**MASTER EN SCIENCES MATHÉMATIQUES**

# Approche alternative du calcul des plus courts chemins dans le projet VirtualBelgium

HENROTIN, William

*Award date:*
2014


**UNIVERSITE DE NAMUR**

**Faculté des Sciences**

# Approche alternative du calcul des plus courts chemins dans le projet VirtualBelgium

Promoteur : Philippe Toint

Co-promoteur : Eric Cornelis

**Mémoire présenté pour l'obtention**

**du grade académique de master en « sciences mathématiques à finalité spécialisée »**

William HENROTIN

Août 2014

# Résumé

Le projet VirtualBelgium est une plateforme de simulation multi-agents créée par les chercheurs du centre GRT à Namur. Cet outil a pour ambition de simuler, via plusieurs aspects, la population belge, sur base d'une population synthétique. À l'heure actuelle, deux importants modules composent ce projet : une simulation temporelle de la population d'un côté et une simulation spatiale de l'autre. Pour la simulation spatiale, le programme tente de reproduire les déplacements de chaque individu de la population étudiée pendant une journée type. Ces déplacements prennent part au déroulement de ce qu'on appelle une chaine d'activités.

Ces déplacements effectués dans un graphe représentant le réseau routier belge sont calculés pour les simulation par un algorithme de Dijkstra optimisé. Ce mémoire présente une alternative et la compare avec cet algorithme dans le but d'optimiser les temps d'exécutions des calculs de ces plus courts chemins.

**Mots clés :** Dijkstra, Floyd, algorithme de plus courts chemins, programmation parallèle

# Abstract

The VirtualBelgium project is a multi-agents based simulation framework built by researchers of the GRT center in Namur. This tool aims to simulate, by many aspects, the belgian population, thanks to a synthetic population. Two main module are built in the project now : The first one produces a time simulation while the other a spatial simulation. For the second one, the program tries to reproduce movements of each person during a common day. These displacements take place into a concept we called activity chains.

These movements of people within a graph representing the belgian road network are computed in the simulation by a optimised Dijkstra algorithm. This master thesis presents an alternative and compares it with this algortihm to bring a potential optimisation about the execution time of the shortest path computing.

**Keywords :** Dijkstra, Floyd, shortest path algorithm, parallele computing

# Remerciements

Je voudrais tout d'abord remercier pour l'encadrement, la documentation, les relectures et les conseils, Philippe Toint, mon promoteur et Éric Cornelis, mon co-promoteur.

Ce mémoire s'inscrit dans un projet global, un grand nombre de personnes y ont apporté leur contribution. Je souhaiterais donc remercier

- André Füzfa, pour le prêt du livre m'ayant permis d'initier ce travail.
- Johan Barthelemy, pour ses réponses précises aux problèmes techniques, sa disponibilité et son soutien bienveillant.
- Stéphanie Marchal, Gaëlle Picard et Élodie Ramelot, pour la partie commune de ce travail ainsi que pour leur aide.

Je tiens aussi à remercier Vasco Henrotin pour la correction de la partie anglaise, Dorothée Dejaegher, Morgane Dumont, Isabelle Jardon, Charlotte Dejaegher pour les nombreuses relectures orthographiques et pour leur soutien moral. Et enfin, je voudrais exprimer ma reconnaissance à tous ceux qui m'ont proposé leur aide.

# Table des matières

# Introduction

La simulation de la vie réelle est devenue ces dernières années un outil très intéressant, tant pour l'analyse que pour prévoir des observations futures. Les problématiques de mobilité, de démographie et de choix résidentiel peuvent être étudiées par ce biais. Un outil permettant ce genre d'étude a été créé par le groupe GRT de l'université de Namur et appelé VirtualBelgium. Il s'agit d'une plateforme multi-agents, pour l'instant appliquée à la Belgique, permettant de simuler certains aspects de la vie d'une population comme les données démographiques (naissance, mort, ...) géographiques (déménagement) et sociales (emploi, mariage, ...).

Cet ambitieux projet est décrit plus en détail au chapitre 1. Dans ce cadre-là, nous présentons ce mémoire comme proposant l'analyse d'une alternative possible à une partie du projet : celle concernant l'évolution spatiale. Cette partie tente de simuler une journée type pour chaque individu de la population synthétique. Ces journées sont symbolisées par une suite d'activités que l'individu pratique l'une après l'autre. Le but de la simulation est de proposer des lieux pour chacune de ces activités sur le réseau. Les lieux possibles sont à choisir parmi les nœuds du réseau routier servant à la simulation. Pour cela, nous connaissons l'endroit de résidence de chaque individu et, pour toute activité, la distance qu'il parcourt pour y arriver.

Ce problème se résume à des calculs de plus courts chemins. Ainsi, nous devons calculer les distances entre le départ et toutes les arrivées possibles. A partir de là, nous isolons celles assez proches du départ. Enfin nous en choisissons une aléatoirement suivant une loi uniforme pour être le lieu de l'activité.

Actuellement, l'algorithme calculant ces plus courts chemins implémenté dans VirtualBelgium est celui de Dijkstra. Il a été longuement optimisé grâce à l'utilisation de structures particulières d'arbres de Fibonacci. Toutes ces notions seront expliquées en détail dans le chapitre 2.

Cette méthode a montré une certaine limite lorsqu'il s'agit de faire tourner une simulation impliquant la population belge entière sur le réseau routier complet. En effet, avec comme données 262.000 nœuds, 830.000 arcs et $10^7$ agents, les temps de calculs dépassent les 5 jours, même sur un *cluster* de 500 processeurs.

Il est évident que de tels temps deviennent problématiques dans une optique d'utilisation fréquente de VirtualBelgium. Cet outil pourrait par exemple prétendre à simuler le trafic routier quotidiennement. Il serait alors primordial de réduire si-

gnificativement ces temps avec une durée maximale d'un jour.

Cette problématique du temps d'exécution sera notre fil conducteur tout au long de ce travail. Pour tenter de la résoudre, nous présenterons une méthode alternative aux calculs des plus courts chemins. Celle-ci s'appelle la méthode de Floyd et sera décrite au chapitre 3. Ensuite, la comparaison entre ces deux algorithmes sera réalisée au chapitre 4.

# Chapitre 1

# Présentation de VirtualBelgium

La plateforme de simulation VirtualBelgium a pour ambition de comprendre l'évolution d'une population via une simulation de différents aspects : démographie, choix résidentiel, mobilité, ... Ce projet est mené par le Groupe de Recherche sur les Transports (GRT-naXys) et constitue une partie de la thèse de J. Barthelemy[BJ14]. Cet outil est actuellement appliqué à la Belgique.

Dans cette section, nous expliquons les différents éléments qui composent la plateforme mais également les différentes données nécessaires à son utilisation. Á l'heure actuelle, l'outil est appliqué à la Belgique.

## 1.1 Génération d'une population synthétique

Avant de pouvoir simuler une quelconque évolution avec l'outil VirtualBelgium, ce dernier a besoin d'une population initiale. Malheureusement, pour des raisons de confidentialité et de budget, il n'est pas possible de récupérer les données personnelles attendues concernant chaque citoyen belge. Il faut donc générer une population synthétique qui doit être aussi statistiquement proche que possible de la population réelle belge.

Cette population synthétique a été générée par J. Barthélemy et Ph. Toint selon une nouvelle méthode qu'ils proposent dans l'article [BJ10]. Celle-ci est composée de 10 262 160 individus répartis en 4 236 202 ménages dans les 589 communes belges. Les données utilisées dans le processus concernent la population belge en 2001 et proviennent des sources suivantes :

- *Directorate-general Statistics and Economic information* du gouvernement fédéral belge (2001)
- *Service public fédéral Mobilité et Transports* du gouvernement fédéral belge (2000)
- *GéDAP*[1] centre de l'université catholique de Louvain(2001)

1. Groupe d'étude de démographie appliquée

- L'enquête nationale de mobilité *MOBEL* [HJP02]

Une fois cette population synthétique créée, son évolution est prise en charge, dans VirtualBelgium, par un programme implémenté en C++ dont la structure est présentée à la figure 1.1. Ce programme simule une évolution temporelle (démographie) et spatiale (mobilité) de la population synthétique, variant selon les données qui lui sont fournies et utilisant le framework Repast HPC, présenté dans la section suivante.

Après l'exécution du programme, des outputs tels que des statistiques concernant l'évolution ou encore des fichiers contenant les déplacements des individus sont créés. Ils sont ensuite exploités par le logiciel MATSim qui permet une visualisation du trafic routier. Ce logiciel sera également présenté dans la section suivante.

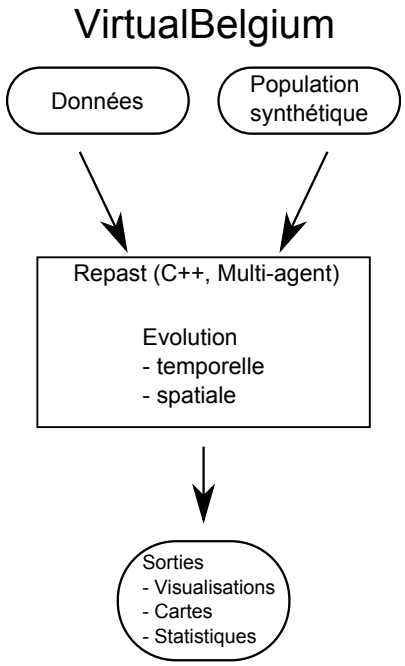



FIGURE 1.1 – Structure de VirtualBelgium

La suite de ce chapitre permet de comprendre l'architecture du programme VirtualBelgium lié à la plateforme. Il est conseillé de connaitre certaines notions des langages de programmation orientée objet.

## 1.2 Les logiciels utilisés

Comme nous venons de le dire dans la section précédente, les deux logiciels principalement utilisés par le programme VirtualBelgium sont Repast HPC et MATSim. Leur utilisation et leurs principales caractéristiques sont décrites ci-dessous.

### 1.2.1 Repast HPC

Repast HPC (REcursive Porous Agent Simulation Toolkit for High Performance Computing) est un framework C++ utilisé pour la simulation et la modélisation de systèmes multi-agents. D'après la référence [mul], on appelle multi-agents un système où plusieurs agents évoluent et qui possède des caractéristiques telles que le parallélisme [2], la robustesse [3] et l'extensibilité [4]. Le terme « agent » peut être interprété de plusieurs manières en informatique ; dans notre cas, un agent est un système informatique situé dans un environnement qu'il peut modifier, qui agit de manière autonome et flexible pour atteindre les objectifs qui lui sont fixés. Les agents sont capables d'interagir entre eux.

Repast est un outil gratuit, open source, et une plateforme de modélisation et simulation multi-agents. Cet outil est surtout utilisé pour des applications en sciences sociales. À l'origine, il a été développé par David Sallach ainsi que d'autres chercheurs de l'Université de Chicago et du Laboratoire National D'Argonne. Désormais, il est dirigé par l'organisation de bénévoles ROAD (Repast Organization for Architecture and Development).

Son architecture permet de représenter chaque agent par un identifiant unique et prend en charge la gestion des opérations parallèles grâce aux librairies MPI et Boost. Dans VirtualBelgium, par exemple, les individus et les ménages de la population synthétique sont représentés par des agents.

Dans l'annexe reprenant un manuel d'utilisation de VirtualBelgium, les indications pour l'installation de Repast HPC sont reprises de même que des informations relatives à la compilation et au débugage dans cet environnement.

### 1.2.2 MATSim

MATSim (Multi-Agent Transport Simulation) est un outil de simulation de transport. Il s'agit d'un projet open source principalement créé par les groupes suivants :

- *Transport Systems Planning and Transport Telematics*, Institute for Land and Sea Transport Systems, Technische Universität Berlin
- *Transport Planning*, Institute for Transport Planning and Systems, Swiss Federal Institute of Technology Zurich
- *Senozon*, entreprise suisse avec une filiale en Allemagne

2. "En informatique, le parallélisme consiste à implémenter des architectures d'électronique numérique permettant de traiter des informations de manière simultanée, ainsi que les algorithmes spécialisés pour celles-ci. Ces techniques ont pour but de réaliser le plus grand nombre d'opérations pour réduire le temps d'exécution." Cette définition provient du site de référence [wik].

3. "La robustesse d'un programme est entre autres, sa capacité à bien fonctionner". Cette définition est extraite du site de référence [inf].

4. "Un système est extensible si on peut étendre ses capacités par ajout de mises à jour ou de modules." Cette définition est extraite du site de référence [inf].

Dans VirtualBelgium, MATSim crée le réseau utilisé par le programme, à partir des données extraites du site OpenStreetMap [ope]. *OpenStreetMap est une carte du monde librement modifiable, collaborative où les données sont libres d'être téléchargées et utilisées, sous les termes d'une licence ouverte.*

## 1.3 Cœur du programme

Regardons à présent l'architecture du programme VirtualBelgium. Celle-ci est représentée par un schéma, en annexe, qui décrit les différentes classes implémentées. Une classe est un principe utilisé dans la programmation orientée objet. Elle est constituée d'un ensemble d'attributs et de fonctions, appelées méthodes.

Sur cette figure, chaque cadre correspond à une classe du programme dont le nom est inscrit en haut. Ces cadres sont divisés en deux parties : la première contient les variables de la classe et la deuxième les méthodes principales. De plus, des liens indiquent les relations éventuelles existantes entre les différentes classes.

Parcourons les différentes classes et résumons leurs rôles et leurs principales caractéristiques.

### 1.3.1 Classes générales

La classe `Data` est chargée de lire et récupérer toutes les données nécessaires au programme VirtualBelgium. Les données utilisées sont les suivantes :

- *propriétés des modèles* telles que l'évolution spatiale ou temporelle ;
- *données socio-démographiques* : pyramide des âges par commune pour les hommes et les femmes, probabilité de naissance par âge, probabilité de mort par âge et sexe, probabilité d'avoir un garçon ou une fille ;
- *données relatives au réseau* : noeuds représentant chaque commune et réseau routier ;
- *données relatives au modèle d'activités* : codes pour les activités, paramètres de distribution concernant les activités (distance, durée du voyage, durée de l'activité et heure de départ et d'arrivée).

La classe `RandomGenerators` contient des générateurs [5] de nombres pseudo-aléatoires suivant une loi de probabilité, utilisés tout au long du programme.

La classe `Model` se charge principalement de l'initialisation du modèle et décrit la procédure d'évolution que subit la population via la méthode `step`. La méthode `step` est utilisée pour itérer sur tous les agents du modèle afin qu'ils accomplissent

5. Un générateur de nombres pseudo-aléatoires est un outil permettant de ressortir une suite de nombres suivant une loi demandée à partir d'un nombre qu'on appelle graine (seed). Le générateur n'est pas valide à 100%. En effet, deux graines identiques dans des générateurs identiques produiront deux suites de nombres identiques ; ce qui n'est pas le but d'un générateur de nombres **aléatoires**.

les tâches demandées par le modèle, telles que vieillir, mourir, déménager, se rendre à leurs activités de la journée, se marier ou divorcer, ...

### 1.3.2 Individus et Ménages

VirtualBelgium comporte deux différents types d'agents : les ménages représentés par la classe `Household` et les individus par la classe `Individual`. À chaque agent sont associés plusieurs variables ainsi qu'un identifiant unique, déterminé par le logiciel Repast.

Un ménage contient les identifiants de chaque individu le composant ; l'identifiant du nœud du réseau correspondant au domicile du ménage ainsi que le code INS[6] de sa commune. La classe `Household` contient également des attributs concernant le nombre d'enfants (de 0 à 5), le nombre d'adultes supplémentaires (de 0 à 2) et le type de ménage parmi :

- Homme seul sans enfant
- Femme seule sans enfant
- Homme seul avec enfant(s)
- Femme seule avec enfant(s)
- Couple sans enfant
- Couple avec enfant(s)

Un individu est décrit par son genre (féminin ou masculin), son âge et la classe d'âge associée, sa catégorie professionnelle (actif, inactif ou étudiant), son niveau d'éducation (aucun diplôme, diplôme primaire, diplôme secondaire ou diplôme supérieur), l'identifiant du ménage dont il fait partie et la place qu'il y occupe (chef de ménage, conjoint, enfant ou adulte supplémentaire) ainsi que son obtention ou non du permis de conduire.

De plus, les individus possèdent également une chaine d'activités, c'est-à-dire une liste d'activités qu'ils doivent effectuer pendant leur journée. Par exemple, l'agenda d'un étudiant peut se résumer à une activité (aller à l'école) nécessitant deux déplacements : le premier étant d'aller à l'école et le deuxième, de retourner au domicile.

Les différentes activités possibles sont les suivantes :

- Déposer, reprendre quelqu'un
- Activités à la maison
- Travail
- École
- Manger à l'extérieur

6. Le code INS est un code à 5 chiffres attribué à chaque commune par l'Institut National de la Statistique. Cette définition provient de la référence [ins]

- Faire des courses
- Activités personnelles (banque, docteur)
- Rendre visite à la famille ou aux amis
- Promenade
- Loisirs
- Autre
- Aucune

Toute activité d'un individu se voit attribuer un lieu et une durée. Une telle modélisation par chaines d'activités permet de simuler les déplacements des individus dans le réseau, en d'autres mots leur évolution spatiale. Les chaines d'activités nous permettent donc d'analyser la mobilité belge en vue de mieux la comprendre.

