Ordres et Parcours de Graphes

V. Limouzy

Ordres et parcours de graphes

Vincent LIMOUZY

10 Juin 2005

Mémoire de DEA

161, rue Ada - F. 34394 Montpellier cedex 5 - Tél. 33 (0) 4 67 41 85 85 - Fax. 33 (0) 4 67 41 85 00 - www.lirmm.fr

ACADÉMIE DE MONTPELLIER

UNIVERSITÉ MONTPELLIER II

— SCIENCES ET TECHNIQUES DU LANGUEDOC —

DIPLÔME D'ÉTUDE APPROFONDIE

présenté à l'Université des Sciences et Techniques du Languedoc

SPÉCIALITÉ	:	**INFORMATIQUE**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Information, Structures, Systèmes**

ORDRES ET PARCOURS DE GRAPHES

par

Vincent LIMOUZY

10 Juin 2005

M. Michel HABIB, professeur, LIRMM : ..... Directeur de Stage
MMe Jocelyne NANARD, professeur, LIRMM : ..... Responsable de Formation
_____:.....Président du jury
_____:.....Rapporteur
_____:.....Rapporteur

Remerciements

Je tiens tout d'abord à remercier Michel HABIB, pour m'avoir permis d'effectuer ce stage sous sa direction, pour la qualité de son encadrement, sa présence, pour m'avoir guidé et encouragé et pour sa bonne humeur communicative.

Je tiens également à remercier Cristophe PAUL, pour sa disponibilité et pour ses conseils avisés.

Je remercie les thésards de l'équipe VAG, notamment Bình Minh BÙI XÛAN, pour son attention et ses conseils, ainsi que Cristophe CRESPELLE pour ses conseils, son humour, et pour m'avoir permis d'occuper, occasionnellement, son bureau (ressource très convoitée au LIRMM).

Je remercie également mes camarades de promotion (ou compagnons de galère), Guillaume BAGAN, pour ses conseils, et son humour (parfois involontaire), Heinrich HOERDEGEN, Clément SAAD, Nicolas MOREAU et Alexandre SVETOSLAVSKY, Leïla AOUATI, Thibaud ZAMORA. Et tout ceux que j'oublie ...

Je suis profondément reconnaissant à Donald Erwin KNUTH, pour avoir conçu le système TEX, ainsi qu'à Leslie LAMPORT pour avoir surenchéri avec LATEX. Sans qui ce rapport n'aurait pu être rédigé de la sorte.

Table des matières

Introduction

L'objectif de ce stage était d'établir un lien entre l'intelligence artificielle et la recherche opérationnelle d'une part et l'algorithmique discrète avancée d'autre part.

Plus précisémment il s'agissait d'étudier le problème des plus courts chemins dans un graphe selon deux points de vue.

Le premier, aux confins de l'intelligence artificielle et de la recherche opérationnelle étudie des modélisations par des graphes valués à l'aide d'ordres partiels [Spa03, PS04, MS03]. Les valuations peuvent être des préférences, ou simplement imprécises (telles que des intervalles). La question de l'algorithmique efficace se pose sur ces modèles.

Le second, provient de la longue traque des algorithmiciens depuis Dijkstra (Fredman et Tarjan [FT87], Thorrup [Tho97], Meyer [Mey03], Goldberg [Gol01] et enfin Hagerup [Hag04])) d'un algorithme simple et linéaire pour calculer les plus courts chemins issus d'un sommet. Malgré tous les efforts engagés, il semblerait qu'un goulet d'étranglement soit le tri des valuations. Les algorithmes récents évitent ce tri et celui de Hagerup, pour ce faire ne calcule qu'un ordre partiel sur les sommets.

Ces deux approches se rejoignent sur la notion de valuation partiellement ordonnées.

Après une brève introduction et un rappel de l'algorithme de Dijkstra, nous établissons une généralisation de Dijkstra valable sur des valuations partiellement ordonnées et nous montrons que l'algorithme d'Hagerup rentre dans ce cadre.

Le document est structuré en deux parties, la première concerne le problème des plus courts chemins, au chapitre 1 nous rappelons l'algorithme de Dijkstra, au chapitre 2 nous présentons l'algorithme d'Hagerup et fournissons une preuve de bon fonctionnement, au chapitre 3 nous proposons une généralisation de l'algorithme de Dijkstra, et enfin au chapitre quatre nous abordons la question du tri.

La seconde partie est consacré à une question posée par le Professeur Derek G. CORNEIL (Toronto) [CK04], le calcul efficace des conditions qui caractérisent les parcours de graphes. Nous répondont à cette question que partiellement. Au chapitre 5, nous présentons la caractérisation des parcours de graphe de Corneil-Krueger, ensuite au chapitre 6, nous présentons des algorithmes robustes pour vérifier ces conditions, et enfin au chapitre 7 nous présentons des vérificateurs plus rapides.

Enfin les annexes comportent les résultats que nous avons obtenus sur les arbres couvrant de poids minimum, ainsi que des problèmes de plus courts chemins aux valuations inhabituelles, qui sont NP-Difficile.

Une partie des ces résultats a fait l'objet d'un article qui a été soumis à la conférence EuroComb 2005.

Première partie

Ordres partiels et plus courts chemins

Chapitre 1

Rappels sur les plus courts chemins

1.1 L'algorithme de Dijkstra

Cet algorithme présenté par Edsger W. Dijkstra dans [Dij59] en 1959, permet de calculer pour un graphe valué et un sommet de départ (sommet source noté s) donné, les plus courts chemins reliant ce sommet de départ à tous les autres sommets. Cet algorithme est dit glouton : à chaque étape, l'algorithme effectue un choix, qui ne sera pas remis en cause par la suite. Son mode de fonctionnement s'apparente à un parcours de graphe (largeur, profondeur ...), et s'appuie sur les valuations des arêtes pour guider sa progression. En effet, le principe de Dijkstra consiste à prendre à chaque étape, le sommet du cocycle, non encore exploré, qui a la plus petite valeur. Augmentant ainsi le nombre de sommets sélectionnés.

Algorithme 1 : Algorithme de Dijkstra

Data : G=(X,U) : un graphe orienté ;une fonction coût $\omega : U \to \mathbb{R}^+$
$s \in V$ sommet source
Result : Une arborescence de plus courts chemins enracinée en s

```
1  Fermés← ∅ ; Ouverts← {s}
2  Parent(s) ← Nil ; d(s) ← 0
3  ∀y ∈ V, y ≠ s , d(y) ← ∞
4  while Ouverts ≠ ∅ do
5      z ← min(Ouverts)
6      Ajout(z, Fermés)
7      Enlever(z, Ouverts)
8      foreach v ∈ V avec (z,v) ∈ E do
9          if v ∈ Fermés then
10             return
11         if v ∉ Ouverts then
12             Ajout(v, Ouverts)
13         if d(z) + ω(z, v) < d(v) then
14             parent(v) ← z ; d(v)← d(z)+ω(z, v)
```

1.2 Invariants de l'algorithme

Nous présentons maintenant quels sont les invariants vérifiés par l'algorithme de Dijkstra (un liste plus exhaustive est disponible dans [HS91]).

Invariant Principal *Pour tous sommets x dans* **Fermés** $\cup$ **Ouverts**, $d(x)$ *est égal à la longueur du plus court chemin allant de s à x en utilisant seulement des sommets appartenant à* **Fermés**, *sauf éventuellement x qui peut appartenir à* **Ouverts**.

Remarque 1 *Cet invariant constitue un élément suffisant pour assurer la validité de l'algorithme. En effet après n itérations, l'invariant nous garantit que les distances calculées sont celles des plus courts chemins de s vers tous les autres sommets.*

1.3 Complexité et Implémentation

L'implémentation naïve de cet algorithme, aboutit à une version dont le temps d'exécution est de l'ordre de $O(n^2)$, où n représente le nombre de sommets du graphe. Cependant, des versions plus astucieuses ont été proposées, notamment par Tarjan et Fredman dans [FT87], en gérant l'ensemble des **Ouverts** avec un tas de Fibonacci. Cette implémentation calcule le plus court chemin en $O(m + nlogn)$. Un récapitulatif des implémentations et de leur complexités respectives peut être trouvé dans [AMO93].

La question d'une implémentation linéaire de l'algorithme de Dijkstra dans le pire des cas reste ouverte. Bien que Thorup présente dans [Tho97] un algorithme linéaire, cet algorithme n'est pas implémentable sur les modèles de machines traditionnels (Word RAM,...). Il existe cependant des algorithmes linéaires en temps moyen tel que que Meyer [Mey03] en 2001 Goldberg [Gol01] ou encore Hagerup en 2004 [Hag04]. Au chapitre suivant nous présenterons ce dernier, afin d'étudier son fonctionnement.

Chapitre 2

L'algorithme de Hagerup

2.1 Introduction : "Historique" L'algorithme

Hagerup fournit dans [Hag04] un algorithme pour résoudre le problème du plus court chemin dans un graphe orienté. A la différence de Dijkstra, Hagerup se place dans un contexte où les valuations sur les arcs sont des réels entre 0 et 1, la fonction de poids des arêtes est la suivante $\omega : A \rightarrow [0, 1]$.

L'algorithme de Hagerup, peut être vu comme une relaxation de l'algorithme de Dijkstra (cf Algorithme 1 p. 3). En effet l'algorithme emploie une loi de visite des sommets assimilable à celle de Dijskstra mais qui n'est pas identique. Cette loi de visite s'appuie sur une observation formulée par Dinitz ([Din78], en russe dans le texte), montrant ainsi dans quelle mesure nous pouvons "relaxer" ou "altérer" l'algorithme de Dijkstra.

L'algorithme de Hagerup permet de calculer les plus courts chemins à source unique, en temps linéaire moyen à condition que la distribution des poids sur les arcs soit uniforme.

2.2 Définitions et Algorithme

2.2.1 Définitions spécifiques

Donnée : $G = (V, A)$ un graphe orienté, avec une pondération sur les arcs comprise entre 0 et 1. Nous représentons par n le nombre de sommets ($|V|$) et par m le nombre d'arcs du graphe ($|A|$).
Pour poursuivre nous avons besoin d'introduire quelques notations :

- k = $\lfloor log_2 m \rfloor$
- $\Delta = \frac{1}{k}$
- $\widehat{d}(u) = \begin{cases} d(u) & \textbf{si } u \in V_1 \\ \lfloor \frac{d(u)}{\Delta} \rfloor \Delta & \textbf{sinon} \end{cases}$

Nous définissons les ensembles V_1 et V_2, deux sous ensembles des sommets du graphe, de la manière suivante : V_1 est constitué de sommets qui ont au moins un arc entrant dont le poids est inférieur à Δ, et V_2 l'ensemble des sommets dont tous les arcs entrants sont de poids supérieur ou égal à Δ (en clair, $V_2 = V \setminus V_1$) par conséquent V_1 et V_2 partitionne l'ensemble des sommets V.

2.2.2 algorithme

L'algorithme de Hagerup est décrit par l'algorithme 2 p. 6. La fonction $\widehat{min}$ à la ligne 5 de l'algorithme 2, se définit comme étant le sommet qui a la distance minimum selon la mesure $\widehat{d}$. La seule différence avec l'algorithme de Dijkstra réside la ligne 5, en effet au lieu de sélectionner le sommet minimum selon min, Hagerup sélectionne le sommet minimum selon la fonction $\widehat{min}$. C'est en cela que Hagerup s'affranchit partiellement du tri du cocycle.

Algorithme 2 : Algorithme générique de Hagerup

Data : G=(X,U) : un graphe orienté ;une fonction coût $\omega : U \rightarrow [0,1]$
$s \in V$ sommet source
Result : Une arborescence de plus courts chemins enracinée en s

```
1  Fermés← ∅ ; Ouverts← {s}
2  Parent(s) ← Nil ; d(s) ← 0
3  ∀y ∈ V , y ≠ s , d(y) ← n − 1
4  while Ouverts ≠ ∅ do
5      z ← min^(Ouverts)
6      Ajout(z,Fermés)
7      Enlever(z,Ouverts)
8      foreach v ∈ V avec (z,v) ∈ E do
9          if v ∈ Fermés then
10             return
11         if v ∉ Ouverts then
12             Ajout(v , Ouverts)
13         if d(z) + ω(z, v) < d(v) then
14             parent(v) ← z ; d(v)← d(z)+ω(z, v)
```

2.3 Invariant de l'algorithme de Hagerup

Notations : Soient deux ensembles notés **Fermés** et **Ouverts**, où **Fermés** représente l'ensemble des sommets qui ont été selectionnés à la ligne 5 de l'algorithme, et **Ouverts** représente l'ensemble des sommets qui sont dans la liste des sommets à explorer (i.e. les sommets qui sont modifiés par les lignes 8-14 et qui ne sont pas encore dans l'ensemble **Fermés**). Initialement seul le sommet source est placé dans **Ouverts**. Les ensembles **Fermés** et **Ouverts** ne se chevauchent pas.

Pour montrer la validité de l'algorithme (*i.e.* qu'il calcule effectivement les plus courts chemins), nous montrons que l'invariant principal de Dijkstra est toujours vérifié.

Théorème 1 (Hagerup) *L'algorithme de Hagerup calcule l'arborescence des plus courts chemins*

Démonstration : La preuve se fait par induction sur le nombre d'itérations (lignes 4-14 de l'algorithme 2). Nous regardons ce qui se passe après la i-ème itération de l'algorithme.

