidable dans ce cas. Ces résultats ont été généralisés à des graphes plus généraux où d'autres complexités apparaissent, cf. [BHK+18]. Pour les arbres enracinés, par exemple, les complexités en $\Theta(n^{1/k})$ font apparition, et il est possible de calculer l'entier k [BBC+22].

Un problème très simple qui semble difficile est celui qui consiste à orienter les arêtes d'un graphe (non orienté) sans qu'il y ait de puit : *sinkless orientation* en Anglais. C'est possible de le faire pour tout graphe sauf si une composante connexe du graphe est un arbre [*Exercice*. Pourquoi ?]. C'est clairement un problème localement vérifiable. Il a été montré qu'il nécessitait $\Omega(\Delta^{-1}\log_\Delta n)$ rondes probabilistes et même $\Omega(\log_\Delta n)$ rondes déterministes dans le pire des cas d'un graphe de degré maximum Δ.

[*Cyril*. Expliquer les arguments de simulation.]

Ces résultats (voir aussi [CKP16][CHL+18]) tendent à montrer ainsi que tout problème localement vérifiable dans un graphe degré borné se décompose en un problème de coloration et en un problème que l'on peut résoudre en un nombre constant de rondes.

7.8.4 Notes bibliographiques

La table 7.1 donne un aperçu des algorithmes distribués de coloration dans le modèle LOCAL en fonction du degré maximum Δ et du nombre de sommets n du graphe. Les algorithmes [Lub86][ABI86] ont été décrits à l'origine pour le calcul d'un ensemble indépendant maximal (cf. le chapitre 8) qui peut être transformé en coloration grâce à la réduction donnée dans [Lin92].

29. En effet, selon le théorème de Brooks, pour tout graphe, une Δ-coloration existe toujours sauf s'il s'agit d'une clique ($\Delta = n-1$) ou d'un cycle impair ($\Delta = 2$).

nombre de couleurs		nombre de rondes	références
$\Delta+1$	(probabiliste)	$O(\log n)$	[Lub86][ABI86]
$O(\Delta^2)$		$O(\log^* n)$	[Lin87][Lin92]
$\Delta+1$		$\Delta^{O(\Delta)}+O(\log^* n)$	[GP87]
$\Delta+1$		$2^{O(\sqrt{\log n \log\log n})}$	[AGLP89]
$\Delta+1$		$O(\Delta \log n)$	[AGLP89]
Δ	(probabiliste)	$O((\log^3 n)/\log\Delta)$	[PS92][PS95]
$\Delta+1$		$2^{O(\sqrt{\log n})}$	[PS92][PS96]
$\Delta+1$		$O(\Delta\log\Delta)+\frac{1}{2}\log^* n$	[SV93]
$O(\Delta)$	(probabiliste)	$O(\sqrt{\log n})$	[KOSS06]
$\Delta+1$	(probabiliste)	$O(\Delta\log\log n)$	[KW06]
$\Delta t+1$	$(t\in[1,\Delta^{1-\varepsilon}])$	$O(\Delta/t)+\frac{1}{2}\log^* n$	[BE09][Kuh09][BEK14]
$\Delta t+1$	$(t\in[1,\sqrt{\Delta^{1-\varepsilon}}])$	$O(\sqrt{\Delta}/t)+\log^* n$	[Mau21]
$\Delta+1$	(probabiliste)	$O(\log\Delta+\sqrt{\log n})$	[SW10]
$\Delta^{1+o(1)}$		$\omega(\log\Delta)\log n$	[BE11]
$\Delta^{1+\varepsilon}$		$O(\log\Delta\log n)$	[BE11]
$O(\Delta)$		$O(\Delta^{\varepsilon}\log n)$	[BE11]
$\Delta+1$	(probabiliste)	$O(\log\Delta)+2^{O(\sqrt{\log\log n})}$	[BEPS12][BEPS16]
$\Delta+1$		$O(\Delta^{3/4}\log\Delta)+\frac{1}{2}\log^* n$	[Bar15]
$\Delta+1$	(probabiliste)	$O(\sqrt{\log\Delta})+2^{O(\sqrt{\log\log n})}$	[HSS16]
$\Delta+1$		$O(\sqrt{\Delta}\log^{5/2}\Delta)+\log^* n$	[FHK16]
$\Delta+1$		$O(\sqrt{\Delta\log\Delta}\cdot\log^*\Delta+\log^* n)$	[BEG18]
$(1+\varepsilon)\Delta$		$O(\sqrt{\Delta}+\log^* n)$	[BEG18]
$\Delta+1$	(probabiliste)	$O((\log\log n)^3)$	[CLP18][CLP20][RG20]
$\Delta\geqslant 4$	(probabiliste)	$O(\log\Delta)+2^{O(\sqrt{\log\log n})}$	[GHKM18]
$\Delta\geqslant 3$	(probabiliste)	$O(\sqrt{\Delta\log\Delta}\cdot\log^*\Delta\cdot(\log\log n)^2)$	[GHKM18]
$\Delta+1$		$O(\sqrt{\Delta\log\Delta})+\frac{1}{2}\log^* n$	[MT20]
$\Delta+1$		$O(\log^7 n)$	[RG20]
$\Delta+1$		$O(\log^5 n)$	[GGR21]
$\Delta+1$	(probabiliste)	$O((\log\log n)^3)$	[HKMT21]
$\Delta\geqslant 3$		$O(\log^2\Delta\cdot\log^2 n)$	[GK22]
$\Delta+1$		$O(\log^2\Delta\cdot\log n)$	[GK22][FGG+23]
$\Delta+1=\omega(\log^7 n)$	(probabiliste)	$O(\log^* n)$	[HKNT22]
$\Delta=\omega(\log^7 n)$	(probabiliste)	$O(\log^* n)$	[HNT22][FHM23]
$\Delta\geqslant 3$	(probabiliste)	$O((\log\log n)^3)$	[FHM23]
$\Delta+1$		$\log^2 n\cdot(\log\log n)^{O(1)}$	[GG23]
$\Delta+1$	(probabiliste)	$(\log\log n)^2\cdot(\log\log\log n)^{O(1)}$	[GG23]
$\Delta+1$	(probabiliste)	$(\log\log n)^{5/3}\cdot(\log\log\log n)^{O(1)}$	[GG24]
$\Delta+1$		$(\log n)^{5/3}\cdot(\log\log n)^{O(1)}$	[GG24]

TABLE 7.1 – Historique des algorithmes de coloration. Les algorithmes de Δ-coloration suppose que le graphe est ni une clique ni un cycle impair [29].

En ce qui concerne les bornes inférieures sur le temps pour la $(\Delta+1)$-coloration, la meilleure connue en fonction de n reste celle en $\Omega(\log^* n)$ pour le cycle [Lin87][Lin92]. Plusieurs preuves autres que celle de Linial existent comme [Nao91], [LS14], ou encore certaines utilisant la théorie de Ramsey [Suo14, chapitre 6]. Il a été montré dans [KW06] que tout algorithme d'une ronde appliqué à un graphe m-colorié doit produire $\Omega(\Delta^2/\log^2\Delta+\log\log m)$ couleurs. Pour produire un $O(\Delta)$-coloration tout algorithme répétant une telle réduction de couleurs nécessitera $\Omega(\Delta/\log^2\Delta+\log^* m)$ rondes. Récemment, il a été montré dans [HKMS16] que sous certaines restrictions de l'algorithme, modèle appelé SET-LOCAL où un sommet ne connaît que les classes de couleurs de son voisinage, alors le nombre de rondes pour produire une $O(\Delta)$-coloration par réduction successive de couleurs doit être $\Omega(\Delta^{1/3}+\log^* n)$.

Les derniers résultats de cette table peuvent être étendus au modèle CONGEST, c'est-à-dire avec des messages de $O(\log n)$ bits, voir [BKM20][GGR21], moyennant une dégradation des performances d'un facteur de $O(\log n)$ rondes. Cependant, le dernier, celui en $O(\log^3 n)$ rondes, est déjà dans le modèle CONGEST.

7.9 Exercices

Exercice 1

Dans l'algorithme Color6 page 138, montrez que si l'on remplace le test « Jusqu'à ce que $\ell' = \ell$ » par « Jusqu'à ce que $x \in \{0,\ldots,5\}$ », en supprimant éventuellement l'instruction (e), alors l'algorithme n'est plus correct.

Exercice 2

Donner la plus petite valeur de n pour que le temps de la borne inférieure dans le théorème 7.4 soit au moins deux ?

Exercice 3

Montrer qu'on peut passer de 6 couleurs à 4 couleurs en une seule ronde pour un cycle non orienté.

Exercice 4

On rappelle que la *maille* d'un graphe est la longueur de son plus petit cycle, et infinie dans le cas d'une forêt.

En supposant qu'il existe pour tout entier $k \geqslant 1$, et tout entier n suffisamment grand un graphe $G_{n,k}$ à n sommets, de nombre chromatique au moins k et de maille $g_k \geqslant \left\lfloor \frac{1}{4} \log_k n \right\rfloor$ (cf. [Erd59]), montrer que tout algorithme distribué de coloration d'un graphe 4-coloriable en un nombre optimal de couleur a une complexité en temps $\Omega(\log n)$.

Exercice 5

Soit G une graphe k-dégénéré. Donnez un algorithme distribué donnant une $O(k)$-coloration en temps $2^{O(k)} \log n$.

Exercice 6

Montrer que dans un graphe planaire au moins $6n/7$ sommets ont un degré au plus 6.

En déduire un algorithme en temps $O(\log n)$ calculant une 7-coloration d'un graphe planaire.

Exercice 7

Soit $C(h)$ le graphe « cylindre » de base 5 et de hauteur $h-1$. Plus précisément c'est un graphe à $5h$ sommets obtenu en faisant le produit cartésien (voir Exercice 2 pour une définition) entre un cycle à 5 sommets et un chemin à h sommets, soit encore h copies de C_5 connectés entre-eux par des chemins.

1) Montrer que ce graphe est planaire et 3-coloriable.

On considère le graphe $K(h)$ obtenu en recollant et identifiant les deux cycles extrémités d'un cylindre $C(h+1)$ et en inversant le sens de rotation des cycles (en supposant qu'initialement tous les cycles du cylindre étaient orientés de la même façon). On obtient alors un graphe plongé sur une bouteille de Klein. On notera $T(h)$ le tore obtenu en recollant et préservant le sens d'orientation des deux cycles. Notes que $C(h)$, $T(h)$ et $K(h)$ ont tous $5h$ sommets précisément.

2) En admettant que les graphes $T(h)$ et $K(h)$ sont sommets-transitifs, c'est-à-dire qu'aucun sommet ne peut être distingué d'un autre, montrer que pour tout sommet u de $T(h)$ ou $K(h)$, le graphe induit par la boule centrée en u et de rayon $\lfloor h/2 \rfloor$ est isomorphe à $C(h)$.

3) En admettant que $\chi(K(h)) = 4$, montrer que tout algorithme distribué produisant une 3-coloration pour les graphes planaires sans triangle à n sommets nécessite plus de $n/10$ rondes.

Bibliographie

[ABI86] N. Alon, L. Babai, and A. Itai, *A fast and simple randomized parallel algorithm for the maximal independent set problem*, Journal of Algorithms, 7

(1986), pp. 567–583. DOI : 10.1016/0196-6774(86)90019-2.

[AGLP89] B. Awerbuch, A. V. Goldberg, M. Luby, and S. A. Plotkin, *Network decomposition and locality in distributed computation*, in 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, May 1989, pp. 364–369. DOI : 10.1109/SFCS.1989.63504.

[Als08] B. Alspach, *The wonderful Walecki construction*, Bulletin of the Institute of Combinatorics & Its Applications, 52 (2008), pp. 7–20.

[Bar15] L. Barenboim, *Deterministic* $(\delta+1)$*-coloring in sublinear (in* δ*) time in static, dynamic and faulty networks*, in 34th Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, July 2015, pp. 345–354. DOI : 10.1145/2767386.2767410.

[BBC+22] A. Balliu, S. Brandt, Y.-J. Chang, D. Olivetti, M. Rabie, and J. Suomela, *Efficient classification of locally checkable problems in regular trees*, in 38th International Symposium on Distributed Computing (DISC), vol. 246 of Lecture Notes in Computer Science (ARCoSS), Springer, October 2022, pp. 8 :1–8 :19. DOI : 10.4230/LIPIcs.DISC.2022.8.

[BE08] L. Barenboim and M. Elkin, *Sublogarithmic distributed MIS algorithm for sparse graphs using Nash-Williams decomposition*, in 27th Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, August 2008, pp. 25–34. DOI : 10.1145/1400751.1400757.

[BE09] L. Barenboim and M. Elkin, *Distributed* $(\delta+1)$*-coloring in linear (in* δ*) time*, in 41st Annual ACM Symposium on Theory of Computing (STOC), ACM Press, May 2009, pp. 111–120. DOI : 10.1145/1536414.1536432.

[BE11] L. Barenboim and M. Elkin, *Distributed deterministic vertex coloring in polylogarithmic time*, Journal of the ACM, 58 (2011), pp. Article No. 23, pp. 1–25. DOI : 10.1145/2027216.2027221.

[BEG18] L. Barenboim, M. Elkin, and U. Goldenberg, *Locally-iterative distributed* $\delta+1$*-coloring below Szegedy-Vishwanathan barrier, and applications to self-stabilization and to restricted-bandwidth models*, in 37th Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, July 2018, pp. 437–446. DOI : 10.1145/3212734.3212769.

[BEK14] L. Barenboim, M. Elkin, and F. Kuhn, *Distributed* $(\delta+1)$*-coloring in linear (in* δ*) time*, SIAM Journal on Computing, 43 (2014), pp. 72–95. DOI : 10.1137/12088848X.

[BEPS12] L. Barenboim, M. Elkin, S. Pettie, and J. Schneider, *The locality of distributed symmetry breaking*, in 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, October 2012, pp. 321–330. DOI : 10.1109/FOCS.2012.60.

[BEPS16] L. Barenboim, M. Elkin, S. Pettie, and J. Schneider, *The locality of distributed symmetry breaking*, Journal of the ACM, 63 (2016), pp. Article No. 20, pp. 1–45. DOI : 10.1145/2903137.

[BHK+16] S. Brandt, J. Hirvonen, J. H. Korhonen, T. Lempiäinen, P. R. Östergård, C. Purcell, J. Rybicki, J. Suomela, and P. Uznański, *LCL problems on grids*, in 35th Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, July 2016, pp. 101–110. doi : 10.1145/3087801.3087833.

[BHK+18] A. Balliu, J. Hirvonen, J. H. Korhonen, T. Lempiäinen, D. Olivetti, and J. Suomela, *New classes of distributed time complexity*, in 50th Annual ACM Symposium on Theory of Computing (STOC), ACM Press, June 2018, pp. 1307–1318. doi : 10.1145/3188745.3188860.

[BKM20] P. Bamberger, F. Kuhn, and Y. Maus, *Efficient deterministic distributed coloring with small bandwidth*, in 39th Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, August 2020, pp. 243–252. doi : 10.1145/3382734.3404504.

[CGMS23] M. Campos, S. Griffiths, R. Morris, and J. Sahasrabudhe, *An exponential improvement for diagonal Ramsey*, Tech. Rep. 2303.09521 [math.CO], arXiv, March 2023.

[CHL+18] Y.-J. Chang, Q. He, W. Li, S. Pettie, and J. Uitto, *The complexity of distributed edge coloring with small palettes*, in 29th Symposium on Discrete Algorithms (SODA), ACM-SIAM, January 2018, pp. 2633–2652. doi : 10.1137/1.9781611975031.168.

[CKP16] Y.-J. Chang, T. Kopelowitz, and S. Pettie, *An exponential separation between randomized and deterministic complexity in the LOCAL model*, in 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, October 2016, pp. 615–624. doi : 10.1109/FOCS.2016.72.

[CLP18] Y.-J. Chang, W. Li, and S. Pettie, *An optimal distributed* $(\delta+1)$*-coloring algorithm ?*, in 50th Annual ACM Symposium on Theory of Computing (STOC), ACM Press, June 2018, pp. 445–456. doi : 10.1145/3188745.3188964.

[CLP20] Y.-J. Chang, W. Li, and S. Pettie, *Distributed* $(\delta + 1)$*-coloring via ultrafast graph shattering*, SIAM Journal on Computing, 49 (2020), pp. 497–539. doi : 10.1137/19M1249527.

[CP17] Y.-J. Chang and S. Pettie, *A time hierarchy theorem for the LOCAL model*, in 58th Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, October 2017, pp. 156–167. doi : 10.1109/FOCS.2017.23.

[CV86] R. Cole and U. Vishkin, *Deterministic coin tossing and accelerating cascades : micro and macro techniques for designing parallel algorithms*, in 18th Annual ACM Symposium on Theory of Computing (STOC), ACM Press, 1986, pp. 206–219. doi : 10.1145/12130.12151.

[Erd59] P. Erdős, *Graph theory and probability*, Canadian Journal of Mathematics, 11 (1959), pp. 34–38. doi : 10.4153/CJM-1959-003-9.

[FGG+23] S. Faour, M. Ghaffari, C. Grunau, F. Kuhn, and V. Rozhoň, *Local distributed rounding : Generalized to MIS, matching, set cover, and beyond*, in 34th Symposium on Discrete Algorithms (SODA), ACM-SIAM, January 2023, pp. 4409–4447. doi : 10.1137/1.9781611977554.ch168.

[FHK16] P. Fraigniaud, M. Heinrich, and A. Kosowski, *Local conflict coloring*, in 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, October 2016, pp. 625–634. doi : 10.1109/FOCS.2016.73.

[FHM23] M. Fischer, M. M. Halldórsson, and Y. Maus, *Fast distributed Brooks' theorem*, in 34th Symposium on Discrete Algorithms (SODA), ACM-SIAM, January 2023, pp. 2567–2588. doi : 10.1137/1.9781611977554.ch98.

[GG23] M. Ghaffari and C. Grunau, *Faster deterministic distributed MIS and approximate matching*, in 55th Annual ACM Symposium on Theory of Computing (STOC), ACM Press, June 2023, pp. 1777–1790. doi : 10.1145/3564246.3585243.

[GG24] M. Ghaffari and C. Gruneau, *Near-optimal deterministic network decomposition and ruling set, and improved MIS*, Tech. Rep. 2410.1951v1 [cs.DS], arXiv, October 2024.

[GGR21] M. Ghaffari, C. Grunau, and V. Rozhoň, *Improved deterministic network decomposition*, in 32nd Symposium on Discrete Algorithms (SODA), ACM-SIAM, January 2021, pp. 2904–2923. doi : 10.1137/1.9781611976465.173.

[GHKM18] M. Ghaffari, J. Hirvonen, F. Kuhn, and Y. Maus, *Improved distributed delta-coloring*, in 37th Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, July 2018, pp. 427–436. doi : 10.1145/3212734.3212764.

[GK22] M. Ghaffari and F. Kuhn, *Deterministic distributed vertex coloring : Simpler, faster, and without network decomposition*, in 62nd Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, February 2022, pp. 1009–1020. doi : 10.1109/FOCS52979.2021.00101.

[GKP94] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics, a foundation for computer science*, Addison-Wesley, 1994.

[GNNW24] P. Gupta, N. Ndiaye, S. Norin, and L. Wei, *Optimizing the CGMS upper bound on Ramsey numbers*, Tech. Rep. 2407.19026v1 [math.CO], arXiv, July 2024.

[GP87] A. V. Goldberg and S. A. Plotkin, *Parallel* $(\delta + 1)$*-coloring of constant-degree graphs*, Information Processing Letters, 25 (1987), pp. 241–245. doi : 10.1016/0020-0190(87)90169-4.

[GPS12] W. I. Gasarch, A. Parrish, and S. Sinai, *Three proofs of the hypergraph Ramsey theorem (an exposition)*, 2012.