### 1.3.3 Réseau

Le réseau utilisé dans VirtualBelgium est le réseau routier belge entier constitué de tous les carrefours et routes du pays et extrait du site OpenStreetMap. Le réseau est ensuite modifié par OSMOSIS[7] pour conserver uniquement les données nécessaires au programme VirtualBelgium correspondant au réseau routier belge avec comme conséquence, par exemple, que les noeuds représentant les gares dans OpenStreetMap ne sont pas pris en considération. Ensuite, ce réseau est converti dans un type de fichier XML exploitable par VirtualBelgium et MATSim.

Dans le programme, le réseau est représenté par un *graphe*. Un graphe est un objet mathématique constitué de deux ensembles finis : le premier, noté $V$, contient $n$ points appelés *noeuds* ou *sommets* labellisés de 1 à $n$ ; le deuxième, noté $E$, contient des liens entre certaines paires de noeuds. Un lien entre deux noeuds est appelé *arc* et est noté $(i, j)$ lorsqu'il relie les noeuds $i$ et $j$.

Notre réseau est constitué d'environ 262.000 noeuds et 830.000 arcs. Les noeuds et arcs représentent respectivement les carrefours du réseau et les routes. Les arcs ne sont pas orientés.

## 1.4 Programmation parallèle

Vu la quantité de données traitées dans VirtualBelgium avec ses 10 262 160 individus et ses 4 326 202 ménages, ce programme ne peut s'exécuter sur une unique machine possédant un seul processeur. En effet, si on exécute le programme uniquement en se restreignant à la population de Namur sur un ordinateur personnel avec deux processeurs, il faut déjà attendre deux jours pour obtenir les résultats.

7. OSMOSIS est une application java pour la manipulation des données OSM, extraites d'OpenStreetMap.

Une solution à ce problème est la programmation parallèle, traitable par le logiciel Repast et par la norme MPI. Ce type de programmation consiste à effectuer différentes tâches en même temps grâce à l'utilisation de plusieurs processeurs[8]. L'utilisation de la norme MPI (Message Passing Interface) permet d'exploiter les ordinateurs multiprocesseurs en assignant l'exécution d'une suite séquentielle d'opérations à chaque processeur. Cette assignation est réalisée en utilisant des opérateurs d'envoi et de réception de messages. Chaque processeur possède son propre espace mémoire. Si on veut une interaction entre les processeurs, il existe des messages spécifiques relatifs à la synchronisation.

Dans le programme VirtualBelgium, chaque processeur va posséder son propre ensemble de ménages et donc son propre ensemble d'individus composant ces ménages. Actuellement, aucune synchronisation n'est effectuée entre ces processeurs, il n'y a donc pas de lien entre les ménages de chaque processeur. Le réseau de VirtualBelgium n'est pas commun. En effet, chaque processeur possède son propre réseau, ce qui veut dire que si on modifie un noeud dans un processeur, il n'est pas modifié dans le réseau des autres processeurs.

En résumé, il faut garder en mémoire le fait que le programme VirtualBelgium est adapté à la programmation parallèle et que chaque processeur possède son propre ensemble d'individus et son propre réseau.

## 1.5 Conclusion

En conclusion, le programme VirtualBelgium possède une multitude d'outils, autant pour comprendre l'évolution de la démographie que pour comprendre les déplacements effectués par les habitants de la Belgique.

Cette introduction au projet VirtualBelgium a été écrite en collaboration avec S. Marchal, G. Picard et E. Ramelot.

8. Un processeur est la partie centrale d'un ordinateur qui effectue les opérations arithmétiques et logiques. Cette définition a été inspirée du site de FUTURA- SCIENCES [def].

# Chapitre 2

# Présentation de l'algorithme de Dijkstra

Le but de ce mémoire est de proposer une alternative à la problématique du temps de calcul des plus courts chemins. Après plusieurs recherches dans la littérature, proposées entre autres par Johan Barthelemy et Philippe Toint, nous sommes arrivé à deux possibilités : soit proposer une optimisation de la méthode existante, soit changer complètement de méthode pour en introduire une nouvelle. La première possibilité a été écartée assez tôt car elle semblait difficilement améliorable. Nous avions trouvé quelques solutions mais elles avaient déjà été testées par J.Barthélemy lors de la conception du programme. Nous nous sommes alors penché vers une alternative nouvelle. Celle-ci sera présentée au chapitre 3. La méthode actuelle, déjà améliorée, est quant à elle expliquée dans ce chapitre.

Ce mémoire se base en grande majorité sur des notions de théorie des graphes. Avant de passer à la présentation de l'algorithme actuel et de ses optimisations, nous allons rappeler certains concepts qui seront utiles pour une bonne compréhension de la suite. La majorité des définitions de la section suivante proviennent du cours de théorie des graphes [Lec11].

## 2.1 Graphes

Le premier concept abordé est celui de graphe (ou réseau). Cette notion est la plus importante car elle nous sert de base, de canevas pour nos futurs algorithmes.

**Définition 1.** *Un* ***graphe****, ou réseau, est un objet mathématique constitué de deux ensembles finis. Le premier, noté $V$, contient $n$ points appelés* ***nœuds*** *ou* ***sommets*** *labellisés de 1 à $n$ et l'autre, $E$, contient des liens entre certaines paires de ces nœuds. Ces liens sont appelés* ***arcs*** *et notés $(i, j)$ lorsque $i$ est relié à $j$.*

Un réseau peut être par exemple un groupe de personnes où un nœud est une personne et un lien entre deux individus représente le fait qu'ils se connaissent.

Pour alléger les notations, il est aussi possible d'indicer les liens de 1 à $m$, le nombre de liens contenus dans le réseau, au lieu de les désigner par la paire départ-arrivée. Pour une meilleure compréhension du concept, nous garderons la première notation qui reprend les nœuds de départ et d'arrivée. La Figure 2.1 présente un exemple de graphe. Dans celui-ci, $V = \{1, 2, ..., 7\}$ et $E = \{(1, 3), (3, 1), (3, 2), ..., (7, 6)\}$. Ce petit réseau nous servira d'exemple pour la suite de ce travail et interviendra pour illustrer chaque concept théorique.

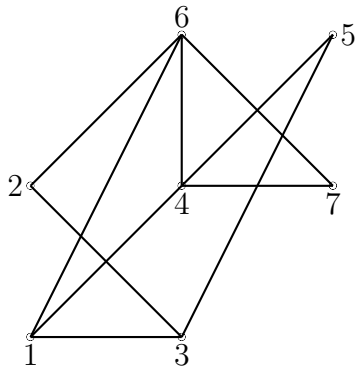



FIGURE 2.1 – Un exemple de graphe

Connaissant maintenant le concept de réseau, nous pouvons introduire des caractéristiques plus précises.

**Définition 2.** *Le **degré** d'un nœud est le nombre de nœuds auquel il est lié par un lien du réseau*

**Propriété 1.** *Un graphe est dit **orienté** si les arcs possèdent un sens.*

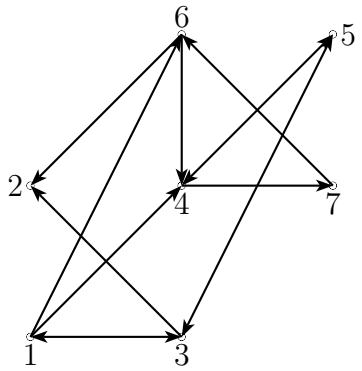



FIGURE 2.2 – Un exemple de graphe orienté

Par exemple, il peut exister un arc reliant le sommet $a$ à $b$ mais pas l'opposé de $b$ à $a$. Dans ce cas-ci, le degré d'un nœud est séparé en deux valeurs : le degré entrant et le degré sortant valant respectivement le nombre de nœuds liés par un lien entrant et le nombre de nœuds liés par un lien sortant. L'exemple à la figure 2.2 est un graphe orienté.

**Propriété 2.** *Un graphe est* ***pondéré*** *si l'on adjoint un* ***poids*** *à chaque arc. Il s'agit d'une valeur propre à un arc, pouvant faire office de coût, de temps de trajet ou encore de pénalité. Ce nombre est souvent positif. Par hypothèse, le poids d'un nœud vers lui-même vaut 0 et celui entre deux sommets qui ne sont pas reliés par un arc du graphe est posé à* $\infty$.

Dans notre exemple, à la figure 2.3, nous pouvons imaginer que le réseau est un réseau routier où les nœuds sont des carrefours et les liens des routes. Dans ce cas, la valeur du poids peut correspondre au temps de trajet, en minutes, pour parcourir le lien.

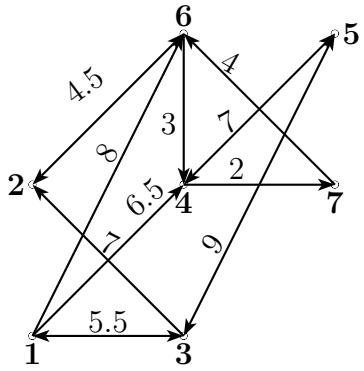



FIGURE 2.3 – Un exemple de graphe orienté et pondéré

L'utilisation de graphes pondérés permet des applications plus intéressantes. Nous pouvons, par exemple, calculer des temps de déplacements sur le réseau ou des plus courts chemins. Pour pouvoir résoudre ces types de problèmes via un ordinateur, nous devons mettre en place une méthode de représentation et de stockage numérique du graphe. Introduisons pour cela la **matrice d'adjacence**, notée $A$, définie comme :

$$A(i,j) = \begin{cases} 1 & \text{si } (i,j) \in E \\ 0 & \text{sinon} \end{cases}$$

Intuitivement, chaque ligne ou colonne de la matrice correspond à un nœud et l'élément $A_{i,j}$ vaut 1 lorsque $i$ est relié à $j$. Pour les graphes pondérés, la définition est légèrement modifiée afin de prendre en compte les poids de chaque lien. Elle devient :

$$A'(i,j) = \begin{cases} w(i,j) & \text{si } (i,j) \in E \\ 0 & \text{si } i = j \\ \infty & \text{sinon} \end{cases}$$

Nous utiliserons cette définition dans la suite du travail. Notre exemple de graphe donne, pour les deux définitions, les matrices d'adjacences suivantes :

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

et en ajoutant les poids :

$$\begin{pmatrix} 0 & \infty & 5.5 & 6.5 & \infty & 8 & \infty \\ \infty & 0 & \infty & \infty & \infty & 4.5 & \infty \\ 5.5 & 7 & 0 & \infty & 9 & \infty & \infty \\ \infty & \infty & \infty & 0 & 7 & \infty & 2 \\ \infty & \infty & 9 & 7 & 0 & \infty & \infty \\ \infty & 4.5 & \infty & 3 & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & 4 & 0 \end{pmatrix}$$

Le concept de graphe est maintenant bien défini et représentable. Il reste à définir le concept reliant deux nœuds qui ne sont pas directement accessibles par un lien du réseau. Un **chemin** entre les nœuds $a$ et $b$ est un ensemble de $m$ arcs appartenant à $E$ de la forme

$$\{(n_0, n_1)(n_1, n_2), ..., (n_m - 1, n_m)\}$$

où $n_0 = a$ et $n_m = b$.

**Définition 3.** *Un graphe est dit* ***connexe*** *s'il est possible de trouver un chemin entre toutes les paires de nœuds du graphe.*

Dans un graphe pondéré, le poids d'un chemin est simplement la somme des poids des arcs qui le composent. Dans un graphe non-connexe, il est parfois impossible de relier deux nœuds. Dans ce cas, le poids du chemin est fixé à l'infini.

**Définition 4.** *Un* ***plus court chemin*** *entre a et b est le chemin de poids minimal parmi tous les chemins possibles entre a et b. Rappelons que le poids d'un chemin partant d'un nœud vers lui-même est fixé à* 0*.*

Grâce à cette notion, nous pouvons définir la matrice des plus courts chemins, notée $D$ telle que $D_{i,j}$ est le poids du plus court chemin allant de $i$ à $j$. Partant de notre exemple, le plus court chemin entre les sommets 7 et 3 de notre graphe est l'ensemble ordonné d'arcs

$$\{(7, 6), (6, 4), (4, 5), (5, 3)\}$$

et est de poids 23. La matrice des plus courts chemins complète vaut quant à elle :

$$\begin{pmatrix} 0 & 12.5 & 5.5 & 6.5 & 13.5 & 8 & 8.5 \\ \infty & 0 & \infty & \infty & \infty & \infty & \infty \\ 5,5 & 7 & 0 & 16 & 9 & \infty & 23 \\ \infty & 10,5 & \infty & 0 & \infty & 6 & 2 \\ 14,5 & 16 & 9 & 7 & 0 & 13 & 8 \\ \infty & 4,5 & \infty & 3 & \infty & 0 & 5 \\ \infty & 8,5 & \infty & 7 & \infty & 4 & 0 \end{pmatrix}$$

On remarque bien que la diagonale est constituée de valeurs nulles et que certaines paires ne sont pas joignables via notre réseau, elles sont donc associées à un chemin de poids infini.

## 2.2 L'algorithme de Dijkstra

Soit un graphe pondéré et $u_0$ un sommet de ce graphe. L'algorithme de Dijkstra permet de trouver les chemins les plus courts entre $u_0$ et tous les autres nœuds du graphe ainsi que leur poids. Ce problème est connu en sciences informatiques comme *Single-Source Shortest Path*. Edsger Wybe Dijkstra proposa une solution efficace dans son article de 1959 ([E.W59]). Comme cela a été spécifié précédemment, cet algorithme est utilisé actuellement dans VirtualBelgium pour calculer tous les parcours et temps de trajets des agents accomplissant leur chaine d'activités.

### 2.2.1 Fonctionnement

Nous allons d'abord expliquer la méthode, puis nous la résumerons de manière plus technique via un pseudo-code. Pour commencer, nous partons d'un nœud source à partir duquel tous les plus courts chemins seront calculés. Les autres nœuds du réseau sont rassemblés dans un ensemble nommé *réservoir*. Chacun sera, à la fin du calcul, caractérisé par une distance à la source (la valeur du poids du plus court chemin entre les deux) et un prédécesseur (un autre nœud se situant juste avant le nœud cible dans le plus court chemin servant à reconstruire le trajet). À l'initialisation, les distances sont fixées à $\infty$ et les prédécesseurs à un élément vide. L'algorithme se déroule comme suit :

1. Au départ, tous les nœuds sont placés dans le réservoir. Parmi eux, seul le nœud source a une valeur nulle car sa distance à lui-même est nulle.
2. On prend dans le réservoir le nœud de valeur minimale. Au premier passage, il s'agit nécessairement du nœud source car il est le seul à posséder une valeur différente de l'infini. Ce nœud est retiré du réservoir. Les nœuds restant sont mis à jour. Pour cela, on vérifie si leur distance diminue en passant par le nœud retiré. Ainsi, soit $v$ le nœud à mettre à jour, $u$ celui retiré du réservoir, $l(v)$ la distance du nœud $v$ et $w(i,j)$ le poids de l'arc $(i,j)$, alors :

$$l(v) = \min(l(v), l(u) + w(u,v))$$

3. Si des nœuds voient leur valeur être diminuée, nous changeons aussi leur prédécesseur pour $u$.
4. On réitère les opérations 2 et 3 tant que le réservoir n'est pas vide.

Le code 2.1 provenant du cours de théorie des graphes [Del09] exprime cet algorithme de manière plus rigoureuse. La procédure insert(Q node i j) permet d'insérer le nœud $i$ de valeur $j$ dans le réservoir $Q$.

```
\\ initialisation du reservoir vide
Q=EmptyQ

\\ insertion des noeuds dans le reservoir
insert(Q, node 0,0) \\la source est a distance 0 d'elle meme
for i from 1 to n
   insert(Q, node i, infinity)
end
while Q non vide
   u:=noeud de Q de distance depuis la source minimale
   Q:=Q\{u}
   for tout v dans Q
      distance de la source a v:=min(distance de la source a
      v,distance de la source a u+poids(u,v))
      mise a jour du predecesseur si changement de distance
   end
end
```

Code 2.1 – Pseudo-code de Dijkstra

**Remarque :** la version de l'algorithme de Dijkstra présentée ici est celle que l'on trouve originellement dans la littérature. Celle implémentée dans VirtualBelgium lui est différente en un point. Comme les nœuds sont retirés du réservoir par ordre croissant de leur valeur, il est garanti que plus nous retirons de nœuds, plus ils seront éloignés de la source. Or, dans notre recherche d'un nœud plausible pour accueillir l'activité de l'individu, nous devons en chercher un parmi ceux se trouvant à une certaine distance tirée aléatoirement. Cela implique que nous pouvons stopper la recherche de nœuds dans l'algorithme de Dijkstra avant que le réservoir ne soit vide si la distance du dernier nœud a y être retiré est supérieure à celle à parcourir pour l'activité.