Base de l'induction après la première itération la situation est la suivante : le sommet s était le seul dans l'ensemble des **Ouverts**, il a par conséquent été choisi en ligne 5 et sa distance est 0, de plus il a été placé dans **Fermés**(ligne 6), et supprimé de **Ouverts**(ligne 7), donc pour s l'invariant est vérifié après la première itération. Pendant l'exécution de la première itération aucun voisin de s n'était dans **Fermés** par conséquent les lignes 9,10 de l'algorithme ne sont pas exécutées. Quelle est la situation des sommets voisins de s ? Concernant le voisinnage de s, aucun des sommets n'était dans **Ouverts** par conséquent il sont tous placés dans **Ouverts**(lignes 11,12) et leur distance est affectée par le poids de l'arête qui les relie à s(lignes 13,14). Par conséquent pour les sommets placés dans **Ouverts** l'invariant est également vérifié puisque le chemin allant de s à ces sommets n'est composé que d'un sommet **Fermés** et d'un seul sommet **Ouverts**. Donc l'invariant est vérifié après la première itération.

Induction Supposons que l'hypothèse d'induction soit vérifiée après la $(i-1)$-ième itération, et montrons que l'invariant reste vérifié après la i-ième itération. Les changements qui interviennent de la $(i$–$1)$-ième itération à la i-ième itération sont les suivants :

1. Un sommet z est choisi en ligne 5 et passe dans **Fermés** et est retiré de **Ouverts**.
2. Des sommets qui sont dans l'ensemble des **Ouverts** voient leur valeur affaiblie.
3. Des sommets qui sont dans l'ensemble des **Ouverts** ne sont pas modifiés.
4. Des sommets entre pour la première fois dans l'ensemble des **Ouverts**.
5. Des sommets de l'ensemble des **Fermés** sont voisins de z

Les cas 2. et 4. sont similaires, en effet ces sommets sont modifiés car voisins de z, or en examinant l'algorithme de plus près les lignes 11-12 servent de pré-traitement au sommet qui n'étaient pas encore dans **Ouverts**, par conséquent lorsque l'algorithme exécute les lignes 13-14 pour éventuellement modifier les distances des sommets **Ouverts**, il est incapable de faire la distinction entre les sommets présents dans **Ouverts** à l'itération précédente et ceux qui viennent d'être ajoutés. C'est la raison pour laquelle nous fusionnons ces deux cas pour effectuer notre preuve.

Le cas 1. Concerne le sommet z, que l'on a choisi en ligne 5 de l'algorithme. Deux possibilités pour ce sommet, soit il appartient à V_1, auquel cas sa distance arrondie ($\widehat{d}(z) = d(z)$) est égale à sa distance réelle, soit il appartient à V_2 et sa distance arrondie peut différer de sa distance réelle. Commençons par étudier la situation lorsque z est dans V_1. Soit v le prédécesseur de z dans l'arborescence. A l'étape $(i-1)$l'hypothèse est vérifiée pour les sommets z et v. De plus z a été choisie à la ligne 5 et comme il appartient à V_1 sa distance arrondie est égale à sa distance réelle. Par conséquent z est le minimum global de l'ensemble des **Ouverts**, de plus les valuations étant positives, nous montrons que tout autre chemin μ reliant s à z en passant par des sommets mixtes est plus grand. Considérons que ce chemin μ sorte des **Fermés** par t, et qu'il aille en z en utilisant soit des sommets **Ouverts** soit des sommets restants, alors la longueur d'un tel chemin est plus grande que celle qui relie s à z en passant seulement par des sommets fermés. Longueur(μ) = Longueur($\mu[s,t]$)+Longueur($\mu[t,z]$). Or Longueur($\mu[s,t]$) est plus grand ou égal à $d(t)$, et Longueur($\mu[t,z]$) est plus grand que 0, par conséquent quand z appartient à V_1 la propriété est vraie après l'étape i.
Lorsque le sommet z appartient à V_2, montrons que l'invariant reste vérifié. Pour les sommets z et v l'hypothèse est vérifié après l'étape $(i-1)$. Comme z appartient à V_2, z est un minimum global arrondi de l'ensemble des **Ouverts**, par conséquent de plus tout chemin analogue à μ ne peut être plus court que le chemin de s à z en empruntant seulement des sommets **Fermés**. En effet on a Longueur(μ)= Longueur($\mu[s,t]$)+ Longueur($\mu[t,z]$). Si t appartient à V_1 on a donc Longueur($\mu[s,t]$) $> d(z) - \Delta$ et de toute façon Longueur($\mu[t,z]$)$> \Delta$, il en résulte que $d(z)$ est inférieur strictement à Longueur(μ).
Deux sommets z et t ont la même distance arrondie si et seulement si on peut exprimer leur distance $d()$ de la façon suivante $\widehat{d}(z) = \widehat{d}(t) = \lfloor \frac{\Delta c+y}{\Delta} \rfloor \Delta$, $d(x)$ est de la forme $\Delta c + y$, où c est un constante et y est compris entre 0 et Δ ($0 \leq y < \Delta$). De cette propriété, il résulte que les distances réelles des sommets z et t, lorsque leur distance arrondie sont égales, sont éloignés d'au plus $\Delta - \epsilon$. Quand maintenant t appartient à V_2, on peut avoir, $d(t) < d(z)$ et $\widehat{d}(z) \leq \widehat{d}(t)$, cependant on a dans ce cas $d(z) - \Delta < d(t)$ par conséquent Longueur(μ)= Longueur($\mu[s,t]$)+ Longueur($\mu[t,z]$), or Longueur($\mu[t,z]$) est plus grand que Δ, et comme $d(t)$ est plus grand que $d(z) - \Delta$ La longueur de μ est plus grande que la longueur du chemin de s à z qui passe seulement par des sommets **Fermés**. Donc après l'étape i la propriété est vérifiée.

Le cas 2-4. L'ensemble de sommets concernés par ce cas, sont l'ensemble des sommets W, ($W \subseteq$ **Ouverts** $\cap N(z)$), voisins de z et qui ont vu leur distance diminuée (ligne 14). Pour l'ensemble des sommets de w, par hypothèse d'induction l'invariant était vérifié à l'étape $(i-1)$, et nous allons montrer que l'invariant reste vérifié après l'étape i. Lorsque l'on modifie leur distance à l'étape i et qu'on leur affecte comme prédécesseur z, c'est que nous parvenons à diminuer leur distance, donc le plus court chemin allant de s à un sommet de W, qui emprunte seulement des sommets de **Fermés** sauf w, est obtenu en passant par z. Par conséquent après l'étape i l'invariant reste vérifié pour les sommets de W.

Le cas 3. L'ensemble des sommets qui sont concernés par le cas 3 à savoir que ces sommets étaient présents dans l'ensemble des **Ouverts** après la $(i-1)$-ième itération et qui n'ont pas étés modifiés par la i-ième itération. Il y a deux raisons possibles à cela, la première étant qu'ils ne font pas partie du voisinage de z, et auquel cas ne sont pas concernés par les lignes 8 à 14 de l'algorithme qui examine le voisinage de z. Soit les sommets appartiennent au voisinage de z mais le test en ligne 13 a échoué car z ne peut affaiblir leur distance. Comme notre hypothèse d'induction était vérifiée à l'itération précédente et que les sommets n'ont pas été modifiés, après la i-ième itération l'invariant est toujours vérifié.

Le cas 5. Ces sommets sont les voisins du sommet z, et qui ont la particularité d'être dans **Fermés**, par conséquent ces sommets étaient dans l'ensemble des **Fermés** à l'étape $(i-1)$, et par hypothèse pour ces

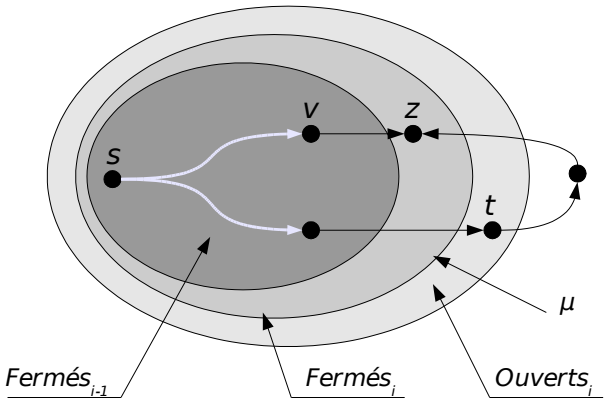

FIG. 2.1 – Situation considérée

sommets l'invariant était vérifié, de plus ces sommets ont étés choisis avant z ils avaient par conséquent une valeur plus faible et les valuations étant positive, il n'est pas possible de diminuer leur valeurs. Etant donné que ces sommets n'ont pas étés modifiés (lignes 9-10) alors l'invariant est toujours vérifié après l'étape i·

□

2.4 Affinage du choix min

L'approche de Hagerup étant légèrement différente de celle de Dijkstra, et comme nous l'avons montré précédemment, cette démarche fonctionne (cf preuve de l'invariant), il faut maintenant montrer que cet algorithme peut être implémenté de manière efficace (plus rapide que Dijkstra), sans quoi, il ne présenterait pas un grand intérêt. C'est pourquoi nous présentons maintenant comment Hagerup accomplit cette tâche, de manière simple et efficace.

La seule différence avec l'algorithme de Dijkstra résidant en ligne 5, à savoir le choix du $\widehat{min}$. Voyons maintenant comment nous pouvons implémenter de manière efficace cette procédure (cf. Algorithme 3 p. 8).

Algorithme 3 : Choix du $\widehat{min}$ dans Hagerup

```
1 x ← min{d̂(v) | v ∈ Ouverts ∩ V1} ∪ {∞}
2 while I_{j*} = ∅ and j* + 1 ≤ x do
3 |   j* ← j* + 1
4 i* ← min(I_{j*} ∪ {∞})
5 if i*Δ ≤ x then
6 |   choose u ∈ L_{i*} arbitrarily
7 else
8 |   choose u ∈ U ∩ V1 with d̂(u) = x
```

L'algorithme 3 travaille sur les deux ensembles V_1 et V_2 (leurs structures seront présentées en détail dans la section suivante). L'algorithme utilise les variables suivantes, j^* et i^*. La variable j^*, est une variable globale, qui représente quel sous ensemble de V_2 est plus petit que l'élément minimum de V_1, cette variable est conservée après chaque appel de la fonction $\widehat{min}$. La variable i^* représente l'indice de la plus petite valeur inférieur à x dans la boîte de j^*.

En ligne 1 de l'algorithme, x est choisi comme le sommet ayant la valeur minimum de l'ensemble V_1(∞, si l'ensemble est vide), le but de l'algorithme 3 est de trouver un sommet de V_2 qui a une valeur (selon $\widehat{d}$)plus petite que celle de x. Les lignes 2-3 permettent de se positionner sur le premier sous-ensemble de V_2 plus petit que x (si un tel ensemble existe). On récupère dans i^* la plus petite valeur du sous ensemble de V_2. En ligne 5, nous testons si le sommet de i^* est effectivement plus petit que le sommet x (il peut ne pas être défini). S'il existe , l'algorithme renvoie ce dernier, sinon l'algorithme renvoie le sommet x.

2.5 Les structures de données mises en œuvre

Maintenant que nous avons vu comment le choix du sommet minimum (selon $\widehat{d}$) était effectué, nous présentons les structures de données utilisées pour accomplir cette tâche, et ce de manière la plus efficace qu'il soit. Hagerup utilise pour cela deux structures :

1. Tas de Fibonacci pour V_1.
2. Paniers multi-niveaux (bucket) et listes doublement chaînées pour V_2.

La première (tas de Fibonacci : [FT87]) sert à stocker les sommets de l'ensemble V_1, les sommets sont stockés selon leur étiquette $\widehat{d}$. Pour rappel cette structure est celle utilisée dans la "meilleure" implémentation de Dijkstra par Tarjan.
Rappelons maintenant quelques résultat de complexité sur les tas de Fibonacci, qui nous seront utiles pour l'analyse de complexité. Tous les résultats fournis sont donnés en complexité amortie (cf [CLRS01]).

Insertion	$O(1)$
Modification de valeur	$O(1)$
Suppresion du minimum	$O(log_2 n)$

La seconde structure (des paniers multi-niveaux contenant des listes doublement chaînées) sert à stocker les sommets de l'ensemble V_2. Cette seconde structure est illustrée par la figure 2.2. La partie constituée de cases représente les paniers (buckets). Ces paniers sont susceptibles de pointer sur une liste de sommets, si tel est le cas la valeur de la case est 1, sinon la valeur est 0. On peut donc voir la structure de panier comme étant un mot binaire car constitué de 0 et de 1. Cet ensemble de paniers est hiérarchisé en mots, en effet au lieu de traiter la chaîne dans sa globalité, nous la divisons en facteurs de longeur k (pourquoi k, l'explication sera donnée par la suite).