[HKMS16] D. Hefetz, F. Kuhn, Y. Maus, and A. Steger, *Polynomial lower bound for distributed graph coloring in a weak LOCAL model*, in 30th International Symposium on Distributed Computing (DISC), vol. 9888 of Lecture Notes in Computer Science (ARCoSS), Springer, September 2016, pp. 99–113. doi : 10.1007/978-3-662-53426-7_8.

[HKMT21] M. M. Halldórsson, F. Kuhn, Y. Maus, and T. Tonoyan, *Efficient randomized distributed coloring in CONGEST*, in 53rd Annual ACM Symposium on Theory of Computing (STOC), ACM Press, June 2021, pp. 1180–1193. doi : 10.1145/3406325.3451089.

[HKNT22] M. M. Halldórsson, F. Kuhn, A. Nolin, and T. Tonoyan, *Near-optimal distributed degree+1 coloring*, in 54th Annual ACM Symposium on Theory of Computing (STOC), ACM Press, June 2022, pp. 450–463. doi : 10.1145/3519935.3520023.

[HNT22] M. M. Halldórsson, A. Nolin, and T. Tonoyan, *Overcoming congestion in distributed coloring*, in 41st Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, July 2022, pp. 26–36. doi : 10.1145/3519270.3538438.

[HSS16] D. G. Harris, J. Schneider, and H.-H. Su, *Distributed* $(\Delta+1)$*-coloring in sublogarithmic rounds*, in 48th Annual ACM Symposium on Theory of Computing (STOC), ACM Press, June 2016, pp. 465–478. doi : 10.1145/2897518.2897533.

[KOSS06] K. Kothapalli, M. Onus, C. Scheideler, and C. Schindelhauer, *Distributed coloring in* $\tilde{O}(\sqrt{\log n})$ *bit rounds*, in 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS), IEEE Computer Society Press, April 2006. doi : 10.1109/IPDPS.2006.1639281.

[KSV11] A. Korman, J.-S. Sereni, and L. Viennot, *Toward more localized local algorithms : removing assumptions concerning global knowledge*, in 30th Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, June 2011, pp. 49–58. doi : 10.1145/1993806.1993814.

[Kuh09] F. Kuhn, *Weak graph colorings : distributed algorithms and applications*, in 21st Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), ACM Press, August 2009, pp. 138–144. doi : 10.1145/1583991.1584032.

[KW06] F. Kuhn and R. Wattenhofer, *On the complexity of distributed graph coloring*, in 25th Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, July 2006, pp. 7–15. doi : 10.1145/1146381.1146387.

[Lin87] N. Linial, *Distributive graph algorithms – Global solutions from local data*, in 28th Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, October 1987, pp. 331–335. doi : 10.1109/SFCS.1987.20.

[Lin92] N. Linial, *Locality in distributed graphs algorithms*, SIAM Journal on Computing, 21 (1992), pp. 193–201. doi : 10.1137/0221015.

[LS14] J. Laurinharju and J. Suomela, *Brief announcement : Linial's lower bound made easy*, in 33rd Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, July 2014, pp. 377–378. doi : 10.1145/2611462.2611505.

[Lub86] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM Journal on Computing, 15 (1986), pp. 1036–1053. doi : 10.1137/0215074.

[Mau21] Y. Maus, *Distributed graph coloring made easy*, in 33rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), ACM Press, July 2021, pp. 362–372. doi : 10.1145/3409964.346180.

[MP23] A. Miller and A. Pelc, *Fast deterministic rendezvous in labeled lines*, in 37th International Symposium on Distributed Computing (DISC), R. Oshman, ed., vol. 281 of LIPIcs, October 2023, pp. 29 :1–29 :22. doi : 10.4230/LIPIcs.DISC.2023.29.

[MT20] Y. Maus and T. Tonoyan, *Local conflict coloring revisited : Linial for lists*, in 34th International Symposium on Distributed Computing (DISC), H. Attiya, ed., vol. 179 of LIPIcs, October 2020, pp. 16 :1–16 :18. doi : 10.4230/LIPIcs.DISC.2020.16.

[MV24] S. Mattheus and J. Verstraëte, *The asymptotics of $r(4,t)$*, Tech. Rep. 2306.04007v5 [math.CO], arXiv, February 2024.

[Nao91] M. Naor, *A lower bound on probabilistic algorithms for distributive ring coloring*, SIAM Journal on Discrete Mathematics, 4 (1991), pp. 409–412. doi : 10.1137/0404036.

[PS92] A. Panconesi and A. Srinivasan, *Improved distributed algorithms for coloring and network decomposition problems*, in 24th Annual ACM Symposium on Theory of Computing (STOC), ACM Press, 1992, pp. 581–592. doi : 10.1145/129712.129769.

[PS95] A. Panconesi and A. Srinivasan, *Improved distributed algorithms for coloring and network decomposition problems*, Combinatorica, 15 (1995), pp. 255–280. doi : 10.1007/BF01200759.

[PS96] A. Panconesi and A. Srinivasan, *On the complexity of distributed network decomposition*, Journal of Algorithms, 20 (1996), pp. 356–374. doi : 10.1006/jagm.1996.0017.

[Rad21] S. Radziszowski, *Small Ramsey numbers*, The Electronic Journal of Combinatorics, Dynamic Survey (2021). doi : 10.37236/21.

[RG20] V. Rozhoň and M. Ghaffari, *Polylogarithmic-time deterministic network decomposition and distributed derandomization*, in 52nd Annual ACM Symposium on Theory of Computing (STOC), ACM Press, June 2020, pp. 350–363. doi : 10.1145/3357713.3384298.

[RS15] J. Rybicki and J. Suomela, *Exact bounds for distributed graph colouring*, in 22nd International Colloquium on Structural Information & Communication Complexity (SIROCCO), vol. 9439 of Lecture Notes in Computer Science, Springer, July 2015, pp. 46–60. doi : 10.1007/978-3-319-25258-2_4.

[Ryb11] J. Rybicki, *Exact bounds for distributed graph colouring*, PhD thesis, University of Helsinki, Dept. of Computer Science, May 2011.

[Suo13] J. Suomela, *Survey of local algorithms*, ACM Computing Surveys, 45 (2013), pp. Article No. 24, pp. 1–40. doi : 10.1145/2431211.2431223.

[Suo14] J. Suomela, *A Course on Deterministic Distributed Algorithms*, Online textbook, 2014. http://users.ics.aalto.fi/suomela/dda/.

[SV93] M. Szegedy and S. Vishwanathan, *Locality based graph coloring*, in 25th Annual ACM Symposium on Theory of Computing (STOC), ACM Press, May 1993, pp. 201–207. doi : 10.1145/167088.167156.

[SW10] J. Schneider and R. Wattenhofer, *A new technique for distributed symmetry breaking*, in 29th Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, July 2010, pp. 257–266. doi : 10.1145/1835698.1835760.

CHAPITRE 8

Ensembles indépendants maximaux

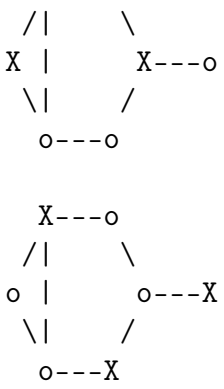
Sommaire

[*Cyril*. Chapitre à développer ...]

8.1 Introduction

- exemple : maximal et maximum

o---o



- parler de : borne inf en $\Omega(\sqrt{\log n / \log\log n})$ de [KMW04].

- parler de l'algo de Luby, différence entre Las Vega et Monté Carlo. Amélioration récente de Mohsen Ghaffari [Gha16].

- algo en $O(\sqrt{\log n})$ d'Elkin pour les planaires.

Un *ensemble indépendant maximal* dans un graphe G (MIS en anglais, pour *Maximal Independent Set*) est un sous-ensemble de sommets M de G tel que : 1) si $x, y \in M$, alors x et y ne sont pas adjacents (M est donc un ensemble indépendant) ; et 2) si l'on ajoute un nouveau sommet à M, alors M n'est plus un ensemble indépendant (il est donc maximal pour l'inclusion).

[EXEMPLE]

Il faut noter que les deux conditions précédentes peuvent être vérifiées localement. Pour l'indépendance, il faut vérifier que tout sommet x dans M n'a aucun voisin dans M, et 2) pour la maximalité, si $x \notin M$, alors il doit avoir au moins un voisin dans M puisque sinon on pourrait rajouter x à M.

Remarques. 1) Un MIS est un ensemble dominant particulier (c'est-à-dire tout $x \notin M$ a un voisin $y \in M$). 2) Dans une coloration (propre) les sommets d'une même couleur forme un ensemble indépendant (mais pas forcément maximal). Coloration, ensemble dominant et MIS sont donc des notions proches et, comme nous allons le voir, il existe des moyens de calculer l'un à partir de l'autre.

L'utilisation des MIS en algorithmique distribuée est importante. On utilise les MIS pour créer du parallélisme. Il est souvent important que des ensembles de sommets puissent calculer indépendamment les uns des autres. Typiquement, pour le calcul d'arbre couvrant de poids minimum (MST), dans l'implémentation de Kruskal, il est assez intéressant que chaque sous arbre (initialement réduit à un sommet) puisse "pousser" indépendamment. Pour ce faire les racines de ces sommets initiateurs doivent être indépendantes pour ne pas se gêner.

8.2 Algorithmes de base

En séquentiel, on utilise un algorithme glouton : tant qu'il existe un voisin u où tout ses voisins ne sont pas dans l'ensemble, on l'ajoute. Calculer un MIS est donc très simple, de même que le calcul d'un ensemble dominant ou d'une coloration. Par contre, si l'on souhaite un MIS de cardinalité maximum (bien faire la différence entre maximal et maximum) le problème devient difficile (NP-complet pour la décision) et reste difficile à approximer. C'est la même chose pour trouver un ensemble dominant de petite taille où une coloration avec peu de couleurs.

En distribué l'approche consiste à calculer une variable b représentant la décision de joindre ($b = 1$) ou de ne pas joindre ($b = 0$) l'ensemble MIS. Initialement $b = -1$.

Il y a deux façons de faire : MIS_{dfs} et MIS_{rank}. Dans le pire des cas, ils ont la même complexité, mais le second à tendance à être plus rapide dans la pratique. Cependant, MIS_{rank} pré-suppose des identité unique, alors que MIS_{dfs} non.

Dans un chemin, MIS_{rank} peut aller aussi lentement que MIS_{dfs}. Cependant, on peut montrer qu'avec grande probabilité, une permutation aléatoire dans un chemin fait que MIS_{rank} calculera un MIS en temps $O(\log n/\log\log n)$, au lieu de n dans le cas du MIS_{dfs}.

8.3 Réductions

Entre MIS et coloration. Si on sait calculer une k-coloration pour G en t rondes, alors on sait calculer un MIS pour G en $t + k$ rondes :

1. on calcule une k-coloration en temps t ;
2. on pose $b := -1$ pour tout sommet u ;
3. pour $i := 0$ à $k - 1$, chaque sommet u de couleur i essaye de s'ajouter au MIS courant. Pour cela u envoie et récupère la valeur b de ses voisins. Si aucune valeur n'est à 1, il pose $b := 1$ sinon il pose $b := 0$.

Ça marche essentiellement parce qu'à cause de la coloration, deux sommets voisins ne peuvent pas essayer de s'ajouter au MIS en même temps.

Il existe une réduction inverse : si l'on sait calculer un MIS en temps $t(n, k)$ pour tout graphe à n sommets de degré maximum k, alors pour tout graphe G de degré maximum Δ on peut calculer une $(\Delta + 1)$-coloration en temps $t(n, k \cdot \Delta)$. ...

8.4 Ensemble dominants

Définition 8.1 *Un ensemble S est dominant pour un graphe G si pour tout sommet $x \in V(G)$, soit $x \in S$, ou x a un voisin $y \in S$.*

Rappelons qu'un MIS est un ensemble dominant, mais pas l'inverse !

Ici on s'intéresse aux ensemble dominants de petite cardinalité pour des motivations liés au routage (notion de sommets spéciaux et peu nombreux qui contrôlent tous les autres).

Prop : tout graphe connexe possède un ensemble dominant de taille $\leqslant n/2$, si $n > 1$.

Arbre couvrant + partition en deux couleurs suivant la parité de la distance à la racine.

On s'intéresse à de petits ensemble dominants (de taille $\leqslant n/2$) pour les arbres enracinés (on suppose donc que chacun connaît son parent).

...

8.5 Exercices

Exercice 1

On définit l'eccentricité d'un graphe G comme $\mathrm{ecc}(G) = \max_u \mathrm{ecc}_G(u)$. Construire un graphe G_n et H_n à n sommets tels que $\mathrm{ecc}(G_n) = \mathrm{diam}(G_n)$ et $\mathrm{ecc}(H_n) = \frac{1}{2}\mathrm{diam}(H_n)$.

Soit $S_0 = \{v : \deg(v) \leqslant 12\}$. Montrer que si G est planaire ($m \leqslant 3n-6$), alors $|S_0| > n/2$. En déduire un algorithme distribué pour calculer un MIS sur tout graphe planaire à n sommets en temps $O(\log n \cdot \log^* n)$.

Exercice 2

Donner un algorithme distribué pour calculer efficacement un MIS dans un graphe bipartie complet.

Bibliographie

[Gha16] M. Ghaffari, *An improved distributed algorithm for maximal independent set*, in 27th Symposium on Discrete Algorithms (SODA), ACM-SIAM, January 2016, pp. 270–277. doi : 10.1137/1.9781611974331.ch20.

[KMW04] F. Kuhn, T. Moscibroda, and R. Wattenhofer, *What cannot be computed locally !*, in 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, July 2004, pp. 300–309. doi : 10.1145/1011767.1011811.

CHAPITRE

9 Graphes couvrant éparses

Sommaire

Nous allons introduire le calcul de *spanners* dans le modèle LOCAL. Les *spanners* sont des sous-graphes couvrant les sommets d'un graphe possédant une faible densité d'arêtes. De plus ils garantissent que les distances dans les sous-graphes ne sont pas trop éloignées des distances originales.

9.1 Introduction

L'utilisation de sous-graphe couvrant, c'est-à-dire qui contient tous les sommets du graphe d'origine, est banale en informatique. Typiquement un arbre couvrant un graphe connexe G à n sommets garantit la connexité alors qu'il ne possède que $n-1$ arêtes. Cela peut être particulièrement intéressant en calcul distribué de minimiser de nombre de liens sans pour autant déconnecter G.

On a vu dans les chapitres précédents que le nombre de messages nécessaire à résoudre certaines tâches peut être réduit si un arbre préexistant était connu des processeurs. Au chapitre 6, on a vu que la connaissance d'un sous-graphe couvrant peu dense pouvait être utilisé pour construire des synchroniseurs.

De manière générale, on souhaite construire un graphe couvrant peu dense préservant les distances d'origines. Bien sûr il n'est pas possible de supprimer des arêtes sans modifier au moins une distance. Néanmoins on souhaite limiter ces modifications. La définition suivante s'applique aussi bien au cas des graphes valués que non valués.

Définition 9.1 *Soient G un graphe et H un sous-graphe couvrant de G, c'est-à-dire avec $V(G) = V(H)$. L'*étirement *de H est la plus petite valeur s telle que, pour tout sommets u, v de G, $d_H(u,v) \leqslant s \cdot d_G(u,v)$.*

EXEMPLES

Dans la littérature on appelle aussi *s-spanner* tout sous-graphe couvrant d'étirement s (*stretch* en Anglais).

Dans la suite, on va essentiellement considérer que les graphes sont non valués.

Le problème qu'on se fixe est donc de calculer de manière distribué, dans le modèle LOCAL, des sous-graphes couvrant avec peu d'arêtes et si possible un étirement faible. On se base essentiellement sur l'article [DGPV08].

Formellement, calculer un sous-graphe H signifie que chaque sommet u doit sélectionner un ensemble d'arêtes qui lui sont incidentes, disons $H(u)$. Le graphe calculé est alors le graphe $\bigcup_{u \in V(G)} H(u)$ composé de l'union de tous ces choix.

Notons qu'il n'est pas formellement imposé que les deux sommets extrémités d'une arête s'accordent sur leur décision, c'est-à-dire $uv \in H(u)$ n'implique pas forcément $uv \in H(v)$. Cependant, avec une ronde supplémentaire, on peut toujours faire en sorte que tous les sommets se mettent d'accord (par exemple sur le fait de garder l'arête si l'un des deux l'a sélectionnée, ou le contraire, de ne pas la garder si l'un des deux ne l'a pas garder).

9.2 Calcul d'un 3-spanner

On va voir un algorithme permettant de calculer un sous-graphe couvrant d'étirement 3 avec significativement moins de n^2 arêtes, en fait $2n^{1.5}$. Il suffit de seulement deux rondes. La décision de sélectionner ou non les arêtes est donc locale.

Avant de présenter l'algorithme, remarquons que le problème n'a rien d'évident. Décider localement de supprimer une arête est *a priori* très risqué, puisque par exemple un sommet n'a aucun moyen de deviner s'il est dans un arbre ou pas (ils ne connaissant pas, par exemple, le nombre d'arêtes du graphe et le degré d'un sommet peut être non borné, ou les cycles passant par le sommet trop grand pour être détecté en deux rondes). Bien sûr pour avoir un l'étirement borné, aucun sommet d'un arbre ne peut supprimer d'arête. D'un autre coté, si aucun sommet de degré élevé ne supprime d'arête on pourrait avoir de l'ordre de n^2 arêtes si le graphe G était très dense.

- présenter l'algo de BKMP05 (slides/Aladdin-PODC08 ...)

Théorème 9.1 ([BKMP05]) *Pour tout graphe G à n sommets, il existe un algorithme probabiliste de type Monté Carlo qui calcule en deux rondes un sous-graphe couvrant de G d'étirement $\leqslant 3$ et en moyenne au plus $2n^{3/2}$ arêtes.*

Preuve.

1. $b_u := 1|0$ avec probabilité $1/\sqrt{n}$, $X := \{u : b_u = 1\}$
2. $[u \in Z]$ Si $B(u,1) \cap X = \varnothing$, alors $S_u := B(u,1)$
3. $[u \in X]$ Si $b_u = 1$, alors $S_u := \mathrm{BFS}(u, B(u,2))$

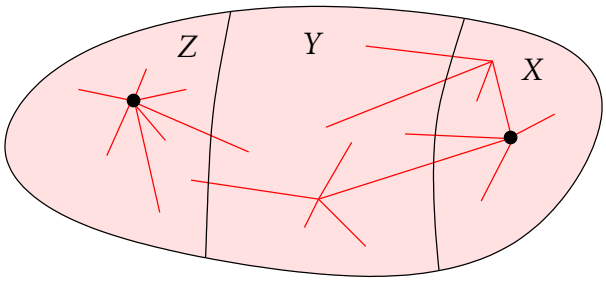


FIGURE 9.1 – Illustration de l'algorithme de type Monté Carlo pour le calcul d'un sous-graphe couvrant d'étirement $\leqslant 3$.

Complexité en temps : 2 rondes

Etirement : 3 (les seules arêtes e dans G et pas dans H ont leur deux extrémités dans Y)

Taille : en moyenne, $|X| = \sqrt{n}$ et si $u \in Z$, alors $\deg(u) \leqslant \sqrt{n}$ en moyenne.

□

Jusqu'à récemment, aucun algorithme déterministe n'était connu pour ce problème. Ils existent des algorithmes en deux rondes, mais qui ne garantissent le nombre d'arêtes seulement en moyenne [BKMP05]. Le problème des algorithmes aléatoires de ce type (Monté Carlo) est qu'on est pas certain du résultat, contrairement aux algorithmes de type Las Vegas. Et contrairement au calcul séquentiel, en distribué il n'est pas toujours facile de vérifier la condition (comment vérifier que le nombre d'arêtes est correct sans une diffusion globale ?). En général, en distribué, on ne peut jamais recommencer !

Dans la suite, on rappelle que $B(u,1)$ représente l'ensemble formé de u et des voisins de u, la boule centrée en u et de rayon 1. Évidemment, $|B(u,1)| = \deg(u) + 1$.