Pour retrouver le plus court chemin de la source à un certain sommet, il suffit de le reconstruire grâce aux prédécesseurs successifs. Le résultat de l'algorithme de Dijkstra sur notre exemple donne, en prenant le nœud 1 comme source :

| Nœuds | Distance à la source | Prédécesseur |
|---|---|---|
| **1** | 0 | Aucun |
| 2 | 12.5 | 6 |
| 3 | 5.5 | 1 |
| 4 | 6.5 | 1 |
| 5 | 13.5 | 4 |
| 6 | 8 | 1 |
| 7 | 8.8 | 4 |

Une représentation sous forme d'arbre permet de mieux rendre compte du résultat. Elle comprend les plus courts chemins de la source vers tous les autres nœuds ainsi que leur poids. Le diagramme 2.4 montre cela.

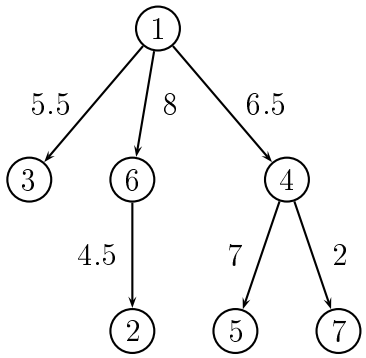



FIGURE 2.4 – Résultat de l'algorithme de Dijkstra sur un exemple

### 2.2.2 Complexité

Un premier élément de comparaison entre deux algorithmes est leur temps de calcul. Un premier indicateur de cette durée peut être donné par leur complexité. L'exécution d'un algorithme demande un certain nombre d'opérations pour arriver à son terme. La notion de complexité est définie par l'ordre de grandeur auquel appartient ce nombre d'opérations dans le pire des cas.

Nous savons que l'algorithme de Dijkstra s'effectue en complexité d'ordre quadratique $\mathcal{O}(n^2)$ où $n$ est la taille du réseau. En effet, le *réservoir*, dans le code 2.2, est classiquement implémenté sous la forme d'un simple tableau. L'algorithme se termine lorsque le réservoir est vide. Or, celui-ci est décrémenté d'un élément à chaque étape, ce qui induit une complexité $\mathcal{O}(n)$. Pour chaque étape, nous cherchons le nœud de distance minimum à la source et modifions la valeur de certains nœuds. Une recherche dans un tableau demande $\mathcal{O}(n)$ opérations. Au final, nous sommes bien en présence d'un problème en temps quadratique.

Si enfin nous désirons construire la matrice des plus courts chemins complète, nous devons appliquer Dijkstra en prenant chaque sommet comme source. La complexité est multipliée par $n$, ce qui donne $\mathcal{O}(n^3)$.

## 2.3 Arbres binaires

L'algorithme actuel de calcul des plus courts chemins se base sur celui de Dijkstra présenté à la section précédente. La partie de l'algorithme qui peut être améliorée en terme de temps de calcul concerne le réservoir. En effet, il existe d'autres structures de données que les tableaux qui permettent la recherche du minimum et la modification de valeur plus efficacement. Il s'agit des arbres binaires et des arbres de Fibonacci. Cette section parle des arbres binaires et la suivante des arbres de Fibonacci qui sont des structures encore plus efficaces dans le cas de l'algorithme de Dijkstra. Ces deux améliorations ont été implémenté successivement dans VirtualBelgium par l'auteur de la thèse [BJ14]. À l'heure où ce mémoire est écrit, la méthode employée est Dijkstra utilisant des arbres de Fibonacci. Nous introduirons quand même la notion d'arbre binaire car celle-ci sert de base à celle des arbres de Fibonacci.

Les **arbres binaires** sont des structures de files prioritaires, c'est-à-dire des structures permettant de stocker une liste de valeurs de façon ordonnée (souvent dans l'ordre croissant) tout en optimisant la complexité de l'insertion de nouvelles données, la modification de valeur, la recherche et la suppression du minimum. Une définition littéraire d'arbre binaire prise du livre d'Herbert Edelsbrunner ([Ede05]) est : « Il s'agit de soit un élément(appelé nœud) vide soit un élément ayant deux arbres binaires comme sous-arbres gauche et droit (appelés enfants) ».

Concrètement, dans la mémoire d'un ordinateur, chaque valeur de l'ensemble de données considéré est stockée dans une variable appelée nœud. Chacun de ces nœuds possède un pointeur vers, au plus, **deux** autres nœuds. Nous appellerons ces derniers *enfants*. Inversement, mis à part le nœud "source", chacun se voit lié par un pointeur depuis un nœud *parent*. C'est cet agencement de nœuds et pointeurs qui définit un arbre binaire. On représentera toujours un triplet parent-enfant-enfant avec le parent au dessus et les enfants sur le même niveau en dessous comme à la figure 2.5.

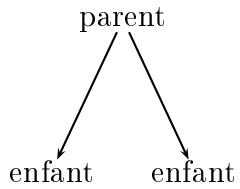



FIGURE 2.5 – Disposition d'un triplet parent-enfant-enfant

Pour pouvoir différencier les valeurs à l'intérieur de l'arbre, chaque nœud est indicé par un nombre compris entre 1 et $n$ où $n$ correspond au nombre d'éléments de l'arbre. La formule utilisée pour numéroter les valeurs est celle du sens de lecture. Soit un arbre $A$, alors un nœud parent stocké en position $i$ de valeur $A(i)$ a ses deux enfants aux positions $2i$ et $2i + 1$. Réciproquement, un nœud stocké à la $j$-

ème position provient d'un parent situé en $\lfloor j/2 \rfloor$. Ceci sera représenté à la figure 2.6.

L'intérêt d'utiliser un arbre binaire est de pouvoir trouver facilement la valeur minimum. Le placement des valeurs n'est donc pas fait de manière arbitraire, celles-ci doivent respecter une propriété de disposition. Il n'existe qu'une seule règle de disposition. Notons $A(i)$ la valeur du nœud stocké en position $i$ dans l'arbre. Alors les valeurs sont agencées de façon suivante :

$$A(i) \leq A(2i) \text{ et } A(i) \leq A(2i+1) \tag{2.1}$$

Autrement dit, la valeur d'un nœud doit toujours être inférieure à celles de ses deux *enfants*. La différence de valeur entre deux enfants liés au même nœud parent n'a pas d'importance.

Le schéma 2.6 représente l'agencement théorique d'un arbre binaire. Les cercles représentent les valeurs stockées, les nombres en indice les places dans la structure et les flèches les pointeurs vers les nœuds enfants.

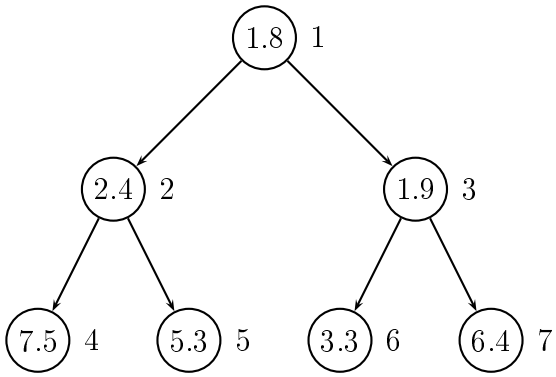



FIGURE 2.6 – Exemple d'arbre binaire.

Enfin, on définit la **hauteur** d'un arbre par le nombre de générations présentes dans celui-ci. Elle vaut $\log_2 n + 1$ lorsque $n$ est la taille de l'arbre.

Voyons maintenant comment il est possible d'accélérer les procédures agissant sur le réservoir de l'algorithme de Dijkstra en le structurant sous la forme d'un arbre binaire. Rappelons le rôle du réservoir dans l'algorithme. Tous les nœuds du réseau s'y trouvent au lancement. Ils vont y être retirés un à un suivant leur éloignement par rapport à la source (le plus proche en premier). Sachant cela, il nous faut implémenter des méthodes, propres aux arbres binaires, permettant de

- Trouver le minimum de l'arbre ;
- Supprimer ce minimum ;
- Ajouter une valeur à l'arbre lors de sa construction ;
- Modifier une valeur dans l'arbre.

La valeur d'un nœud de l'arbre sera la distance de ce nœud à la source au cours de l'algorithme.

### 2.3.1 Retirer le minimum

Les deux premières méthodes de cette liste peuvent être rassemblées en une fonction *Extraire le minimum*. Rappelons que les arbres binaires sont définis par une propriété de disposition, celle énoncée par l'équation 2.1, imposant que la valeur d'un nœud parent doit toujours être inférieure ou égale à celle de ses nœuds "enfants". Cet aspect permet de mettre en place très facilement la fonction *Extraire le minimum*. Il suffit de désolidariser le nœud racine de l'arbre comme repris à la figure 2.7.

Cependant, le fait d'avoir supprimé le nœud source a changé l'aspect général de l'arbre. Il ne commence plus par une unique valeur source. Il faut réarranger les nœuds restants pour revenir à une architecture d'arbre binaire bien définie, c'est-à-dire comprenant un et un seul nœud source et respectant la propriété 2.1.

Pour commencer, nous remplaçons la source supprimée par le dernier nœud de l'arbre, $A(n)$, se situant à l'extrême droite du dernier étage. Schématiquement, cela donne la figure 2.8.

Ensuite, la deuxième étape consiste à permuter certains nœuds afin de retrouver la propriété (2.1). La méthode à suivre est la suivante : Partant du nœud source, nous le changeons de place avec son enfant de plus petite valeur si sa valeur lui est supérieure. On réitère cette opération jusqu'à arriver au dernier étage de l'arbre, ou avant si ce n'est plus nécessaire. Illustrons cela par la figure 2.9.

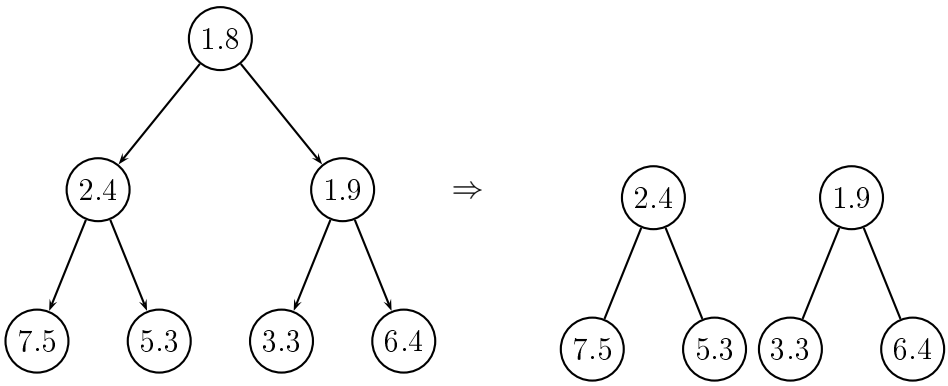



FIGURE 2.7 – Extraction du minimum

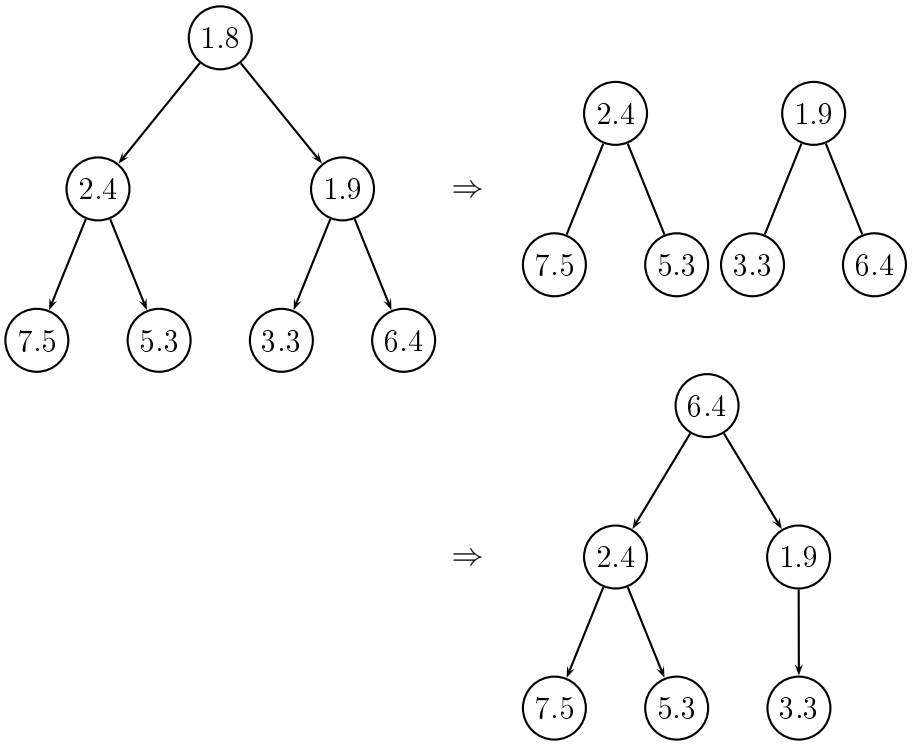



FIGURE 2.8 – Remplacement du nœud source

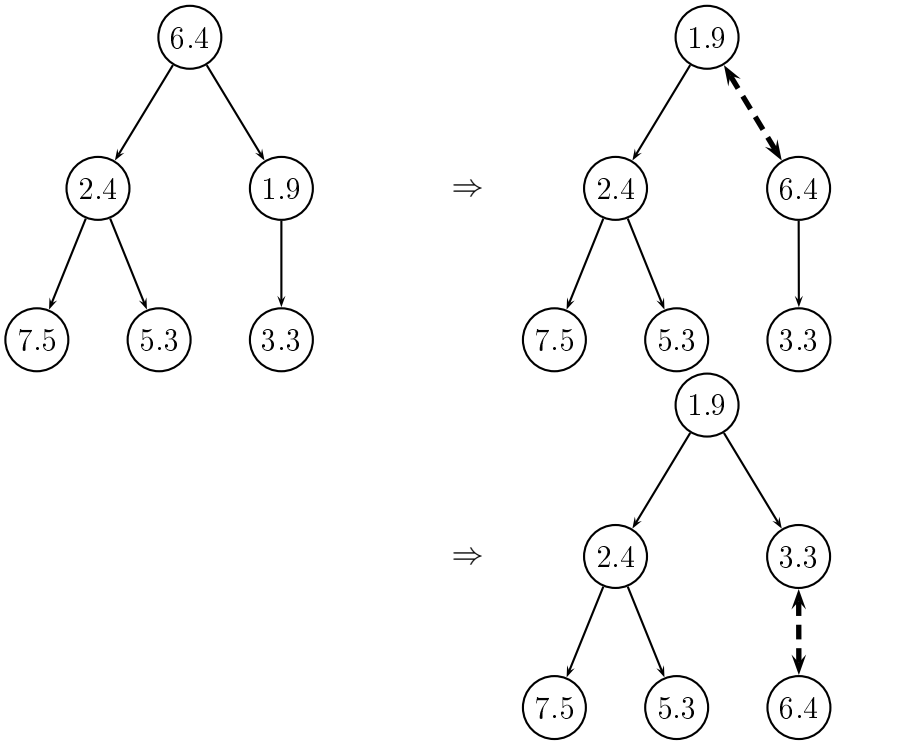



FIGURE 2.9 – Réparation de l'arbre en intervertissant les nœuds

En résumé, la fonction *extraire le minimum* est exprimée par le code 2.2 où

REPAIR() est donné au code 2.3 et la fonction CHANGE(I,K) change de position les nœud $i$ et $k$.

```
**In : aucun
**Out : valeur minimale
ExtractMinimal()
    min= A[1]
    A[1] = A[n]
    delete A[n]
    repair(1)
```

Code 2.2 – Extraction de l'élément minimal

```
**In : indice du premier noeud
**Out : Aucun
repair(int i)
    if 2i ≤ n then
        k = arg min{A(2i), A(2i + 1)}
        if A(k) < A(i) then
            change(i,k)
            repair(k)
        end
    end
```

Code 2.3 – Réarrangement d'un arbre binaire

Ces deux codes proviennent du livre [Ede05].

Au niveau de la complexité, nous avons vu à la section 2.3 que la hauteur d'un arbre de taille $n$ vaut $\log_2 n + 1$. Un nœud à déplacer doit donc être au maximum interverti avec un autre $\log_2 n + 1$ fois. Il y a donc au plus $\log_2 n$ changements de nœuds. La complexité de cet algorithme est $\mathcal{O}(\log_2 n)$.

### 2.3.2 Ajouter une valeur

La deuxième procédure dont nous avons besoin est celle permettant d'ajouter un nœud dans notre arbre. Elle nous est utile au début de l'algorithme de Dijkstra puisque nous devons insérer tous les nœuds dans le réservoir. Nous les ajoutons un à un à la fin de l'arbre. Le premier à entrer est le nœud source. Ensuite, pour chaque ajout, nous réarrangeons l'arbre mais cette fois-ci à l'envers. Dans la fonction de réparation précédente, nous comparions un nœud avec son "enfant" de valeur minimale et nous les permutions de place au besoin. Dans ce cas-ci, c'est le contraire, nous comparons les valeurs du nœud considéré et de son parent. Imaginons que nous insérons le nœud de valeur 2.2 dans notre arbre comme à la figure 2.10. Cette valeur est, pour commencer, stockée à l'indice $n+1$, dans notre cas $n+1 = 8$. Le code 2.4 résume cela où REPAIRINV() est donné par le code 2.5.