Nous localisons les sommets de V_2 en stockant une partie de leur étiquette, en effet nous stockons seulement la partie :$\lfloor \frac{d(x)}{\Delta} \rfloor$, ce résultat est un entier, qui est compris entre 0 et $nk-1$. On peut donc assimiler cette structure à un bucket sort ([Knu98]), en limitant de manière astucieuse le nombre de paniers(bucket) à nk. Nous définissons D_j comme étant l'ensemble des paniers qui contiennent des sommets tels que $D_j = \{i | jk \leq i < (j+1)k \; et \; L_i \neq 0\}$, sont dans un même mot les sommets qui ont leur distance ($\widehat{d}$) comprise entre j et $j+1$. Chaque mot D_j est composé d'au plus k cases, par conséquent les sommets qui sont dans la même case (le même indice i) leur distance réelle ont une différence avec les autres sommets de la case d'au plus Δ. k étant borné par $log_2 m$, nous pouvons représenter D_j sous forme d'entier défini comme suit :

$$I = \{0, \ldots, k-1\}$$

$$D_j = \sum_{i \in I} 2^{k-1-i} \in \{0, \ldots, m-1\}$$

Cela permet d'effectuer les opérations suivantes en temps constant : trouver l'élément minimum de D_j, ajouter un élément, enlever un élément, tester si l'ensemble est vide. Les procédures pour effectuer les opérations précédemment citées, sont les suivantes :

Extraire le minimum	$k-1-\lfloor log_2 D_j \rfloor$
Insérer un élément i	$D_j + 2^{k-1-i}$
Retirer un élément i	$D_j - 2^{k-1-i}$
D_j est vide ?	$D_j = 0$

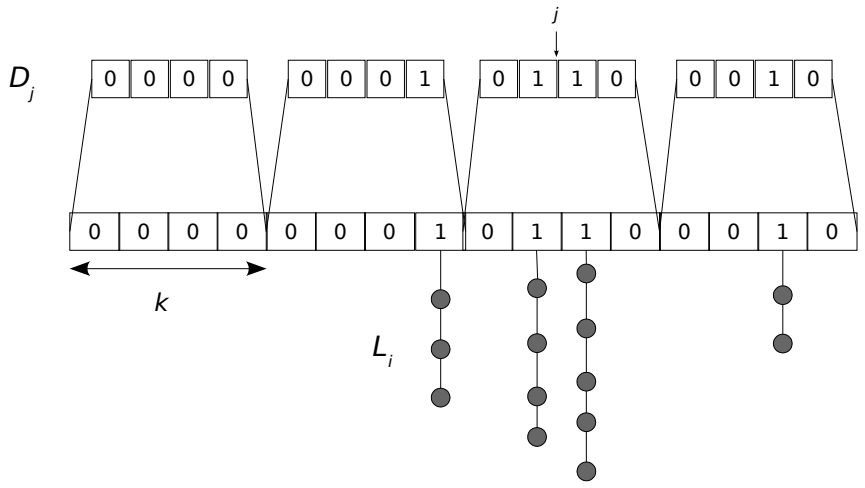

FIG. 2.2 – Structure utilisée pour la gestion des sommets V_2

Les valuations étant entre 0 et 1, et les longeurs des chemins croissent strictement, par conséquent l'ensemble des sommets à sélectionner sont dans l'ensemble D_j, c'est pourquoi la variable j^* (dans l'algorithme 3)pointe sur le plus petit ensemble D_j non vide (*i.e.* une fois que nous avons incrémenté j^* nous n'avons pas besoin de revenir en arrière).

Bien qu'en apparence cette structure semble complexe, elle est en fait relativement simple à utiliser et à mettre en place.

2.6 Complexité

Nous montrons, maintenant quelle complexité admet l'algorithme de Hagerup. Nous présentons la complexité moyenne, et la complexité dans le pire des cas. Nous montrons également, que cet algorithme est linéaire en espace.

Théorème 2 (Temps $\Theta(n+m)$) *L'algorithme de Hagerup résout le problème SSSP en temps moyen $\Theta(n+m)$.*

Démonstration : Nous montrons maintenant le théorême précédent. L'algorithme de Hagerup est constitué de deux algorithmes, l'algorithme 2 et l'algorithme 3. L'algorithme 2 est constitué d'une boucle **while** qui est exécuté $n-1$ fois (lignes 4-14). Ajouter un sommet à l'ensemble **Fermés** prend un temps $O(1)$, enlever un sommet de **Ouverts** prend un temps $O(1)$ si le sommet est dans V_2 et $O(logn)$ si le sommet est dans V_1. La ligne 5 est en fait un appel à l'algorithme 3($\widehat{min}$). Ce dernier est également constitué d'une boucle **while** qui est effectuée $n-1$ fois de manière globale (amortie). Toutes les opérations sur les ensembles sont effectuées en temps constant en utilisant les opérations présentées précédemment, excepté en ligne 1 extraire l'élément minimum de V_1 qui s'effectue en temps $O(logn)$. Or la ligne 1 est effectuée seulement pour les sommets de V_1. Le nombre de sommets de V_1 est borné par le nombre d'arêtes de poids inférieur à Δ noté S. L'espérance de S : $E(S)$ est, si la distribution des poids sur les arêtes est uniforme, de la forme $m\Delta$. Par conséquent la complexité est en $O(n+m+|V1|log_2 n)$, qui est borné par $O(n+m+m\Delta log_2 n)$ ce qui est équivalent à $O(n+m)$. Nous en déduisons donc que l'algorithme de Hagerup est linéaire en temps moyen, lorsque la distribution des poids sur les arêtes est uniforme.

□

Théorème 3 (Temps $O(m+nlogn)$) *L'algorithme dans le pire des cas s'exécute en temps $O(m+nlogn)$.*

Démonstration: Le pire des cas se produit quand aucun des sommets n'est dans l'ensemble V_2, par conséquent tout le travail s'effectue avec le tas de Fibonacci dans V_1, il s'agit donc d'une exécution "conventionnelle" de l'algorithme de Dijkstra avec les tas de Fibonacci, laquelle s'effectue en temps $O(m + nlogn)$.

□

Théorème 4 ($O(n + m)$ espace) *L'algorithme utilise un espace (en mémoire) linéaire en la taille de la donnée (i.e. $O(n + m)$).*

Démonstration: A tout moment, chaque sommet appartient soit à une liste, (*i.e.* un sommet de V_2), soit au tas de Fibonacci. De plus un sommet n'est que dans une liste à la fois, il n'y a pas de réplication des sommets. Nous avons donc n sommets dans au plus n listes. La structure en bucket est constitué de nk bits, ou autrement dit $n\lfloor log_2 m \rfloor$ bits, ce qui est linéaire en la taille de la donnée. □

Chapitre 3

Dijkstra Etendu

Lorsque l'on parle de coût d'un chemin, l'interprétation naturelle qui nous vient tout de suite à l'esprit, est que le coût d'un chemin est la somme des coûts de chaque tronçon (ou arête) qui le composent. C'est évidemment le cas dans le cadre des réseaux routiers, dans le cas des réseaux de communications. Cependant certains problèmes de modélisations n'admettent pas une fonction de coût aussi élémentaire. Nous nous demandons donc si toutefois, il est possible de résoudre cette famille de problèmes, et si oui à quelles conditions. La réponse est qu'il est possible de résoudre ces problèmes à l'aide de l'algorithme de Dijkstra (légèrement modifié) si la fonction de valuation est monotone croissante et faiblement pré-additive.

3.1 Conditions suffisantes pour l'algorithme de Dijkstra

Nous présentons ici des conditions suffisantes sur la fonction de coût de chemins, afin que l'algorithme de Dijkstra fournisse le bon résultat. Pour cela nous utilisons une fonction de coût f, cette fonction prend pour argument un chemin et renvoie son coût.

$$f : \mu \rightarrow \mathbb{R}^+$$

Nous notons μ_x comme étant un chemin reliant s à x, et est défini par l'arborescence des prédécesseurs. Nous notons de manière analogue μ_{xy} comme étant un chemin reliant le sommet x au sommet y (sans existence particulière dans l'arborescence des prédécesseurs). Nous définissons l'opérateur $.$ comme étant l'opérateur de concaténation de chemin (ex : $\mu_x.\mu_{xy}$ est un chemin de s à y).

3.1.1 Pré-additivité faible de la fonction coût

Nous présentons maintenant la notion de pré-additivité faible de la fonction f. Elle se définit comme suit, soit f notre fonction de coût, soient μ_x et μ'_x deux chemins allant de s à x (sans hypothèse supplémentaire : disjoint...). La fonction f est dite pré-additive faible si et seulement si :

$$f(\mu_x) \leq f(\mu'_x) \Rightarrow f(\mu_x.xy) \leq f(\mu'_x.xy)$$

La pré-additivité se définissant avec un si et seulement si.

Corollaire 1 (Pré-additivité faible récursive) *De la formule de pré-additivité, nous pouvons présenter une version récursive qui s'exprime de la façon suivante :*

$$f(\mu_x) \leq f(\mu'_x) \Rightarrow f(\mu_x.\mu_{xt}) \leq f(\mu'_x.\mu_{xt})$$

où μ_{xt} est un chemin allant de x à t.

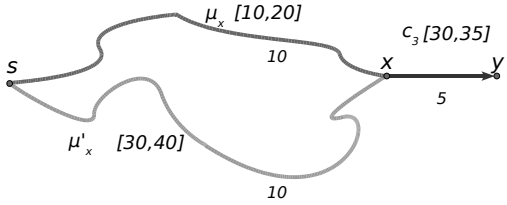

FIG. 3.1 – Exemple de la fonction $max - min$

Démonstration : La démonstration du corollaire précédent se fait par induction sur la longueur (en nombres d'arêtes) du chemin μ_{xy}. □

Cette notion nous permet de ne conserver, en cas d'égalité de deux chemins (μ_x et μ'_x) qui arrivent sur un même sommet x, qu'un seul des chemin. Remarquons que la fonction f, définie comme étant la somme des valuations qui constituent le chemin μ, respecte cette condition (celle employée dans l'algorithme classique de Dijkstra).

A contrario nous présentons ici, une fonction de valuation qui ne vérifie pas cette condition : $f : \mu \rightarrow max(\mu) - \min(\mu)$ (cf FIG. 3.1 p. 13). En effet $f(\mu_x)$ et $f(\mu'_x)$ ont la même valeur, cependant $f(\mu_x.xy)$ a pour valeur 25, alors que $f(\mu'_x.xy)$ a pour valeur 10.

3.1.2 Fonction monotone croissante

Nous appelons fonction monotone croissante f, toute fonction de valuation de chemin qui vérifie la condition suivante : soit μ_x, et xy un arête, f est monotone croissante si et seulement si pour tout chemin μ_x et pour toutes arêtes xy nous avons

$$f(\mu_x) \leq f(\mu_x.xy)$$

Corollaire 2 (Montonie croissante récursive) *Nous exprimons de la même façon la monotonie croissante récursive : avec μ_{xy} un chemin de x à y*

$$f(\mu_x) \leq f(\mu_x.\mu_{xy}) \; \forall x, y \in V$$

Démonstration : La démonstration du corollaire précédent se fait par induction sur la longueur (en nombres d'arêtes) du chemin μ_{xy}. □

3.1.3 Condition nécessaire et suffisante pour l'arborescence des plus courts chemins

Dans le cas de l'addition, il existe une condition nécessaire et suffisante sur l'arborescence des plus courts chemins, en effet T est une arborescence de plus courts chemins si et seulement si $\forall (x, y) \in E, d_T(y) \leq d_T(x) + \omega(xy)$. Est-ce toujours valable quand la fonction f est pré-additive faible et monotone croissante ? Voici la réponse.

Définition 1 (Arborescence de plus courts chemins) *T est une arborescence de plus courts chemins si et seulement si $f_T(\mu_x)$ (la distance dans l'arbre de s à x) est égale à la distance du plus court chemin élémentaire reliant s à x dans le graphe G.*

Théorème 5 (Arborescence des plus courts chemins) *T est une arborescence des plus courts chemins si et seulement si pour toute arête (x, y) de E et pour tout sommet y de G on a $f(\mu_y) \leq f(\mu_x.xy)$*

Démonstration : Nous allons montrer que le théorème précédent est vrai lorsque la fonction f vérifie la pré-additivité faible et la monotonie.
$\Rightarrow$ Montrons le par l'absurde, supposons que T soit effectivement une arborescence de plus courts chemins et que la condition sur les arêtes n'est pas vérifiée. Par conséquent il existe au moins une arête, que l'on

notera (x, y) qui vérifie $f(\mu_y) > f(\mu_x.xy)$. Par conséquent en procédant à l'échange de l'arête (xy) avec la dernière de μ_y, on obtient une arborescence T' qui a un chemin plus court pour y, or par hypothèse T est un arborescence de plus courts chemins donc $f_T(\mu_y)$ est la distance du plus court chemin reliant s à y dans G, or on a un chemin μ'_y $(= \mu_x.xy)$ qui est plus court que μ_y, ce qui est absurde car μ_y est un plus court chemin. $\Leftarrow$ Si la condition est vérifiée pour toutes les arêtes (*i.e.* $\forall e = (x, y) \in E,\ f_T(\mu_y) \leq f_T(\mu_x.xy)$ et $(xy) \in G \setminus T$) alors T est une arborescence de plus courts chemins. Supposons que la condition sur les arêtes et les sommets soit vérifiée, et que l'arborescence ne soit pas celle des plus courts chemins. Il existe un sommet y pour lequel sa distance dans T est plus grande que celle du plus court chemin. Soit μ_y le chemin dans T et soit μ'_y un chemin plus court qui n'est pas dans T. Considérons que le dernier sommet de μ'_y soit x, nous avons $f_T(\mu_y) > f(\mu'_y)$, de plus comme la condition est respectée pour tous les sommets, on a donc μ'_x qui est le plus court chemin de s à x, en effet si on avait le chemin μ'_x qui ne passe par les arêtes de l'arbre, la condition étant vérifiée, cela implique que les chemins qui ne sont pas dans l'arbre ont la même valeur. On a donc finalement $f_T(\mu_y) > f_T(\mu_x.xy)$, ce qui est absurde car par hypothèse ce n'est pas le cas.

□

3.1.4 Dijkstra étendu

Nous allons maintenant montrer que l'algorithme de Dijkstra reste valide lorsque la fonction est faiblement pré-additive et que la fonction de coût est monotone croissante.

Algorithme 4 : Algorithme de Dijkstra Générique

Data : G=(X,U) : un graphe orienté ;une fonction coût $\omega : U \to \mathbb{R}^+$ $s \in V$ sommet source
Result : Une arborescence de plus courts chemins enracinée en s

```
1  Fermés← ∅ ; Ouverts← {s}
2  Parent(s) ← Nil ; f(s) ← 0
3  ∀y ∈ V , y ≠ s , f(y) ← ∞
4  while Ouverts ≠ ∅ do
5      z ← min(f,Ouverts)
6      Ajout(z, Fermés)
7      Enlever(z, Ouverts)
8      foreach v ∈ V avec (z,v) ∈ E do
9          if v ∈ Fermés then
10             return
11         if v ∉ Ouverts then
12             Ajout(v, Ouverts)
13         if f(μ_z.zv) < f(μ_v) then
14             parent(v) ← z ; f(μ_v) ← f(μ_z.zv)
```

Théorème 6 (Dijkstra Etendu) *La conjonction des deux conditions présentées précédemment est suffisante pour que l'invariant principal de Dijkstra soit vérifié.*

Démonstration : La démonstration se fait par induction sur le nombre d'itérations (lignes 4-14 de l'algorithme 1). Nous nous plaçons après la i-ième itération de l'algorithme.