Algorithme Span3
(code du sommet u)

1. NewRound
2. Envoyer son identifiant et recevoir l'identifiant de tous ses voisins.
3. Choisir un ensemble R_u composé de son identifiant et d'un sous-ensemble arbitraire des identifiants reçus tel que $|R_u| = \min\left\{|B(u,1)|, \left\lceil \sqrt{n} \right\rceil\right\}$.
4. NewRound

5. Envoyer R_u et recevoir la sélection R_v de chaque voisin v.
6. Poser $H(u) := \{uv : v \in R_u, v \neq u\}$ et $W := B(u,1) \setminus \{v : u \in R_v\}$.
7. Tant qu'il existe $w \in W$:
   (a) $H(u) := \{uw\} \cup H(u)$.
   (b) $W := W \setminus \{v \in W : R_v \cap R_w \neq \varnothing\}$.

Il y a une variante où u ne connaît pas le nombre $\lceil\sqrt{n}\rceil$, mais qui nécessite une ronde de plus. L'algorithme peut aussi être adapté pour fonctionner avec des poids arbitraires sur les arêtes : il faut choisir R_u parmi ses plus proches voisins et le sommet w parmi les plus proches restant de W.

Théorème 9.2 *Pour tout graphe G à n sommets, l'algorithme* Span3 *calcule en deux rondes un sous-graphe couvrant de G d'étirement $\leqslant 3$ et avec moins de $2n^{3/2}$ arêtes.*

Preuve. L'algorithme comprends deux rondes de communications.

Étirement. Soit uv une arête de G qui n'est pas dans le sous-graphe couvrant $H = \bigcup_u H(u)$ (figure 9.2(a)). Clairement toutes les arêtes de la forme uw avec $w \neq u$, $u \in R_u \cup R_w$ et $w \in B(u,1)$ existent dans H. Donc si cette arête a été supprimée c'est que v a été supprimé de W à l'étape 7b de l'algorithme (figure 9.2(b)). Si v est supprimé de W à l'étape 7b, c'est que R_v et R_w s'intersectent (figure 9.2(c)). Or dans H toutes les arêtes de v vers R_v et de w vers R_w existent. Donc il existe dans H un chemin de u à v de longueur trois.

Il suit qu'un plus court chemin de x à y dans H est de longueur au plus trois celle d'un plus court chemin P de G, chaque arête de P qui n'est pas dans H étant remplacée par un chemin de H d'au plus trois arêtes.

Nombre d'arêtes. Montrons que tout sommet w de l'ensemble W calculé à l'étape 6 est tel que $|B(w,1)| > \lceil\sqrt{n}\rceil$. En effet, si $|B(w,1)| \leqslant \lceil\sqrt{n}\rceil$, alors $|R_w| = \min\left\{|B(w,1)|, \lceil\sqrt{n}\rceil\right\} = |B(w,1)|$. Donc R_w contiendrait tous les voisins de w, en particulier u. Or ce n'est pas le cas, donc $|B(w,1)| > \lceil\sqrt{n}\rceil$. Il suit que $|R_w| = \lceil\sqrt{n}\rceil$.

Les sommets sélectionnés dans la boucle « Tant que » ont, par construction, des ensembles deux à deux disjoints, c'est-à-dire $R_w \cap R_{w'} = \varnothing$ (figure 9.2(e-d)) puisque ceux qui intersectent R_w sont supprimés à jamais de W. Il suit que le nombre de sommets sélectionnés lors de la boucle « Tant que » est au plus $|B(u,2)|/\lceil\sqrt{n}\rceil \leqslant n/\sqrt{n} = \sqrt{n}$ chaque ensemble R_w étant inclus dans $B(u,2)$ qui est de taille au plus n.

Le nombre d'arêtes sélectionnées par u est donc au plus $|R_u| - 1 + \sqrt{n} \leqslant \lceil\sqrt{n}\rceil - 1 + \sqrt{n} < (\sqrt{n+1}+1) - 1 + \sqrt{n} = 2\sqrt{n}$. Au total H n'a pas plus de $2n^{3/2}$ arêtes. □

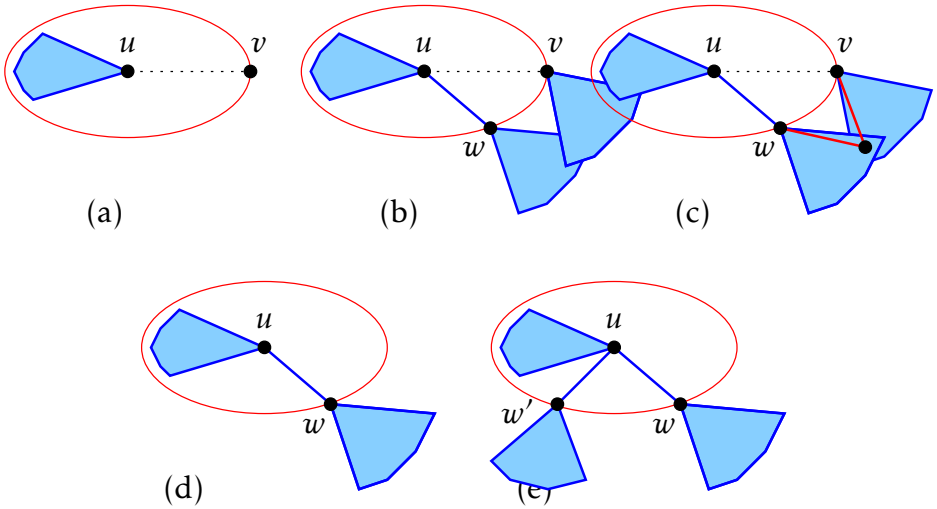


FIGURE 9.2 – Illustration de la preuve du théorème 9.2.

9.3 Exercices

Étendre l'algorithme à un algorithme qui ne nécessite pas la connaissance de n. Combien de rondes cela nécessite ?

Problème ouvert : Peut-on réaliser la tâche précédente avec seulement deux rondes ?

Bibliographie

[BKMP05] S. BASWANA, T. KAVITHA, K. MEHLHORN, AND S. PETTIE, *New constructions of (α, β)-spanners and purely additive spanners*, in 16th Symposium on Discrete Algorithms (SODA), ACM-SIAM, January 2005, pp. 672–681.

[DGPV08] B. DERBEL, C. GAVOILLE, D. PELEG, AND L. VIENNOT, *On the locality of distributed sparse spanner construction*, in 27th Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, August 2008, pp. 273–282. DOI : 10.1145/1400751.1400788.

CHAPITRE

10 Information quantique

« Je crois pouvoir dire sans risque de me tromper que personne ne comprend la mécanique quantique. »

— Richard Feynman

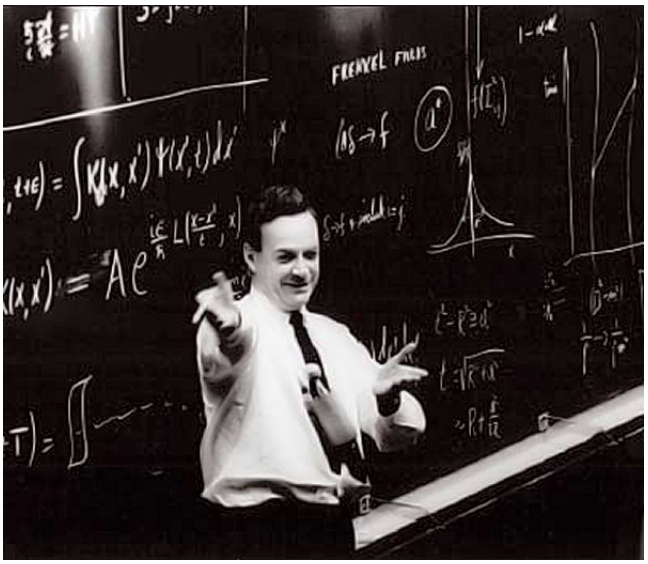


FIGURE 10.1 – Richard Feynman, ici en 1965 après avoir reçu son prix Nobel, est l'un des pères fondateurs de la théorie des ordinateurs quantiques (source © IOP, Cern).

Sommaire

L'objectif de ce chapitre est de donner quelques intuitions sur l'information quantique, même si l'intuition est *a priori* difficile à percevoir, de montrer comment exploiter la non localité de l'information dans le monde quantique. Nous montrerons la différence subtile qu'il existe en calcul distribué entre des ressources aléatoires partagées (*shared randomness*) et des ressources quantiques intriqués (*entangled qubits*).

Mots clés et notions abordées dans ce chapitre :

- principe de causalité, principe de localité
- qubits, superposition, intrication, notation de Dirac
- expérience de Bell, jeu CHSH, jeu GHZ

Notons aussi qu'il existe des tentative de langages dédiés au calcul quantique, comme QCL (Quantum Computation Language [1]), dont on ne parlera pas.

10.1 Introduction

Ils existent deux principes fondamentaux en physique :

- le principe de *causalité*, et
- le principe de *localité*.

Le principe de causalité exprime le fait que la cause précède toujours la conséquence. Un événement ne peut avoir de conséquences que sur des événements futurs. En terme de calcul, cela signifie aussi que le résultat d'un calcul ne peut dépendre que de données déterminées avant le résultat. (Pour calculer $y := f(x)$, il faut que la donnée x soit déterminée. Le calcul de y ne peut pas déterminer x.)

Le principe de localité (parfois appelé principe de séparabilité) exprime le fait que des entités suffisamment éloignées (dans le temps et l'espace) ne peuvent interagir instantanément.

Des expériences reproductibles ont montré que ces deux principes fondamentaux ne peuvent être vrais tous les deux. Plus précisément, Alain Aspects a montré expérimentalement en 1982 que les inégalités de Bell étaient violées, ce qui implique la non localité de la physique quantique en supposant vrai le principe de causalité. Ce fait est utilisé dans certains protocoles de cryptographie quantique où la lecture d'un message par un espion est détectée par le fait que ces inégalités ne sont plus violées une fois les effets quantiques dissipés. Car l'acquisition d'information, comme la lecture, dissipe par nature les effets de superposition quantique.

La réalité physique du monde est donc différente de ce qu'on s'imagine *a priori*, tout au moins à l'échelle microscopique. Ces effets ont été vérifiés pour les particules élé-

1. http://tph.tuwien.ac.at/~oemer/qcl.html

mentaires (photons, électrons) mais aussi pour des atomes et même pour des molécules de plus en plus complexes [2]

La physique quantique manipule des états (ou particules quantiques) qui remettent en cause *a priori* le principe de localité, car des deux principes fondamentaux on s'accorde plutôt pour penser que c'est celui de causalité qui est vrai, c'est-à-dire celui qui est le plus en accord avec le monde réel. Des travaux récents en physique mathématique [DHFRW20] tendent à montrer que les *ensembles causaux* [3] pourrait être une base solide et plus fondamentale à la fois pour la physique quantique et la théorie de la gravitation.

Concernant l'information, remettre en cause la localité signifie qu'un bit d'information n'est pas forcément localisé en un lieu bien déterminé (sur une particule par exemple), mais peut être en même temps dans plusieurs endroit à la fois. C'est ce genre de propriété que l'on tente d'exploiter dans le calcul distribué quantique.

Contrairement à la théorie de la relativité, la physique quantique ne se déduit pas de manière logique à partir d'axiomes évidents ou faciles à admettre, comme par exemple le principe d'invariance (les lois sont les mêmes quel que soit le repère) et de causalité. D'ailleurs de ces deux principes on peut en déduire qu'il existe une vitesse limite c pour la propagation des ondes électromagnétiques. Du coup, la façon d'aborder la physique quantique est différente et moins intuitive. Comme Richard Feymann aimait à le dire « [...] personne ne comprend la mécanique quantique. »

10.2 Préliminaires

10.2.1 Information élémentaire

On pourra aussi se reporter aux articles [BCMdW10][RP00]. En première approximation un bit quantique, ou qubit, est une extension du bit probabiliste. Il est donc important de bien comprendre en premier lieu ce qu'est un bit classique déterministe, puis un bit probabiliste.

Bit déterministe B : représenté par une valeur $B \in \{0, 1\}$.

Bit probabiliste P : représenté par un vecteur P de deux réels $\begin{pmatrix} p \\ q \end{pmatrix}$ où p représente la probabilité d'obtenir la valeur 0 et q celle d'obtenir 1. Bien sûr pour être des probabilités, il faut avoir $0 \leqslant p, q \leqslant 1$ et $p + q = 1$.

2. Le record (fin 2019) est une molécule de 2 000 atomes observée en état de superposition en observant des franges d'interférence [FGZ+19].

3. En gros, ils s'agit de graphes orientés où les arcs correspondent aux liens de causalité. Émergent alors les notions de positions, d'espace, de temps (local), de gravité, de champs quantique... Je vous conseille vivement la vidéo de Passe-science #20 (causal sets) sur ce sujet.

En notant la valeur 0 par le vecteur $\begin{pmatrix}1\\0\end{pmatrix}$ et la valeur 1 par le vecteur $\begin{pmatrix}0\\1\end{pmatrix}$, on peut aussi représenter le bit probabiliste P comme une somme vectorielle des vecteurs de base :

$$P = p\begin{pmatrix}1\\0\end{pmatrix} + q\begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}p\\q\end{pmatrix}.$$

Cette représentation arbitraire à juste pour vocation de donner un cadre plus formelle qui va se généraliser aux bits quantiques. Cette une façon de préparer le terrain.

Bit quantique (ou qubit) Q : representé par un vecteur Q de deux complexes $\begin{pmatrix}\alpha\\\beta\end{pmatrix}$ où $|\alpha|^2$ représente la probabilité d'observer 0 et $|\beta|^2$ celle d'observer 1. Bien sûr, comme le bit probabiliste il faut $|\alpha|^2 + |\beta|^2 = 1$.

On parle d'*état* $\begin{pmatrix}\alpha\\\beta\end{pmatrix}$ pour un qubit Q, puisque les nombres α, β ne sont pas à proprement parlés des probabilités. On garde le terme de *probabilités* pour un bit probabiliste P et de *valeur* pour le bit déterministe B.

Pour récupérer l'information « utile » d'un bit B, P ou Q, il faut faire :

- une *lecture* de sa valeur pour B ;
- un *tirage* aléatoire pour P ;
- une *mesure* de l'état pour Q (voir le paragraphe 10.2.6).

La présence de nombres complexes pour décrire l'état d'un qubit Q s'explique par la nature ondulatoire des particules quantiques. Chaque coefficient α ou β représente l'amplitude et la phase d'une onde qui est de manière générale un nombre complexe selon la théorie ondulatoire classique [4]. La probabilité est proportionnelle à l'intensité de l'onde (disons lumineuse) observée/mesurée (disons sur un écran). Et dans la théorie ondulatoire l'intensité (ou taux d'énergie) n'est rien d'autre que le carré du module de l'amplitude.

Comme précédemment, en notant la valeur 0 par le vecteur $\begin{pmatrix}1\\0\end{pmatrix}$ et la valeur 1 par le vecteur $\begin{pmatrix}0\\1\end{pmatrix}$, on peut aussi représenter le bit quantique Q comme une somme vectorielle des vecteurs de base :

$$Q = \alpha\begin{pmatrix}1\\0\end{pmatrix} + \beta\begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}\alpha\\\beta\end{pmatrix}.$$

On rappelle que $|z|$ représente le module du complexe $z \in \mathbb{C}$. Si $z = x + iy$ alors son module z est $|z| = \sqrt{x^2 + y^2}$ et $z = |z| \cdot (\cos\varphi + i \sin\varphi)$ où $\varphi = \arg(z)$ est l'argument de z. La formule de De Moivre énonce que $\cos\varphi + i\sin\varphi = e^{i\varphi}$.

4. Pour faire simple, une onde transporte de l'énergie sans transporter de matière. C'est une perturbation locale et réversible du milieu physique.

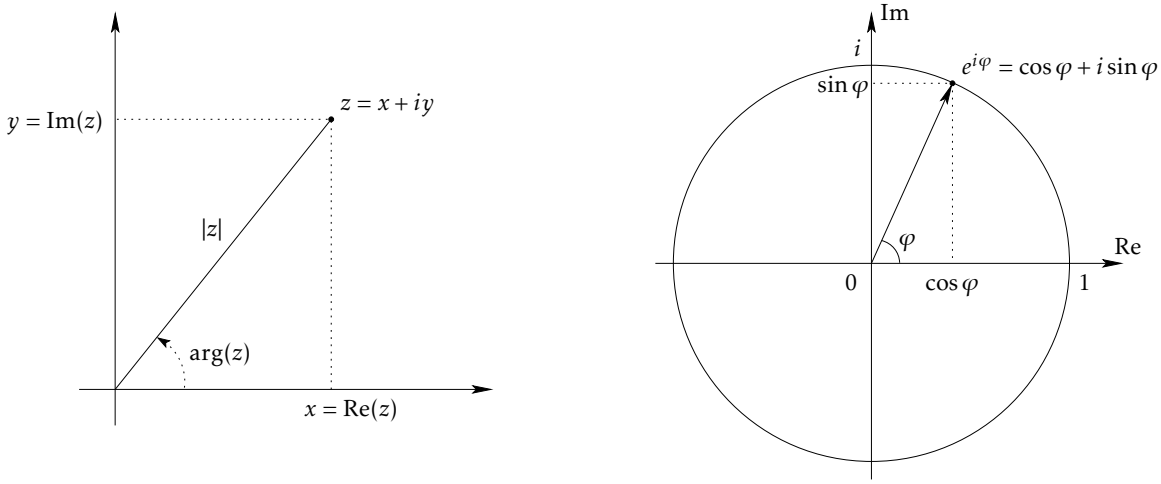


FIGURE 10.2 – Interprétation géométrique d'un nombre complexe $z \in \mathbb{C}$.

Un qubit est implémenté par une particule physique [5] : un photon, un électron, un atome. La mesure d'un qubit correspond, par exemple, à la mesure de la polarisation d'un photon, le spin [6] d'un électron ou le niveau d'énergie d'un atome. Polarisation, spin ou énergie, mais aussi position, vitesse ou impulsion [7], est ce qui s'appelle un *observable*.

Généralement ces particules sont refroidies par laser afin de contrôler et limiter leurs interactions avec l'environnement. Car dès qu'une particule interagit avec l'environnement, l'état du système ⟨particule,environnement⟩ change, en particulier l'état de la particule. Par exemple observer la position d'une particule modifie non seulement l'état de l'observateur (qui sait maintenant où elle se situe) mais aussi celui de la particule observée (par exemple en modifiant sa position ou sa vitesse).

Le formalisme mathématique de la théorie quantique n'a pour vocation que de prédire les états quantiques et de calculer les probabilités d'obtenir tels ou tels résultats. Le point est que jusqu'à présent, les prédictions selon cette théorie (et donc des calculs de probabilités qui en découlent) se sont toujours révélées exactes.

La théorie quantique affirme (c'est en fait l'un de ses axiomes) que des états colinéaires (plus précisément des états dont les vecteurs d'ondes sont colinéaires [8]) sont

5. De même qu'un bit a un support physique : le transistor.

6. Chaque particule possède un *spin* qui est un peu son moment magnétique, comme si la particule était un petit aimant tournant sur lui-même (ce qu'elle ne fait pas). Malheureusement, cette propriété n'a pas d'équivalent dans le monde classique, contrairement aux propriétés comme la charge électrique, la masse, la position, la vitesse, etc. Son intérêt est qu'elle est quantifiée (prend des valeurs discrètes) et facilement observable. Pour la mesurer on fait passer un jet de particules semblables horizontalement à travers un champs magnétique perpendiculaire et chaque particule est alors déviée aléatoirement vers le haut (+1) ou vers le bas (-1) exactement du même angle.

7. C'est une notion proche de celle de quantité de mouvement, soit le produit de la masse par la vitesse.

8. On parle de vecteurs colinéaires $\vec{u}, \vec{v} \in \mathbb{C}^2$, c'est-à-dire tels qu'il existe $\lambda \in \mathbb{C}^*$ avec $\vec{u} = \lambda \cdot \vec{v}$.