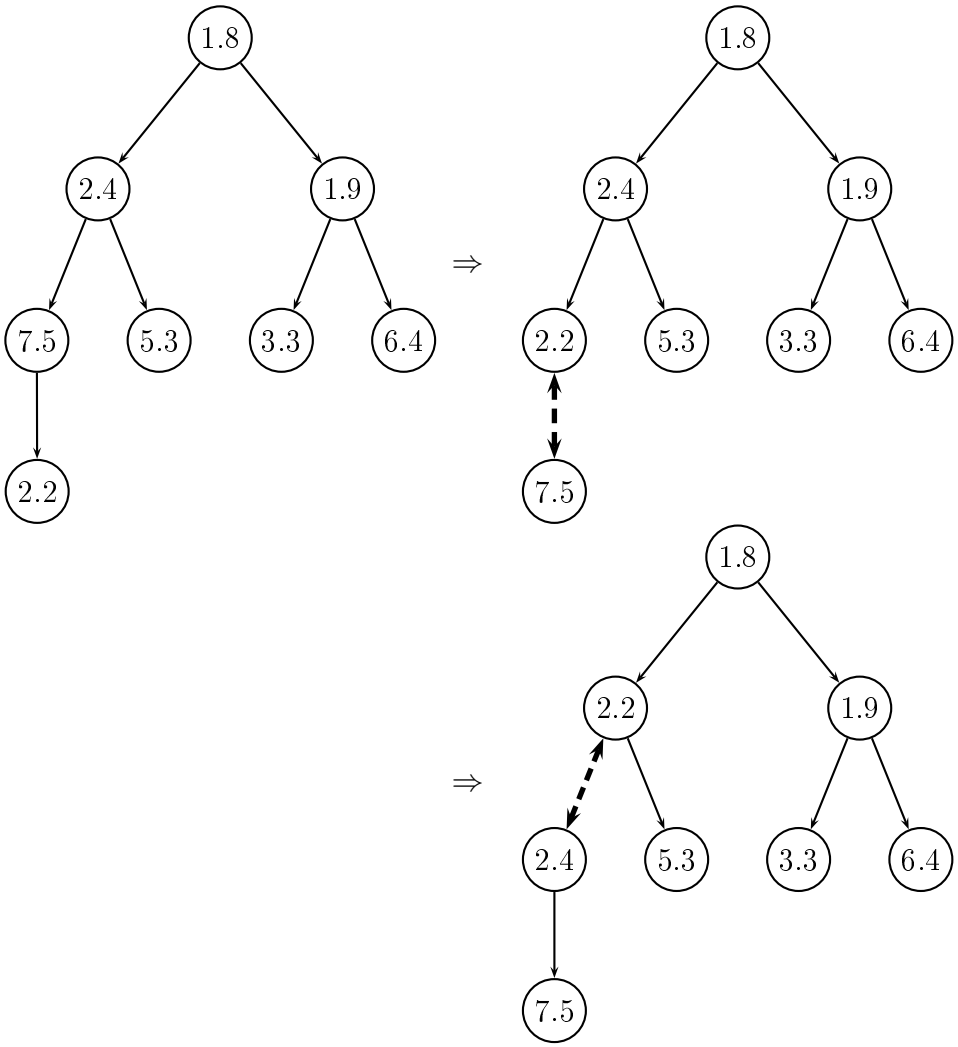



FIGURE 2.10 – Remplacement du nœud source

```
**In : valeur du noeud ajoute
**Out : Aucun
repairInv(double x)
    A[n+1]=x
    repairInv(n+1)
```

Code 2.4 – Construction/insertion

```
**In : indice du premier noeud
**Out : Aucun
repairInv(int i)
    if i ≥ n then k = ⌊i/2⌋
        if A(i) < A(k) then
            change(i,k)
            repairInv(k)
        end
```

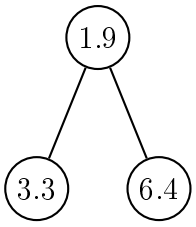
**end**

Code 2.5 – Réparation inverse

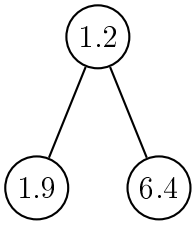
La complexité d'un réarrangement inverse est la même que celle d'un réarrangement. L'insertion d'une valeur a donc une complexité d'ordre $\mathcal{O}(\log_2 n)$. L'ajout de tous les nœuds du graphe dans le réservoir est de complexité $\mathcal{O}(n.\log_2 n)$

### 2.3.3 Modifier une valeur

L'utilité du réservoir dans l'algorithme de Dijkstra est de garder en mémoire les nœuds du réseau pour lesquels on ne connait pas encore le plus court chemin depuis la source. Tous nœuds du graphe n'étant pas joignables depuis la source par un lien commencent l'algorithme de Dijkstra avec une distance fixée à l'infini. Petit à petit, à force de retirer des nœuds du réservoir et donc de mettre à jour les plus courtes distances, ces valeurs décroissent pour arriver à leur valeur minimale. Cette observation implique que les modifications de valeurs dans l'arbre binaire se feront toujours dans le même sens : en décroissant. Ces changements de valeurs impliquent potentiellement une mauvaise structure de l'arbre. Imaginons le simple arbre suivant :




Si le nœud de gauche passe de 3.3 à 1.2, il est clair que l'arbre ne respecte plus la propriété 2.1. Il faut le réparer. Une inversion de ce nœud avec son parent permet de satisfaire cette propriété.




Nous avons simplement réparé l'arbre de la même manière que celle vue précédemment au code 2.3.

A nouveau, la complexité du changement d'une valeur est d'ordre $\mathcal{O}(\log_2 n)$. En effet, supposons, dans le pire des cas, que le nœud modifié soit dans le bas de l'arbre et prenne comme nouvelle valeur une valeur inférieure à celle du minimum. Il faut alors réarranger l'arbre de bas en haut. Ceci justifie la valeur de la complexité.

### 2.3.4 Arbres $m$-aires

Remarquons que, dans la définition d'arbre binaire, chaque nœud avait au plus 2 enfants. Nous pouvons utiliser aussi des arbres dont les nœuds peuvent être liés à $m > 2$ éléments. Dans son livre, Robert Endre Tarjan ([Tar83]) précise que suivant le nombre d'opérations d'insertions et d'extractions dont nous aurons besoin, le choix de $m$ peut avoir une influence sur la rapidité de celles-ci.

Cependant, cette légère modification a déjà été testée par Johan Barthélemy lors de plusieurs lancements de simulation de VirtualBelgium. Les temps de calculs sont restés d'ordre similaire.

## 2.4 Arbres de Fibonacci

Les arbres binaires permettent une représentation sous forme de liste prioritaire en informatique. Ils peuvent cependant être remplacés par une autre structure encore plus efficace partant de la même idée de disposition sous forme d'arbre : les arbres de Fibonacci. Ce nom ne vient pas de leur structure mais bien du calcul de la complexité des opérations qui s'y rapportent, et où les nombres de Fibonacci interviennent. Ces arbres utilisent l'évaluation paresseuse (ou moins péjorativement évaluation par nécessité) comme nous le verrons dans cette section. Cette façon d'implémenter les programmes signifie n'exécuter un travail que si cela est nécessaire au résultat final.

Nous présentons ces deux structures car, d'une part, elles ont été utilisées successivement toutes deux dans l'algorithme de Dijkstra de VirtualBelgium et d'autre part parce que la première (arbre binaire) est moins complexe mais inspire fortement la seconde (arbre de Fibonacci).

**Définition 5.** *Un arbre de Fibonacci est une structure constituée de plusieurs arbres $m$-aire ($m \geq 1$).*

La figure 2.11 montre un exemple d'arbre de Fibonacci tiré du cours [Way07] au chapitre intitulé *Fibonacci Heaps*.

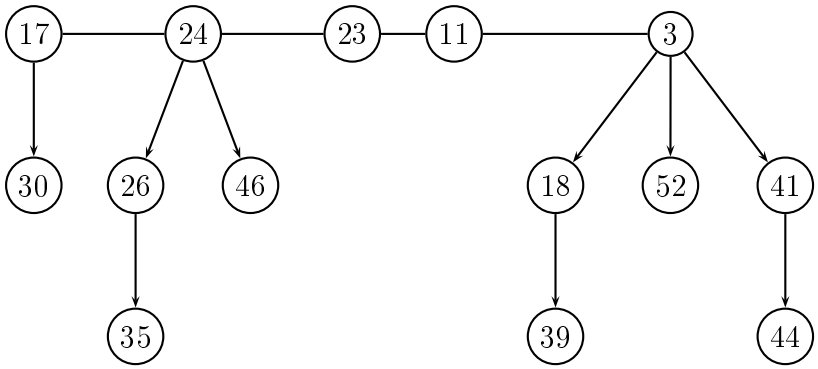



FIGURE 2.11 – Exemple d'arbre de Fibonacci.

Avant d'analyser plus en détail les particularités de cette structure, introduisons quelques termes :

**Définition 6.** *Le **rang d'un nœud** est le nombre de ses nœuds enfants*

**Définition 7.** *Le **rang de l'arbre** est le rang maximum des nœuds de l'arbre*

**Définition 8.** *Un **nœud marqué** est un nœud dont un des enfants a été détaché. Cette notion intervient lors d'un réarrangement de l'arbre à la section 2.4.3.*

Comme montré par l'exemple de la figure 2.11, un arbre de Fibonacci peut être constitué de plusieurs arbres $m$-aires. Chacun d'eux respecte la propriété (2.1), présentée précédemment, qui dit que tout nœud parent doit posséder une valeur inférieure ou égale à celles de ses enfants. En revanche, concernant la disposition des arbres, il n'y a pas de règle spécifique. Les deux arbres présentés à la Figure 2.12 sont similaires et ont une structure correcte.

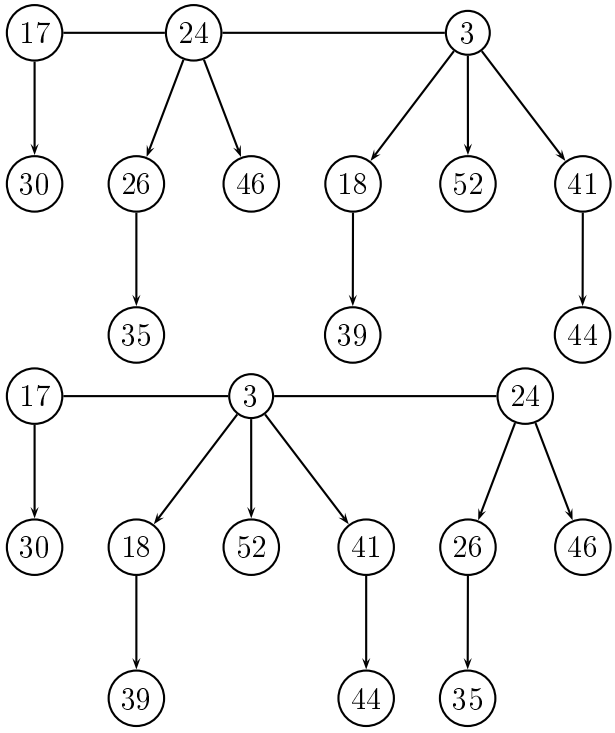



FIGURE 2.12 – Deux arbres de Fibonacci similaires.

### 2.4.1 Retirer le minimum

Maintenant que la structure est détaillée, nous pouvons présenter les procédures de modification de ces arbres. Celles-ci nous seront nécessaires lors de l'utilisation de ces structures dans l'algorithme de Dijkstra. Rappelons que la fonction principale est de trouver le minimum des valeurs dans la structure. Dans ce cas-ci, comme dans celui des arbres binaires, ce travail est très simple. En effet, un pointeur garde

toujours en mémoire la position de ce minimum. Il s'agit d'une des racines des sous-arbres.

L'enlèvement de ce minimum est plus complexe. Dans l'exemple, à la figure 2.11, le minimum est le nœud racine de l'arbre à l'extrême droite de valeur 3. Sa suppression transforme le sous-arbre dont il faisait partie en trois arbres distincts, illustrés à la figure 2.13.

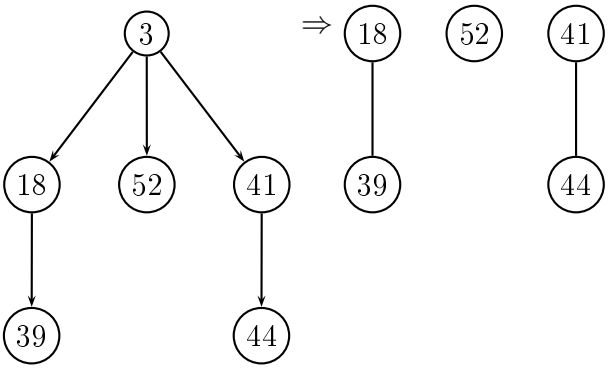



FIGURE 2.13 – Suppression de la racine *minimum*

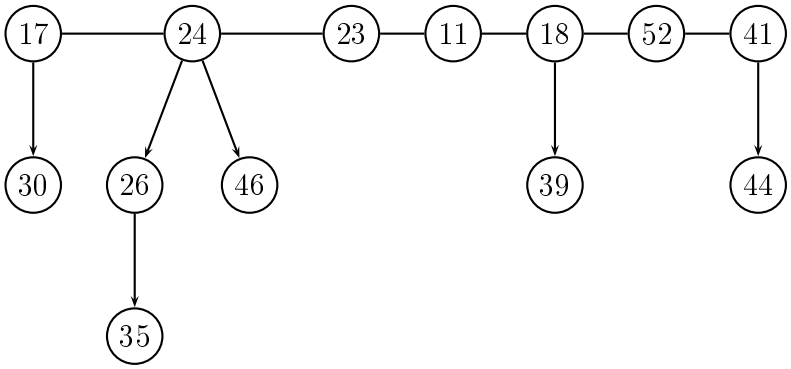
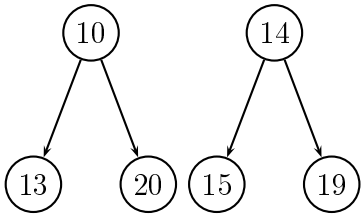
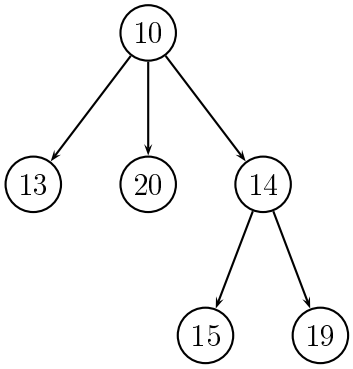



FIGURE 2.14 – Arbre de Fibonacci après suppression du minimum

L'arbre général devient alors simplement l'union de tous les sous-arbres comme nous pouvons le voir à la figure 2.14. Cependant, on peut rapidement constater qu'après quelques suppressions de minima, l'arbre ressemblera vite à une grande union de nœuds singletons. Cette structure ne se prête pas bien à la recherche d'un minimum car elle se rapproche plus de celle d'un simple tableau. Afin d'éviter cette dispersion des nœuds, nous allons grouper certains arbres de telle sorte qu'aucun nœud racine n'ait le même rang qu'un autre. On appelle cette action *consolider* l'arbre. Lorsque deux racines ont un rang égal, l'arbre de celle de plus grande valeur devient un enfant de celui de la racine de plus petite valeur. Par exemple, pour consolider l'arbre de Fibonacci suivant :




L'arbre de droite devient un enfant de celui de gauche (car $14 > 10$) pour former la structure suivante :




Voici les étapes successives de consolidation de l'arbre illustré à la figure 2.14. Sa forme finale est quant à elle montrée à la figure 2.15.

1. La racine 17 a un rang de 1, la 24 un rang de 2 et la 23 un rang de 0. Il ne faut en grouper aucune.
2. La racine 11 a un rang de 0, comme la 23 qui devient son nœud enfant.
3. L'arbre ainsi créé a une racine de rang 1 (comme le premier qui devient son enfant).
4. À nouveau, le groupement provoque la similitude de rang, avec l'arbre de racine 24. La valeur 24 étant supérieure à 11, l'arbre de cette racine devient le troisième enfant de celui de la racine 11.
5. La racine 18 est maintenant la seule de rang 1.
6. Même constat pour la racine 52 qui est la seule de rang 0.
7. Enfin, la racine 41 est de rang 1 comme la racine 18. Elle devient donc son enfant.
8. Plus aucune racine ne possède un rang identique à celui d'une autre.

L'arbre consolidé est formé à présent de trois arbres, respectivement de rang 3,0 et 2.