Base de l'induction Après la première itération la situation est : le sommet s a été choisi en ligne 5 et a été placé dans l'ensemble des **Fermés**. Les sommets voisins de s ont été ajoutés dans l'ensemble des **Ouverts** en lignes 11-12. Concernant le sommet source s l'invariant est trivialement vérifié. Concernant les sommets qui sont dans l'ensemble des **Ouverts**, ce sont les voisins de s par conséquent le seul chemin de s à

ces sommets qui n'emprunte que des sommets **Fermés** est l'arête qui les relient à s par conséquent pour ces sommets l'invariant est vérifié. L'invariant est donc vérifié après la première itération.

Induction Supposons que l'hypothèse d'induction soit vérifiée après la $(i–1)$-ième itération et montrons que l'invariant est toujours vérifié après la i-ième itération. Afin de mener à bien la démonstration, nous distingons trois cas :

1. Un sommet z est choisi en ligne 5.
2. Les sommets dans l'ensemble des **Ouverts**.
3. Les sommets dans l'ensemble des **Fermés**.

Le cas 1. En ligne 5 de l'algorithme, le sommet z est sélectionné. Il est selectionné car il est le sommet qui la valeur minimum(selon f) parmi les sommets des **Ouverts**. Nous allons démontrer que pour ce sommet z l'invariant reste vérifié après la i-ième itération. Soit v son prédécesseur dans l'ensemble des **Fermés**, pour v comme pour z, l'hypothèse était vérifiée à l'étape précédente. Comme $f(z)$ est minimum parmi les **Ouverts** et que f est une fonction monotone croissante (cf FIG. 2.1 p. 8) soit t un sommet de **Ouverts** pour lequel $f(t)$ est plus grand que $f(z)$. La longueur du chemin de s à z en passant par t. $f(\mu_t.\mu_{tz})$ est plus grand que $f(\mu_t)$ et $f(\mu_{tz})$, or $f(\mu_t)$ est plus grand que $f(\mu_z)$ par conséquent la propriété reste vérifiée pour z après la i-ième itération.

Le cas 2. Regardons ce qu'il se passe pour les sommets de l'ensemble des **Ouverts**. Nous considérons deux cas, pour cela nous partionnons les **Ouverts** en deux sous-ensembles. $\mathbf{Ouverts} = Y_1 \cup Y_2$. Où Y_1 est l'ensemble des sommets qui appartiennent à **Ouverts** et qui n'ont ne sont pas voisins de z :
$Y_1 = \{y \mid y \in \mathbf{Ouverts} \cap \overline{N}(z)\}$

Y_2 est l'ensemble des sommets qui appartiennent à **Ouverts** et qui sont dans le voisinage de z. $Y_2 = \{y \mid y \in \mathbf{Ouverts} \cap N(z)\}$

Y_1 Pour les sommets de Y_1, n'étant pas dans le voisinage de z, ils n'ont pas étés modifiés par l'algorithme (ils ne sont pas concernés par les lignes 8-14). De plus comme pour ces sommets l'invariant était vérifié et qu'aucune modification n'a été apportée alors, l'invariant reste vérifié.

Y_2 Pour les sommets voisins de z, nous distinguons deux cas : les sommets qui sont "améliorés" par z et ceux qui ne sont pas modifiés.
$Y_{21} = \{y \mid y \in \mathbf{Ouverts} \cap N(z)\ et\ f(\mu_y) > f(\mu_z.zy)\}$ et ceux qui ne sont pas modifiés notés Y_{22}. ($Y_{22} = \{y \mid y \in \mathbf{Ouverts} \cap N(z)\ et\ f(\mu_y) \leq f(\mu_z.zy)\}$). Pour Y_{21}, l'ajout de z dans **Fermés** a provoqué une amélioration des distances pour ces sommets par conséquent pour les sommet de Y_{21} la propriété reste vérifiée. En ce qui concerne les sommets de Y_{22}, ils n'ont pas été modifiés par l'algorithme car soit la modification que l'on proposait était une aggravation et dans ce cas là la propriété reste trivialement vérifiée, soit il y a égalité, et comme la fonction f vérifie la condition de pré-additivité faible on peut choisir sans conséquence soit l'un soit l'autre, c'est pourquoi l'algorithme ne procède à aucune modification.

Le cas 3. Sont concernés par ce cas les sommets qui appartiennent à l'ensemble des **Fermés** durant la i-ième itération. Ces sommets sont dans **Fermés** car leur coût était inférieur à celui de z par conséquent la fonction distance étant monotone croissante tout ce que peut faire z c'est d'augmenter leur valeur. C'est pourquoi en lignes 9-10 les sommets **Fermés** sont ignorés. Etant donné que pour ces sommets l'invariant était vérifié à l'étape précédente et qu'aucune modification n'a été apporté, l'invariant reste vérifié après la i-ième itération.

□

3.2 Hagerup : cas particulier de Dijkstra étendu

Ci-après une version modifiée de l'algorithme de Hagerup, exprimée comme un cas particulier de l'algorithme de Dijkstra étendu. Nous utilisons dans le test ligne 13 l'opérateur de comparaison $\prec$ définit de la manière suivante :

$$\widehat{d}(v) \prec \widehat{d}(u) = \begin{cases} \widehat{d}(v) < \widehat{d}(u) & si\ \widehat{d}(v) \neq \widehat{d}(u) \\ d(v) < d(u) & si\ \widehat{d}(v) = \widehat{d}(u) \end{cases}$$

Algorithme 5 : Algorithme générique de Hagerup modifié

Data : G=(X,U) : un graphe orienté ;une fonction coût $\omega : U \rightarrow [0, 1]$
$s \in V$ sommet source
Result : Une arborescence de plus courts chemins enracinée en s

```
1  Fermés← ∅ ; Ouverts← {s}
2  Parent(s) ← Nil ; d(s) ← 0
3  ∀y ∈ V , y ≠ s , d(y) ← n − 1
4  while Ouverts ≠ ∅ do
5      z ← min(d̂,Ouverts)
6      Ajout(z,Fermés)
7      Enlever(z,Ouverts)
8      foreach v ∈ V avec (z,v) ∈ E do
9          if v ∈ Fermés then
10             return
11         if v ∉ Ouverts then
12             Ajout(v , Ouverts)
13         if d̂(μ_z.zv) ≺ d̂(μ_v) then
14             parent(v) ← z ; d̂(μ_v) ← d̂(μ_z.zv)
```

Corollaire 3 (Cas particulier) *L'algorithme de Hagerup constitue un cas particulier de l'algorithme de Dijkstra : à savoir qu'une modification des valuations sur une instance de Hagerup provoque le même ordre de sélection des sommets par l'algorithme de Dijkstra sur cette instance transformée.*

Sachant que le min en ligne 5 est choisi selon $<$, et que pour calculer $\widehat{d}(x)$, nous avons besoin de stocker $d(x)$.

Chapitre 4

L'étape du tri dans le calcul des plus court chemin

Nous présentons maintenant les liens étroits qui existent entre le problème du tri et l'algorithme de Dijkstra. L'algorithme de Dijkstra est aussi puissant que le tri, en effet avec cet algorithme nous pouvons trier des valeurs, il suffit pour cela de calculer le plus court chemin sur une étoile. Il a de plus été montré par Thorup, que si nous sommes capable de proposer une implémentation en temps linéaire, alors le problème du tri est également solvable en temps linéaire. Afin de comprendre en quoi l'algorithme de Hagerup est plus "efficace" que l'algorithme de Dijkstra, il convient de comparer leurs exécutions respectives.

Sur la figure 4.1 p. 18, en a) nous avons l'instance initiale. En b), nous procèdons à une normalisation des étiquettes, c'est à dire que nous multitplions chaque étiquette par $1/M$, (où M représente l'étiquette de valeur maximum). On obtient ainsi des valeurs entre 0 et 1. De plus sur la figure est présent l'ordre de sélection des sommets par Dijkstra. Enfin en c) est noté l'exécution de Hagerup sur la même instance, d'abord sont sélectionnés les sommets de V_1 (hachure oblique) dans l'ordre des valeurs, puis sont sélectionnés les sommets qui ont pour valeur arrondi $1/3$ (hachure verticale), l'ordre n'importe pas, puis ensuite les sommets qui ont pour valeur arrondie $2/3$, il y en a 4 (les sommets grisés), l'ordre n'importe toujours pas, et enfin le dernier sommet qui a pour valeur 1. On remarque sur cette exécution que l'algorithme de Hagerup est plus souple que l'algorithme de Dijkstra, en effet Dijkstra n'autorise qu'une seule "permutation" (un seul ordre de sélection $1/12!$) des sommets alors que Hagerup autorise plusieurs sélections ($1 \times 3! \times 4! \times 1/12!$).

Cet exemple permet de remarquer, que l'algorithme de Hagerup, n'est pas aussi puissant que le tri, il retourne cependant le résultat optimal. Nous en concluons, que le modèle de tri est "trop" puissant pour résoudre le problème de plus court chemin. C'est pourquoi il est possible que l'on voit apparaître un jour des algorithmes pour résoudre le problème du plus court chemin en temps linéaire, ce qui semble plus difficile pour le problème du tri.

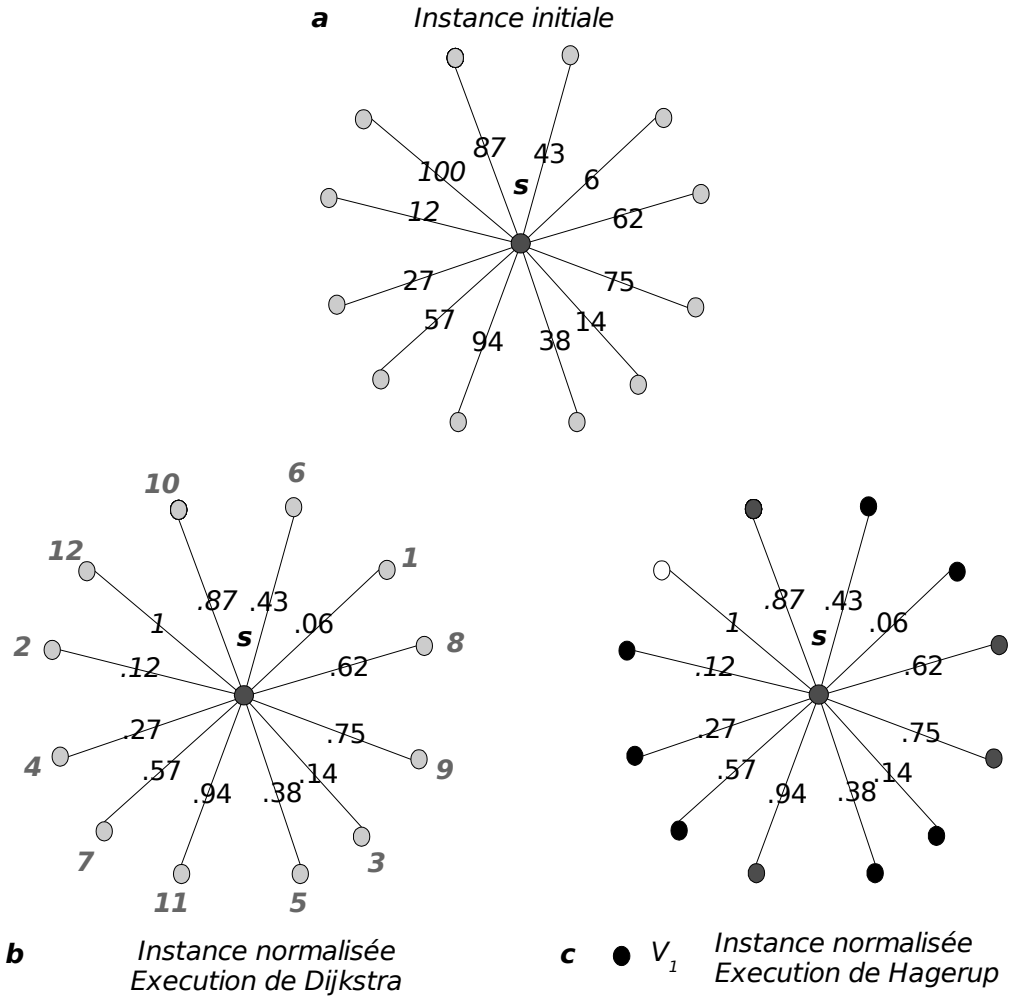

FIG. 4.1 – a) Plus court chemin sur une étoile, b) Exécution de Dijkstra sur l'instance normalisée et c) exécution de Hagerup

Deuxième partie

Caracterisations de parcours à l'aide d'ordre totaux

Chapitre 5

Caractérisations des parcours

Récemment Corneil et Krueger ([CK04]) ont proposés une classification à l'aide de caractérisations des des différents parcours(en largeur, profondeur, Lexicographique)que l'on peut effectuer sur un graphe. Cette caractérisation s'exprime sous la forme d'un quadruplet de sommets. Ces sommets a, b, c et d sont de la forme suivante : ab n'est pas une arête du graphe, par contre ac en est une, et si l'ordre de découverte (sélection) est $\sigma(a) < \sigma(b) < \sigma(c)$ (où σ représente l'ordes des sommets), cela signifie qu'il existe un sommet d tel que db est une arête du graphe et que d a été découvert avant b. Le parcours est en profondeur si et seulement si pour tout triplet a, b, c vérifiant les conditions précédentes, le sommet d a été découvert avant b et après a. Le parcours est en largeur si le sommet d a été découvert avant a. Nous appelons ancêtre de x tout sommets y adjacent au sommet x tel que $\sigma(y) < \sigma(x)$. De la même manière nous appelons successeurs de x tout sommet y adjacent x tel que $\sigma(x) < \sigma(y)$. Cette classification a permis en outre de découvrir un nouveau parcours, le parcours LexDFS (en profondeur lexicographique). En effet la caractérisation du parcours LexDFS est symétrique à celle du LexBFS.