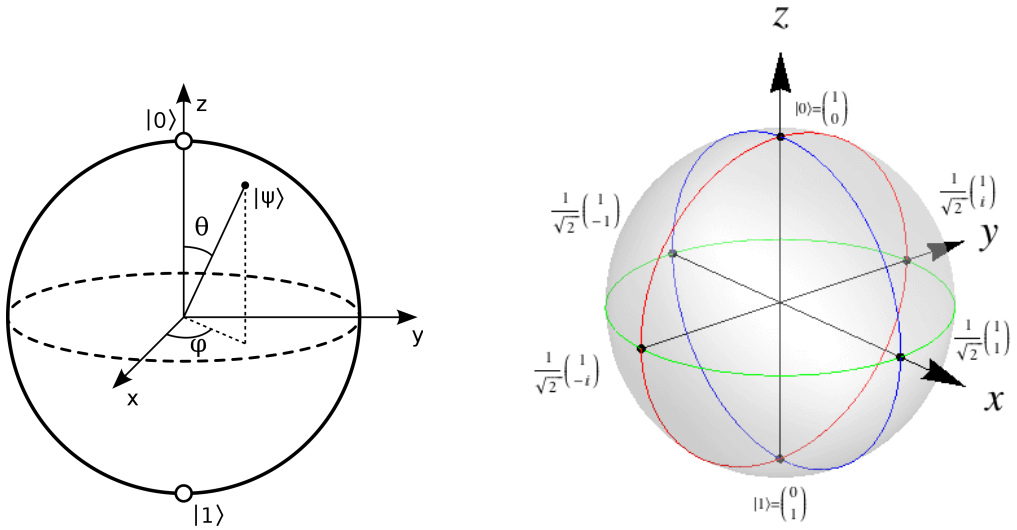


FIGURE 10.3 – Sphère de Bloch : un point de la sphère représente l'état quantique d'une particule.

indiscernables – les projections lors des mesures seront les mêmes. Du coup, l'état d'un qubit est bien un point de la surface unité d'une sphère : les coefficients complexes α, β nous donnent *a priori* 4 degrés de liberté ; la condition $|\alpha|^2 + |\beta|^2 = 1$ fait que le qubit évolue dans un espace isomorphe à $\mathbb{R}^3$; et finalement la co-linéarité fait qu'on peut normaliser les vecteurs à l'unité. D'où la sphère unité de $\mathbb{R}^3$ comme sur la figure 10.3.

10.2.2 Superposition, interférence et décohérence

La *superposition* exprime le fait qu'un qubit peut être dans la superposition de plusieurs états. On l'exprime formellement en écrivant que l'état d'un qubit Q vaut :

$$Q = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}.$$

Dans le monde physique, cela traduit la dualité onde-corpusculaire. Par exemple, un photon individuel va passer par les deux fentes de Young et les probabilités d'observer un résultat sur l'écran vont dépendre des longueurs des deux chemins utilisés avec un effet constructif ou destructif comme une onde (voir la figure 10.4(c)). Le photon est alors dans un état dit de *superposition*. Il passe réellement par les deux fentes à la fois, comme le ferai une onde. Quant au résultat observé, c'est comme si le photon interfèrerait avec lui-même par recombinaison. D'un autre coté on n'observe jamais un photon à deux endroits à la fois.

Il est important de noter que ce phénomène d'*interférence* se produit pour *une seule* particule. Ce n'est pas un phénomène lié à un flux de particules. Cela a été testé physiquement pour des photons individuels par Alain Aspect dans les années 1980, puis sur

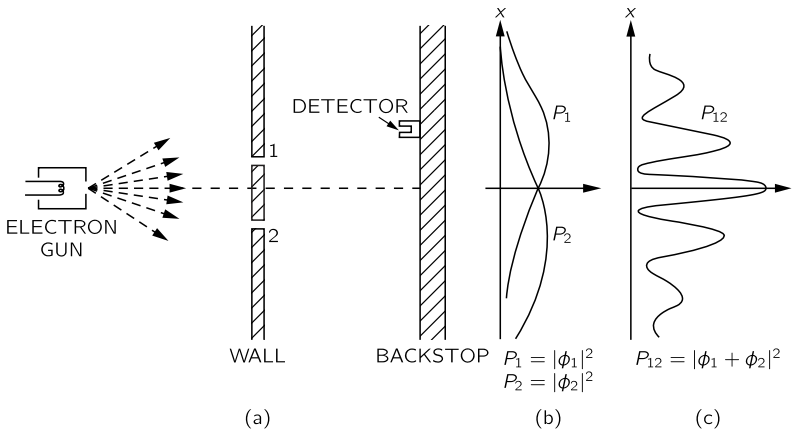


FIGURE 10.4 – Le détecteur permet de capter ou non un électron individuel à une position déterminée (axe x). Pour obtenir la distribution P_1 par exemple (fente 1 ouverte), il faut déplacer uniformément le détecteur et compter. La ditribution P_{12} est celle observée lorsque les fentes 1 et 2 sont ouvertes simultanément. Extrait des cours de Richard Feymann (caltech.edu).

des électrons, protons, neutrons et même des atomes et plus récemment des molécules. C'est bien une seule particule qui passe par les deux fentes à la fois.

Si l'on obstrue une des deux fentes (figure 10.4(b)), la distribution devient quasiment uniforme sur l'écran : on observe une tâche diffuse. Après l'observation, l'état de superposition est détruit, car dans la réalité telle qu'on la conçoit on observe jamais le même photon à deux endroits à la fois.

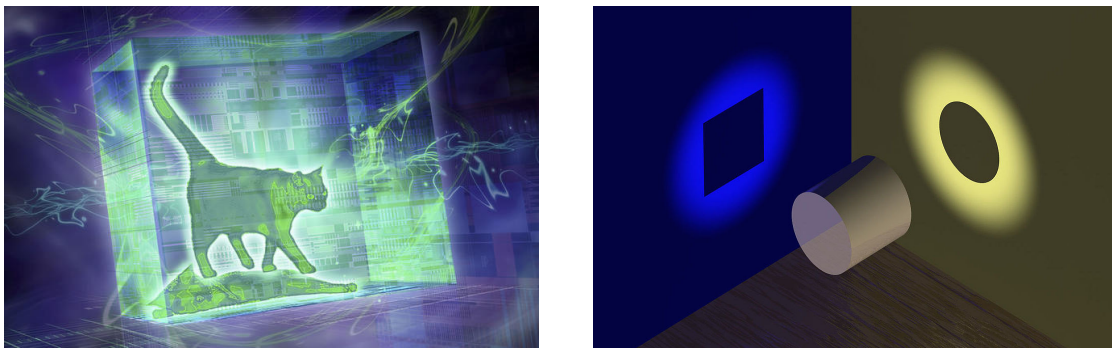


FIGURE 10.5 – Illustration du phénomène de superposition par le Chat de Schrödinger (source © H. Ritsch/Science Photo Library), et la métaphore du cylindre (source Wikipédia).

La métaphore du cylindre (voir figure 10.5) est l'exemple d'un objet ayant des propriétés (comme être un cercle ou un carré) apparemment inconciliables. Mais une projection peut nous faire apparaître l'une ou l'autre de ces propriétés. Pour une particule dans un état quantique c'est la même chose : onde et particule sont deux facettes (ob-

servations) d'une même réalité mais pas la réalité elle-même. Notons que la notion de *mesure* d'un état quantique revient à faire aussi une projection (cf. paragraphe 10.2.6). Voir aussi l'extrait de la BD sur la figure 10.6.

FIGURE 10.6 – « Le Mystère du Monde Quantique » par T. Damour et M. Burniat © Dargaud 2016.

La *décohérence* est le fait une particule dans un état de superposition va au bout d'un moment devenir une simple particule probabiliste à cause des interaction avec l'extérieur au système. Dans l'expérience de Young, si le photon interagit avec les atomes près d'un des deux parcours possibles, alors le choix de la trajectoire devient simplement aléatoire et plus superposé.

10.2.3 Système à plusieurs qubits

Pour l'instant il n'y a pas de différence fondamentale entre un bit probabiliste $P = p\begin{pmatrix}1\\0\end{pmatrix}+q\begin{pmatrix}0\\1\end{pmatrix}$ et un qubit $Q = \alpha\begin{pmatrix}1\\0\end{pmatrix}+\beta\begin{pmatrix}0\\1\end{pmatrix}$ hormis que α, β peuvent être complexes. Dans les deux cas, les coefficients donnent la probabilité d'observer 0 (p ou $|\alpha|^2$) et la probabilité d'observer 1 (q ou $|\beta|^2$). Il va en être autrement lorsqu'on considère des systèmes de plusieurs bits et qubits.

En déterministe un système composé d'une paire de valeurs v_1, v_2 se représente tout simplement par un couple (v_1, v_2), un vecteur ligne $(\,v_1\ v_2\,)$ ou encore un vecteur colonne ${}^t(\,v_1\ v_2\,) = \begin{pmatrix}v_1\\v_2\end{pmatrix}$.

On pourrait représenter deux bits probabilistes (P_1, P_2) par un couple de vecteurs ou

encore un vecteur colonne à quatre composantes :

$$(P_1, P_2) = \left(\begin{pmatrix} p_1 \\ q_1 \end{pmatrix}, \begin{pmatrix} p_2 \\ q_2 \end{pmatrix}\right) \quad \text{ou} \quad \begin{pmatrix} p_1 \\ q_1 \\ p_2 \\ q_2 \end{pmatrix} \quad \text{ou} \quad \begin{pmatrix} p_1p_2 \\ p_1q_2 \\ q_1p_2 \\ q_1q_2 \end{pmatrix} = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}$$

Ces représentations sont en fait toutes équivalentes, ce qui nécessite un effort de vérification pour la dernière. En utilisant le fait que $p_1 + q_1 = 1$ et que $a/c = p_1/q_1$, on déduit que $p_1 = a/(a+c)$ et $q_1 = c/(c+a)$. De la même manière, $p_2 + q_2 = 1$ et $a/b = p_2/q_2$ impliquent que $p_2 = a/(a+b)$ et $q_2 = b/(b+a)$. On peut donc facilement passer d'une représentation à n'importe qu'elle autre.

La dernière représentation a l'avantage de donner les probabilités d'obtenir chacune des quatre paires de valeurs $(0,0)$, $(0,1)$, $(1,0)$ et $(1,1)$. De plus elle correspond à un produit *tensoriel*, opération vectorielle notée $\otimes$, appelé aussi *produit de Kronecker*. De manière générale, si $A = (a_{i,j})$ et B sont deux matrices de dimensions respectives $r_A \times s_A$ et $r_B \times s_B$, alors $A \otimes B = (a_{i,j}B)$ est une matrice $r_A r_B \times s_A s_B$.

Ainsi,

$$\begin{pmatrix} p_1 \\ q_1 \end{pmatrix} \otimes \begin{pmatrix} p_2 \\ q_2 \end{pmatrix} = \begin{pmatrix} p_1 \cdot \begin{pmatrix} p_2 \\ q_2 \end{pmatrix} \\ q_1 \cdot \begin{pmatrix} p_2 \\ q_2 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} p_1p_2 \\ p_1q_2 \\ q_1p_2 \\ q_1q_2 \end{pmatrix}.$$

Notez en passant que ce produit n'est pas commutatif en général, car par exemple si $p_2q_1 \neq p_1q_2$, alors

$$\begin{pmatrix} p_2 \\ q_2 \end{pmatrix} \otimes \begin{pmatrix} p_1 \\ q_1 \end{pmatrix} = \begin{pmatrix} p_2p_1 \\ \boxed{p_2q_1} \\ q_2p_1 \\ q_2q_1 \end{pmatrix} \neq \begin{pmatrix} p_1p_2 \\ \boxed{p_1q_2} \\ q_1p_2 \\ q_1q_2 \end{pmatrix} = \begin{pmatrix} p_1 \\ q_1 \end{pmatrix} \otimes \begin{pmatrix} p_2 \\ q_2 \end{pmatrix}.$$

On remarque que ce produit tensoriel se décompose en une combinaison linéaire de quatre vecteurs unitaires coefficientés par les probabilités d'obtenir les valeurs $(0,0)$, $(0,1)$, $(1,0)$ et $(1,1)$.

$$\begin{pmatrix} p_1 \\ q_1 \end{pmatrix} \otimes \begin{pmatrix} p_2 \\ q_2 \end{pmatrix} = \begin{pmatrix} p_1p_2 \\ p_1q_2 \\ q_1p_2 \\ q_1q_2 \end{pmatrix} = p_1p_2 \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + p_1q_2 \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} + q_1p_2 \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + q_1q_2 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

D'ailleurs on vérifie que la somme $p_1p_2 + p_1q_2 + q_1p_2 + q_1q_2 = (p_1+q_1)(p_2+q_2) = 1$ puisque $p_1 + q_1 = p_2 + q_2 = 1$. Ainsi le produit tensoriel permet une simple généralisation du bit probabiliste $P = \begin{pmatrix} p \\ q \end{pmatrix} = p\begin{pmatrix} 1 \\ 0 \end{pmatrix} + q\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ a un système qui se décomposent en une somme de vecteur unitaires coefficientés par des probabilités.

De même, un système (A,B) composé de deux qubits A et B se représentera comme une combinaison linéaire des 4 vecteurs unitaires

$$(A,B) = \begin{pmatrix} c_{00} \\ c_{01} \\ c_{10} \\ c_{11} \end{pmatrix} = c_{00}\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + c_{01}\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} + c_{10}\begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + c_{11}\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

avec $c_{ij} \in \mathbb{C}$ et $\sum_{ij} |c_{ij}|^2 = 1$, la probabilité d'observer $A = i \in \{0,1\}$ et $B = j \in \{0,1\}$ étant contrôlée par $|c_{ij}|^2$. Pour un système à 3 qubits, on aurait une description décrivant la combinaison linéaire des 8 vecteurs de base (soit une superposition)

$$(A,B,C) = \begin{pmatrix} c_{000} \\ c_{001} \\ c_{010} \\ c_{011} \\ c_{100} \\ c_{101} \\ c_{110} \\ c_{111} \end{pmatrix} = c_{000}\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + c_{001}\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \cdots + c_{110}\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + c_{111}\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

L'état du système (A,B,C) donné par l'equation précédente est déterminé par les coefficients (*a priori* complexes) $c_{000},\ldots,c_{111}$. Si l'on fait une mesure sur chacun des trois qubits (mesure simultanée ou pas), la probabilité d'observer par exemple $A = 1$, $B = 0$ et $C = 1$ sera $|c_{101}|^2$.

Plus généralement l'état d'un système à n qubits se décrit comme la combinaison linéaire de 2^n vecteurs de taille 2^n. De même pour un système de n bits probabilistes. Bien sûr, si certains bits sont indépendants, ce n'est pas la façon la plus compacte de faire. S'ils sont tous indépendants par exemple, alors la suite $p_1,\ldots,p_n$ des probabilités suffit à décrire le système.

10.2.4 Intrication

Le point important est qu'on peut avoir des systèmes à $n > 1$ qubits qui sont dans des états qui ne peuvent correspondre à aucun produit tensoriel. On parle d'états non séparables ou *intriqués*. Inversement on dira que l'état (A,B) d'un système est *séparable* si $(A,B) = A \otimes B$ ce qui revient à dire que le système est composé de deux sous-systèmes indépendants. Deux particules intriquées forment donc un système unique, solidaire, inséparable (cf. figure 10.7).

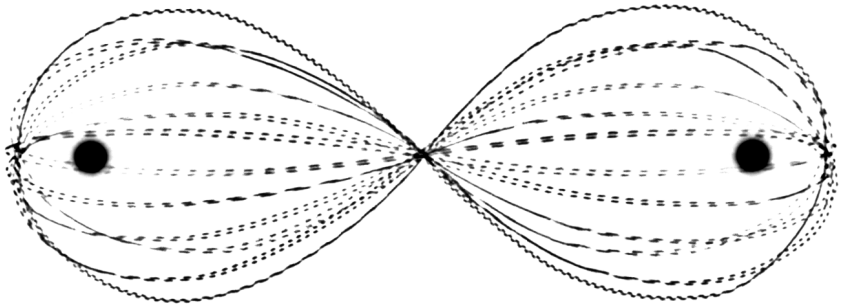


FIGURE 10.7 – Illustration d'un système de deux particules intriquées.

Par exemple prenons un système (A, B) de deux qubits défini [9] par :

$$(A,B) \;=\; \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\0\\0\end{pmatrix} + \frac{1}{\sqrt{2}}\begin{pmatrix}0\\0\\0\\1\end{pmatrix}.$$

Cet état signifie que la probabilité d'obtenir $(0,0)$ est $|1/\sqrt{2}|^2 = 1/2 = 50\%$, soit la même que d'obtenir $(1,1)$. Il n'est pas possible d'obtenir $(1,0)$ ou $(0,1)$. Le résultat de la mesure des qubits A et B est bien aléatoire mais corrélé.

Ce système n'est pas séparable. S'il l'était alors on aurait :

$$(A,B) = \begin{pmatrix}\alpha_0\\\beta_0\end{pmatrix} \otimes \begin{pmatrix}\alpha_1\\\beta_1\end{pmatrix} \quad \Rightarrow \quad (A,B) = \alpha_0\alpha_1\begin{pmatrix}1\\0\\0\\0\end{pmatrix} + \beta_0\alpha_1\begin{pmatrix}0\\1\\0\\0\end{pmatrix} + \alpha_0\beta_1\begin{pmatrix}0\\0\\1\\0\end{pmatrix} + \beta_0\beta_1\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$$

pour $\alpha_0, \alpha_1, \beta_0, \beta_1 \in \mathbb{C}$ à déterminer. Or il faut $\alpha_0\alpha_1 = \beta_0\beta_1 = 1/\sqrt{2} \neq 0$ et $\beta_0\alpha_1 = \alpha_0\beta_1 = 0$, ce qui n'est pas possible, même dans $\mathbb{C}$.

Remarques. Pour générer un système de qubits intriqués, les particules support doivent préalablement interagir. En pratique on peut récupérer deux photons issus d'une même « cascade atomique », comme l'a fait Alain Aspect dans sa célèbre expérience. On excite un atome, par exemple de calcium, à un niveau d'énergie élevé avec un laser. Lorsque les électrons reviennent au niveau fondamental, en passant par un niveau d'énergie intermédiaire, deux photons intriqués sont émis (un pour chaque niveau). Leur polarisation est alors corrélées et on peut leur faire prendre des directions différentes et contrôlées (voir [Teo16]). Un même système peut avoir une propriété en état d'intrication

9. On aurait pu prendre un système $(P_1, P_2) = {}^t(p_1p_2, 0, 0, q_1q_2)$ de deux bits probabilistes corrélés. La non séparabilité n'est pas le propre de la mécanique quantique.

(ici la polarisation) et une autre dans un état classique (ici la position). Il se trouve aussi que les deux électrons de l'atome d'hélium sont toujours dans un état intriqué. De manière générale, deux éléments quantiques indiscernables ou issus d'un même processus sont intriqués tant que les éléments n'ont pas subi de mesure qui détruit l'état quantique. Par exemple, un photon qui passe à travers un cristal non linéaire va être coupé en deux photons intriqués (voir figure 10.8).

La propriété d'intrication a été vérifiée, non seulement pour des systèmes de deux particules, mais aussi des systèmes de deux objets macroscopiques (faisceaux de silicium de la taille d'une bactérie) d'une dizaine de milliards d'atomes et espacés de 20 cm, l'effet pouvant être maintenu pendant plusieurs minutes.