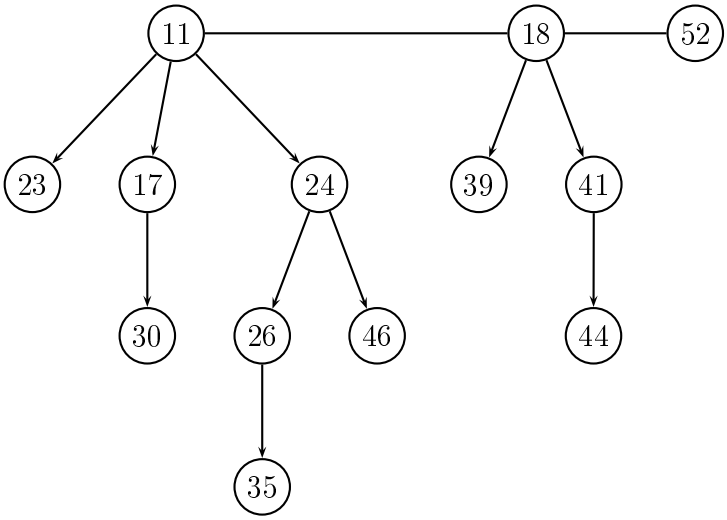



FIGURE 2.15 – Arbre de Fibonacci après suppression du minimum et consolidation

La suppression du minimum est presque terminée. Il ne reste plus qu'à repositionner le pointeur vers le nouveau minimum. Une simple recherche parmi les trois nouvelles racines suffit pour retrouver le minimum : 11.

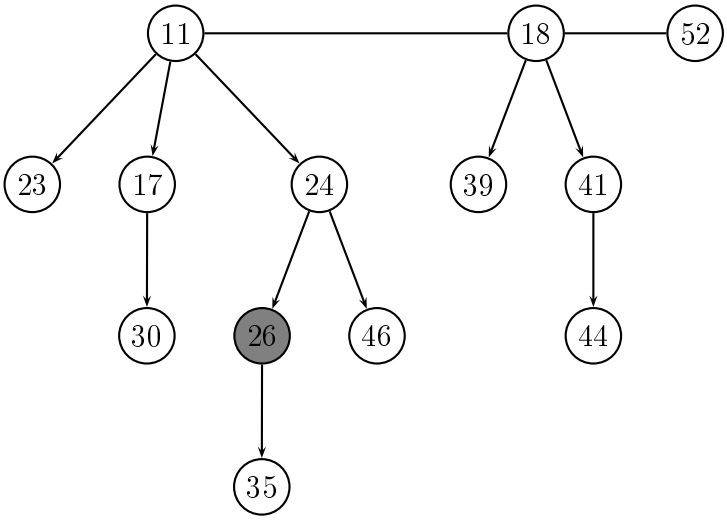
### 2.4.2 Ajouter une valeur

Il reste deux opérations à implémenter pour rendre notre structure d'arbre de Fibonacci capable de servir de liste prioritaire : l'insertion et la modification de valeur. L'ajout de nouvelle valeur est très simple ; il suffit de créer avec elle un arbre singleton et de l'ajouter à la suite des sous-arbres, puis de mettre à jour le minimum si nécessaire.

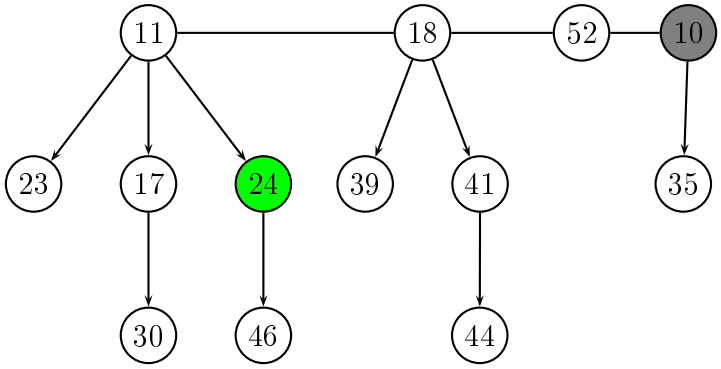
Rappelons cependant que l'algorithme de Dijkstra ne remplit son réservoir qu'à un seul moment et que toutes les valeurs y entrent en même temps. L'arbre de Fibonacci ressemble à une simple liste d'arbres singletons. Il n'est pourtant pas nécessaire de consolider l'arbre à ce moment car le remplissage du réservoir est directement suivi du retrait du premier minimum. Ce retrait implique nécessairement une consolidation de l'arbre.

### 2.4.3 Modifier une valeur

La modification de valeur est plus complexe. C'est ici qu'entre en compte la notion de *nœud marqué*. Partons de l'exemple précédent où nous avions supprimé le minimum et supposons que le nœud 26 change de valeur pour 10.




(a) Modification d'une valeur




(b) Désolidarisation du nœud enfant et marquage du nœud 24

FIGURE 2.16 – Modification d'une valeur dans un arbre de Fibonacci

Il est évident que si, après la modification de la valeur, l'arbre respecte toujours la structure, il peut rester tel quel. Seul le pointeur vers le minimum peut éventuellement changer. Mais si la modification a entrainé le non-respect de la structure, il faut la réparer. C'est le cas dans notre exemple : la nouvelle valeur, 10, est inférieure à 24, celle de son nœud parent. Il faut restructurer l'arbre. Une manière évidente de rendre l'arbre correct est de désolidariser le nœud posant problème et de le transformer en sous-arbre $m$-aire à la suite des autres. On voit directement que ce type de modification peut, comme dans le cas de la suppression du minimum, rendre la structure de Fibonacci moins efficace car la structure se rapproche de celle d'un tableau.

Dans ce cas de figure, le principe de base à respecter lors de la restructuration est qu'un nœud doit être désolidarisé de son arbre dès qu'il perd deux de ses enfants. Le concept de nœud marqué entre alors en jeu. Dès qu'un enfant est coupé de son nœud parent, ce dernier garde en mémoire cette séparation en étant *marqué*. Si un autre de ses enfants change de valeur et doit être détaché, on isole aussi le parent avec les potentiels nœuds restants ainsi que tous les ancêtres marqués. L'exemple de la figure 2.17 illustre cela. On suppose que le nœud ayant la valeur 24 est déjà

marqué (nœud de couleur verte) et que son enfant de gauche prend la valeur 10 comme précédemment (2.17a). L'arbre formé par les nœuds de valeur 10 et 46 est désolidarisé ainsi que le singleton parent de valeur 24. Ce dernier perd son marquage après l'opération.

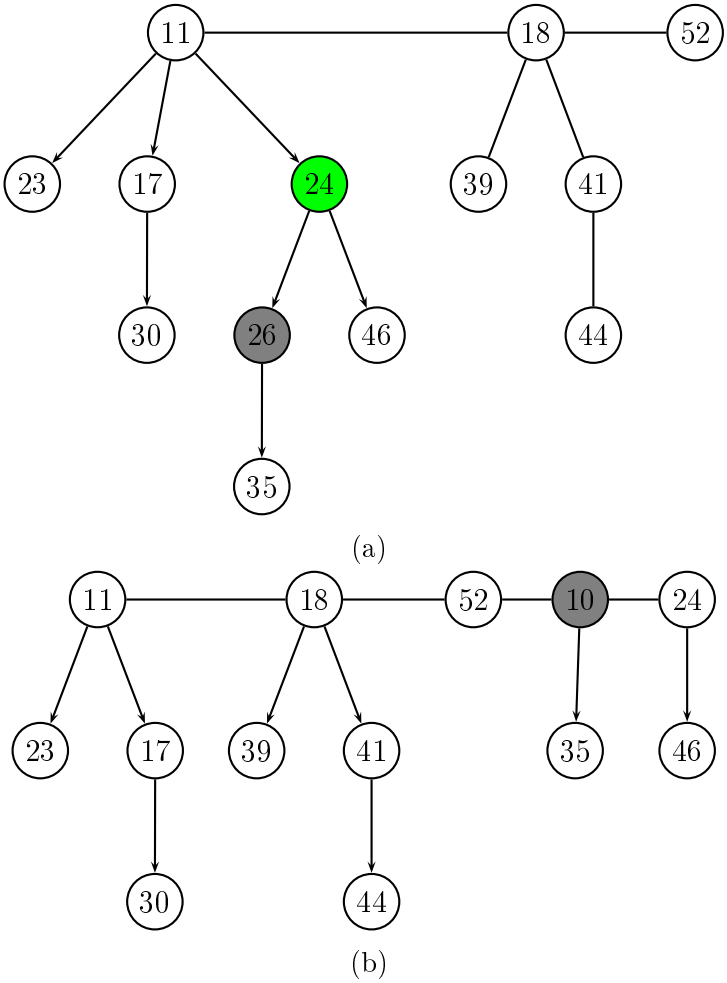



FIGURE 2.17 – Illustration des nœuds marqués

## 2.5 Association des arbres de Fibonacci et de l'algorithme de Dijkstra

Pour terminer ce chapitre, détaillons comment l'algorithme de Dijkstra a été optimisé grâce aux arbres de Fibonacci pour devenir celui implémenté à l'heure actuelle dans VirtualBelgium. Nous allons reprendre les étapes de l'algorithme de base décrit dans la deuxième section de ce chapitre.

1. Initialisation du réservoir par un arbre de Fibonacci vide.
2. Insertion de tous les nœuds du réseau dans l'arbre de Fibonacci. Leur valeur est fixée à l'infini et leur prédécesseur à un élément vide excepté pour ceux joignables depuis le nœud source. Leur valeur correspond alors au poids du lien

les liant et leur prédécesseur est le nœud lui-même. Le pointeur vers le nœud de valeur minimale est associé au nœud voisin de la source le plus proche.

3. Identification du nœud de valeur minimale (immédiat grâce au pointeur de l'étape précédente). Appelons-le $N$. Pour chaque voisin de ce nœud dans le réseau, la relation suivante est testée :

   **Si** la distance du voisin est supérieure à la somme de la distance depuis la source vers $N$ et du poids du lien de $N$ à son voisin, **Alors** mettre à jour la distance du nœud voisin dans l'arbre. Réarranger l'arbre si ce changement de valeur viole la propriété (2.1). Mettre à jour le prédécesseur du voisin par le nœud $N$
4. Supprimer le nœud $N$ de l'arbre pour le réarranger.
5. Répéter les deux dernières opérations tant que l'arbre de Fibonacci n'est pas vide

La complexité de cet algorithme optimisé est donnée par Johan Barthélemy dans sa thèse ([BJ14]) comme étant d'ordre $\mathcal{O}(n \log n + m)$ où $n$ est le nombre de nœuds du réseau et $m$ le nombre de liens.

## 2.6 Conclusion

Ce chapitre a décrit de manière détaillée le fonctionnement de l'algorithme de Dijkstra ainsi que son utilisation des arbres de Fibonacci. Rappelons que cette méthode est actuellement utilisée dans le projet VirtualBelgium pour calculer les plus courts chemins dans les chaines d'activités des agents. Le chapitre suivant va décrire un candidat potentiel pour le même rôle.

# Chapitre 3

# Un autre algorithme de plus courts chemins

Actuellement, le calcul des plus courts chemins pour le choix des destinations dans VirtualBelgium est obtenu grâce à l'algorithme de Dijkstra modifié avec les arbres de Fibonacci. Ce chapitre présente une autre solution pour calculer ces plus courts chemins.

## 3.1 L'algorithme de Floyd, principe général

Au lieu de partir d'un nœud source et de calculer les plus courts chemins vers tous les autres nœuds, une autre approche serait de construire en entier la matrice des plus courts chemins $D$. Les destinations seraient alors choisies simplement en parcourant la matrice à la recherche d'un nœud situé à la bonne distance de la source. L'algorithme de Floyd permet de construire cette matrice. Contrairement à l'algorithme de Dijkstra, il a la particularité de ne devoir calculer les plus courts chemins qu'une seule fois au début du programme. La suite ne demande qu'une recherche dans une ligne de la matrice.

Rappelons que la matrice des plus courts chemins $D$ est définie de façon suivante :

$$D(i,j) = \begin{cases} \text{Le poids du plus court chemin entre } i \text{ et } j \text{ si } i \neq j \\ 0 \text{ sinon} \end{cases}$$

Ce chapitre décrit l'algorithme de Floyd ainsi que ses caractéristiques. Le suivant regroupera les tests mis en œuvre qui permettront de comparer cette alternative avec l'actuelle.

## 3.2 Fonctionnement

Commençons par comprendre en détail l'idée de cet algorithme. Le raisonnement se fait par récurrence. Supposons que nous connaissions le plus court des chemins entre les sommets $i$ et $j$ qui n'utilise que les $k$ premiers nœuds du réseau. Notons son poids $c_k(i,j)$. La notion de $k$ premiers nœuds provient d'une façon arbitraire de

les dénoter. La disposition du réseau en elle-même n'entre pas en compte dans cette numérotation.

S'il est impossible de relier $i$ et $j$ avec cette restriction, la distance est fixée à l'infini. Ajoutons un $k+1$ième nœud : soit le plus court chemin, entre $i$ et $j$, est inchangé ; soit le chemin emprunté change, de telle sorte que le poids du chemin diminue. Dans ce second cas, le nouveau chemin passe nécessairement par le nouveau nœud. Le poids du chemin $c_k(i,j)$, entre $i$ et $j$, est remplacé par $c_{k+1}(i,k+1) + c_{k+1}(k+1,j)$, la somme des chemins de $i$ à $k+1$ et de $k+1$ à $j$. L'idée générale de l'algorithme est d'itérer cette opération jusqu'à utiliser le réseau tout entier. Mathématiquement, nous pouvons résumer tout cela par la relation de récurrence suivante :

$$c_k(i,j) = \begin{cases} \min\{c_k(i,j), c_{k+1}(i,k+1) + c_{k+1}(k+1,j)\} & \text{si} \quad k \geq 1 \\ w(i,j) & \text{si} \quad k = 0 \end{cases}$$

## 3.3 Code

L'algorithme de Floyd provenant du livre [AGK03] est donné par le code 3.1.

```
\\initialisation de la matrice des plus courts chemins par
  celle d'adjacence
D = A
\\Corps de l'algorithme
for k from 1 to n
    for i from 1 to n
        for j from 1 to n
            if (i==j) OR (i==k) OR (j==k)
                continue
            D_{i,j} = min D_{i,j}, D_{i,k} + D_{k,j}
        end
    end
end
```

Code 3.1 – pseudo-code de Floyd

La première instruction de l'intérieur des trois boucles successives exprime le fait que la mise à jour de l'élément de la matrice n'est pas nécessaire si deux des trois indices $i, j, k$ sont égaux. En effet, si $i = j$, l'élément $D_{i,j}$ est sur la diagonale et est donc toujours nul. Si $i = k$, le calcul du minimum revient à faire la comparaison entre $D_{i,j}$ et $D_{i,i} + D_{i,j} = 0 + D_{i,j}$ qui sont égaux entre eux. L'expression devient :

$$D_{i,j} = \min(D_{i,j}, D_{i,j})$$

On voit ainsi que $D_{i,j}$ est inchangé. Enfin le cas $j = k$ s'explique de façon similaire que le précédent.

L'avantage de ce code est qu'il n'est pas nécessaire d'allouer en mémoire la place pour une matrice temporaire à l'intérieur des trois boucles. Les mises à jour de valeurs peuvent être faites directement dans la matrice d'adjacence comme on peut le voir dans le code 3.1.

## 3.4 Complexité

Le calcul de la complexité est très simple. L'algorithme est composé de trois boucles imbriquées. Chacune parcourant le nombre de nœuds du réseau : $n$. La complexité est donc d'ordre $\mathcal{O}(n^3)$.

## 3.5 Forme parallèle

Calculer les déplacements de millions d'agents sur un réseau aussi conséquent que le réseau routier belge peut rapidement devenir très gourmand en ressources. Un ordinateur de bureau classique mettrait beaucoup trop de temps pour effectuer toutes les opérations. Une grande partie de la tâche sera donc effectuée en calcul parallèle. Cette technique de programmation est le fait de décomposer son programme en morceaux plus restreints. Ceux-ci peuvent alors effectuer des opérations, relativement indépendamment, sur des processeurs différents et surtout en même temps pour réduire le temps de calcul.

Plusieurs types de parallélisme existent. Nous détaillerons plus précisément celui que nous utiliserons en même temps que la présentation des algorithmes comparés. VirtualBelgium intègre le calcul parallèle grâce à la librairie C++ *MPI* dont la documentation peut être trouvée sur le site officiel de sa documentation [MPI].

Dans le livre *Parallele Computing* ([AGK03]), les auteurs proposent de partager les calculs des éléments de la matrice $D$ en $p$ processeurs. Nous supposons avoir à disposition un nombre carré de processeurs numérotés de $P_{1,1}$ à $P_{\sqrt{p},\sqrt{p}}$. Chacun d'eux s'occupe d'une sous-matrice de $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ éléments, comme présenté à la figure 3.1. Ainsi, le processeur $P_{i,j}$ calcule les éléments de la matrice compris entre $\left((i-1)\frac{n}{\sqrt{p}}+1, (j-1)\frac{n}{\sqrt{p}}+1\right)$ et $\left(i\frac{n}{\sqrt{p}}, j\frac{n}{\sqrt{p}}\right)$. Rappelons que le calcul de $D_{i,j}$ nécessite de connaitre, à l'étape $k$ de la boucle extérieure, les valeurs $D_{i,k}$ et $D_{k,j}$. Or elles ne se trouvent pas nécessairement dans les données du même processeur. Nous devons partager ces informations.