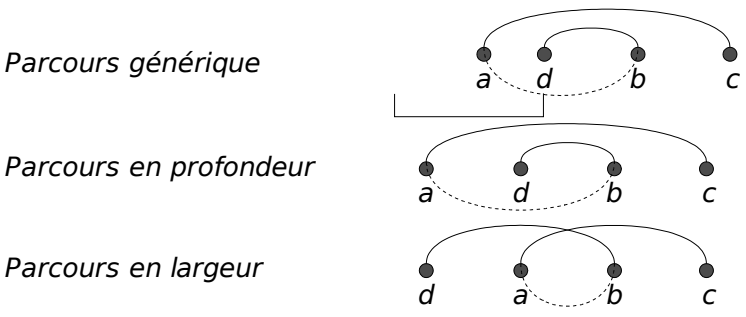

FIG. 5.1 – Caractérisations des parcours

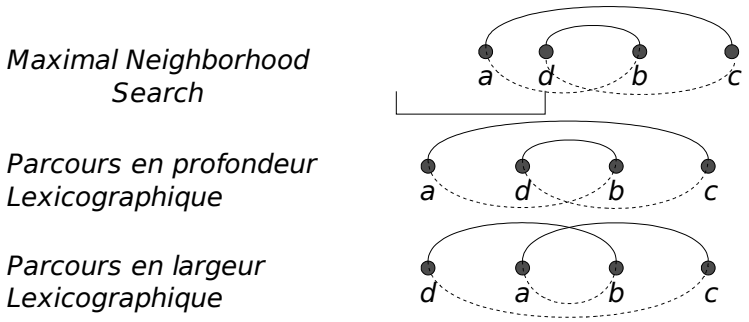

FIG. 5.2 – Caractérisations des parcours

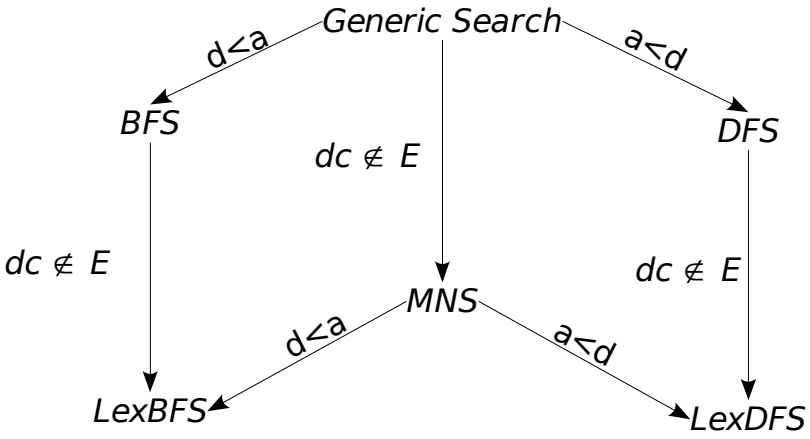

FIG. 5.3 – Récapitulatif des parcours

Chapitre 6

Vérification des parcours

6.1 Vérification du parcours

Nous pouvons vérifier si un parcours est en largeur ou en profondeur (ou si cela correspond à un parcours "générique") en vérifiant la condition sur tous les triplets. Ce qui nous amène à une vérification en $O(n^3)$.

Nous pouvons, en améliorant quelque peu la technique, procéder à une vérifcation de ces trois parcours en $O(n^2)$. La technique employée est la suivante : Soit T le tableau contenant les sommets du graphes, ordonnés selon σ, la permutation du parcours. Pour chaque sommet, nous sommes capable de trouver le plus jeune ancêtre dans σ en temps $O(1)$ (*i.e.* le plus proche), et le plus vieil ancêtre en temps $O(1)$(*i.e.* le plus éloigné). Pour ce faire notre graphe doit être codé sous forme de listes d'adjacences, pour chaque sommet une liste des prédécesseurs triés selon σ, et une liste de successeurs triés selon σ. Trouver le plus petit prédécesseur (*i.e.* le plus vieil ancêtre) consiste à choisir le premier élément de la liste. Trouver le plus grand prédécesseur (*i.e.* le plus jeune ancêtre) consiste à prendre le dernier élément de la liste.

En ce qui concerne le parcours, générique, la condition est vérifiée si pour tout triplet nous pouvons trouver un ancêtre de b. Ensuite, pour les parcours en largeur, est si l'on peut trouver un sommet d qui a été sélectionné avant a. Et enfin pour le parcours en pronfondeur, si on trouve un sommet d qui a été choisi après a et avant d.

Algorithme 6 : Algorithme de vérification du parcours générique

Data : T_1 un tableau les sommets ordonnés selon σ

Result : {Oui (la propriété est vérifiée pour tous les quadruplets), Non}

```
for a ← 1 à n do
    marquer c les voisins de a dans T_2. ;
    for b ← a + 1 à min(n, DernierVoisin(a)) do
        if ∄ d un ancêtre de b then
            return Non
return Oui
```

6.1.1 Robustesse

Remarquons que les algorithmes précédents sont robustes, à savoir qu'il produisent un certificat pour les deux réponses oui-non. En effet dans les deux cas, soit la permutation n'est pas un parcours et nous sommes capable d'exhiber la cause (le triplet fautif), soit la permutation correspond à un parcours et nous pouvons garantir que pour tout triplet la condition est vérifiée.

Algorithme 7 : Algorithme de vérification du parcours en Profondeur

Data : T_1 un tableau les sommets ordonnés selon σ
Result : {Oui (la propriété est vérifiée pour tous les quadruplets), Non}

```
1 for a ← 1 à n do
2     marquer c les voisins de a dans T_2. ;
3     for b ← a + 1 à min(n, DernierVoisin(a)) do
4         d ← plus jeune ancêtre de b. ;
5         if σ(a) > σ(d) then
6             return Non
7 return Oui
```

Algorithme 8 : Algorithme de vérification du parcours en Profondeur

Data : T_1 un tableau les sommets ordonnés selon σ
Result : {Oui (la propriété est vérifiée pour tous les quadruplets), Non}

```
1 for a ← 1 à n do
2     marquer c les voisins de a dans T_2. ;
3     for b ← a + 1 à min(n, DernierVoisin(a)) do
4         d ← plus vieil ancêtre de b. ;
5         if σ(a) < σ(d) then
6             return Non
7 return Oui
```

6.2 Complexité

La complexité des algorithmes précédents est en $O(n^2)$. L'algorithme est constitué de deux boucles "for", la première est exécutée n fois, et la seconde est exécutée au plus $n - a$ fois ($n - 1 + n - 2 + \cdots + 1$) ce qui aboutit en $O(n^2)$. Ce résultat constitue une amélioration majeure par rapport à l'approche naïve en $O(n^3)$.

Chapitre 7

Vérificateurs

Face aux difficultés rencontrées pour concevoir des algorithmes robustes pour déterminer si un parcours vérifie ou non les propriétés qui sont les siennes, nous abordons le problème selon une autre approche. Notre approche consiste à fournir et véfifier un certificat seulement si le parcours n'est pas celui annoncé.

7.1 Nouvelle approche

La technique présentée maintenant consiste à effectuer le parcours que l'on cherche à reconnaître en s'aidant de la permutation à vérifier. La permutation que l'on cherche à vérifier sera noté σ_1, la permutation que l'on va calculer en s'aidant de σ_1 sera noté σ_2. Notre algorithme valide σ_1 si en en exécutant l'algorithme, nous obtenons $\sigma_1 = \sigma_2$. Si par contre lorsque nous effectuons le parcours que l'on cherche à vérifier, nous rencontrons un blocage, alors notre algorithme est capable d'exhiber un triplet de sommets problématique.

7.1.1 Blocage et Permutation

Nous appelons blocage sur deux sommets x et y, la configuration suivante : Nous exécutons l'algorithme du parcours que l'on cherche à vérifier (σ_1 est la permutation de tel parcours (BFS,...)), et l'algorithme que nous sommes en train d'exécuter nous impose de choisir le sommet y, alors que dans σ_1 c'est x qui a été choisi. L'algorithme de "vérification" s'arrête sur le premier blocage qui intervient.

Théorème 7 *Soit deux sommet x et y de G. Si le premier blocage intervient sur ces sommets, alors $\sigma_1(y) > \sigma_1(x)$.*

Démonstration: Montrons le par l'absurde : supposons que x, y soit le premier blocage et qu'on ait $\sigma_1(y) < \sigma_1(x)$. Si x, y est le premier blocage, cela signifie que le préfixe de σ_1 jusqu'à la position précédent x est identique au préfixe de σ_2 jusqu'à la position précédent y. Or si on a $\sigma_1(y) < \sigma_1(x)$, alors il y a un blocage dans σ_1 sur y, ce qui est absurde puisque x, y est le premier blocage. □

7.1.2 "Generic Search"

Nous appelons parcours générique, un parcours qui vérifie la condition sur les triplets de sommets a, b, c. Pour tout triplet a, b, c, tel que $ab \notin E$, et $ac \in E$, il existe un sommet d qui est avant b dans σ tel que $db \in E$.

Propriété 1 (O-GS) *Soit σ une permutation, si l'on a $a < b < c$ et $ac \in E$ et $ab \notin E$ alors il n'existe aucun sommet d, tel que $d < b$ et $db \in E$.*

Théorème 8 (Existence de triplet a, b, c) *Lorsqu'il y a blocage, en x et y, on peut trouver un tripet a, b, c, tel que $ac \in E$ et $ab \notin E$.*

Démonstration lorsque nous obtenons un blocage en x, y, cela signifie que le sommet y doit être choisi avant le sommet x. On fixe alors x comme b et y comme c. Nous avons alors x qui n'est pas dans le cocycle $\{\mathbf{Fermés}, \mathbf{Ouverts}\}$. Par conséquent cela signifie qu'il n'y a pas dans σ_1 de sommet d qui est un prédécesseur de b. Par conséquent on a un triplet a, b, c, qui ne vérifie pas la condition. □

Théorème 9 *S'il existe dans σ_1 un triplet a, b, c pour lequel la propriété O-GS est vérifiée alors, σ_1 n'est pas la permutation d'un parcours générique.*

Démonstration: S'il existe un triplet a, b, c pour lequel nous ne sommes pas capable de trouver un sommet d prédécesseur de b. Cela signifie que b n'était pas dans la liste des sommets à traiter par conséquent il ne s'agit pas d'un parcours générique.

7.1.3 BFS

Nous nous intéressons maintenant au parcours BFS, (le parcours en Largeur). Un parcours est en largeur si pour tout triplet a, b, c, (avec les mêmes conditions d'adjacence), on peut trouver un sommet d prédécesseur de b qui se trouve avant a dans σ.

Propriété 2 (O-BFS) *Soit a, b, c, trois sommets de G, tel que $ac \in E$, et $ab \notin E$, alors il n'existe aucun sommet d tel que $db \in E$ et $\sigma_1(d) < \sigma_1(a)$ (i.e. il existe des sommets d prédécesseurs de b mais qui ne sont pas avant a).*

Théorème 10 *En x, y, lorsqu'il y a blocage, on peut trouver un triplet a, b, c qui vérifie la condition d'adjacence.*

Démonstration: Lorsqu'il y a un blocage sur x, et y, nous fixons x en b et y en c. Comme il y a blocage cela signifie que b n'est pas voisin du plus petit prédécesseur de c dans σ. Sinon il n'y aurait pas de blocage. Nous avons donc un triplet qui vérifie la condition. □

Théorème 11 (BFS-Particulier) *Si la permutation des sommets σ_1 n'est pas celle d'un parcours générique alors, σ_1, n'est pas la permutation d'un parcours en largeur.*

Démonstration: Le parcours en largeur étant un cas particulier du parcours générique, il est clair qu'une permutation de parcours en largeur est aussi la permutation d'un parcours générique. □

Théorème 12 *Si il existe dans σ_1 un triplet a, b, c qui vérifie la propriété O-BFS, ou la propriété O-GS alors σ_1 n'est pas la permutation d'un parcours en largeur.*

Démonstration: Pour O-GS la démonstration est donnée par le théorème 11. Intéressons nous à la propriété O-BFS, s'il existe un tel triplet, cela signifie que b est dans la liste des sommet à explorer mais après c, par conséquent le plus petit ancêtre de b est plus grand que celui de c. □

7.1.4 DFS

Un parcours est en largeur si pour tout triplet a, b, c, (avec les mêmes conditions d'adjacence), on peut trouver un sommet d prédécesseur de b qui se trouve entre a et d dans σ.