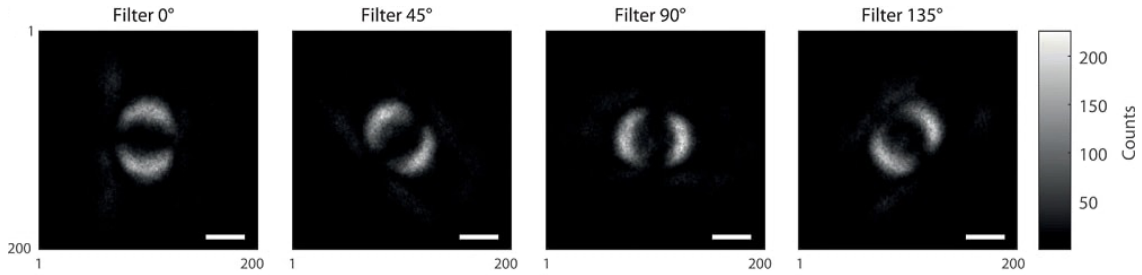


FIGURE 10.8 – Premières photos de photons intriqués (juillet 2019). Il s'agit de flux de photons intriqués selon différents filtres, l'image résultante étant la superposition de chacun des deux flux.

Actuellement, on est capable de construire des systèmes jusqu'à plusieurs dizaines [10] de qubits intriqués (cf. figure 10.9). La question de savoir s'il sera possible un jour de construire des ordinateurs généraliste avec beaucoup plus de qubits est largement débattue [Kal19]. Les effets non locaux d'intrications (cf. paragraphe 10.2.10) ont été testés jusqu'à des distances supérieures à plusieurs centaines de kilomètres en conditions extérieures [11] et même dans l'eau.

10.2.5 Notation de Dirac

Comme on vient de le voir, lorsqu'on considère des systèmes à n qubits on est amené à manipuler des vecteurs de dimension 2^n qui ont tendances à être très peu denses. En

10. IBM a annoncé en mai 2017 un processeur à 17 qubits, et à moyen terme l'objectif d'un processeur à 50 qubits avec 90 microsecondes de temps quantique.
11. Le record en 2017 était de 1 200 km détenu par la Chine (voir l'article).

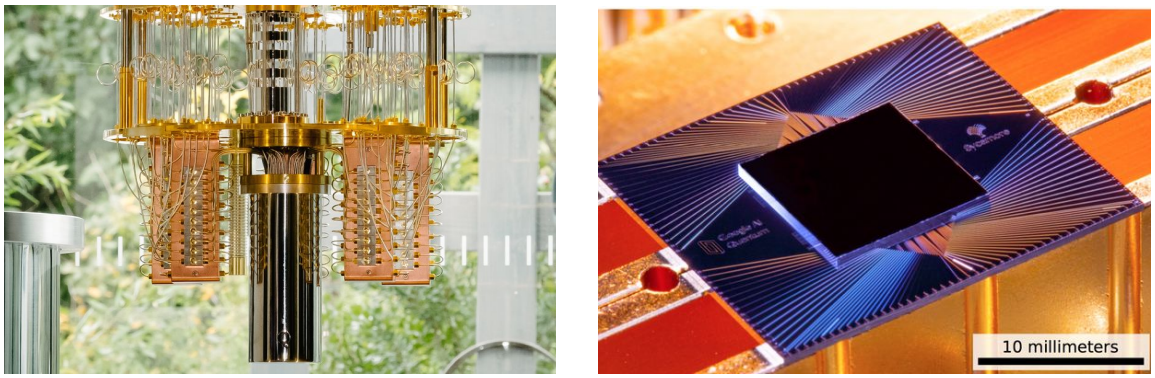


FIGURE 10.9 – Cryostat d'IBM câblé pour le processeur de 50 qubits, et le processeur Sycamore de 54 qubits de Google. © IBM & Google.

particulier, les 2^n vecteurs de la base de calcul

$$\begin{pmatrix}1\\0\\\vdots\\0\end{pmatrix},\begin{pmatrix}0\\1\\\vdots\\0\end{pmatrix},\ldots,\begin{pmatrix}0\\\vdots\\1\\0\end{pmatrix},\begin{pmatrix}0\\0\\\vdots\\1\end{pmatrix}$$

contiennent qu'une seule valeur non nulle pour 2^n composantes. On utilise donc la notation de Dirac ou notation « bra-ket », qui pour les vecteurs de la base utilise n caractères au lieu de 2^n. Le vecteur de base pour la dimension $i = 0,\ldots,n-1$, donc celui ayant un seul « 1 » à la ligne i, est simplement noté $|\text{bin}_n(i)\rangle$ où $\text{bin}_n(i)$ est l'écriture binaire de l'entier i sur n bits.

Par exemple, $|0\rangle$, $|01\rangle$ et $|111\rangle$ représenteront respectivement les vecteurs

$$\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}0\\1\\0\\0\end{pmatrix},\begin{pmatrix}0\\0\\0\\0\\0\\0\\0\\1\end{pmatrix}$$

ce qui pour $n = 3$ représente un gain de place non négligeable.

Aussi, on notera l'état d'un système S, composé d'un ou plusieurs qubits, par $|S\rangle$ qui est donc un vecteur d'une certaine dimension (2^n si S est composé de n qubits). Pour

$n = 2$, on pourra par exemple écrire :

$$|S\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\0\\0\end{pmatrix} + \frac{1}{\sqrt{2}}\begin{pmatrix}0\\0\\0\\1\end{pmatrix} = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle = \begin{pmatrix}2^{-1/2}\\0\\0\\2^{-1/2}\end{pmatrix}.$$

Il est facile de voir que, pour tout mot binaire $w \in \{0,1\}^+$,

$$|0\rangle \otimes |w\rangle = \begin{pmatrix}1\\0\end{pmatrix} \otimes |w\rangle = \begin{pmatrix}1 \cdot |w\rangle\\0 \cdot |w\rangle\end{pmatrix} = \begin{pmatrix}|w\rangle\\0\\\vdots\\0\end{pmatrix} = |0w\rangle\,.$$

De même, $|1\rangle \otimes |w\rangle = |1w\rangle$, si bien que, plus généralement, on a $|v\rangle \otimes |w\rangle = |vw\rangle$ pour tout mot binaire v. (Cela se montre par induction sur la longueur du mot v et par l'associativité de $\otimes$.)

On note la transposition d'un vecteur $|A\rangle$ non pas par ${}^t|A\rangle$ mais par $\langle A|$. [*Cyril.* À revoir. En fait $\langle A|$ est le trans-conjugué, c'est-à-dire la matrice transposée où chaque élément complexe de A est remplacé par son conjugué : $a + ib \mapsto a - ib$. On note que $(a+ib)\cdot(a-ib) = a^2 - (ib)^2 = a^2 + b^2$.] Si bien que $|A\rangle \cdot \langle B|$ représente le produit scalaire de $|A\rangle$ par $|B\rangle$. Comme on va le voir bientôt, le produit scalaire représentera l'application d'opérateur sur les qubits. Produit scalaire et vecteur à coefficients complexes font que l'état d'un système quantique S à n qubits est finalement représenté par un vecteur de l'espace de Hilbert de dimension 2^n.

10.2.6 Mesure

Faire une mesure d'un système S consiste à observer son état. En terme informatique, c'est produire le résultat d'un calcul. En fait, une mesure produit deux effets : 1) elle modifie le système qui se trouve alors dans un état particulier ; et 2) elle permet de récupérer une probabilité d'observer ce nouvel état.

Prenons un exemple déjà rencontré d'un système S composé de deux qubits dans l'état

$$|S\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle\,.$$

Après la mesure du système S, on obtient $|00\rangle$ ou $|11\rangle$ avec probabilité $\frac{1}{2}$. Plus généralement, la mesure d'un système S de n qubits initialement dans l'état :

$$|S\rangle = \sum_{i=0}^{2^n-1} \alpha_i \cdot |\mathrm{bin}_n(i)\rangle$$

à pour effet : 1) de modifier S qui passe alors dans un état $|S'\rangle = |\text{bin}_n(i)\rangle$ qui est déterminé (la probabilité que l'état de S' soit $|\text{bin}_n(i)\rangle$ est précisément de $|1|^2 = 1$) ; et 2) de fournir à l'expérimentateur l'indice i en question. Et donc la probabilité que cela se produise est $|\alpha_i|^2$. Dans ce cas les valeurs des n qubits deviennent déterminées à 0 ou à 1 suivant la chaîne $\text{bin}_n(i)$.

À vrai dire il est possible d'effectuer d'autres sortes de mesures sur un état $|S\rangle$, comme par exemple mesurer un seul ou une partie des n qubits. Les mesures précédentes sont dites selon la « base de calcul ». Un des axiomes de la mécanique quantique stipule que pour toute base orthogonale $|B_i\rangle$, la mesure de l'état $|S\rangle = \sum_i \beta_i \cdot |B_i\rangle$ fait passer S dans un état $|B_i\rangle$ avec la probabilité connue $|\beta_i|^2$.

10.2.7 Opérateurs

De manière générale, l'évolution d'un système quantique suit l'équation d'Erwin Schrödinger (1925) qui est très jolie mais dont on ne parlera pas trop dans ce cours [12] :

$$i\hbar \cdot \frac{\partial}{\partial t}|\Psi(r,t)\rangle \;=\; \hat{H} \cdot |\Psi(r,t)\rangle\,.$$

En gros, les états successifs (dérivée selon le temps) s'obtiennent en appliquant une certaine opération $\hat{H}$ (qu'on appelle hamiltonien du système). C'est l'évolution de l'équation d'onde d'un système quantique Ψ.

On modifie l'état d'un système S par l'application d'un *opérateur* qui n'est rien d'autre que le produit du vecteur d'état $|S\rangle$ par une matrice U d'un espace de Hilbert de dimension 2^n pour un système de n qubits.

$$|S'\rangle \;=\; U \cdot |S\rangle\,.$$

Les opérateurs sont des matrices carrées *unitaires* $2^n \times 2^n$ à coefficients dans $\mathbb{C}$, c'est-à-dire un opérateur U vérifiant $U^* \cdot U = U \cdot U^* = \text{Id}_{2^n}$ où U^* est l'adjoint de U et Id_{2^n} est la matrice identité de dimension 2^n. Ce sont aussi les matrices de changement de base. Notons que la matrice nulle (avec des zéros partout) par exemple n'est pas un opérateur, elle n'est pas unitaire. En quelque sorte, on doit toujours pouvoir revenir à l'état initial, car :

$$U^* \cdot |S'\rangle \;=\; U^* \cdot U \cdot |S\rangle \;=\; |S\rangle\,.$$

Les calculs quantiques doivent donc être réversibles, sinon il s'agit d'une mesure. Dans ce cas le système perd de l'information qui est transférée à l'expérimentateur.

Pour $n = 1$, les opérateurs sont des matrices 2×2. Par exemple :

$$\text{NOT} := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \text{NOT} \cdot \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \;=\; \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \;=\; \begin{pmatrix} \beta \\ \alpha \end{pmatrix}.$$

12. $\hbar \approx 1.054\,571\,800 \times 10^{-34} J \cdot s$ est la constante de Planck réduite, r le vecteur de position, t le temps, et $\hat{H}$ est l'hamiltonien du système, son énergie totale.

Il suit que $\text{NOT} \cdot |0\rangle = |1\rangle$ et $\text{NOT} \cdot |1\rangle = |0\rangle$. C'est donc l'opérateur NOT classique lorsque l'entrée est 0 ou 1, mais de manière générale il s'agit d'un échange de coordonnées. Sur la sphère de Bloch, cela correspond à la symétrie centrale autour de l'origine. En classique et pour $n = 1$, c'est, avec l'identité, le seul opérateur réversible. En quantique par contre, dès $n = 1$ il existe déjà une infinité d'opérateurs comme cette famille paramétrée par $\theta \in \mathbb{R}$:

$$R(\theta) := \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}. \tag{10.1}$$

Cet opérateur revient à faire une rotation du vecteur représentant l'état du qubit, qui on le rappelle est un vecteur. On a aussi des opérateurs plus exotiques comme (ici i est le nombre complexe tel que $i^2 = -1$)

$$\sqrt{\text{NOT}} := \frac{1}{2}\begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix}$$

son nom étant justifié par le fait que $\sqrt{\text{NOT}} \cdot \sqrt{\text{NOT}} = \text{NOT}$. Si l'opérateur NOT pouvait être vu comme une symétrie ou rotation du vecteur d'état sur la surface de Bloch, $\sqrt{\text{NOT}}$ peut être vu comme une rotation intermédiaire.

Et pour $n = 2$, on a par exemple l'opérateur *Controlled-Not* :

$$\text{CNOT} := \begin{pmatrix} \text{Id}_2 & 0 \\ 0 & \text{NOT} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

L'effet de cet opérateur sur un système composé de deux qubits $|x\rangle \otimes |y\rangle = |x, y\rangle$ est le suivant : $\text{CNOT} \cdot |x, y\rangle = |x, x \oplus y\rangle$ (voir figure 10.10). Donc si $x = 1$, y est inversé sinon il est laissé tel quel. Notons que pour être réversible il faut garder en mémoire (dans la sortie) x et/ou y. Ici c'est x qui est conservé.

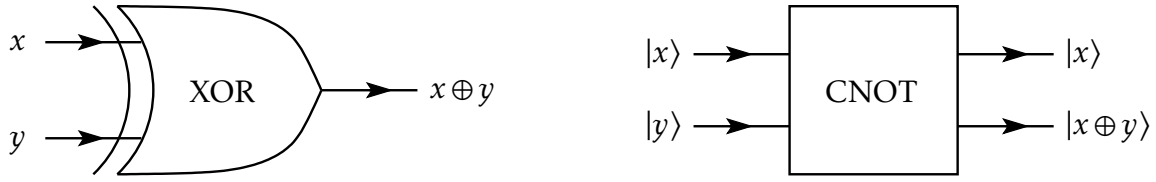


FIGURE 10.10 – Opérateurs classique XOR et quantique CNOT.

En fait, tous les circuits booléens classiques peuvent être construits par un circuits quantiques à la différence que les opérateurs doivent avoir une ou plusieurs entrées supplémentaires pour les rendre réversibles. La porte universelle classique comme NAND (pour *Not-And*) peut être réalisée par un opérateur sur 3 qubits, donc par une matrice 8×8. C'est l'opérateur *Toffoli* appelé aussi CCNOT (pour *Controlled-Controlled-Not*). Cet opérateur se comporte ainsi : $\text{CCNOT} \cdot |x, y, z\rangle = |x, y, z \oplus (x \wedge y)\rangle$. En particulier $\text{CCNOT} \cdot |x, y, 1\rangle = |x, y, \neg(x \wedge y)\rangle = |x, y, \text{NAND}(x, y)\rangle$. Mais aussi $\text{CCNOT} \cdot |1, y, z\rangle =$

$|1, y, z \oplus y\rangle = |1, \text{CNOT} \cdot |y, z\rangle\rangle$. Enfin, $\text{CCNOT} \cdot |1, 1, z\rangle = |1, 1, \neg z\rangle$. Il peut aussi être défini par :

$$\text{CCNOT} := \begin{pmatrix} \text{Id}_6 & 0 \\ 0 & \text{NOT} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

10.2.8 Commutation des opérations locales

Certains opérateurs quantiques binaires précédents, comme par exemple CNOT, ne sont pas locaux dans le sens où il faut avoir accès à la fois au qubit $|x\rangle$ et au qubit $|y\rangle$ pour calculer $\text{CNOT} \cdot |x, y\rangle$. Intuitivement, les entrées doivent passer dans le même lieu au même moment. Dans un certain nombre de cas, les qubits sont très éloignés (Alice possède $|x\rangle$ et Bob $|y\rangle$) et on veut pouvoir effecter une certaine opération sur le système $|x, y\rangle$ seulement en interagissant sur $|x\rangle$ ou sur $|y\rangle$. Il s'agit alors d'une opération « locale » réalisée par Alice (disons l'opération A) ou par Bob (disons l'opération B). Dans le premier cas, l'opérateur appliqué sur le système est $A \otimes \text{Id}$. Dans le second cas, il s'agit de l'opérateur $\text{Id} \otimes B$. Si Alice et Bob appliquent simultanément [13] leur propre opération, il s'agit de l'opérateur $A \otimes B$.

Si Alice et Bob appliquent leurs opérateurs localement chacun sur leur qubit, on obtient *a priori* trois évolutions possibles du système $|x, y\rangle$, en se rappelant que le produit de matrice est associatif mais pas commutatif. Plus précisément, le système devient :

$$\begin{aligned} (\text{Id} \otimes B) \cdot (A \otimes \text{Id}) \cdot |x, y\rangle \quad & \text{(Alice, puis Bob)} \\ (A \otimes \text{Id}) \cdot (\text{Id} \otimes B) \cdot |x, y\rangle \quad & \text{(Bob, puis Alice)} \\ (A \otimes B) \cdot |x, y\rangle \quad & \text{(Alice et Bob).} \end{aligned}$$

Une propriété importante du produit tensoriel qui va nous servir est que

$$(A \otimes X) \cdot (Y \otimes B) \;=\; (A \cdot Y) \otimes (X \cdot B)$$

[*Cyril*. À vérifier sur des matrices (blocs) 2×2.] En particulier, si $X = Y = \text{Id}$, alors

$$\begin{aligned} (A \otimes \text{Id}) \cdot (\text{Id} \otimes B) \;&=\; (A \cdot \text{Id}) \otimes (\text{Id} \cdot B) \;=\; A \otimes B \\ &=\; (\text{Id} \cdot A) \otimes (B \cdot \text{Id}) \\ &=\; (\text{Id} \otimes B) \cdot (A \otimes \text{Id})\,. \end{aligned}$$

13. Pour peu que cela ait un sens, cf. la figure 1.10.

Et donc les opérations « locales » appliquées à un système binaire, soient $(A \otimes \mathrm{Id})$ et $(\mathrm{Id} \otimes B)$, commutent. L'état du système résultant ne dépend pas de l'ordre d'application des opérations. C'est d'ailleurs le même résultat que d'appliquer simultanément l'opération $A \otimes B$ sur le système.

On peut déduire de cette propriété de commutation des opérateurs locaux une propriété fondamentale de la théorie quantique. C'est une théorie dite *non-signalling* ou non signalante (Voir [Bar06][Corollaire 1, pp. 5].) Cela serait très gênant si cela n'était pas le cas, car on violerait le principe de causalité. On y reviendra plus tard.

10.2.9 Copie et effacement

En admettant le principe que l'on peut copier que ce que l'on connaît, il devient alors évident que les bits probabilistes et quantiques ne peuvent pas être copiés. Seuls les bits déterministes (donc connus) le peuvent. Plus formellement, on peut montrer qu'il n'existe pas d'opérateur unitaire U permettant de passer d'un état [14] $|\psi, 0\rangle$ à un état $|\psi, \psi\rangle$, soit

$$U \cdot |\psi, 0\rangle \; = \; |\psi, \psi\rangle$$

à condition bien sûr que $|\psi\rangle$ soit une superposition d'états, c'est-à-dire un vecteur $\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ dont les coefficients vérifient $0 < |\alpha|^2, |\beta|^2 < 1$, puisque les états dont les coordonnées sont 0 ou 1 sont déterminées de manière certaine et donc copiables. Pour la même raison il n'est pas possible d'« effacer » un qubit : il n'y a pas d'opérateur unitaire U tel que $U \cdot |\psi\rangle = |0\rangle$ pour un vecteur $|\psi\rangle$ en état de superposition.

Le principe de copier seulement que ce que l'on connaît peut se voir aussi comme le principe suivant : « pour copier il faut mesurer ». Or faire une mesure change l'état du système. En particulier, la mesure d'un qubit à pour effet de fixer sa valeur, désormais on la connaît. De même un appel à une fonction `random()` aura pour effet de fixer la valeur d'un bit aléatoire. Maintenant, pour les opérations, c'est différent. Bien entendu il faut admettre qu'on puisse modifier un état sans forcément en connaître sa valeur. Lorsqu'on fait `NOT(x)` par exemple, on ne dit pas qu'on connaît `x`. Cette opération pourrait correspondre à la rotation d'une boîte spéciale contenant `x` telle que si on l'ouvre par le haut on lit `x` et si c'est par le bas on lit `NOT(x)`. Donc tant que la boîte n'est pas ouverte, on ne sait pas si l'on va lire un 0 ou un 1, l'état de `x` n'est pas connue (et potentiellement elle n'est peut-être même pas encore fixée). Par contre, en inversant le haut et le bas on sait que l'on lira 0 si `x` était dans l'état 1 et 0 si `x` était dans l'état 1.