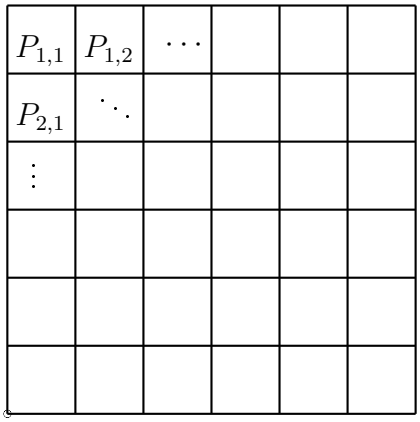



FIGURE 3.1 – Matrice $D$ divisée en sous-blocs

Pour rendre notre algorithme calculable par plusieurs processeurs simultanément, ceux-ci doivent partager l'information calculée. À l'itération $k$, les processeurs, dont la mémoire associée contient les éléments de la ligne $k$ de la matrice, envoient leurs données aux processeurs travaillant sur les mêmes colonnes que les leurs. De façon similaire, les processeurs contenant les éléments de la colonne $k$ de la matrice envoient leurs données à ceux de la même ligne. Quand toutes les données sont partagées, les processeurs peuvent mettre à jour la matrice à la manière du code 3.1. Voyons cela avec l'exemple suivant : il s'agit du même réseau que celui utilisé pour illustrer l'algorithme de Dijkstra à la page 17 mais auquel nous avons retiré un noeud afin de faciliter le découpage en sous-matrices. Supposons aussi que nous avons quatre processeurs à notre disposition.

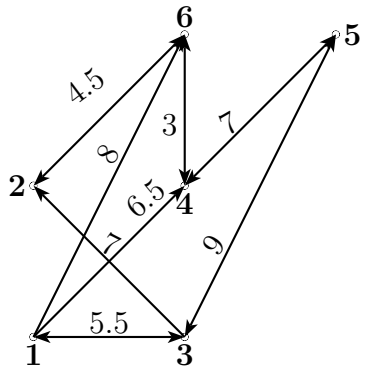



FIGURE 3.2 – Un exemple de réseau pour tester l'algorithme de Floyd

La matrice d'adjacence correspondante est la suivante :

$$A = \begin{pmatrix} 0 & \infty & 5.5 & 6.5 & \infty & 8 \\ \infty & 0 & \infty & \infty & \infty & 4.5 \\ 5.5 & 7 & 0 & \infty & 9 & \infty \\ \infty & \infty & \infty & 0 & 7 & 3 \\ \infty & \infty & 9 & 7 & 0 & \infty \\ \infty & 4.5 & \infty & 3 & \infty & 0 \end{pmatrix}$$

Chaque processeur s'occupe d'une sous-matrice de taille $3 \times 3$ de la façon suivante :

| $P_{1,1}$ | $P_{1,2}$ |
|---|---|
| $\begin{pmatrix} 0 & \infty & 5.5 \\ \infty & 0 & \infty \\ 5.5 & 7 & 0 \end{pmatrix}$ | $\begin{pmatrix} 6.5 & \infty & 8 \\ \infty & \infty & 4.5 \\ \infty & 9 & \infty \end{pmatrix}$ |
| $\begin{pmatrix} \infty & \infty & \infty \\ \infty & \infty & 9 \\ \infty & 4.5 & \infty \end{pmatrix}$ | $\begin{pmatrix} 0 & 7 & 3 \\ 7 & 0 & \infty \\ 3 & \infty & 0 \end{pmatrix}$ |
| $P_{2,1}$ | $P_{2,2}$ |

- $P_{1,1}$ s'occupe de la sous-matrice partant de $A_{1,1}$ à $A_{3,3}$.
- $P_{1,2}$ s'occupe de la sous-matrice partant de $A_{1,4}$ à $A_{3,6}$.
- $P_{2,1}$ s'occupe de la sous-matrice partant de $A_{4,1}$ à $A_{6,3}$.
- $P_{2,2}$ s'occupe de la sous-matrice partant de $A_{4,4}$ à $A_{6,6}$.

Détaillons maintenant la première étape de l'algorithme pour comprendre comment les processeurs doivent partager l'information calculée.

## $\mathbf{k = 1}$

**Premier processeur :** $P_{1,1}$

Ce processeur doit effectuer ces neuf opérations :

$$\begin{aligned} A_{1,1} &= min(A_{1,1}, A_{1,1} + A_{1,1}) \\ A_{1,2} &= min(A_{1,2}, A_{1,1} + A_{1,2}) \\ A_{1,3} &= min(A_{1,3}, A_{1,1} + A_{1,3}) \\ A_{2,1} &= min(A_{2,1}, A_{2,1} + A_{1,1}) \\ A_{2,2} &= min(A_{2,2}, A_{2,1} + A_{1,2}) \\ A_{2,3} &= min(A_{2,3}, A_{2,1} + A_{1,3}) \\ A_{3,1} &= min(A_{3,1}, A_{3,1} + A_{1,1}) \\ A_{3,2} &= min(A_{3,2}, A_{3,1} + A_{1,2}) \\ A_{3,3} &= min(A_{3,3}, A_{3,1} + A_{1,3}) \end{aligned}$$

Il n'a besoin d'aucune autre information que celles contenues dans sa sous-matrice.

**Deuxième processeur :** $P_{1,2}$

$$
\begin{aligned}
A_{1,4} &= min(A_{1,4}, \mathbf{A_{1,1}} + A_{1,4}) \\
A_{1,5} &= min(A_{1,5}, \mathbf{A_{1,1}} + A_{1,5}) \\
A_{1,6} &= min(A_{1,6}, \mathbf{A_{1,1}} + A_{1,6}) \\
A_{2,4} &= min(A_{2,4}, \mathbf{A_{2,1}} + A_{1,4}) \\
A_{2,5} &= min(A_{2,5}, \mathbf{A_{2,1}} + A_{1,5}) \\
A_{2,6} &= min(A_{2,6}, \mathbf{A_{2,1}} + A_{1,6}) \\
A_{3,4} &= min(A_{3,4}, \mathbf{A_{3,1}} + A_{1,4}) \\
A_{3,5} &= min(A_{3,5}, \mathbf{A_{3,1}} + A_{1,5}) \\
A_{3,6} &= min(A_{3,6}, \mathbf{A_{3,1}} + A_{1,6})
\end{aligned}
$$

Ce processeur a besoin de connaitre les valeurs des éléments $A_{1,1}, A_{2,1}$ et $A_{3,1}$. Autrement dit, la première colonne de la sous-matrice calculée par le processeur $P_{1,1}$. Remarquons que $k = 1$ et qu'il manque les valeurs de la première colonne du processeur travaillant sur les mêmes lignes de $A$ que $P_{1,2}$.

**Troisième processeur :** $P_{2,1}$

$$
\begin{aligned}
A_{4,1} &= min(A_{4,1}, A_{4,1} + \mathbf{A_{1,1}}) \\
A_{4,2} &= min(A_{4,2}, A_{4,1} + \mathbf{A_{1,2}}) \\
A_{4,3} &= min(A_{4,3}, A_{4,1} + \mathbf{A_{1,3}}) \\
A_{5,1} &= min(A_{5,1}, A_{5,1} + \mathbf{A_{1,1}}) \\
A_{5,2} &= min(A_{5,2}, A_{5,1} + \mathbf{A_{1,2}}) \\
A_{5,3} &= min(A_{5,3}, A_{5,1} + \mathbf{A_{1,3}}) \\
A_{6,1} &= min(A_{6,1}, A_{6,1} + \mathbf{A_{1,1}}) \\
A_{6,2} &= min(A_{6,2}, A_{6,1} + \mathbf{A_{1,2}}) \\
A_{6,3} &= min(A_{6,3}, A_{6,1} + \mathbf{A_{1,3}})
\end{aligned}
$$

Ce processeur a besoin de connaitre les valeurs des éléments $A_{1,1}, A_{1,2}$ et $A_{1,3}$. Autrement dit, la première ligne de la sous-matrice calculée par le processeur $P_{1,1}$. Remarquons que $k = 1$ et qu'il manque les valeurs de la première ligne du processeur travaillant sur les mêmes colonnes de $A$ que $P_{2,1}$.

**Quatrième processeur :** $P_{2,2}$

$$\begin{aligned}
A_{4,4} &= min(A_{4,4}, \mathbf{A_{4,1}} + \mathbf{A_{1,4}}) \\
A_{4,5} &= min(A_{4,5}, \mathbf{A_{4,1}} + \mathbf{A_{1,5}}) \\
A_{4,6} &= min(A_{4,6}, \mathbf{A_{4,1}} + \mathbf{A_{1,6}}) \\
A_{5,4} &= min(A_{5,4}, \mathbf{A_{5,1}} + \mathbf{A_{1,4}}) \\
A_{5,5} &= min(A_{5,5}, \mathbf{A_{5,1}} + \mathbf{A_{1,5}}) \\
A_{5,6} &= min(A_{5,6}, \mathbf{A_{5,1}} + \mathbf{A_{1,6}}) \\
A_{6,4} &= min(A_{6,4}, \mathbf{A_{6,1}} + \mathbf{A_{1,4}}) \\
A_{6,5} &= min(A_{6,5}, \mathbf{A_{6,1}} + \mathbf{A_{1,5}}) \\
A_{6,6} &= min(A_{6,6}, \mathbf{A_{6,1}} + \mathbf{A_{1,6}})
\end{aligned}$$

Ce processeur a besoin de connaitre les valeurs des éléments $A_{1,4}, A_{1,5}, A_{1,6}, A_{4,1}, A_{5,1}$ et $A_{6,1}$. Les trois premiers forment la première ligne de la sous-matrice calculée par le processeur $P_{1,2}$. Les trois derniers forment la première colonne de la sous-matrice calculée par le processeur $P_{2,1}$. Remarquons que $k = \mathbf{1}$, qu'il manque les valeurs de la **première** ligne du processeur travaillant sur les mêmes colonnes de $A$ que $P_{2,2}$ et qu'il manque les valeurs de la première colonne du processeur travaillant sur les mêmes lignes de $A$ que $P_{2,2}$.

Remarquons que l'envoi de données se fait avant les mises à jour des éléments car les processeurs ont besoin de ces ressources pour pouvoir effectuer le travail demandé.

## 3.6 Complexité de la forme parallélisée

Le temps d'exécution de l'algorithme sous forme parallèle se décompose en deux. Il y a d'un côté le temps de calcul à part entière que les processeurs utilisent et de l'autre, le temps nécessaire pour partager les informations entre processeurs : la communication. Pour le premier temps, la complexité est tout simplement d'ordre $\mathcal{O}\left(\frac{n^3}{p}\right)$ puisqu'un total de $n^3$ opérations doivent être effectuées simultanément par $p$ processeurs .

Pour la communication, nous savons que les processeurs s'occupant des sous-matrices contenant une partie de la $k$-ième ligne ou de la $k$-ieme colonne de $A$ envoient $\frac{n}{\sqrt{p}}$ éléments [1] aux autres processeurs. Tous les processeurs ne recevant pas l'information en même temps, il faut attendre que tout soit bien distribué. Ce temps s'appelle la synchronisation et sa longueur est d'ordre $\log p$. Enfin, cela vaut pour une valeur de $k$. Or il y en a un total de $n$. Le temps total de communication est de l'ordre de $\mathcal{O}\left(n.\frac{n}{\sqrt{p}}.\log p\right)$.

1. correspondant à l'entièreté d'une ligne ou d'une colonne de sous-matrice

La complexité totale de l'algorithme de Floyd en version parallèle est la somme des deux complexité :

$$\mathcal{O}\left(\frac{n^3}{p} + \frac{n^2}{\sqrt{p}}\log p\right)$$

Lorsque $n > \sqrt{p}\log p$ , le terme dominant de cette complexité est $\frac{n^3}{p}$. Si $n$ vaut 262.000 (le nombre de nœuds dans le plus grand réseau de VirtualBelgium) alors ce terme est dominant tant que $p$, le nombre de processeurs, est inférieur à 189008000, ce qui est logiquement toujours le cas.

## 3.7 Intégration dans VirtualBelgium

Le code C++ complet de la forme parallélisée se trouve dans les annexes. Cette section explique en français les étapes du code afin d'aider le lecteur à mieux le comprendre. Nous détaillerons cela de manière fort structurée et non sous forme de texte continu. De plus, les étapes seront expliquées dans l'ordre dans lequel elles interviennent dans le code.

Avant de commencer, voici une remarque générale d'ordre technique. En C++ (le langage de programmation dans lequel est écrit VirtualBelgium), les indices des éléments dans une structure (tableau, matrice, etc.) commencent toujours à zéro et non à un, comme cela peut être le cas dans d'autres langages.

La partie parallèle se base, comme dit dans l'introduction, sur la librairie MPI incluse dans la librairie Boost. Les différentes commandes utilisées sont :

**rank().** Le code de VirtualBelgium a beau être exécuté sur plusieurs processeurs, il est exactement le même pour tous. Or ces processeurs ne doivent pas faire le même travail au même moment. Il faut donc pouvoir les distinguer pour spécifier les tâches. La fonction *rank()* renvoie un nombre entre 0 et $p-1$ où $p$ est le nombre de processeurs affectés. Ce nombre sert d'identifiant au processeur qui "lit" cette commande.

**size().** En plus de savoir quel processeur lit le code, il est intéressant de connaitre combien y sont affectés pour répartir le mieux possible la tâche. La fonction *size()* renvoie le nombre de processeurs alloués au programme.

**send(rank,tag,data).** Une fonction dont l'utilité est très facile à comprendre, est celle permettant d'envoyer de l'information d'un processeur vers un autre. La fonction *send(rank,tag,data)* envoie la valeur *data* vers le processeur identifié par *rank*. Cet envoi possède lui aussi un identifiant propre donné par *tag*. Dans notre cas, la variable *data* sera un tableau comprenant une partie de la $k$-ième ligne ou colonne de la matrice d'adjacence.

**recv(rank,tag,data).** Après l'envoi de l'information par les processeurs s'occupant des sous-matrices comprenant la $k$-ième ligne ou colonne, il faut bien sûr que les autres reçoivent cette information et pas une autre. En effet, deux processeurs qui s'occupent de sous-matrices non alignées ne partagent jamais de données. Il ne faut donc pas tout envoyer à tout le monde. La fonction *recv(rank,tag,data)*

récupère l'information désignée par *tag*, envoyée par le processeur identifié par *rank*, pour la stocker dans la variable *data*. Cette variable doit être allouée antérieurement.

L'argument *tag* dans les fonctions *send* et *recv* est simplement la valeur de l'indice de boucle $k$. Nous sommes donc sûr que chaque envoi est différent d'un autre : soit par son *tag*, soit par le processeur émetteur/récepteur.

## Étape 1 : les coordonnées des processeurs

Cette première étape est cruciale si nous voulons faciliter les calculs des étapes suivantes. En effet, rappelons que les processeurs sont indexés par un entier de 0 à $p-1$ et non par un couple d'entiers. Cette deuxième façon de faire est bien plus adéquate lors d'une disposition en grille (voir la figure 3.1). Les sous-matrices sont distribuées de gauche à droite puis de haut en bas comme le montre le schéma ci-dessous.

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | ⋮ | |
| | | | |
| | | | |

Pour avoir les coordonnées en deux dimensions, nous calculons la division euclidienne de l'identifiant du processeur par la racine carrée du nombre total alloué au programme pour trouver la ligne. Le reste de la division de ces deux nombres donne la colonne. Par exemple, la position du numéro 5 dans notre schéma est

$$(5/4,\ 5 \mod 4) = (1, 1)$$

## Étape 2 : les processeurs contiennent-ils les données à envoyer ?

À chaque étape $k$, certains processeurs doivent envoyer une ligne (ou une colonne) vers certains autres. Pour savoir quel rôle occupe chacun, nous nous servons des coordonnées obtenues précédemment. Un processeur envoie une ligne de sa sous-matrice si celle-ci contient une partie de la $k$-ième ligne de la matrice d'adjacence à l'étape $k$ de l'algorithme. Il en va de même pour la $k$-ième colonne.

Nous connaissons la taille $n$ de la matrice et le nombre $p$ de processeurs alloués. Une sous-matrice est donc de dimension :

$$\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$$

A nouveau, une division euclidienne permet de dire quelles sous-matrices contiennent les parties des $k$-ième ligne et $k$-ième colonne. Ainsi, le processeur ayant pour coordonnées $(i, j)$ s'occupe de la $k$-ième ligne si et seulement si

$$k/\frac{n}{\sqrt{p}} = i$$

et s'occupe de la $k$-ième colonne si et seulement si

$$k/\frac{n}{\sqrt{p}} = j$$

### Étape 3 : l'envoi et la réception des données

Sachant quels processeurs doivent envoyer des données, nous pouvons écrire une boucle sur une des coordonnées (la première si on envoie une ligne ou la deuxième si on envoie une colonne). Il faut tout de même faire attention à ne pas s'envoyer les données à soi-même. En effet, d'une part, c'est une perte de temps et d'autre part le programme peut s'arrêter s'il reste des processeurs qui n'ont pas "reçu" de données (grâce à la fonction *recv*).

### Étape 4 : calcul des mises à jour des éléments des sous-matrices

Maintenant que les mémoires des processeurs possèdent toutes les données nécessaires, ces dernier peuvent effectuer leur calculs sur les éléments de leur sous-matrice respective.