Propriété 3 (O-DFS) *Soit a, b, c, trois sommets de G, tel que $ac \in E$, et $ab \notin E$, alors il n'existe aucun sommet d tel que $db \in E$ et $\sigma_1(a) < \sigma_1(d)$ (i.e. il existe des sommets d prédécesseurs de b mais qui sont avant a).*

Théorème 13 *En x, y, lorsqu'il y a blocage, on peut trouver un triplet a, b, c qui vérifie la condition d'adjacence.*

Démonstration: Lorsqu'il y a un blocage sur x, et y, nous fixons x en b et y en c. Comme il y a blocage cela signifie que b n'est pas voisin du plus grand prédécesseur de c dans σ. Sinon il n'y aurait pas de blocage. Nous avons donc un triplet qui vérifie la condition. □

Théorème 14 (DFS-Particulier) *Si la permutation des sommets σ_1 n'est pas celle d'un parcours générique alors, σ_1, n'est pas la permutation d'un parcours en profondeur.*

Démonstration: Le parcours en profondeur étant un cas particulier du parcours générique, il est clair qu'une permutation de parcours en profondeur est aussi la permutation d'un parcours générique. □

Théorème 15 *Si il existe dans σ_1 un triplet a, b, c qui vérifie la propriété O-DFS, ou la propriété O-GS alors σ_1 n'est pas la permutation d'un parcours en largeur.*

Démonstration: Pour O-GS la démonstration est donnée par le théorème 14. Intéressons nous à la propriété O-DFS. S'il y a blocage, cela signifie que x est plus bas dans la pile par conséquent le plus grand ancêtre de c est plus grand que le plus grand ancêtre de d, on obtient donc un triplet pour lequel nous ne pouvons trouver aucun sommet d entre a et b. □

7.1.5 LexBFS

Nous étudions maintenant le parcours LexBFS (parcours en largeur lexicographique).

Propriété 4 (O-LexBFS) *Soit a, b, c trois sommets de G tel que $ac \in E$ et $ab \notin E$, alors il n'existe aucun sommet d tel que $db \in E$ et $\sigma_1(d) < \sigma_1(a)$ avec $dc \notin E$.*

Théorème 16 *Lorsque survient un blocage sur les sommets x, y, nous pouvons trouver un triplet a, b, c qui vérifie la condition d'adjacence.*

Démonstration: Lors d'un blocage sur x, et y, nous fixons x en b et y en c. De plus s'il y a blocage, cela signifie que c a une étiquette plus grande que b. Par conséquent en regardant les étiquettes de b et c, nous pouvons trouver un sommet a tel que $ac \in E$ et $ab \notin E$. Il suffit de regarder pour cela le préfixe de chaque étiquette et de s'arrêter sur la première différence. Nous pouvons ainsi trouver un triple a, b, c qui vérifie les conditions d'adjacence. □

Théorème 17 (LexBFS-Particulier) *Si la permutation σ_1 n'est pas celle d'un parcours en largeur, alors σ_1 n'est pas la permutation d'un parcours en largeur lexicographique.*

Démonstration: Le parcours LexBFS étant un cas particulier du parcours en largeur, il est clair que si la parcours ne vérifie pas les conditions du parcours en largeur, il ne vérifiera pas les conditions du parcours LexBFS □

Théorème 18 *S'il existe dans σ_1 un triplet qui vérifie la propriété O-BFS ou O-LexBFS, alors σ_1 n'est pas la permutation d'un parcours LexBFS.*

Démonstration: Pour O-BFS, la démonstration est donnée par le théorème 17. Concentrons nous sur la propriété O-LexBFS, s'il existe un tel triplet, cela signifie que b est dans la liste des sommets à explorer, mais que son étiquette est plus petite, par conséquent s'il existe des sommets d prédécesseurs de b plus petit que a, cela signifie que ces sommets sont reliés également à c, par conséquent l'étiquette de c est plus grande que celle de b. Il ne peut donc s'agir d'un parcours LexBFS. □

7.1.6 LexDFS

Nous étudions maintenant le parcours LexDFS (parcours en profondeur lexicographique). Ce parcours a été "découvert" suite à la classification proposée par Corneil-Krueger, c'est le parcours "symétrique" au LexBFS.

Propriété 5 (O-LexDFS) *Soit a, b, c trois sommets de G tel que $ac \in E$ et $ab \notin E$, alors il n'existe aucun sommet d tel que $db \in E$ et $\sigma_1(a) < \sigma_1(d) < \sigma_1(b)$ avec $dc \notin E$.*

Théorème 19 *Lorsque survient un blocage sur les sommets x, y, nous pouvons trouver un triplet a, b, c qui vérifie la condition d'adjacence.*

Démonstration: Lors d'un blocage sur x, et y, nous fixons x en b et y en c. De plus s'il y a blocage, cela signifie que c a une étiquette plus grande que b. Par conséquent en regardant les étiquettes de b et c, nous pouvons trouver un sommet a tel que $ac \in E$ et $ab \notin E$. Il suffit de regarder pour cela le préfixe de chaque étiquette et de s'arrêter sur la première différence. Nous pouvons ainsi trouver un triplet a, b, c qui vérifie les conditions d'adjacence. □

Théorème 20 (LexDFS-Particulier) *Si la permutation σ_1 n'est pas celle d'un parcours en profondeur, alors σ_1 n'est pas la permutation d'un parcours en profondeur lexicographique.*

Démonstration: Le parcours LexDFS étant un cas particulier du parcours en profondeur, il est clair que si la parcours ne vérifie pas les conditions du parcours en profondeur, il ne vérifiera pas les conditions du parcours LexDFS. □

Théorème 21 *S'il existe dans σ_1 un triplet qui vérifie la propriété O-DFS ou O-LexBFS, alors σ_1 n'est pas la permutation d'un parcours LexDFS.*

Démonstration: Pour O-DFS, la démonstration est donnée par le théorème 20. Concentrons nous sur la propriété O-LexDFS, s'il existe un tel triplet, cela signifie que l'étiquette de b est plus petite que celle de c. Si l'étiquette de c est plus grande, nous aurions dû sélectionner c au lieu de b, il ne peut donc s'agir d'un parcours LexDFS.

□

7.2 Calcul LexDFS

Algorithme de calcul du LexDFS par affinage de partition.

Algorithme 9 : LexDFS par affinage de partition

Data : $G = (V, E)$ un graphe, s un sommet de départ
Result : σ un permutation des sommets de V correspondant au parcours LexDFS à partir de s

1 $\sigma(s) \leftarrow 1$
2 Couper les boîtes en $N_i(s)$ et $\overline{N_i}(s)$
3 Translater les boîtes $N_i(s)$ à droite de s.
4 **for** *i de 2 à n* **do**
5 Choisir un sommet x dans la boîte la plus à gauche des sommets non sélectionnés
6 Isoler x de sa boîte
7 $\sigma(x) \leftarrow i$
8 Couper les boîtes à droite de x en $N_i(x)$ et $\overline{N_i}(x)$
9 Translater les boîtes $N_i(x)$ à droite de x.

L'opération de translation de boîtes vers x consiste à déplacer certaines boîtes et les placer à droite de x. Cette opération préserve la précédence des boîtes déplacées. En clair, pour deux boîtes translatées A et B, cela signifie que si une boîte A est avant une boîte B dans σ_{i-1} alors A reste avant B dans σ_i ($A <_{i-1} B \Rightarrow A <_i B$).

Invariant 1 (Monotonie des étiquettes) *Dans l'algorithme LexDFS les étiquettes des sommets non sélectionnés, à chaque itération augmente ou reste inchangée (i.e. ne décroissent jamais).*

Démonstration: dans l'algorithme lorsqu'un sommet voit son étiquette modifiée, cela signifie qu'on préfixe cette étiquette par un nombre plus grand que celui qui constitué son préfixe, ainsi son étiquette ne peut qu'augmenter lorsque le sommet est dans le voisinage d'un sommet sélectionné. Si un sommet n'est pas dans le voisinage d'un sommet sélectionné alors son étiquette n'est pas modifiée. □

Invariant 2 (Lexicographique Décroissant) *Après chaque itération les boîtes non sélectionnées sont ordonnées (de gauche à droite) par ordre lexicographique décroissant.*

Démonstration: Nous effectuons la preuve par induction sur le nombre d'itérations. Base de l'induction : après la ligne 3 (ligne 1-3), nous avons à coté de s les sommets étiquetés par "$, 1\}$" (les voisins de s) "ε" (les non voisins de s), où ε représente le mot vide.

Supposons que la propriété soit vérifiée jusqu'à la $i-1$-ième itération. Montrons qu'elle reste vérifiée après la i-ième itération : soit $N_i(x)$ les boîtes qui sont translatées, et soit $\overline{N_i}(x)$ les sommets qui ne sont pas dans le voisinage de x. Par hypothèse d'induction les boîtes, à l'étape précédente, les boîtes était ordonnées, par conséquent cela reste vrai pour les boîtes non adjacente à x. Qu'en est il pour les boîtes des sommets adjacent à x ? Les sommets adjacent à x on vu leur étiquette modifiée en "$i, Anciennevaleur\}$". Etant donné que ces boîtes ont été translatées dans l'ordre, elle reste ordonnée (par hypothèse d'induction). Nous devons donc montrer que la plus petite boîte des voisins de x b_x (la plus à droite) est plus grande que la plus grande boîte des non voisins $b_{\bar{x}}$ (la plus à gauche). Les sommets de b_x ont tous étés préfixés par i, or i ne cesse d'augmenter par conséquent les sommets non voisins de x dans $b_{\bar{x}}$ ont leur première lettre qui est inférieure à i, on a donc $b_x > b_{\bar{x}}$. L'invariant reste donc vérifié après la i-ième itération. □

Théorème 22 (Equivalence d'étiquettes) *Lors de l'éxecution de LexDFS par affinage de partition, les sommet non sélectionnés sont dans une même boîte si et seulement si ils ont la même étiquettes.*

Démonstration:
$\Rightarrow$ S'ils sont dans la même boîte, cela signifie qu'ils ont les mêmes prédécesseurs(dans σ) par conséquent les prédécesseurs leurs communiquant chacun leurs valeurs, il en résulte qu'ils ont tous la même étiquette.
$\Leftarrow$ Par l'absurde, supposons que des sommets dans une même boîte n'ait pas la même étiquette, par conséquent cela signifie qu'il existe au moins deux sommets x et y qui n'ont pas la même étiquette. Pour cela, supposons que l "manque" à x, par conséquent x et y sont séparés par l, ce qui est absurde puisqu'ils sont dans la même boîte. □

Corollaire 4 (Sélection Maximum) *L'algorithme LexDFS par affinage de partition sélectionne, à chaque itération un des sommets qui a l'étiquette maximum.*

Démonstration: Evidente d'après le théorème "Lexicographiqe décroissant", en effet l'algorithme maintenant les boîtes triées par ordre décroissant, la première qui suit le sommet x est celle qui a la plus grande valeur. □

7.3 Algorithmes de vérification

7.3.1 Algorithme de vérification pour le parcours générique

En ce qui concerne le parcours générique il suffit de vérifier, une fois le triplet a, b, c, obtenu, qu'il n'existe aucun sommet d prédécesseur de b (*i.e.* $\sigma_1(d) < \sigma_1(b)$). Cette opération se fait en $O(1)$, il suffit pour cela de regarder la liste des prédécesseurs, cette dernière doit être vide. Nous sommes donc capable d'exhiber et vérifier un certificat en temps $O(n+m)$.

7.3.2 Algorithme de vérification pour BFS

Afin de vérifier que la permutation des sommets σ_1 n'est pas celle d'un parcours en largeur, nous devons un exhiber un triplet a, b, c pour lequel nous ne pouvons trouver aucun sommet d tel que $db \in E$ et $\sigma_1(d) < sigma_1(a)$. Pour ce faire, il suffit, une fois que nous avons trouvé b et c, de prendre le sommet a comme étant le plus petit prédécesseur de c (*i.e.* $a \leftarrow min\{N^-(c)\}$). Nous devons ensuite vérifier que tous les sommets d sont entre a et b.

Trouver le sommet a se fait en temps $O(1)$, ensuite vérifier que tous les sommets d sont entre a et b se fait en $O(|N^-(b)|)$, au pire en $O(n)$. Nous sommes donc capable d'exhiber un certificat et le vérifier en temps $O(n+m)$.

7.3.3 Algorithme de vérification pour DFS

Afin de vérifier que la permutation des sommets σ_1 n'est pas celle d'un parcours en profondeur, nous devons exhiber un triplet a, b, c pour lequel nous ne pouvons trouver aucun sommet d (prédécesseur de b) entre a et b dans σ_1. Nous devons tout d'abord trouver un sommet a, pour cela nous choisissons le sommet a comme étant le prédécesseur de c qui a la plus grande valeur (*i.e.* $a \leftarrow max\{N^-(c)\}$). Nous vérifions ensuite que tous les sommets d sont avant a.

La complexité de cette étape est en $O(n)$. En effet, trouver le sommet a se fait en temps $O(1)$, et vérifier que tous les sommets d sont avant a se fait en temps $O(|N^-(b)|)$, donc au pire $O(n)$. Nous sommes donc capable d'exhiber un certificat et le vérifier en temps $O(n+m)$.

7.3.4 Algorithme de vérification pour LexBFS

Pour vérifier que la permutation des sommets σ_1 n'est pas celle d'un parcours en largeur lexicographique (LexBFS), nous devons exhiber un triplet de sommet a, b, c, pour lequel il n'existe aucun sommet d tel que $db \in E$ et d avant a et $dc \notin E$. Pour cela nous devons trouver un sommet a prédécesseur de c.

Algorithme 10 : Vérification de LexBFS

Data : $N^-(c)$, $N^-(b)$ la liste des prédécesseurs de b et c ordonnés selon σ
Result : Sommet a

```
T_1 ← N^-(c)
T_2 ← N^-(b)
i ← 1
while T_1[i] ≠ T_2[i] do
    i ← i + 1
a ← i
```

Cela revient en quelque sorte à regarder les étiquettes des sommets b et c, et de regarder à partir de quel sommet a, cela ne marche plus.

7.3.5 Algorithme de vérification pour LexDFS

Pour vérifier que la permutation des sommets σ_1 n'est pas celle d'un parcours en profondeur lexicographique (LexDFS), nous devons exhiber un triplet de sommet a, b, c, pour lequel il n'existe aucun sommet d tel que $db \in E$ et d entre a et b avec $dc \notin E$. Pour cela nous devons trouver un sommet a prédécesseur de c.