On pourrait imaginer copier un bit probabiliste P généré par une certaine fonction `random()` en copiant le code source du générateur, ainsi que l'initialisation de sa graine `seed`. Mais dans ce cas, le bit P serait déterminé par la donnée (`seed`, `random`) ce qui contredit le fait que P soit aléatoire (on parle de nombre pseudo-aléatoire). Si P est

14. On peut remplacer $|\psi, 0\rangle$ par $|\psi, b\rangle$ où b est une constante arbitraire indépendante de ψ.

réellement aléatoire alors au moins un des deux éléments `seed` ou `random` ne peut avoir de description finie (comme un entier de 128 bits ou un programme `C` de 10 Ko).

10.2.10 Impact non local d'actions locales

Pour illustrer les notions rencontrées, nous allons considérer un système composé de deux qubits (A, B) dans l'état suivant :

$$|A,B\rangle = \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle - \frac{1}{2}|11\rangle = \frac{1}{2}\begin{pmatrix} 1 \\ 1 \\ 1 \\ -1 \end{pmatrix}$$

Dans cet état, et après mesure suivant la base de calcul, on a donc la probabilité $|1/2|^2$ d'observer $|00\rangle$, $|1/2|^2$ d'observer $|01\rangle$, $|1/2|^2$ d'observer $|10\rangle$ et $|-1/2|^2$ d'observer $|11\rangle$. Bref, il y a exactement une chance sur quatre d'observer chacune des quatre possibilités.

On peut supposer que le qubit A est à disposition d'Alice et que le qubit B à disposition de Bob, Alice et Bob étant arbitrairement éloignés l'un de l'autre et dans l'incapacité de communiquer pendant le temps de l'expérience. En particulier, Alice a exactement une chance sur deux d'observer 0. Idem pour Bob, et pour la valeur 1.

Supposons que Bob décide d'appliquer l'opérateur $R(-\pi/4)$ à son qubit. Alice, quant à elle ne fait rien. Formellement elle applique l'opérateur Id_2. C'est une action locale qui se traduit par l'application de l'opérateur $\mathrm{Id}_2 \otimes R(-\pi/4)$ sur l'état $|A,B\rangle$.

Par définition de R (équation 10.1), on a :

$$R(-\pi/4) = \begin{pmatrix} \cos(-\pi/4) & -\sin(-\pi/4) \\ \sin(-\pi/4) & \cos(-\pi/4) \end{pmatrix} = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \tag{10.2}$$

L'opérateur appliqué au système est :

$$\mathrm{Id}_2 \otimes R(-\pi/4) = \begin{pmatrix} R(-\pi/4) & 0 \\ 0 & R(-\pi/4) \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & -1 & 1 \end{pmatrix}$$

Il suit qu'après la rotation de Bob, l'état du système devient :

$$|A,B\rangle' = (\mathrm{Id}_2 \otimes R(-\pi/4)) \cdot |A,B\rangle = \frac{1}{2} \cdot \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ -1 \end{pmatrix}$$

$$= \frac{1}{2\sqrt{2}} \begin{pmatrix} 2 \\ 0 \\ 0 \\ -2 \end{pmatrix} = \frac{1}{\sqrt{2}}|00\rangle - \frac{1}{\sqrt{2}}|11\rangle \,.$$

La probabilité qu'a Alice d'observer 1 vaut $|-1/\sqrt{2}|^2 = 1/2$, comme Bob. Alice et Bob ont aussi toujours une chance sur deux d'observer 0. En fait, la probabilité qu'a Alice d'observer une valeur donnée ne peut pas être influencée par l'action locale de Bob. Sinon, il y aurait transmission d'information [15]. Avant la mesure, et quoi qu'ait fait Bob localement, Alice ignore toujours l'information aléatoire codé par le qubit de Bob. D'ailleurs, peut-être que Bob a déjà fait la mesure sur son qubit... et Alice ne le sait pas.

En revanche, ce qui a changé c'est que conjointement Alice et Bob ne peuvent observer que 00 ou 11. Si Bob fait une mesure locale et observe 1, Alice mesurera également 1 de manière certaine. Ce n'était pas le cas pour l'état initial $|A,B\rangle$ où 01 et 10 pouvaient être observés. Bob a fait quelque chose qui fait qu'Alice et Bob observeront à coup sûr la même valeur. C'est donc les corrélations qui ont changées. Notons que le changement de corrélation est opérationnel dès que l'action de Bob sur son qubit est achevée, indépendamment de la distance séparant Alice de Bob.

Pour illustrer la situation, on pourrait imaginer qu'un grand nombre de particules intriquées, dans l'état ci-dessus, soit générées successivement. Pour fixer les idées supposons que les particules sont des pièces quantiques enfermées dans des boîtes, donnant pile ou face aléatoirement lorsqu'Alice et Bob ouvre leur boîte respectivement. On pourrait alors vérifier au fur et à mesure des générations de pièces intriquées que, quelle que soit la distance séparant les pièces quantiques, les quatre sorties possibles se produisent uniformément : PP, PF, FP, FF. Cependant, a un moment donné, Bob pourrait effectuer une opération sur sa pièce lui permettant d'affirmer à Alice : « cette fois je suis sûr certain que tu auras le même résultat que moi ». Cela paraît bien évidemment étonnant.

Pour résumer, une action locale peut changer les corrélations de manière instantanées [16] sans permettre de transmission d'information entre les parties du système. On dit parfois que le calcul quantique est *contextuel* : le calcul d'un système (ici réduit à

15. Alice, en mesurant ou estimant la probabilité, pourrait déterminer si Bob a appliqué son opérateur ou pas, et ce instantanément quelque soit la distance.

16. On a pu tester expérimentalement dans [SBB+08] « l'instantanéité » de ce changement et vérifier qu'il se produisait à une vitesse supérieur à 10^4 fois la vitesse de la lumière.

Alice) quantique peut être influencé par des actions locales extérieures à ce système (ici Bob). Cela ne peut pas se produire dans le cas de calcul classique.

10.3 Sur les inégalités de Bell

Les inégalités de John Stewart Bell prédisent les résultats d'une expérience, en terme de seuil de probabilité, pour que la théorie des variables locales cachées soit valide. On va revenir sur ces expériences qui ont été réalisées par Alain Aspect en 1982. Elles confirment que les inégalités de Bell sont effectivement violées, que la théorie des variables locales cachées ne peut expliquer les résultats expérimentaux.

FIGURE 10.11 – John Bell et Alain Aspect.

Les expériences d'Alain Aspect ont été successivement améliorées afin d'éliminer tous les « échappatoires [17] « (*loophole*) possibles à la non localité des effets quantiques, et en 2015 on considérait que tous les échappatoires avait été éliminés [HBD+15]. De très bonnes vidéos (part. 1 | part. 2 | part. 3 | part. 4) réalisées par L'Institut d'Optique retracent l'histoire de l'intrication quantique et revient sur ces découvertes.

10.3.1 L'expérience de pensée

Dans l'expérience de pensée (cf. figure 10.12 de gauche) un générateur C produit deux particules (des photons intriqués dans l'expérience d'Alain Aspect) qui partent en direction opposée vers des détecteurs A et B. Elles partent à une vitesse qui rend impossible toute communication entre elles pendant leur parcours. Les détecteurs, qui sont typiquement des polariseurs comme sur la figure 10.13, sont indépendants, sans lien de communication. Ils sont orientés selon les positions à 1, 2 ou 3 de façon aléatoire, indépendante et uniforme. Au passage d'une particule émise depuis C, chaque détecteur émet un flash soit rouge (R) soit vert (V). Il est important de remarquer que le choix du

17. Comme la certification de la source aléatoire ou l'effet de la distance (6m pour Alain Aspect), etc.

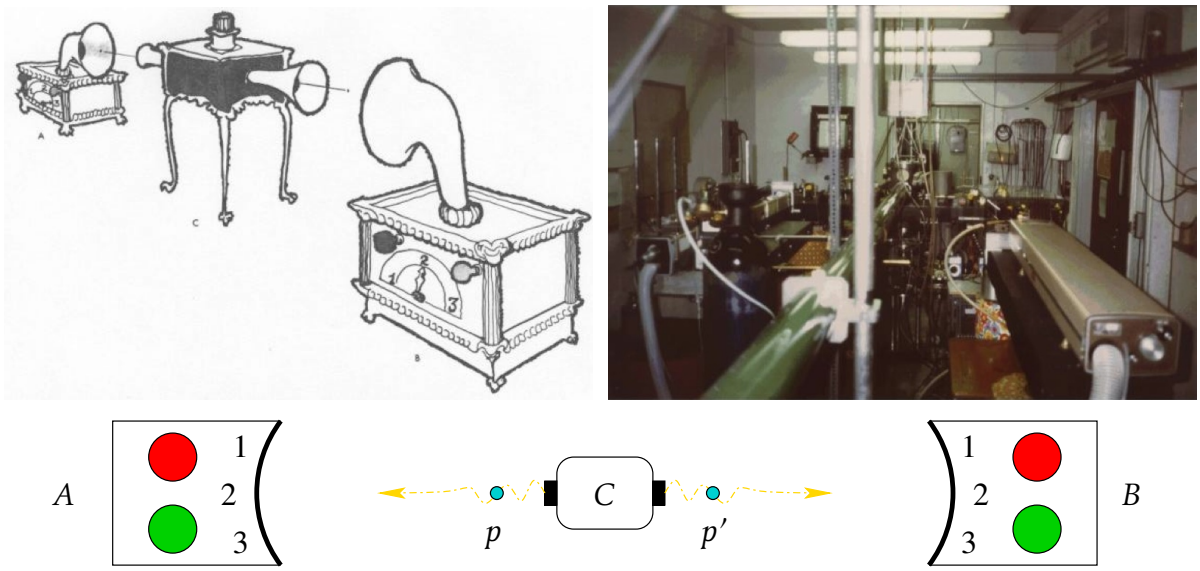


FIGURE 10.12 – Le choix 1-2-3 des détecteurs A et B (angle des polariseurs) est déterminé par le libre choix de l'expérimentateur après le départ des particules depuis le générateur C. Dans l'expérience réelle d'Alain Aspect en 1982 (à droite) le choix est déterminé par un générateur aléatoire quantique. Source [Mer85].

positionnement est fait après la génération des particules en C. Il n'y a donc pas de lien de causalité entre la position des détecteurs A et B, et la génération des particules en C.

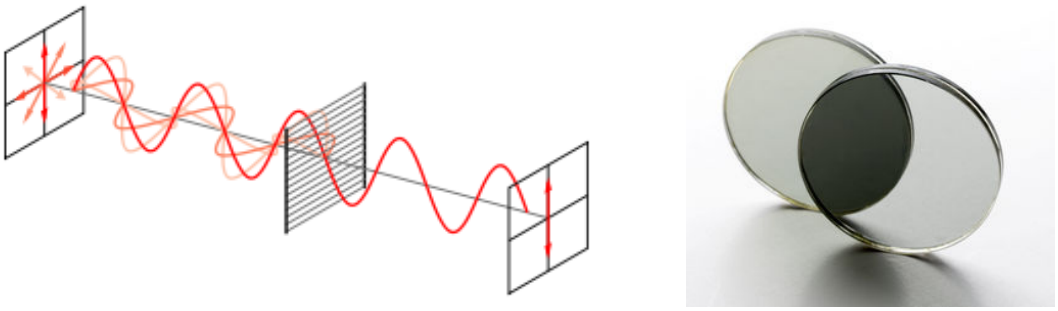


FIGURE 10.13 – Un polariseur permet de ne garder que les particules/ondes d'une orientation (polarité) donnée.

La table 10.1 présente les résultats hypothétiques d'une telle série d'expériences. Par exemple « 12VR » dans la table signifie que le réglage du détecteur A était sur 1 et a flashé V, et que le détecteur B était sur 2 et a flashé R.

12VR	32RV	32VR	22RR	33RR	12RV	22VV	13RV	21RV
22VV	11VV	31RV	22RR	21RV	21VV	21VR	11VV	13VR
11VV	31RV	32RV	21VR	31VR	31RV	12VR	23RV	11RR
21VR	13VR	13RV	33VV	21RR	12RV	33RR	23RV	23VR
33VV	33RR	12RR	21RV	31VR	32VR	23RR	13VV	23VV
33RR	12VR	12RV	31VV	33VV	13VV	32VR	32RR	23RV
11RR	31VR	22RR	11RR	23VR	23VR	22VV	13RR	33VV
22VV	11VV	12VR	32RV	32VV	31RR	13RV	22RR	11RR

TABLE 10.1 – Résultats d'une expérience de pensée[18] comme schématisée sur la figure 10.12 comportant 72 détections. La distribution des 9 paires de positions possibles (11, 12, ..., 33) est uniforme, soit 8 fois chacune. La distribution des flashs (RR, RV, VR, VV) est uniforme, soit 18 fois chacune. En particulier, 50% des détections (36 fois sur 72) flashent de la même couleur (RR ou VV). Les corrélations (mêmes positions et mêmes flashes) sont sous-lignées.

Les données ont deux caractéristiques (vérifiées pour la table 10.1) :

1. **Corrélation.** Lorsque les détecteurs sont sur les mêmes positions, leurs flashs sont toujours de couleur identique : 22VV, 33RR, 22RR, etc.
2. **Uniformité.** Lorsqu'on ne considère que les flashs, faisant abstraction de la position des détecteurs, le résultat est aléatoire uniforme. En particulier, la moitié du temps les flashs sont de la même couleur, la moitié du temps les flashs sont de couleur différente.

Il faut bien réalisé que de vraies expériences physiques montrent que ces deux caractéristiques se trouvent être réalisées simultanément. On donnera plus loin au paragraphe 10.3.2 une explication de cette observation grâce au formalisme quantique.

> **Hypothèse.** Supposons que ce que mesurent les détecteurs est une propriété intrinsèque des particules qui est fixée avant d'être détectée.

Cette propriété pourra être fixée par exemple lorsque les particules sont créées en C ou un peu plus tard pendant leur parcours de C aux détecteurs. En fait on supposera que la propriété est fixée avant de positionner les détecteurs (le choix de la mesure). C'est l'hypothèse de la théorie des variables locales cachées : les particules possèdent une propriété préexistante qui n'est accessible aux expérimentateurs seulement au moment de réaliser la mesure.

18. Il est assez facile de générer une telle distribution, et donc de se convaincre que la table 10.1 n'est pas qu'un accident ou juste impossible. Il suffit d'une part de générer le même nombre de chaque paire de positions (un multiple de 9), et d'autre part de générer le même nombre de chaque paire de flashs (un multiple de 4). Lorsque ces deux quantités de paires sont identiques (disons $9\times4 = 36$), il suffit d'apparier les paires de positions et paires de flashs de façon à respecter la corrélation.

Sous cette hypothèse et à cause de la caractéristique (1) des données, pour que les détecteurs flashent de la même couleur lorsqu'ils sont réglés de la même façon, il faut que les deux particules possèdent une propriété commune indiquant comment les détecteurs doivent se comporter. Si cela n'était pas le cas, puisqu'on génère suffisamment de particules et effectue suffisamment de détections avec des positions aléatoires, on devrait observer une différence de flash pour un même positionnement de détecteur (qui on le rappelle a lieu après la génération). Or cela n'arrive jamais. Les propriétés qu'on observent sont donc les mêmes pour chaque paire de particules. Entre deux générations successives, ces propriétés sont évidemment arbitraires mais identiques pour p et p'.

Un détecteur ne peut distinguer que 8 possibilités, et donc 8 propriétés d'une particule : un flash R ou V en position 1, un flash R ou V en position 2, et un flash R ou V en 3. Par exemple, une particule qui flasherait R en position 1, d'après notre hypothèse, possède cette propriété (de flasher R en position 1) avant la mesure. Si elle flashe V en position 2, c'est qu'elle possède aussi cette propriété. Enfin, si elle flashe R en position 3, c'est qu'elle possède cette propriété supplémentaire. Comme le détecteur est positionné après la génération, une telle particule doit posséder ces trois propriétés qu'on notera par RVR dans ce cas-ci. On dira plus simplement que la particule est de type RVR. Encore une fois, à cause de la caractéristique (1) des données les deux particules doivent avoir le même type, RVR par exemple, puisque sinon au bout d'un certain nombre de mesures les détecteurs pareillement positionnés mesureraient une différence. Il est exclus que le générateur C génère un même type seulement lorsque les détecteurs ont la même position car le positionnement est déterminé après la génération des particules. Encore une fois, sous notre hypothèse, chaque particule générée possède un type déterminé mais absolument inaccessible avant la mesure. Le type est aléatoire et gardé secret jusqu'à la mesure.

Pour le type RVR, les deux particules flasheront de la même couleur si les détecteurs sont en position 11, 22 ou 33, 13 ou 31. Ils flasheront de couleurs différentes pour les positions 12, 21, 23 et 32. Puisque les positions des détecteurs sont fixées indépendamment, aléatoirement et uniformément, chacune des 9 paires de positions (11, 12, 13, 21, ..., 33) se produit avec la même fréquence, soit 1/9. Donc les mêmes couleurs devraient flasher $5/9 = 1/2 + 1/18 > 55\%$ du temps pour le type RVR. Cela reste valable pour les six types ayant exactement deux lettres identiques, soient RRV, RVR, VRR, VVR, VRV, RVV. Pour les deux types restant RRR et VVV, les mêmes couleurs devraient flasher 100% du temps. Voir la table 10.2.

Au final, quelque soit la distribution du type des particules générées en C au cours de l'expériences, les détecteurs devraient flasher de la même couleur au moins 55% du temps. C'est incompatible avec la caractéristique (2) des données. Les distributions des résultats vérifiant (1) et (2) violent les inégalités de Bell.

Il suit que les résultats de cette expérience de pensée, qui ont été, et c'est le point fondamental, réellement vérifiés par Alain Aspect, sont incompatibles avec l'hypothèse que les particules possèdent, avant la mesure, une quelconque propriété. La propriété

type	positions	proba
RVR	11,22,33,13,31	5/9
VRV	11,22,33,13,31	5/9
RRV	11,22,33,12,21	5/9
VVR	11,22,33,12,21	5/9
VRR	11,22,33,13,31	5/9
RVV	11,22,33,23,32	5/9
VRR	11,22,33,23,32	5/9
RRR	toutes	9/9
VVV	toutes	9/9

TABLE 10.2 – Table donnant, en fonction du type de la particule, les positions des détecteurs qui aboutissent à un flash de la même couleur, ainsi que la probabilité de cet évènement.

de la particule est déterminée seulement au moment de sa mesure en A et est de plus corrélée au résultat de B.

Une présentation similaire de la même expérience de pensée peut-être trouvé sur le site de l'écrivain et astrophysicien Adam Becker. Cette fois il s'agit de jeu de roulettes dans un casino, cf. figure 10.14.

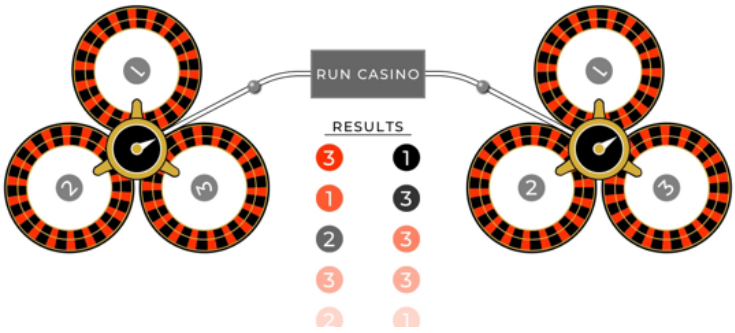


FIGURE 10.14 – Application interactive sur freelanceastro.github.io/bell permettant de simuler un lancer de billes intriquées dans un casino avec un système de tables de roulettes sélectionnables, dont le résultat est soit rouge soit noir.