## 3.8 Utilisation de la matrice

La matrice des plus courts chemins étant construite, elle ne donne pas encore de nœud situé à une distance acceptable du point de départ. Il faut chercher l'information dans la matrice. Pour cela, partons de la définition d'un élément $(i, j)$ :

$$D_{i,j} = \text{la valeur du plus court chemin entre } i \text{ et } j$$

Ainsi, la $i$-ième ligne de la matrice comprend les distances des plus courts chemins entre le nœud $i$ et chaque autre nœud du réseau.

Pour déterminer le nœud d'arrivée où sera associé l'activité, nous commençons par filtrer toutes les distances et les identifiants des nœuds respectifs. Si, après avoir passé en revue tous les éléments de la ligne, aucun ne convient, nous ajoutons à la valeur de recherche un paramètre $\epsilon$ fixé à 250 mètres et la $i$-ième ligne est de nouveau entièrement parcourue. Tant qu'aucun nœud n'est isolé, $\epsilon$ est doublé.

Lorsqu'au moins un nœud est candidat, il reste à choisir celui qui "accueillera" l'activité considérée. Cela est simplement fait en en tirant un aléatoirement suivant une loi uniforme.

## 3.9 Conclusion

Ce chapitre était l'équivalent du second mais cette fois dans le but d'expliquer le fonctionnement de l'algorithme de Floyd et d'en présenter son intégration dans le projet VirtualBelgium. Nous allons maintenant passer à la comparaison de ces deux méthodes.

# Chapitre 4

# Comparaison des algorithmes

Les deux chapitres précédents ont permis de présenter et décrire deux algorithmes de calcul de plus courts chemins. Le premier, celui de Dijkstra optimisé grâce aux arbres de Fibonacci, est implémenté dans VirtualBelgium. Le second, celui de Floyd, propose une alternative à la problématique du temps d'exécution. Ce chapitre a pour but de comparer ces deux algorithmes dans le cas précis de VirtualBelgium.

La problématique de départ étant le temps élevé d'exécution quand la taille du réseau est importante, nous avons axé les tests de comparaison des deux procédés sur les temps d'exécution. Mais nous allons décrire au préalable les critères théoriques de comparaison à la section 4.3.

## 4.1 Notions préliminaires

### 4.1.1 Structure des fichiers de stockage

Cette section détaille l'architecture des fichiers dans lesquels sont stockés les réseaux. Lors d'une simulation par VirtualBelgium, le réseau utilisé est donné sous forme d'un fichier XML. L'architecture de ce fichier est la suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<network>
    <nodes>
        <node id x y/>
        ⋮
    </nodes>