Le principe est le même que pour LexBFS, mais au lieu de commencer par le préfixe des étiquettes, nous commençons par le suffixe. Une manière plus pratique de procéder consister à inverser la liste (ou le tableau), pour une manipulation plus aisée des indices.

Algorithme 11 : Vérification de LexDFS

Data : $N^-(c)$, $N^-(b)$ la liste des prédécesseurs de b et c ordonnés selon σ
Result : Sommet a

```
T_1 ← N^-(c)
T_2 ← N^-(b)
Inverser T_1
Inverser T_2
i ← 1
while T_1[i] ≠ T_2[i] do
    i ← i + 1
a ← i
```

Conclusions et perspectives

Nous avons étudié le problème des plus courts chemins à source unique, nous avons rappelé quels sont les algorithmes qui permettent de résoudre ce problème , et nous avons présenté un algorithme, dû à Hagerup, simple et linéaire en temps moyen pour résoudre ce même problème. Nous avons ensuite fourni une généralisation de l'algorithme de Dijkstra.

Nous avons également abordé la question de caractérisation de parcours, et nous avons présenté des méthodes simples et efficaces pour vérifier qu'une permutation ne correspond pas au parcours annoncé.
Dans le prolongement de ce stage de DEA, nous envisagons de traiter les problèmes suivants :

- Prolonger l'étude sur les plus courts chemins, notamment l'étude de Hagerup lors d'une école d'été sur les plus courts chemins à Copenhague (Danemark).
- Etudier le problème de plus courts chemins pour toutes les paires de sommets, lorsque la fonction de valuation est faiblement pré-additive et monotone croissante.
- Adapter la technique dû à Hagerup pour l'algorithme A^*.
- Etudier d'autres problèmes de plus courts chemins qui admettent des valuations non conventionnelles.
- Caractériser le parcours de l'algorithme de Dijkstra.
- Calculer un parcours LexDFS en temps linéaire.
- Etudier les propriétés du parcours LexDFS. Chercher les applications qu'il peut avoir.
- Vérifier de manière robuste et efficace les caractérisations de parcours de graphes.
- Concevoir des algorithmes pour effectuer les parcours, sur des graphes évolutifs, qui supportent l'ajout et la suppression de sommets et l'ajout et la suppression d'arêtes.

Au cours de ce stage, j'ai été amené à étudier et à généraliser des algorithmes de plus court chemins, concevoir des algorithmes efficaces, manipuler des structures de données plus ou moins complexes.

C'est pourquoi je souhaite prolonger cette expérience, en effectuant un thèse dans le domaine de l'algorithmique de graphes.

Bibliographie

[adH01] Friedhelm Meyer auf der Heide, editor. *Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings*, volume 2161 of *Lecture Notes in Computer Science*. Springer, 2001.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows : Theory, Algorithms and Applications*. Prentice Hall, 1993.

[Bar99] Régis Barbanchon. Etude de l'algorithme de thorup plus courts chemins à source fixée en temps quasi-linéaire. Master's thesis, GREYC Caen, 1999.

[CK04] Derek G. Corneil and Richard Krueger. A unified view of graph searching, 2004.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest, and Clifford Stein. *Introduction to algorithms, Second Edition*. The MIT Press, 2001.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Matematik*, Volume 1 :269–271, 1959.

[Din78] E.A. Dinic. Economical algorithms for finding shortest path in a network. *Transportation Modeling Systems*, pages 36–44, 1978.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3) :596–615, 1987.

[Gol01] Andrew V. Goldberg. A simple shortest path algorithm with linear average time. In auf der Heide [adH01], pages 230–241.

[Hag04] Torben Hagerup. Simpler computation of single-source shortest path in linear average time. In Habib Diekert, editor, *STACS 2004*, pages 362–369. spinger, march 2004.

[HS91] M Habib and G Simonet. Quelques remarques sur A*. Research report, LIRMM, july 1991.

[Kin95] Valerie King. A simpler minimum spanning tree verification algorithm. In *Workshop on Algorithms and Data Structures*, pages 440–448, 1995.

[KKT95] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2) :321–328, 1995.

[KLS91] Bernhard Korte, László Lovász, and Rainer Schrader. *Greedoids*, volume 4 of *Algorithmic and Combinatorics*. Springer-Verlag, 1991.

[Knu98] Donald E. Knuth. *The art of computer programming, Sorting and Searching, Second Edition*, volume 3. Addison-Wesley, 1998.

[Mey01] Ulrich Meyer. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *Symposium on Discrete Algorithms*, pages 797–806, 2001.

[Mey03] Ulrich Meyer. Average-case complexity of single-source shortest-paths algorithms : lower and upper bounds. *J. Algorithms*, 48(1) :91–134, 2003.

[MS91] B.M.E. Moret and H.D. Shapiro. *Algorithms from P to NP*, volume 1 ,Design & Efficiency. Benjamin - Cimmings, 1991.

[MS03] J. Monnot and O. Spanjaard. Bottleneck shortest paths on a partially ordered scale. *4OR*, 1(3) :225–241, 2003.

[PS04] P. Perny and O. Spanjaard. A preference-based approach to spanning trees and shortest paths problems. *European Journal of Operational Research, to appear*, 2004.

[RL68] F. Roberts and R. Luce. Axiomatic thermodynamics and extensive measurement. *Synthese*, tm 18 :311–326, 1968.

[Spa03] Olivier Spanjaard. *Exploitation de préférences non-classiques dans les problèmes combinatoires : modèles et algorithme pour les graphes*. PhD thesis, LAMSADE - Paris IX - Dauphine, december 2003.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[Tho97] Mikkel Thorup. Undirected single source shortest path in linear time. In *IEEE Symposium on Foundations of Computer Science*, pages 12–21, 1997.

Annexe A

Variations sur les plus courts chemins

Dans le cadre de l'étude du problème de plus court chemin selon des valuations partiellement ordonnées, nous avons préalablement étudié des problèmes qui admettent des valuations non conventionnelles ou des ordres non conventionnels. Nous avons tout d'abord étudié le problème de plus court chemin selon le produit, nous avons ensuite regardé ce qui ce passait lorsque l'on cherche un plus court chemin modulo, et enfin les plus court chemin selon un treillis d'inclusion. Il résulte de cette étude que l'ensemble de ces problèmes sont NP-Complets.

A.1 Plus court chemin produit

Nous étudions dans cette partie le calcul d'un plus court chemin dans un graphe valué, lorsque l'opérateur de «concaténation» est le produit (*i.e.* la valeur d'un chemin est le produit des valuations qui le composent). Le problème du plus court chemin selon le produit s'énonce de la manière suivante :
Donnée : un graphe $G = (V, E)$, $\omega : E \to \mathbb{R}$, et s un sommet source. V représente l'ensemble des sommets et E représente l'ensemble des arêtes. ω est une fonction de valuation des arêtes.
Résultat : Une arborescence de plus court chemin allant de s à tous les autres sommets qui sont minimum selon le produit $\times$. Le problème sera abrégé par PCCΠ.
Nous allons montrer que PCCΠ sur l'intervalle $[1, +\infty)$ est polynômial et que le même problème sur $(0, 1)$ est NP-Complet.
Rappel sur les logarithmes :

$$ln(a.b) = ln(a) + ln(b)$$

Pour résoudre se problème on se ramène au problème de plus court chemin selon l'addition. Pour ce faire, nous transformons toutes les valuations par leurs logarithmes, $t : x \to ln(x)$. Une fois cette transformations effectuée on cherche un plus court chemin selon l'addition. On sait, dans le cas de l'addition, que lorsque les valuations sont positives, le problème est polynomial et quand les valuations sont négatives, le problème est NP-Complet, (On se ramène à la recherche d'un plus long chemin quand les valuations sont positives). Une fois que l'on a obtenu un résultat sur l'instance transformée, on souhaite retrouver la valeur du chemin obtenu dans l'instance initiale. Pour cela, il suffit d'effectuer la somme des éléments du chemin, et ensuite passer à l'exponentielle. On s'appuie pour cela sur les propriétés du logarithme et de l'exponentielle.

Par conséquent l'étude de variation de la fonction logarithme nous dit que $ln(x)$ est positif sur l'intervalle $[1, +\infty)$, et négative entre 0 et 1.

Malheureusement la fonction logarithme n'est définie que pour les réels positifs, par conséquent notre transformation ne nous indique rien sur la complexité du problème lorsque les valuations sont négatives. Cela étant la, version du problème, lorsque les valuations sont négatives, est NP-Complet. Nous distingons deux cas : la situation sur $(-\infty, -1)$ et sur $(-1, 0)$.

Remarquons que le produit de deux nombres négatifs est un nombre positif. Par conséquent dans un graphe où les valuations sont négatives, tout chemin composé d'un nombre impair d'arêtes est plus petit que tout chemin composé d'un nombre pair d'arêtes. Nous effectuons la réduction à partir de cycle Hamiltonien.

$$\text{Cycle Hamiltonien} \leq_K \text{PCCΠ}(-\infty, -1)$$

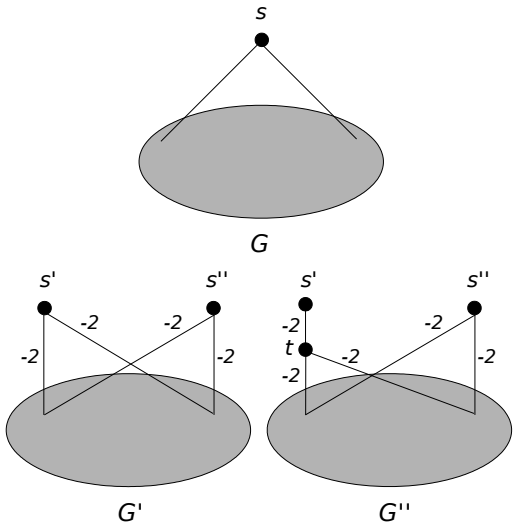

FIG. A.1 – Réduction pour le produit avec des valuations négatives

Réduction Soit G le graphe dans lequel on cherche un cycle Hamiltonien. Soit V l'ensemble des sommets, et E l'ensemble des arêtes. Le nombre de sommets est représenté par $n = |V|$, et le nombre d'arêtes par $m = |E|$. Dans G on choisit un sommet quelconque s. Nous transformons G en $G' = (V', E')$ de la manière suivante : on dédouble le sommet s en s' et s'' dans G'. On conserve dans G' le voisinage de s, c'est à dire $N(s) = N(s') = N(s'')$ et $(s', s'') \notin E$. Une fois le graphe G' obtenu, nous valuons toutes les arêtes par un nombre négatif (par exemple -2). Si n est impair, la transformation telle quelle est suffisante, cela revient à chercher de s' à s'' un plus court chemin élémentaire de coût -2^n. Si par contre n est pair il -2^n sera positif, il existe peut être un chemin de s' à s'' dont le coût est inférieur mais qui ne passe pas par tous les sommets. C'est pourquoi lorsque n est pair dans G' on rajoute un sommet au dessus de s', qui devient s'. Cela a pour but d'assurer que le chemin de s' à s'' ait un nombre impair d'arêtes. Dans cette version on cherche un chemin de coût -2^{n+1} (cf FIG. A.1 p. 34).

Polynômialité de la réduction La réduction présentée ci-dessus est effectivement polynômiale, en effet le nombre de sommet dans G' est soit $n + 1$ soit $n + 2$, de plus le nombre d'arêtes dans G' est de l'ordre de m, en effet on ajout $|N(s)|$ arêtes, plus éventuellement une (dans le cas de n pair) par conséquent on a au plus $m + n$ arêtes dans G'. Nous ajoutons en plus des valuations sur les arêtes, ces valuations étant toutes identiques (x avec $x \in (-\infty, -1)$), il n'est pas absolument nécessaire de coder ces valuations (puisqu'identiques). Néanmoins si l'on souhaite tout de même coder ces valuations, on doit pouvoir s'en tirer avec 1 bit par arêtes (du fait qu'il n'y qu'une seule valeur). Il en résulte que notre réduction est polynômiale en la taille de l'instance initiale.

Validité de la réduction Démontrons maintenant que la réduction est valide. A savoir que si l'instance du problème cycle Hamiltonien admet une solution alors, l'instance transformée admet un plus court chemin élémentaire de s' à s'' de coût -2^n (ou -2^{n+1} dans le cas pair).
$\Rightarrow$ Si l'instance cycle Hamiltonien admet une solution alors l'instance transformée admet chemin de s' à s'' de coût -2^n (dans le cas impair). Soit C le cycle Hamiltonien dans G, il correspond dans G' à un chemin μ, de plus C est élémentaire par conséquent, il en est de même pour μ. Donc le chemin μ passe par tous les sommets, il est donc composé de n arêtes. Les arêtes étant valuées par -2, le coût est donc -2^n. Dans le cas pair, on a rajouté une sommet ainsi qu'un arête, c'est à dir qu'au cycle C correspond un chemin μ qui relie t à s'', cependant on peut sans se tromper, ajouter à ce chemin μ l'arête (s', t), par conséquent le chemin ainsi obtenu reste élémentaire, et de coût -2^{n+1}.
$\Leftarrow$ Dans le cas impair, soit μ un plus court chemin de coût -2^n reliant s' à s'', ce chemin étant élémentaire,

on en déduit, par son coût, qu'il passe par tous les sommets, on peut donc lui associer dans G un cycle Hamiltonien passant par G. Dans le cas pair, si on a un chemin élémentaire μ de coût -2^{n+1}, on a donc un chemin qui passe par tous les sommets de G' (toujours à cause du coût et du fait qu'il est élémentaire). On peut donc en "extraire" un cycle Hamiltonien dans G, pour cela dans μ on supprime le sommet t, et on relie s' au successeur de t dans μ, on obtient ainsi un chemin de taille n qui passe par tous les sommets de G' privé de t, et trivialement on obtient dans G un cycle Hamiltonien en contractant s' et s''.