Notons que c'est cette combinaison (propriété indéterminée et corrélation) qui est difficile à expliquer. La corrélation seule pourrait être expliquée par des variables locales cachées : quand on met un gant issu d'une même paire chacun dans une boîte fermée alors si on observe un gant droit dans l'une, on observera de façon certaine un gant gauche dans l'autre. De même il est concevable que la propriété mesurée ne soit pas déterminée avant la mesure, comme par exemple une pièce de monnaie tournante

sur une table qui se fixerait sur pile ou face quand elle n'aurait plus assez d'énergie (cf. figure 10.15) ou quand l'expérimentateur le déciderait.

FIGURE 10.15 – Pièce de monnaie dont l'état « pile » ou « face » n'est pas encore déterminé.

L'expérience de pensée que l'on vient de décrire ressemble en certains points au *Free Will Theorem* de John Conway *et al.* [CK06]. Sur la base de trois axiomes de physique assez simples (appelés SPIN [19], TWIN [20] et FIN [21]), il est démontré que si le choix de la mesure (la direction du spin d'une particule) est libre (*Free Will*), c'est-à-dire n'est pas fonction d'une information pré-déterminée accessible aux expérimentateurs (par exemple le passé de l'Univers), alors la réponse de la particule n'est pas non plus fonction d'une information pré-déterminée accessible aux particules. Dit autrement, si l'expérimentateur a le choix de la mesure, la particule a le choix du résultat : le résultat d'une mesure n'est pas pré-déterminé, la propriété mesurée n'existe pas avant sa mesure.

Ce phénomène de non déterminisme des propriétés quantiques des particules peut donc se déduire d'expériences de pensées basées sur des axiomes qui sont plus simples à admettre que le formalisme de la physique quantique.

10.3.2 L'explication

[*Cyril*. À FINIR]

On va maintenant vérifier par le calcul qu'en effet l'expérience de pensée, si on la réalise, produira des données vérifiant à la fois la propriété de corrélation et d'uniformité des flashs comme présentée ci-dessus.

Notons par X et Y les deux particules générées par C, juste après leur création mais avant leur détection en A et B. Ce sont des qubits et ils forment un système quantique

19. Quand l'expérimentateur fait trois mesures de directions mutuellement orthogonales, il obtient les résultats $1, 0, 1$ dans un ordre quelconque.
20. Si les expérimentateurs A et B font la même mesure de direction sur deux particules intriquées/jumelées, le résultat sera identique.
21. La vitesse de transmission de l'information est finie.

dans l'état intriqués suivant :

$$|X,Y\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\0\\1\end{pmatrix}.$$

Le choix de l'orientation effectuée dans les détecteurs revient à une mesure locale sur chacune des particules selon une base qui a été tournée d'un angle de 0°, 120° ou 240°.

[*Cyril*. À FINIR. On pourrait définir un jeu non-local, avec comme entrée $x, y \in \{1,2,3\}$ et comme sortie $a, b \in \{0,1\}$. Objectif : montrer qu'on obtient bien l'uniformité et la corrélation comme annoncé.]

[*Cyril*. D'ailleurs on pourrait introduire d'autres jeux non-locaux : $x, y \in \{0,1\}$ et $a, b \in \{0,1\}$ tels que si $x = y$ alors $a \neq b$ et sinon la sortie est aléatoire uniforme. Avec un état de $|01\rangle + |10\rangle$ on va pouvoir le faire, avec une mesure identique (selon la base de calcul $|0\rangle$ ou $|1\rangle$ suivant l'entrée) donc si $x = y$ ce qui va donner 01 ou 10 avec proba 1/2. Sinon, on aura des sorties aléatoires. Mais on peut le faire sans aucun effet quantique : Alice lit x et fait un xor avec un bit aléatoire r_i. Bob lit y et fait xor avec le bit aléatoire $1 - r_i$. Il semble que l'on ne peut pas le faire avec du non-shared randomness. NB : Dans les jeu non-locaux, c'est-à-dire sans communication entre les joueurs, partager un état quantique intriqué ou shared-randomness, partagent le fait que Alice et Bob, pour disposer de ces ressources, ont du de rencontrer.]

10.4 Téléportation

Il est possible de téléporter des états quantiques. Mais il faut bien l'avouer, les explications des réalisations expérimentales laissent perplexe (voir figure 10.16).

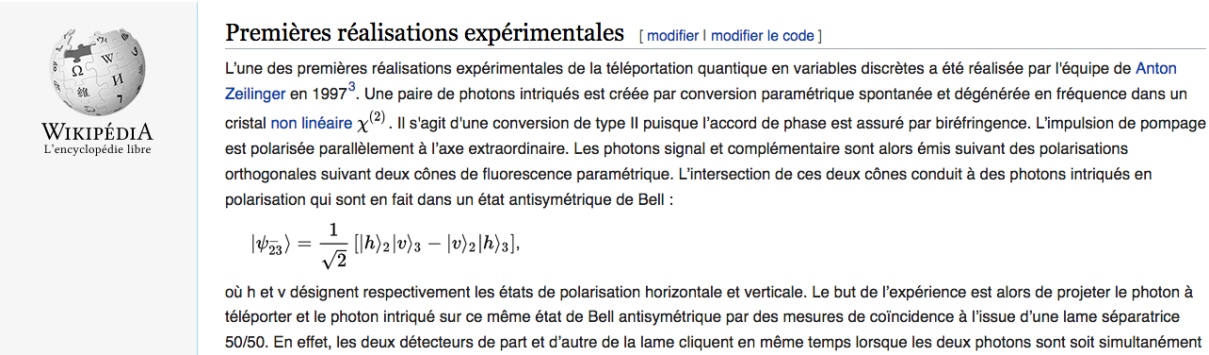


FIGURE 10.16 – Extrait, plutôt cryptique, de la page Wikipédia sur les expériences de téléportations quantiques.

Le principe est pourtant assez simple. [*Cyril*. À FINIR... Il faut qu'Alice fasse un CNOT entre le qubit dont elle veut téléporter l'état et son qubit intriqué avec celui

de Bob. Elle fait ensuite un Hadamard et une mesure qu'elle transmet à Bob par voie classique. Bob effectue alors un Hadamard en fonction du résultat d'Alice. Le qubit de Bob est alors dans le même état que celui d'Alice.] Pour cela il faut une paire de particules intriquées, donc *a priori* indiscernables et qui ont précédemment interagi. La téléportation est instantanée [16] quelle que soit la distance. Il faut bien noter que c'est l'état qu'on téléporte d'une particule à une autre. En rien on ne téléporte une particule, de l'énergie ou une information (voir figure 10.17).

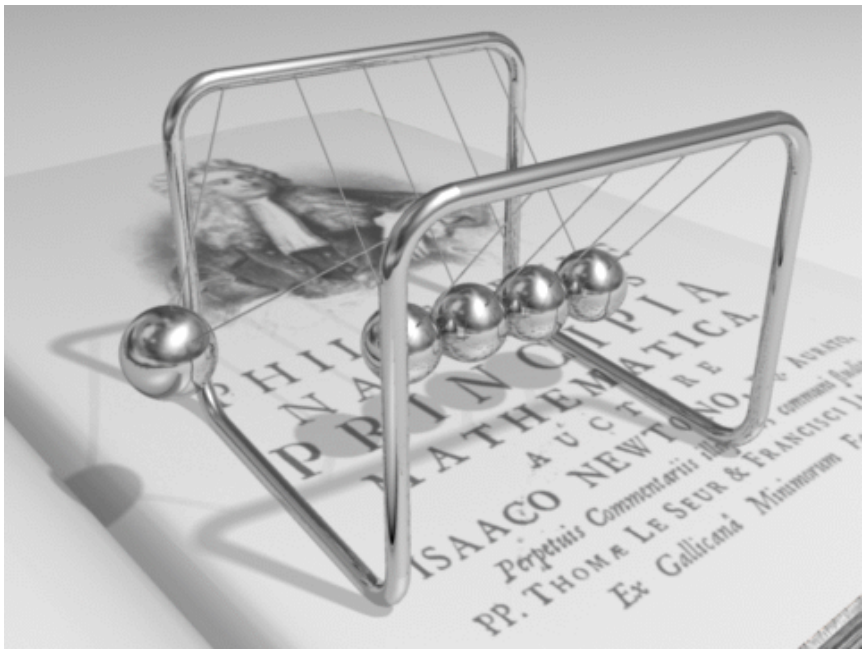


FIGURE 10.17 – Analogie avec le pendule de Newton pour comprendre que ce n'est pas la particule, support du qubit, qui se téléporte, mais son état (ici la quantité de mouvement de la bille à l'extrémité). On remarquera que : 1) les deux billes extrémités doivent être indiscernables vis à vis de l'observable (ici la masse pour préserver la quantité de mouvement) ; et 2) l'état initial qui est « téléporté » est aussi détruit. Bien évidemment, pour le pendule, il s'agit d'une transmission, et non d'une téléportation, de la quantité de mouvement qui ne se détruit pas mais plutôt s'annule. Voir l'animation.

10.5 Jeu CHSH

On va expliquer, grâce à une expérience de pensée (un jeu), la différence entre ressource aléatoire (partagée) et ressource quantique (intriquée). Ce jeu fait écho à l'expérience de pensée de Bell et l'expérience bien réelle d'Alain Aspect. Ce type de jeu, tout comme celui de la section 10.6, est parfois appelé jeu *pseudo-télépathique*, les joueurs semblant deviner comme par magie les cartes des autres joueurs.

Dans ce jeu, du noms des auteurs, Clauser-Horne-Shimony-Holt [CHSH69], il y a deux joueurs : Alice (A) et Bob (B) et ils ne peuvent pas communiquer (cf. figure 10.18). Il s'agit pour les joueurs de calculer un bit de sortie a et b en fonction d'un bit d'entrée x et y, et d'éventuellement de ressources extérieures accessibles seulement par les joueurs. Plus précisément, pour qu'Alice et Bob gagnent, if faut $a \oplus b = x \wedge y$. Dit autrement, il faut que $a = b$, sauf si $x = y = 1$.

Dans le cas déterministe, la ressource est réduite à l'algorithme seul. Dans le cas probabiliste, la ressource (en plus de l'algorithme) est alors une source de bits aléatoires partagées (*shared randomness*) inconnue de l'expérimentateur (secrète donc). Et dans le cas quantique, la ressource sont des qubits dans un état intriqué (*entangled qubits*).

Les physiciens nomment plus communément la source de bits aléatoires partagées par *variable locale cachée*, une quantité constante déterminée inaccessible aux expérimentateurs mais connues de toutes les particules.

Proposition 10.1 *Aucun algorithme déterministe ne permet à Alice et Bob de gagner au jeu CHSH à tous les coups.*

Preuve. Supposons qu'il existe un tel algorithme, et soient $a(x)$ et $b(y)$ les fonctions calculées par Alice et Bob ; dans le cas déterministe ces fonctions ne dépendent que des l'entrées, Alice et Bob ne pouvant pas communiquer. Pour gagner, il faut $a(x) \oplus b(y) = x \wedge y$. Les fonctions doivent donc vérifier les quatre équations suivantes, une pour chacune entrée (x, y) possibles :

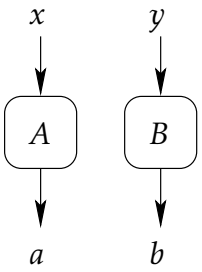


FIGURE 10.18 – Le jeu CHSH : pour gagner, le OU-exclusif de la sortie doit être égale au ET de l'entrée, soit $a \oplus b = x \wedge y$. Autrement dit, on doit avoir les mêmes sorties, sauf si les deux entrées sont à 1.

$$
\begin{aligned}
a(0) \oplus b(0) &= 0 \\
a(0) \oplus b(1) &= 0 \\
a(1) \oplus b(0) &= 0 \\
a(1) \oplus b(1) &= 1
\end{aligned}
$$

En utilisant le fait que $a \oplus b \equiv a + b \pmod 2$, la somme modulo deux, on peut transformer ces équations en sommes classiques et considérer le résultat modulo 2. La somme des termes des membres à gauche fait $2 \cdot (a(0) + a(1) + b(0) + b(1))$ ce qui est nécessairement paire. Malheureusement, la somme des termes des membres de droite est impaire. Il n'existe donc pas de fonctions $a(x)$ et $b(y)$ satisfaisant ces quatre contraintes. Il n'existe donc pas d'algorithme déterministe qui permet de gagner au jeu CHSH à tous les coups. □

En revanche, comme on va le voir, il existe un algorithme probabiliste qui résout CHSH avec une probabilité de 75% si Alice et Bob partagent un secret aléatoire. Notez bien qu'il faut arriver à gagner avec cette probabilité quelle que soit la distribution des entrées. Par exemple, répondre toujours 0, qui satisfait les contraintes dans $\frac{3}{4}$ des cas, ne marche que si la distribution des entrées est uniforme (même chance d'obtenir les quatre entrées (x, y) possibles). Alice et Bob se battent donc contre un adversaire (en fait l'expérimentateur) qui connaît par avance leur stratégie, et qui dans ce cas là proposera toujours $x = y = 1$ pour faire échouer à coup sûr l'algorithme d'Alice et Bob. Notez qu'on admet que l'adversaire, aussi fort soit-il, n'a pas accès à la ressource aléatoire commune (variables locales cachées).

Proposition 10.2 *Si Alice et Bob partagent des bits secrets aléatoires, alors il peuvent gagner au jeu CHSH dans 75% des cas, et ce quel que soit l'adversaire.*

Preuve. Pour atteindre 75% de réussite, il suffit que les joueurs, avant la proposition de chaque entrée (x,y) proposée par l'adversaire, se mettent d'accord sur une des quatre équations à ne pas satisfaire (en utilisant une même paire de bits aléatoires partagés), choix inconnu de l'adversaire. Il est alors toujours possible de résoudre le système composé des trois équations restantes.

Par exemple, s'ils décident de ne pas satisfaire la 4e équation soit $a(1) \oplus b(1) = 0$ au lieu de $a(1) \oplus b(1) = 1$, alors $a(x) = b(y) = 0$ est une solution. Ainsi, les joueurs perdent seulement si l'adversaire propose l'entrée (x,y) correspondant à l'équation choisie (ici $x = y = 1$), soit seulement dans $\frac{1}{4}$ des cas.

On peut alors vérifier que s'ils choisissent de ne pas satisfaire l'équation correspondant à l'entrée (x,y) :

- $(0,0)$, alors $a(x) = \neg x$ et $b(y) = y$ est une solution.
- $(0,1)$, alors $a(x) = 0$ et $b(y) = y$ est une solution.
- $(1,0)$, alors $a(x) = x$ et $b(y) = 0$ est une solution.
- $(1,1)$, alors $a(x) = 0$ et $b(y) = 0$ est une solution.

Donc quels que soient les choix de l'adversaire, Alice et Bob gagneront dans $\frac{3}{4}$ des coups. □

Mais peut-on faire mieux ? Et bien non.

Proposition 10.3 *Aucun algorithme probabiliste ne peut gagner au jeu CHSH avec plus de* 75% *de chance.*

Preuve. Un algorithme probabiliste est un algorithme qui, pour chaque entrée, fait un certain nombre de tirages aléatoires avant de renvoyer sa sortie. Ces choix aléatoires sont autant d'exécutions et de sorties possibles. En fait on peut regrouper les sorties possibles et voir un algorithme probabiliste comme un algorithme qui affiche chaque sortie avec une certaine probabilité.

Il y a au plus quatre fonctions $a(x)$ à une variable binaire [22] qui sont $0, 1, x, \neg x$. De même pour $b(y)$, si bien qu'Alice et Bob ont a leur disposition au plus 16 algorithmes ou paires de fonctions $(a(x), b(y))$. En fait, certaines paires de fonctions ont toujours les mêmes sorties, en particulier à cause de la symétrie de la formule $a(x) \oplus b(y)$ qui définit la sortie de l'algorithme. Mais pas seulement. Par exemple, la paire de fonctions $(a(x), b(y)) = (\neg x, \neg y) = (x, y)$ puisque $\neg x \oplus \neg y = x \oplus y$.

Ainsi, on peut vérifier qu'il n'existe que 8 fonctions de sorties différentes, qui sont (cf. la table ci-dessous) : 0, 1, x, $\neg x$, y, $\neg y$, $x \oplus y$, $\neg x \oplus y$.

22. De manière générale il y a 2^{2^n} fonctions $f : \{0,1\}^n \to \{0,1\}$ à n variables, cf. [Gav24][Section 2.6].

	0	1	x	$\neg x$
0	0	1	x	$\neg x$
1	1	0	$\neg x$	x
y	y	$\neg y$	$x \oplus y$	$\neg x \oplus y$
$\neg y$	$\neg y$	y	$\neg x \oplus y$	$x \oplus y$

L'algorithme probabiliste d'Alice et Bob choisit donc une de ces sorties suivant une certaine distribution de probabilité $p_1, \ldots, p_8$, p_i étant la probabilité de choisir la i-ème sortie. On a bien sûr que $\sum_i p_i = 1$. La distribution $p_1, \ldots, p_8$ détermine entièrement l'algorithme probabiliste d'Alice et Bob qui cherchent à optimiser les p_i pour maximiser leurs chances de succès quel que soit l'adversaire.

On notera P_{xy} la probabilité de succès de cet algorithme pour l'entrée (x, y). La table ci-dessous résume les différentes probabilités.

x	y	$x \wedge y$	0	1	x	$\neg x$	y	$\neg y$	$x \oplus y$	$\neg x \oplus y$	probabilité de succès
0	0	0	p_1	0	p_3	0	p_5	0	p_7	0	$P_{00} = p_1 + p_3 + p_5 + p_7$
0	1	0	p_1	0	0	p_4	p_5	0	0	p_8	$P_{01} = p_1 + p_4 + p_5 + p_8$
1	0	0	p_1	0	p_3	0	0	p_6	0	p_8	$P_{10} = p_1 + p_3 + p_6 + p_8$
1	1	1	0	p_2	p_3	0	p_5	0	0	p_8	$P_{11} = p_2 + p_3 + p_5 + p_8$

L'objectif est de montrer que, quel que soit le choix des probabilités $p_1, \ldots, p_8$, la probabilité de succès de l'algorithme d'Alice et Bob ne dépassera pas 75% pour les quatre entrées possibles. Dit autrement on veut montrer que :

$$\min\{P_{00}, P_{01}, P_{10}, P_{11}\} \leqslant \frac{3}{4}.$$

Pour cela on va calculer la somme $S = P_{00} + P_{01} + P_{10} + P_{11}$ et montrer que $S \leqslant 3$ ce qui permettra de conclure que le plus petit des quatre termes de la somme est $\leqslant 3/4$. C'est immédiat car :

$$S = 3p_1 + p_2 + 3p_3 + p_4 + 3p_5 + p_6 + p_7 + 3p_8 \leqslant 3 \cdot (p_1 + \cdots + p_8) = 3$$

ce qui termine la preuve. On notera que pour maximiser la probabilité de succès, Alice et Bob ont intérêt à choisir $p_1 = p_3 = p_5 = p_8 = 1/4$ et $p_2 = p_4 = p_6 = p_7 = 0$, ce qui revient à choisir aléatoirement une des quatre paires de fonctions : $0 = 0 \oplus 0$, $x = x \oplus 0$, $y = 0 \oplus y$ et $\neg x \oplus y$. C'est précisément la solution présentée dans la proposition 10.2. Dans ce cas on a $P_{00} = P_{01} = P_{10} = P_{11} = 3/4$. □

On va maintenant montrer que s'ils possèdent chacun un qubit formant un système dans un état intriqué alors ils peuvent gagner avec une probabilité strictement supérieure à 75%.

Proposition 10.4 *Si Alice et Bob disposent de ressources quantiques, alors ils peuvent gagner au jeu CHSH avec une probabilité de* $\cos^2(\pi/8) = (1+1/\sqrt{2})/2 > 85\%$.

Preuve. Cette fois-ci, Alice et Bob partagent des paires de qubits intriqués, une paire pour chaque nouvelle entrée de l'adversaire. Alice possède un qubit $|A\rangle$ et Bob un qubit $|B\rangle$. Les qubits ont interagit dans le passé, avant bien sûr que n'ai été fixée l'entrée (x,y) de l'adversaire, pour former le système :

$$|A,B\rangle \;=\; \frac{1}{\sqrt{2}}\left(|00\rangle - |11\rangle\right) .$$

L'algorithme, pour Alice, consiste à appliquer une rotation $R(\theta) = \begin{pmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{pmatrix}$ suivant un angle θ de $-\pi/16$ si $x=0$ et $3\pi/16$ si $x=1$, soit de manière générale un angle de $\theta_A = (4x-1)\pi/16$. La stratégie est identique pour Bob avec une rotation de $\theta_B = (4y-1)\pi/16$.

Une fois cette opération effectuée sur chacun de leur propre qubit, Alice et Bob affiche le résultat de la mesure dans la base de calcul, soit 0 ou 1 suivant que l'état devient $|0\rangle$ ou $|1\rangle$.

En terme de calcul, l'action d'Alice se traduit par l'application de l'opérateur $R(\theta_A)\otimes \mathrm{Id}_2$ (soit une matrice 4×4) à l'état $|A,B\rangle$ représentant l'état initial du système. De même pour Bob, son action va se traduire par l'application de l'opérateur $\mathrm{Id}_2 \otimes R(\theta_B)$ à $|A,B\rangle$.

Chacune des opérations de rotation affecte non seulement le qubit sur laquelle elle est effectuée mais le système en entier. On peut montrer que l'ordre d'application des opérateurs n'a pas d'influence sur l'état du système final (cf. paragraphe 10.2.8) si bien qu'on peut appliquer directement l'opérateur $R(\theta_A)\otimes R(\theta_B)$ à $|A,B\rangle$. L'état du système devient :

$$(R(\theta_A)\otimes R(\theta_B))\cdot|A,B\rangle \;=$$

$$\begin{pmatrix} \cos\theta_A \cdot \begin{pmatrix}\cos\theta_B & -\sin\theta_B \\ \sin\theta_B & \cos\theta_B\end{pmatrix} & -\sin\theta_A \cdot \begin{pmatrix}\cos\theta_B & -\sin\theta_B \\ \sin\theta_B & \cos\theta_B\end{pmatrix} \\ \sin\theta_A \cdot \begin{pmatrix}\cos\theta_B & -\sin\theta_B \\ \sin\theta_B & \cos\theta_B\end{pmatrix} & \cos\theta_A \cdot \begin{pmatrix}\cos\theta_B & -\sin\theta_B \\ \sin\theta_B & \cos\theta_B\end{pmatrix} \end{pmatrix} \cdot \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix}$$

En utilisant les formules

$$\begin{cases} \cos(x+y) \;=\; \cos x\cos y - \sin x \sin y \\ \sin(x+y) \;=\; \sin x \cos y + \cos x \sin y \end{cases}$$

l'état se simplifie en :

$$(R(\theta_A)\otimes R(\theta_B))\cdot|A,B\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}\cos\theta_A\cos\theta_B-\sin\theta_A\sin\theta_B\\ \cos\theta_A\sin\theta_B+\sin\theta_A\cos\theta_B\\ \sin\theta_A\cos\theta_B+\cos\theta_A\sin\theta_B\\ \sin\theta_A\sin\theta_B-\cos\theta_A\cos\theta_B\end{pmatrix} = \frac{1}{\sqrt{2}}\cdot\begin{pmatrix}\cos(\theta_A+\theta_B)\\ \sin(\theta_A+\theta_B)\\ \sin(\theta_A+\theta_B)\\ -\cos(\theta_A+\theta_B)\end{pmatrix}$$
$$= \frac{1}{\sqrt{2}}(\cos(\theta_A+\theta_B)(|00\rangle-|11\rangle)+\sin(\theta_A+\theta_B)(|01\rangle+|10\rangle))$$

Il faut maintenant vérifier que la probabilité de succès, c'est-à-dire d'avoir $a\oplus b = x\wedge y$, est bien de $\cos^2(\pi/8)$.

La probabilité d'obtenir $a\oplus b=0$ est la probabilité d'obtenir $|00\rangle$ ou $|11\rangle$, soit

$$2\cdot\left(\frac{1}{\sqrt{2}}\cos(\theta_A+\theta_B)\right)^2 = \cos^2(\theta_A+\theta_B)\,.$$

Si l'entrée est $x=0$ et $y=0$, alors la probabilité d'avoir $a\oplus b = x\wedge y = 0$ vaut $\cos^2(-2\pi/16)=\cos^2(\pi/8)$.

Si l'entrée est $x=1$ et $y=0$ (ou $x=0$ et $y=1$, les formules sont symétriques en x et y), alors la probabilité d'avoir $a\oplus b = x\wedge y = 0$ vaut $\cos^2(3\pi/16-\pi/16)=\cos^2(\pi/8)$.

La probabilité d'obtenir $a\oplus b=1$ est la probabilité obtenir $|01\rangle$ ou $|10\rangle$, soit $\sin^2(\theta_A+\theta_B)$.

Si l'entrée est $x=y=1$, alors la probabilité d'avoir $a\oplus b = x\wedge y = 1$ vaut $\sin^2(6\pi/16)=\sin^2(3\pi/8)=\cos^2(\pi/8)$, car sur le cercle unité on s'aperçoit que les angles $3\pi/8$ et $\pi/8$ sont symétriques par rapport à l'angle $\pi/4$ (voir figure 10.19). □

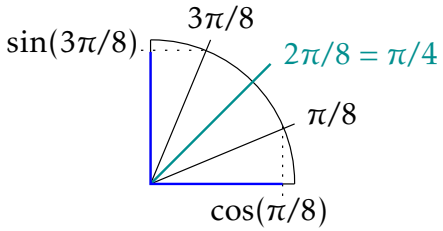


FIGURE 10.19 – Géométriquement, par symétrie autour de l'axe d'angle $\pi/4$, on remarque que $\cos(\pi/8)=\sin(3\pi/8)$.

On peut même montrer qu'aucun algorithme quantique ne peut le faire avec une probabilité plus grande que $\cos^2(\pi/8)$. C'est la borne de Tsirelson [Cir80]. Il faut pour cela utiliser de l'algèbre linéaire (et des valeurs propres) dans les espaces de Hilbert, et on ne le fera pas dans ce cours.

[*Exercice.* 1 Montrer qu'Alice et Bob peuvent gagner au jeu CHSH avec 50% de chance s'ils ont accès à une ressource probabilistes non partagées. 2* Montrer que 50% est la meilleure probabilité possible.]

10.6 Jeu GHZ

Il s'agit en fait d'une variante simplifiée du jeu GHZ, du nom des auteurs Greenberger-Horne-Zeilinger [GHZ89], voir aussi [Boy04]. Il y a maintenant trois joueurs : Alice (A), Bob (B) et Carole (C). On peut en fait le généraliser à n joueurs. Comme précédemment ils ne peuvent communiquer avant de recevoir les entrées qui sont des valeurs booléennes, respectivement x, y, z. Ils doivent sortir des valeurs a, b, c respectivement avec leurs propres resources, mais sans communiquer après l'arrivée des entrées (figure 10.20).

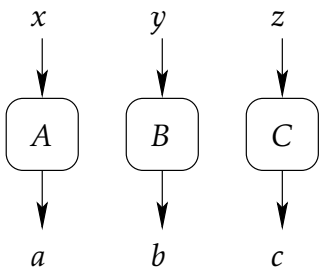


FIGURE 10.20 – Le jeu GHZ : pour gagner, la parité de la somme de la sortie doit être égale à la parité de la demi-somme de l'entrée, sachant que la somme de l'entrée est toujours paire. Ce jeu peut être généralisé à n joueurs.

Il y a une contrainte (promesse) sur les entrées : leur somme doit être paire. Pour gagner, il faut que la parité de la somme de la sortie soit égale à la parité de la demi-somme de l'entrée. Autrement dit, $x+y+z \in \{0,2\}$ et $a+b+c \equiv \frac{1}{2}(x+y+z) \pmod 2$. Pour résumer :

x	y	z	$\frac{1}{2}(x+y+z)$
0	0	0	0
1	1	0	1
1	0	1	1
0	1	1	1

ce qui revient aussi à distinguer le cas $xyz = 000$ des trois autres.

Soient $a(x)$, $b(y)$ et $c(z)$ les fonctions calculées par Alice, Bob et Carole. Alors, pour gagner, les fonctions doivent donc vérifier les quatre équations suivantes :

$$\begin{aligned} a(0)+b(0)+c(0) &\equiv 0 \\ a(1)+b(1)+c(0) &\equiv 1 \\ a(1)+b(0)+c(1) &\equiv 1 \\ a(0)+b(1)+c(1) &\equiv 1 \end{aligned}$$

De manière similaire au jeu précédent, la somme des termes de gauche fait $2\cdot(a(0)+a(1)+b(0)+b(1)+c(0)+c(1))$ ce qui est paire. Malheureusement, la somme des termes de droite est impaire. Il n'existe donc pas de fonctions $a(x), b(y), c(z)$ satisfaisant toutes les contraintes.

Une stratégie, similaire au jeu précédent, atteint 75% pour $n = 3$, où il suffit que tous les joueurs choisissent, conjointement via 2 bits aléatoires partagés, une des quatre contraintes à ne pas respecter, et résolvent ainsi le système modifié. Par exemple, si c'est la 3e équation que les joueurs décide de ne pas respecter, alors $a(x) = c(z) = 0$ et $b(y) = y$ est une solution. Et si c'est la 1ère, c'est encore plus simple, puisque $a(x) = b(y) = c(z) = 1$ est une solution.

De manière générale, on peut montrer qu'aucune stratégie, même celle utilisant des bits aléatoires partagées, ne peut gagner à ce jeu avec une probabilité plus grande que $\frac{1}{2} + \left(\frac{1}{2}\right)^{\lceil n/2 \rceil}$ où n est le nombre de joueurs. Pour $n = 3$, cela fait $\frac{1}{2} + \left(\frac{1}{2}\right)^2 = \frac{3}{4} = 75\%$.

Si maintenant les trois participants Alice, Bob et Carole, possèdent chacun un qubit préalablement intriqués dans l'état

$$|A,B,C\rangle \;=\; \frac{1}{2}|000\rangle - \frac{1}{2}|011\rangle - \frac{1}{2}|101\rangle - \frac{1}{2}|110\rangle$$

alors ils peuvent gagner à 100% (cf. [BCMdW10, §A.]). L'algorithme consiste à faire une mesure sur chacun des qubits selon la base de calcul si l'entrée vaut 0 et selon la base d'Hadamard $\{H|0\rangle, H|1\rangle\} = \left\{\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle), \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right\}$ si l'entrée vaut 1. Il se trouve que

$$H \;=\; \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \;=\; R(\pi/4) \qquad \text{(cf. l'équation (10.2).)}$$

On peut montrer que mesurer selon la base d'Hadamard revient en fait à appliquer H puis à effectuer une mesure selon la base de calcul.

Si $xyz = 000$, les trois mesures de l'état $|A,B,C\rangle$ se font selon la base de calcul et donc donneront $a + b + c \equiv 0$ dans 100% des cas car l'état est une superposition d'états ayant un nombre pair de 1. Si $xyz = 011$, cela revient à faire une mesure selon la base de calcul

sur l'état :

$$\begin{aligned}(I\otimes H\otimes H)\cdot|A,B,C\rangle &= (I\otimes H\otimes H)\cdot\left(\frac{1}{2}|000\rangle-\frac{1}{2}|011\rangle-\frac{1}{2}|101\rangle-\frac{1}{2}|110\rangle\right)\\ &= (I\otimes H\otimes H)\cdot\left(\frac{1}{2}|0\rangle(|00\rangle-|11\rangle)-\frac{1}{2}|1\rangle(|01\rangle+|10\rangle)\right)\\ &= \frac{1}{2}|0\rangle(|01\rangle+|10\rangle)-\frac{1}{2}|1\rangle(|00\rangle-|11\rangle)\\ &= \frac{1}{2}|001\rangle+\frac{1}{2}|010\rangle-\frac{1}{2}|100\rangle+\frac{1}{2}|111\rangle\end{aligned}$$

et donc $a+b+c\equiv 1$ après mesures dans 100% des cas, puisqu'il s'agit d'une superposition d'état ayant un nombre impair de 1. Les autres cas ($xyz=101$ et $xyz=110$) se traitent de manière similaire.

Il est à noter que le résultat n'est pas déterministe : pour les mêmes entrées la sortie peut être différente. Cependant les joueurs gagnent le jeu à coup sûr.

10.7 Le modèle φ-LOCAL

Dans un jeu à deux joueurs, l'hypothèse *non-signalling* signifie que le choix de mesure d'un joueur n'est pas *signalé* à l'autre. Et en physique le choix de la mesure est l'analogue de l'entrée en informatique, la donnée qui va déterminer le calcul. Donc dans un modèle de calcul multi-joueurs respectant l'hypothèse *non-signalling*, la sortie d'un joueur ne peut dépendre de l'entrée d'un autre sans communication. Il est par contre autorisé que les sorties soient corrélées. C'est précisément ce qu'il se passe dans le modèle quantique : il y a corrélation (entre sorties distantes) sans pour autant être *signalling* (entre entrées et sorties distantes).

Dans cette extension *non-signalling* du modèle local, nommée φ-LOCAL comme *physical locality*, on s'autorise toutes les correlations (potentiellement non locales) des résultats, sans toute fois autoriser celles qui découlent d'une communication instantanée à distance. Donc dans le modèle φ-LOCAL à distance t on s'autorise tout sauf de communiquer au-delà de nœuds à distance t en t unités de temps (ou rondes). La non communication entre deux nœuds quelconques suffisamment loin s'exprime par le fait que les sorties de l'un d'entre eux sont indépendantes des entrées de l'autre. Et plus généralement, les sorties conjointes d'un groupe de nœuds (on parle de distribution des marginales) sont indépendantes des entrées de tout groupe de sommets suffisamment éloignés.

Plus formellement, lorsque deux sommets A et B ne sont pas en mesure de communiquer, cela signifie que pour une entrée (x,y), le résultat (a,b) doit vérifier :

$$\Pr(a|x,y)=_{\text{def}}\sum_b \Pr(a,b|x,y)=\Pr(a|x)\quad\text{et}\quad \Pr(b|x,y)=_{\text{def}}\sum_a \Pr(a,b|x,y)=\Pr(b|y)\,.$$

Dit autrement, la première somme doit être indépendante de y alors que la seconde indépendante de x.

Par exemple, la distribution suivante vérifie cette propriété, étant sous-entendu que les lignes absentes arrivent avec une probabilité nulle comme par exemple $\Pr(a=0,b=1|x=0,y=0)=0$. Pour résumer la distribution, on sort 00 ou 11 avec probabilité 1/2 sauf lorsqu'on a $x=y=1$. Dans ce cas on sort 01 ou 10 avec probabilité 1/2.

x	y	a	b	Pr
0	0	0	0	1/2
		1	1	1/2
0	1	0	0	1/2
		1	1	1/2
1	0	0	0	1/2
		1	1	1/2
1	1	0	1	1/2
		1	0	1/2

En effet, $\Pr(a=0|x,y)=1/2$ quels que soient x et y. De même $\Pr(a=1|x,y)=1/2$. Donc $\Pr(a|x,y)=\Pr(a|x)$. Et de manière similaire, on vérifie que $\Pr(b=0|x,y)=\Pr(b=1|x,y)=1/2$ quels que soient x et y. Cette distribution est donc *non-signalling* : a et b sont corrélées, mais à partir de a on ne tire aucune information sur y, et à partir de b on ne tire aucune information sur x.

On remarquera que $a \oplus b = x \wedge y$. Autrement dit, dans ce modèle, Alice et Bob gagnent au jeu CHSH avec une probabilité de 100%.

[*Cyril*. À FINIR]

On peut vérifier que les distributions issues des propositions 10.2 et 10.4 sont également *non-signalling*.

...

On peut étendre la propriété de non communication (ou *non-signalling*) de la manière suivante. Tout d'abord une *tâche* sur un graphe G de sommets $v_1,\dots,v_n$ est une fonction faisant correspondre, pour tout sommet v_i, une entrée x_i avec une sortie a_i. On notera $\vec{x}=(x_1,\dots,x_n)$ et $\vec{a}=(a_1,\dots,a_n)$.

...

Bibliographie

[Bar06] J. Barrett, *Information processing in generalized probabilistic theories*, Tech. Rep. 0508211v3 [quant-ph], arXiv, November 2006.

[BCMdW10] H. Buhrman, R. Cleve, S. Massar, and R. de Wolf, *Nonlocality and communication complexity*, Reviews of Modern Physics, 82 (2010), pp. 665–697. doi : 10.1103/RevModPhys.82.665.

[Boy04] M. Boyer, *On Mermin's n-player games parity game*, August 2004. http://www.iro.umontreal.ca/~boyer/mermin.

[CHSH69] J. F. Clauser, M. A. Horne, A. Shimony, and R. A. Holt, *Proposed experiment to test local hidden-variable theories*, Physical Review Letters, 23 (1969), pp. 880–883. doi : 10.1103/PhysRevLett.23.880.

[Cir80] B. Cirel'son, *Quantum generalizations of Bell's inequality*, Letters in Mathematical Physics, 4 (1980), pp. 93–100. doi : 10.1007/BF00417500.

[CK06] J. H. Conway and S. Kochen, *The free will theorem*, Foundations of Physics, 36 (2006), pp. 1441–1473. doi : 10.1007/s10701-006-9068-6.

[DHFRW20] E. Dable-Heath, C. J. Fewster, K. Rejzner, and N. Woods, *Algebraic classical and quantum field theory on causal sets*, Tech. Rep. 1908.01973v3 [math-ph], arXiv, February 2020.

[FGZ+19] Y. Y. Fein, P. Geyer, P. Zwick, F. Kiałka, S. Pedalino, M. Mayor, S. Gerlich, and M. Arndt, *Quantum superposition of molecules beyond 25 kDa*, Nature Physics, 15 (2019), pp. 1242–1245. doi : 10.1038/s41567-019-0663-9.

[Gav24] C. Gavoille, *Analyse d'algorithme – Cours d'introduction à la complexité paramétrique et aux algorithmes d'approximation*, 2024. http://dept-info.labri.fr/~gavoille/UE-AA/cours.pdf. Notes de cours.

[GHZ89] D. M. Greenberger, M. A. Horne, and A. Zeilinger, *Going beyond Bell's Theorem*, in Bell's Theorem, Quantum Theory, and Conceptions of the Universe, Kluwer, 1989, pp. 69–72.

[HBD+15] B. Hensen, H. Bernien, A. E. Dréau, A. Reiserer, N. Kalb, M. S. Blok, J. Ruitenberg, R. Vermeulen, R. N. Schouten, C. Abellán, W. Amaya, V. Pruneri, M. W. Mitchell, M. Markham, D. J. Twitchen, D. Elkouss, S. Wehner, T. H. Taminiau, and R. Hanson, *Loophole-free Bell inequality violation using electron spins separated by 1.3 kilometres*, Nature, 526 (2015), pp. 682–686. doi : 10.1038/nature15759.

[Kal19] G. Kalai, *The argument against quantum computers*, Tech. Rep. 1908.02499 [quant-ph], arXiv, August 2019.

[Mer85] N. D. Mermin, *Is the moon there when nobody looks ? Reality and the quantum theory*, Physics Today, 38 (1985), pp. 38–47. doi : 10.1063/1.880968.

[RP00] E. Rieffel and W. Polak, *An introduction to quantum computing for non-physicists*, ACM Computing Surveys, 32 (2000), pp. 300–335. doi : 10.1145/367701.367709.

[SBB+08] D. Salart, A. Baas, C. Branciard, N. Gisin, and H. Zbinden, *Testing the speed of 'spooky action at a distance'*, Nature, 454 (2008), pp. 861–864. doi : 10.1038/nature07121.

[Teo16] M. Teodorani, *Entanglement : l'intrication quantique, des particules à la conscience*, Macro Edition, 2016.