□

A.1.1 PCCΠ sur $(-1, 0)$

Dans l'intervalle $(-1, 0)$, on remarque que lorsque le graphe est uni-valué, trouver le plus court chemin entre deux sommets consiste à trouver soit un plus court chemin impair (en nombres d'arêtes), soit un plus long chemin pair (toujours en nombres d'arêtes). Par conséquent cette version du problème est également NP-Complète.

A.2 Plus court chemin modulaire $\mathbb{Z}/p\mathbb{Z}$

Intéressons-nous maintenant au problème de plus-court chemin lorsque les valuations sont sur $\mathbb{Z}/p\mathbb{Z}$, et que l'opérateur de «concaténation» est l'addition. Le problème s'énonce de la manière suivante :
Données : Un graphe $G = (V, E)$, une fonction de valuation ω sur les arêtes définie de la manière suivante $\omega : E \to \mathbb{Z}/p\mathbb{Z}$, un entier k appartenant à $\mathbb{Z}/p\mathbb{Z}$, un sommet source s et un puits t.
Résultat : Une chaîne élémentaire allant de s à t de coût inférieur ou égal à h.

Il est clair que le problème est dans NP, en effet le certificat est la chaîne. Nous pouvons vérifier ce certificat en temps linéaire.

Nous allons démontrer que lorsque les valuations sont sur $\mathbb{Z}/p\mathbb{Z}$, et que l'opérateur considéré est l'addition alors le problème est NP-Complet. La réduction s'effectue à partir du problème cycle Hamiltonien.

$$\text{Cycle Hamiltonien} \leq_K \text{plus court chemin } \mathbb{Z}/p\mathbb{Z}$$

Réduction La transformation s'effectue de la manière suivante (cf FIG. A.2, p 36) : nous modifions le graphe initial, G en G', en sélectionnant un sommet s, et en dédoublant celui-ci, (notre sommet s devient s' et s''). Nous préservons l'adjacence du sommet s pour les sommets s' et s'', (i.e. $N(s) = N(s') = N(s'')$ et $(s', s'') \notin E(G')$). Dans G', nous pondérons les arêtes avec un poids de 1, nous fixons le paramètre p (de $\mathbb{Z}/p\mathbb{Z}$) à n (où $n = |V(G)|$) et nous fixons le paramètre k à 0. Ainsi, dans G' nous allons chercher un chemin allant de s' à s'' de poids inférieur ou égal à 0. Si un tel chemin existe alors nous avons trouvé un cycle Hamiltonien, sinon il n'y a pas de cycle Hamiltonien dans notre problème initial.

Polynômialité de la réduction Montrons maintenant que la réduction est polynômiale, la réduction que nous effectuons ajoute un sommet à l'ensemble V, lorsque nous dédoublons le sommet s (*i.e.* $|V'| = |V| + 1$). Le nombre d'arrêtes dans le graphe transformé est égal au nombre d'arêtes dans E, plus $|N(s)|$, le nombre de voisins de s dans G (*i.e.* $|E| < |E'| \leq |E| + n - 1$). On ajoute également $|E'|$ valuations sur les arêtes. Par conséquent la réduction est polynômiale.

Validité de la réduction Démontrons maintenant que la réduction formulée précédemment est valide. Le problème initial admet un cycle Hamiltonien si et seulement si le problème transformé admet un chemin de s' à s'' de coût 0.
$\Rightarrow$ Supposons que le problème initial admette un cycle Hamiltonien, que l'on notera C. Ce cycle C étant Hamiltonien dans G, il passe nécessairement par s, il existe alors dans G' un chemin élémentaire reliant s' à s'' qui n'est autre que le cycle C qui a été coupé au niveau de s. Par conséquent le chemin reliant s' à s'' est composé de n arêtes, et il admet donc un coût de 0. Donc si l'instance initiale admet un cycle Hamiltonien alors l'instance transformé admet un chemin reliant s' à s'' de coût nul.
$\Leftarrow$ Supposons maintenant que le problème transformé admette un chemin élémentaire allant de s' à s'' de coût nul. Nous notons ce chemin μ. Le chemin μ étant élémentaire et ayant un coût nul, cela signifie qu'il

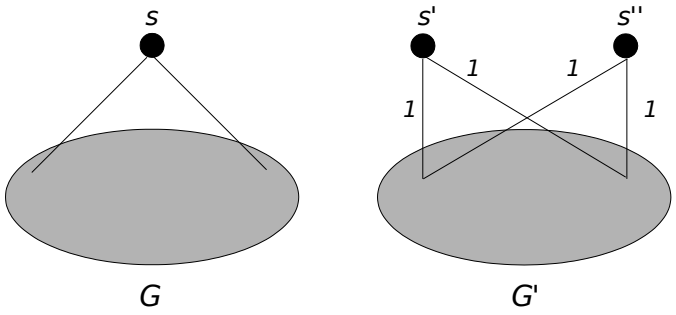

FIG. A.2 – Transformation du problème cycle Hamiltonien

passe par tous les sommets. Nous pouvons aisément associer ce chemin dans G' à un cycle Hamiltonien dans G, cela se fait en contractant les sommets s' et s'' en un sommet s ; notre chemin μ devient dans G un cycle Hamiltonien.

□

A.3 Plus court chemin selon un ordre treillis

Nous nous intéressons maintenant au problème du plus court chemin dans un graphe lorsque les valuations sont plongées dans un treillis.

Rappel du treillis : pour tout couple d'éléments (x, y) deux à deux distincts, il existe toujours une borne inférieure unique (le maximum des minorants), et une borne supérieure unique(le minimum des majorants). Afin de comparer deux chemins, nous considérons les antichaînes de leurs éléments maximaux.
Nous allons montrer que cette fois encore le problème est NP-Complet. La réduction se fait à partir de 3-SAT (voir FIG. A.3 p 37). Nous transformons les instances de 3-SAT en un problème de plus court chemin lorsque l'opérateur de concaténation est l'union, et la relation d'ordre est un treillis d'inclusion.

$$\text{3-}SAT \leq_K \text{Plus Court Chemin Treillis Union}$$

Réduction Nous transformons une instance de 3-SAT de la manière suivante : nous créons un graphe G constitué de deux parties, une partie représentant les variables (*i.e.* x et $\bar{x}$) et une autre représentant les clauses. Le graphe est constitué d'un sommet source qui noté $\emptyset$, et d'un sommet «arrivée» noté $\top$. Les variables sont transformées en deux sommets, un représentant le littéral et l'autre son opposé. Une illustration de la transformation est donné FIG. A.3 p 37. Les clauses sont transformées de la même manière.

Polynômialité de la réduction Nous montrons maintenant que la réduction est polynômiale. Soit n le nombre de variable dans le problème initial et soit m le nombre de clauses. Notre réduction est constitué de $n-1$ $K_{2,2}$ et de $m-1$ $K_{3,3}$ plus deux arêtes pour le point de départ ($\emptyset$) et trois arêtes pour l'arrivée en $\top$. Où $K_{2,2}$ est le graphe biparti complet constitué de deux sommets dans chaque ensemble X et Y, un tel graphe admet 4 arêtes. $K_{3,3}$ est le graphe composé de 3 sommets dans chaque ensemble X et Y, un tel graphe admet 9 arêtes. Par conséquent le nombre de sommets engendré par la réduction peut être exprimé de la manière suivante : $|V| = 2n + m + 2$. Et le nombre d'arêtes : $|E| = 4n + 9m + 5$. La taille de la réduction est exprimable par un polynôme fonction de la taille de la donnée initiale (*i.e.* la réduction est polynômiale).

Validité de la réduction Nous démontrons que la réduction formulée ci-dessus est valide. Toute instance de 3-SAT admet une solution si et seulement si l'instance transformée admet un chemin élémentaire reliant $\emptyset$ à $\top$ dont l'antichaîne des éléments maximaux est de taille n.

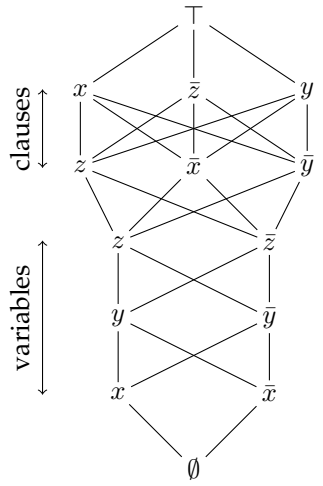

FIG. A.3 – Réduction à 3 SAT

$\Rightarrow$ Si l'instance de 3-SAT admet une affectation des variables telle que la formule soit satisfiable, alors avec cette affectation on peut construire dans G un chemin allant de $\emptyset$ à $\top$ en garantissant que la taille de l'antichaîne est n, cela se fait comme suit : si une variable est affecté à vrai alors le chemin passe par son littéral (ex : x) sinon on passe par son opposé (*i.e.* $\bar{x}$) dans la partie des variables, ensuite pour "traverser" les clauses on choisit dans la clause une variable qui apparaît dans l'affectation, une telle variable existe toujours car l'instance est satisfiable.
$\Leftarrow$ Si l'instance du plus-court chemin selon le treillis d'inclusion admet un chemin élémentaire allant de $\emptyset$ à $\top$ avec une antichaîne de taille n, alors l'instance de 3-SAT qui lui est associée est satisfiable. En effet, si l'instance de 3-SAT était insatisfiable, dans notre graphe G on serait obligé de passer par une variable et son opposé dans une clause ce qui entrainerait obligatoirement une antichaîne de taille au moins $n + 1$, ce qui n'est pas le cas. Donc la réduction est valide.

□

La figure A.3 est la transformation effectuée pour l'instance $(x \vee \bar{z} \vee y) \wedge (z \vee \bar{x} \vee \bar{y})$.

Annexe B

Arbre recouvrant de poids minimum

B.1 Historique

Le problème de l'arbre couvrant de poids minimum (ACPM) (ou MST pour Minimum Spanning Tree), est un problème qui a été très étudié. La première méthode pour résoudre le problème est dû à Borůvka (1926), la technique était destinée au placement d'un réseau electrique en Moravie du sud. Par la suite Kruskal puis Prim proposèrent des algorithmes, pour résoudre ce problème. La technique proposé par Kruskal consiste à seléctionner à chaque étape, l'arête qui a un poids minimum, et qui ne crée pas de cycle. La technique de Prim est sensiblement différente, à savoir que la construction s'effectue à partir d'un sommet, et on sélectionne à chaque étape la plus petite arête du cocyle.

Le problème de l'ACPM, est intéressant selon plusieurs points de vues : les algorithmes qui permettent de le résoudre sont gloutons, le poids de l'arbre n'est jamais calculé, en effet "tous" les algorithmes effectuent des considérations locales (trouver une arête de poids minimum) sans jamais effectuer une seule somme. Il est intéressant de voir qu'une structure de matroïde se "cache" derrière ce problème, ce qui n'est pas le cas pour les plus court chemins.

B.2 Plus court chemin Max

Nous nous intéressons maintenant aux relations qu'il y a entre le problème du plus court chemin selon la fonction Max (*i.e.* $f : \mu \rightarrow Max(\mu)$) et l'arbre couvrant de poids minimum. En effet calculer un ACPM d'un graphe valué revient à calculer les plus courts chemins selon Max, malheureusement la réciproque n'est pas vraie. (cf FIG. B.1 p. 39)

B.3 Quand Dijkstra rencontre Prim

Nous nous intéressons maintenant à un nouveau problème, qui est à la fois un problème de plus court chemin, et à la fois un problème d'arbre recouvrant de poids minimum. En effet, dans le cadre des plus court chemins selon la somme, il est possible de rencontrer des sommets pour lesquels, il existe plusieurs chemins de coût identique. Par conséquent un arbre des plus court chemins de poids minimum, serait un arbre des plus court chemin classique, avec la propriété particulière, d'avoir un poids minimum.

La solution proposée consiste lors de l'exécution de l'algorithme de Dijkstra de regarder en cas d'ex aequo, quelle arête a le coût minimum. La modification à apporter est la gestion en cas d'ex aequo et, il faut également regarder les sommets **Fermés**.

B.4 Arbre Couvrant de poids minimum et plus court chemin

Le "dual" du problème précédent serait, étant donné un graphe G, et un sommet racine r, trouver l'arbre couvrant de poids minimum qui soit en plus un arbre des plus court chemin enraciné en r. En clair on

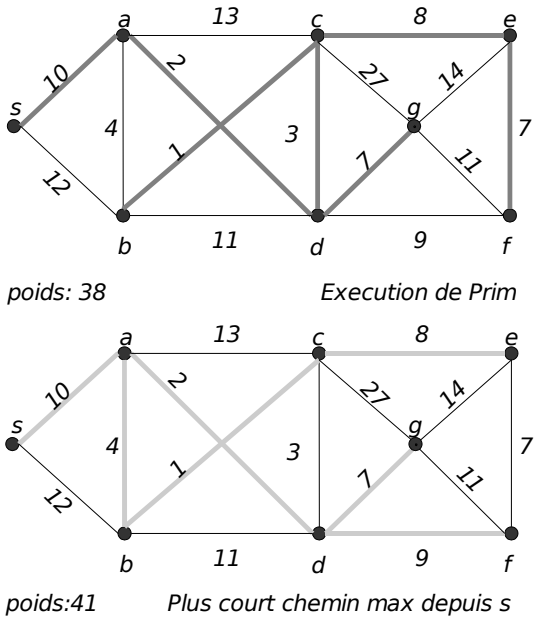

FIG. B.1 – Cas critique pour la fonction Max

cherche un arbre de poids minimum enraciné en r qui minimise les la longeur des chemins r à tous les autres sommets.