    <links capperiod="01:00:00" effectivecellsize="7.5"
     effectivelanewidth="3.75">
        <link id from to length freespeed capacity permlane oneway
         modes origid />
        ⋮
    </links>
```

</network>

On retrouve les deux ensembles $V$ et $E$ composant un graphe. Dans le premier, celui des nœuds, chaque élément est caractérisé par trois paramètres : un identifiant unique (**id**), une longitude (**x**) et une latitude (**y**). Le deuxième comprend les liens reliant ces nœuds. Rappelons que le réseau représente un réseau routier et donc que ces liens sont en réalité des routes. Elles ont trois caractéristiques communes et onze individuelles.

- Les communes sont :

**-capperiod="01 :00 :00"** : la période pour laquelle la capacité maximale du tronçon est calculée (voir le détail de **capacity**). Elle est fixée à une heure.

**-effectivecellsize="7.5"** : la longueur d'occupation d'un véhicule sur la route, en mètres.

**-effectivelanewidth="3.75"** : la largeur de la route, en mètres.

- Les individuelles sont :

**-id** : un identifiant unique

**-from** : l'identifiant du nœud de départ (présent dans l'ensemble "nodes" du fichier XML)

**-to** : l'identifiant du nœud d'arrivée.

**-length** : la longueur du tronçon routier, en mètres.

**-freespeed** : la vitesse maximale autorisée, en kilomètres par secondes

**-capacity** : la capacité maximale que peut supporter le tronçon en terme de véhicules, sur une période donnée par **capperiod**.

**-permlane** : le nombre de voies du tronçon

**-oneway** : l'indicateur de sens unique ("1" si la voie est à sens unique)

**-modes** : la liste des véhicules autorisés sur la voie.

**-origid** : L'ID de l'ancien lien à double sens. Lorsqu'un lien représente une route à double sens, il est plus facile de le séparer en deux liens à sens unique. L'**origid** reprend l'identifiant de l'ancien lien à double sens.

Ces informations proviennent du site [doc]. Pour les calculs de plus courts chemins, nous devons définir un poids pour les liens. Dans notre cas, il correspond à la longueur (**length**) de la route. Seule cette valeur est donc à prendre en compte dans notre algorithme.

### 4.1.2 Null model

**Définition 9.** *Le **null model** d'un graphe est un réseau qui lui ressemble dans sa structure générale mais dont une partie est modifiée de façon aléatoire.*

Par exemple, un null model souvent utilisé, illustré à la Figure 4.1 consiste, à partir d'un graphe, à permuter certains liens tout en gardant pour chaque nœud

ses degrés entrants et sortants initiaux. Pour garantir ce respect des degrés, il suffit de tirer aléatoirement, dans une distribution uniforme, deux liens du réseau et d'intervertir soit leur nœud d'arrivée soit leur nœud de départ. On répète cette opération jusqu'à atteindre une condition d'arrêt prédéfinie. Celle-ci peut, par exemple, concerner le nombre d'arcs à interchanger. Plus nous intervertissons de liens, plus le réseau diffère de l'original. Nous avons décidé, pour nos tests, de réaliser autant de changements qu'il y a de paires de nœuds dans le réseau de départ.

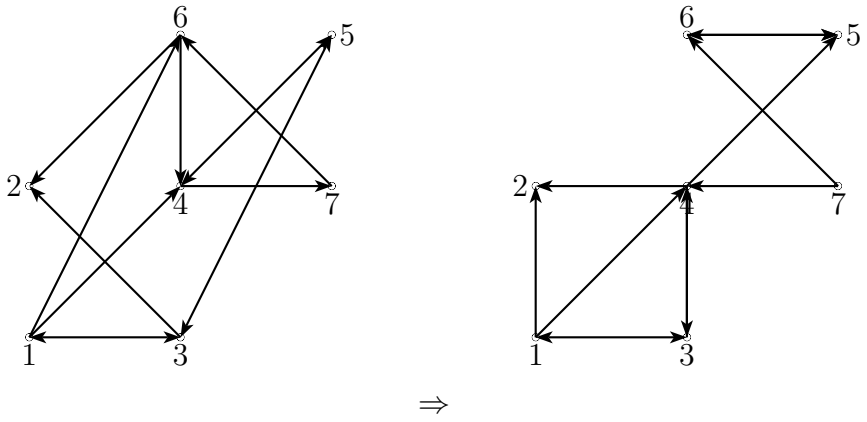



FIGURE 4.1 – Un exemple null model

On remarque à la figure 4.1 que les liens ne sont pas du tout disposés de manière similaire. En revanche, tous les nœuds ont gardé le même nombre de liens entrants et sortants.

L'intérêt de la comparaison entre un réseau et son équivalent *null model* est d'analyser si l'original possède certaines structures particulières comme, par exemple, les clusters. En effet, en introduisant de l'aléatoire dans le réseau (typiquement en changeant les origines et destinations de certains arcs), on "casse" sa structure. Si une propriété du réseau original est observée dans le réseau modifié, on peut alors en déduire que cette propriété n'est pas due à la structure, mais dépend d'autres critères. Donc, si une propriété est observée dans le réseau original et dans son équivalent modifié, on peut déduire que la structure de l'original n'en est pas responsable. La propriété sera alors vue comme dépendante d'un autre aspect des graphes qui n'a pas été modifié (par exemple le degré moyen des nœuds lors d'une répartition nouvelle des arcs). L'utilisation du *null model* est expliquée plus en détail dans l'article [PP12]. Nous verrons dans la section 4.5 comment nous avons utilisé cette notion.

Pour pouvoir utiliser la technique du null model, nous avons dû créer un nouveau réseau lisible par VirtualBelgium. C'est-à-dire un réseau stocké dans le format XML. Pour pouvoir générer des réseaux sous ce format, nous utilisons le langage Python. Ce choix est justifié par les atouts suivants : il est très simple d'utilisation, multiplateformes et très riche en librairies complémentaires. L'un de ses atouts majeurs, dans notre cas, est sa bibliothèque standard. Elle possède un module spécialement

dédié à la lecture et à l'écriture de fichiers XML, le module *xml minidom* [xml] et un autre dédié à la gestion de réseaux : *networkX* [nxP]. Il permet ainsi de hiérarchiser et de structurer automatiquement des données. Les codes écrits pour cette partie sont disponibles en annexe.

## 4.2 Structure des tests

Le but de ce mémoire étant la comparaison entre l'algorithme de Dijkstra optimisé par les arbres de Fibonacci et l'algorithme de Floyd, nous avons testé ces deux procédés sur trois réseaux différents. La comparaison se base sur les temps d'exécution d'une simulation des chaines d'activités par VirtualBelgium. Mais chacun des tests comporte une comparaison supplémentaire ; il nous a paru en effet intéressant d'effectuer aussi une comparaison entre l'algorithme de Dijkstra optimisé par les arbres de Fibonacci et l'algorithme de Floyd, mais sans tenir compte du temps mis pour calculer la matrice des plus courts chemins. En effet la complexité pour construire cette matrice étant $\mathcal{O}(n^3)$, il semble intéressant, à priori, de faire des tests sans ce calcul. Une piste d'application possible pour VirtualBelgium est de pouvoir éventuellement utiliser cette procédure, si elle s'avère efficace, pour des problématiques où le réseau de départ reste inchangé et qui garde donc la même matrice de plus courts chemins.

En résumé, chaque test concernera un réseau différent et comportera trois procédés :

1. Dijkstra optimisé par les arbres de Fibonacci
2. Floyd complet
3. Floyd sans le calcul de la matrice des plus courts chemins

Mais avant de passer aux tests à proprement parler, nous allons préalablement comparer les trois procédures d'un point de vue théorique.

## 4.3 Comparaison théorique des trois procédures

Sur base des chapitres 2 et 3, nous pouvons construire le tableau suivant qui rappelle les caractéristiques de chaque algorithme.

| | **Dijkstra optimisé par les arbres de Fibonacci** | **Floyd complet** | **Floyd sans calculer la matrice des plus courts chemins** |
|---|---|---|---|
| **Complexité** | $\mathcal{O}(n \log n + m)$ où $n$ est le nombre de nœuds du réseau et $m$ le nombre d'arcs. Voir la thèse de Johan Barthelemy ([BJ14] | $\mathcal{O}(n^3)$ | $\mathcal{O}(n)$ car il faut parcourir une ligne d'une matrice de taille $n \times n$ |
| **Obligation ou non de parcourir l'entièreté des nœuds pour choisir un candidat valable pour accueillir l'activité de l'individu** | Non car les nœuds sont parcourus de manière ordonnée | Oui | Oui |
| **Nécessité de calculer ou non les plus courts chemins entre les différents nœuds** | Seulement ceux dont le départ est un lieu de résidence d'un individu | Tous | Aucun |
| **Espace mémoire nécessaire** | $\mathcal{O}(n)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2)$ |

## 4.4 Test 1 : un petit réseau

Dans ce premier test, nous avons choisi comme réseau une petite partie du réseau complet de la Belgique dans VirtualBelgium. Il comprend 2280 nœuds, ce qui correspond à environ 1% du nombre de nœuds du réseau total. Les nœuds ont été choisis aléatoirement suivant une loi uniforme, sans critère géographique. Tous les liens partant ou arrivant à un nœud supprimé ont donc aussi été supprimés.

### 4.4.1 Processeur et ordinateur utilisés

La petite taille du réseau nous permet de pouvoir effectuer nos trois tests sur un seul processeur d'un ordinateur de bureau. Les spécificités de cette machine sont les suivantes :

- Un seul processeur Intel(R) Core(TM) i5-3317U CPU @ 1.70GHz
- 4Go de RAM
- Architecture 64 bits
- Système d'exploitation Debian 7 en machine virtuelle depuis Windows 8.1.

### 4.4.2 Résultats

Les temps obtenus sur dix simulations sont les suivants :

| Observations | Dijkstra | Floyd complet | Floyd sans calcul de la matrice |
|---|---|---|---|
| Temps en secondes | 18.32 | 32.71 | 5.13 |
| | 18.79 | 33.09 | 5.03 |
| | 18.88 | 32.68 | 4.98 |
| | 18.89 | 32.54 | 5.12 |
| | 18.87 | 32.52 | 5.04 |
| | 18.34 | 32.94 | 5.16 |
| | 18.2 | 33.35 | 5.08 |
| | 18.45 | 33.16 | 4.94 |
| | 18.90 | 32.69 | 5.08 |
| | 19.01 | 32.24 | 5.02 |
| Moyennes | 18.67 | 32.8 | 5.06 |
| Variances | 0.08 | 0.10 | 0.004 |
| Écarts types | 0.29 | 0.32 | 0.07 |

La figure 4.2 reprend ces valeurs sur un même graphique.

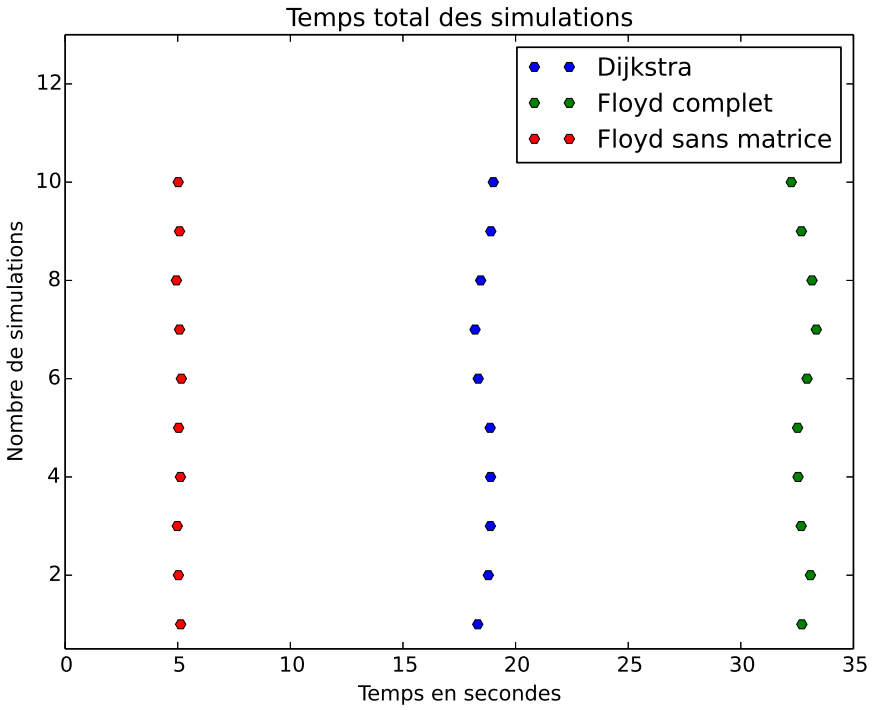



FIGURE 4.2 – Différents temps pour les simulations sur le petit réseau

On peut ici observer que la version de Dijkstra est plus rapide que celle de Floyd complète. En revanche, la version de Floyd sans la construction de la matrice est plus efficace.

Par conséquent, sur un petit réseau, lorsque l'on veut faire plusieurs simulations sans modifier la topologie de la carte, Floyd sera plus efficace que Dijkstra, car il pourra utiliser à nouveau la matrice de plus courts chemins précédemment calculée. Cela est probablement dû au fait que Dijkstra risque de calculer plusieurs fois les mêmes plus courts chemins, puisque beaucoup d'individus sont disposés sur un réseau restreint et vont donc certainement parcourir les mêmes chemins. Floyd lui, a déjà préalablement calculé tous les plus courts chemins et se contente de chercher des valeurs dans la ligne d'une matrice. Dans ce cas, le réseau étant petit, cette matrice n'est pas extrêmement lourde et le programme accède sans perdre trop de temps à chaque cellule des lignes.

## 4.5 Test 2 : le même petit réseau, mais modifié en null model

Nous avons effectué le test sur le même réseau que le précédent mais en le modifiant aléatoirement suivant la méthode des *null model* expliquée à la section 4.1.

### 4.5.1 Processeur et ordinateur utilisés

Étant donné l'analogie avec le premier test et pour garantir la rigueur de notre comparaison, nous avons gardé le même matériel à savoir :

- Un seul processeur Intel(R) Core(TM) i5-3317U CPU @ 1.70GHz
- 4Go de RAM
- Architecture 64 bits
- Système d'exploitation Debian 7 en machine virtuelle depuis Windows 8.1.

### 4.5.2 Résultats

Les temps obtenus sur dix simulations sont les suivants :

| Observations | Dijkstra | Floyd complet | Floyd sans calcul la matrice |
|---|---|---|---|
| Temps | 18.84 | 32.83 | 5.02 |
| en secondes | 18.18 | 32.31 | 4.93 |
| | 18.18 | 32.28 | 4.94 |
| | 18.8 | 32.52 | 7.03 |
| | 18.03 | 32.63 | 5.09 |
| | 18.59 | 32.26 | 5.03 |
| | 18.58 | 32.53 | 5.12 |
| | 18.3 | 32.29 | 5.18 |
| | 18.17 | 32.1 | 5.2 |
| | 18.22 | 32.24 | 5.09 |
| Moyennes | 18.39 | 32.4 | 5.26 |
| Variances | 0.075 | 0.044 | 0.35 |
| Écarts types | 0.27 | 0.21 | 0.6 |

La figure 4.3 reprend ces valeurs sur un même graphique.

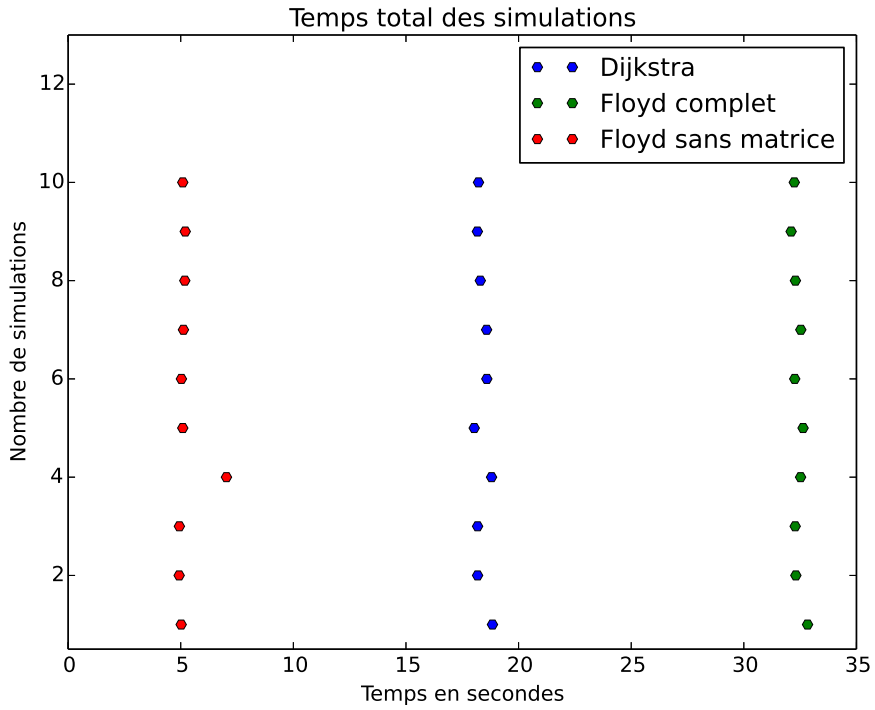



FIGURE 4.3 – Différents temps pour les simulations sur le réseau *null model*

nous pouvons à nouveau observer que la version de Dijkstra est plus rapide que celle de Floyd complète. En revanche, la version de Floyd sans la construction de la matrice est toujours la plus efficace.

En confrontant les résultats du premier et du deuxième test, nous pouvons conclure que la structure des réseaux n'a pas d'effet sur le temps d'exécution des trois algorithmes. En effet, quand on casse la structure du réseau, les résultat restent analogues.

Cette constatation empirique peut s'expliquer de façon théorique pour chacun des trois algorithmes :

- Pour l'algorithme de Dijkstra utilisant les arbres de Fibonacci, le nombre de nœuds étant identique dans les deux tests, le réservoir de l'algorithme garde la même taille. Le nombre d'exécutions nécessaires pour le vider est donc similaire dans les deux cas.
- Pour l'algorithme de Floyd complet, la matrice d'adjacence servant à initier l'algorithme dépend uniquement du nombre de nœuds. Ce nombre restant inchangé d'un test à l'autre, les deux matrices sont donc de même taille. Le temps pour les mettre à jour est similaire.
- Pour l'algorithme de Floyd sans matrice, le raisonnement est similaire au précédent. Les deux matrices de plus courts chemins sont de même taille.

## 4.6 Test 3 : le réseau de Namur

Dans ce troisième test, nous avons choisi un réseau de taille moyenne comprenant, 23234 nœuds. Il s'agit du réseau décrivant le réseau routier du Grand Namur.

### 4.6.1 Processeurs et ordinateur utilisés

Vu la taille du réseau, nous avons opté pour l'utilisation d'un ordinateur utilisant quatre processeurs. Les spécificités générales sont :

- Quatre processeurs Intel(R) Core(TM) i5-3317U CPU @ 1.70GHz
- 6Go de RAM
- Architecture 64 bits
- Système d'exploitation Debian 7 en machine virtuelle depuis Windows 8.1.

Nous avons utilisé les algorithmes de Dijkstra et de Floyd dans leur version parallélisée.

### 4.6.2 Résultats

Les temps obtenus sur sept simulations sont les suivants :

| Observations | Dijkstra | Floyd complet | Floyd sans calcul la matrice |
|---|---|---|---|
| Temps | 24.73 | 4234.58 | 556.14 |
| en secondes | 27.05 | 4223.8 | 554.82 |
| | 25.54 | 4234.5 | 552.94 |
| | 26.07 | 4237.36 | 554.34 |
| | 25.39 | 4229.65 | 555.43 |
| | 25.21 | 4233.71 | 559.43 |
| | 26.17 | 4236.65 | 556.1 |
| Moyennes | 25.73 | 4232.89 | 555.6 |
| Variances | 0.49 | 19.05 | 3.50 |
| Écarts types | 0.70 | 4.36 | 1.87 |

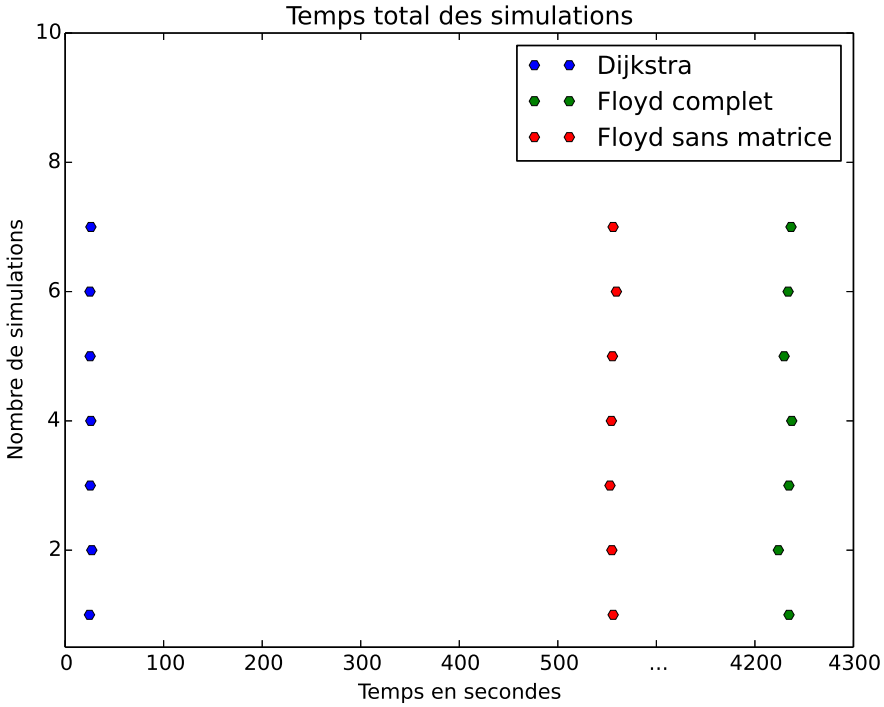



FIGURE 4.4 – Différents temps pour les simulations sur le réseau de Namur

La figure 4.2 illustre les valeurs du tableau comme pour le premier réseau. Notons que l'axe des abscisses n'est pas continu. Cela est dû au fait que le troisième jeu d'observations est très éloigné des deux autres. Nous avons eu recours à cette présentation pour pouvoir placer toutes les observations sur le même graphe.

La première observation que nous pouvons faire est la spectaculaire augmentation du temps d'exécution pour la simulation de l'algorithme de Floyd complet. La taille de la matrice des plus courts chemins ayant augmenté par rapport au test précédent, il semble logique d'avoir une augmentation significative du temps d'exécution. Néanmoins, nous avons été surpris par l'ampleur de cette augmentation. Rappelons que

la complexité de l'algorithme de Floyd est d'ordre $\mathcal{O}(n^3)$, ce qui pourrait expliquer cette énorme augmentation. Les temps d'exécution pour la simulation se passant du calcul de la matrice ont eux aussi augmenté et dépassent les valeurs obtenues avec Dijkstra utilisant les arbres de Fibonacci. On constate que les temps d'exécution induits par l'algorithme actuel de VirtualBelgium sont les meilleurs et restent dans les mêmes ordres de grandeur que précédemment.

Une autre source de ce ralentissement pourrait être aussi la parallélisation. En effet, rappelons que les tests précédents ont été effectués sur un seul processeur. Cependant, après avoir mis en place ce troisième test de comparaison, nous avons lancé une simulation de VirtualBelgium sur le réseau de Namur utilisant l'algorithme de Floyd complet et sans parallélisation. Nous avons stoppé l'exécution de la simulation après les 2500 premières étapes sur 23234 (ce nombre d'étapes correspond au nombre de nœuds dans le réseau de Namur) de la boucle extérieure de l'algorithme. Un total de 25 minutes s'était écoule pour ce début de simulation. Cela correspond à environ 10% du temps d'exécution. Une rapide extrapolation permet d'affirmer que la simulation aurait requis autour de quatre heures de temps pour arriver au bout de la construction de la matrice des plus courts chemins.

Nous voyons ainsi que la parallélisation a sans doute aidé à l'accélération du temps d'exécution, mais malheureusement pas de façon assez significative. Nous concluons donc que la taille du réseau est bien la cause du ralentissement et non la parallélisation.

Aussi, il est logique que Dijkstra soit moins influencé par un changement de taille du réseau, car il ne se préoccupe à aucun moment de la totalité du réseau. Cet algorithme sera exécuté une fois par personne et par activité. À chaque exécution, il ne se concentrera que sur les nœuds proches du domicile de la personne. Les autres, ceux dont la distance est supérieure à la valeur tirée comme distance prête à être parcourue pour se rendre à son activité, ne sont pas pris en compte. Au contraire, Floyd commence par calculer tous les plus courts chemins. Lors d'un ajout de nœud dans le réseau, la matrice passe d'une taille $n \times n$ à $(n+1) \times (n+1)$. Ce qui signifie qu'un nœud supplémentaire dans le réseau provoque l'ajout d'une colonne et d'une ligne complète. Tous ces éléments de matrice en plus nécessitent également des calculs. Le temps de création de la matrice des plus courts chemins est donc très sensible à la taille du réseau associé.

## 4.7 Conclusion des trois tests

Il apparait clairement que l'algorithme de Floyd complet produit des temps d'exécution catastrophiques par rapport à celui de Dijkstra optimisé. Cela apparaissait déjà pour les deux tests sur un réseau de petite taille. C'est devenu une évidence avec le troisième test utilisant le réseau de taille moyenne. A fortiori, le test avec un réseau de grande taille comme celui de la Belgique entière produirait des temps d'exécution encore plus gigantesques. Nous avons donc trouvé inutile de réaliser un quatrième test sur un grand réseau.

Par contre, si l'on compare entre eux l'algorithme de Dijkstra optimisé avec les arbres de Fibonacci et l'algorithme de Floyd sans le calcul de la matrice des plus courts chemins, nous constatons que, même si le premier est plus performant, il y a une similitude dans les ordres de grandeur des temps d'exécution. Ce résultat nous interpelle car il nous semblait plus logique que ce soit le contraire, étant donné que le deuxième ne doit plus calculer aucune valeur mais seulement les rechercher dans une ligne de la matrice (contrairement à l'autre). Cette différence entre les résultats empiriques et nos prévisions basées sur la théorie nous ont interpellé. Nous avons trouvé adéquat d'approfondir nos investigations en analysant pourquoi l'algorithme de Floyd sans matrice de plus courts chemins ne donnait pas de meilleurs résultats. Cette analyse fera l'objet du chapitre suivant.

# Chapitre 5

# Analyse de l'algorithme de Floyd

Comme annoncé dans le chapitre précédent, le temps de calcul de la matrice des plus courts chemins par l'algorithme de Floyd que nous avons implémenté était beaucoup trop important. Nous avons donc décidé de clôturer nos réflexions sur ce sujet pour nous concentrer sur la partie du chargement en mémoire et du parcours de la matrice des plus courts chemins. Nous justifions ce choix par le fait que le réseau grâce auquel VirtualBelgium produit ses simulations est un réseau routier. Cette structure est, dans la vie courante, assez statique. Nous pouvons donc nous passer des mises à jour à son sujet entre plusieurs simulations. Cette hypothèse permet donc ainsi de laisser le temps nécessaire à la construction de la matrice des plus courts chemins hors du temps consacré aux simulations.

Ce chapitre a donc comme but d'analyser l'algorithme de Floyd sans le calcul de la matrice des plus courts chemins afin de comprendre les causes éventuelles de la longueur interpellante des temps de calcul. Il tentera aussi, à la lumière de cette analyse, de remédier à ce défaut.

## 5.1 Recherche du passage plus lent dans l'algorithme

Nous avons repris le réseau utilisé pour le troisième test du chapitre précédent. Nous avons décomposé l'exécution de la simulation en plusieurs étapes spécifiques à une tâche pour pouvoir calculer leur temps d'exécution respectif. Nous espérons découvrir de cette manière l'étape de l'algorithme qui coute le plus en temps d'exécution. Les quatre étapes que nous avons isolées sont :

1. Calcul préliminaire permettant de lire les fichiers de données, de construire les lois de distribution des générateurs de nombres aléatoires et de générer la population d'agents.
2. Chargement de la matrice des plus courts chemins, calculée précédemment et stockée dans un fichier.
3. Calcul des chaines d'activités pour chaque individu de la population par les parcours de lignes de la matrice des plus courts chemins.
4. Sauvegarde des activités et de la population, et fin du programme.

Après avoir lancé 3 simulations successives, les temps d'exécution, en secondes, pour chacune de ces étapes sont :

| étape 1 | étape 2 | étape 3 | étape 4 |
|---|---|---|---|
| 1 | 7 | 589 | 2 |
| 2 | 8 | 561 | 2 |
| 1 | 8 | 545 | 3 |

Il apparait clairement que la partie la plus longue de cette procédure est la troisième. C'est donc cette troisième partie qu'il faut analyser en profondeur si nous voulons trouver des pistes d'amélioration.

Rappelons comment se déroule cette troisième étape. Pour chaque individu, il faut choisir des nœuds représentant les lieux de ces différentes activités. Le choix de chaque nœud se fait en tirant aléatoirement parmi un échantillon de nœuds se trouvant assez proches de celui de départ de l'individu. Cet échantillon est construit à partir de la bonne ligne de la matrice des plus courts chemins chargée préalablement. Ce sont précisément les parcours de ces lignes dans la matrice qui caractérisent et différencient, à cette étape, l'algorithme de Floyd et celui de Dijkstra. C'est donc bien cet endroit de la simulation qui prend du temps, et qui nécessite une amélioration si on veut le rendre plus performant que l'algorithme actuel.

## 5.2 Première piste : le tri de la ligne

La première piste d'amélioration se base sur une des propriétés de l'algorithme de Dijkstra : la possibilité de ne pas devoir parcourir l'ensemble des nœuds (voir tableau 4.3 dans le chapitre 4). Partant de cette idée, il serait opportun que les éléments de la ligne de la matrice soit parcourues de manière ordonnée. Cela impliquerait en effet de pouvoir arrêter l'algorithme avant la fin, dès qu'il détecte un nœud trop éloigné de celui de départ. Supposons que ce nœud trop éloigné soit en position $n'$ dans la ligne triée, où $n' \leq n$.

Cela nécessite de trier préalablement les éléments de la ligne de la matrice, ce qui est techniquement faisable mais demande évidemment un cout en terme de temps de calcul. Comparons donc les complexités dans les deux cas. Pour une matrice de taille $n \times n$, un tri quicksort étant d'ordre $\mathcal{O}(n \log n)$, cette alternative demanderait $n \log n + n'$ opérations pour trier puis parcourir une partie de la ligne. En revanche, sans le tri, le parcours complet demande $n$ opérations. À première vue, trier la ligne pourrait paraitre trop long. Cette piste mérite cependant, à notre sens, d'être explorée pour les cas de simulations de chaines d'activités où un grand nombre d'individus partent d'un même nœud. En effet, dans ce cas, le tri effectué serait réutilisé. Ainsi, le coût excessif induit par ce tri serait en quelque sorte réparti sur ce nombre d'individus.

Partant de ce principe, il faudrait donc dans l'exécution d'une simulation de VirtualBelgium, avant tout calcul de chaine d'activités, associer aux nœuds du réseau le nombre d'individus y résidant. Cette information permettrait de déterminer un

seuil d'individus résidant dans un nœud à partir duquel la ligne de la matrice associée devrait être triée.

## 5.3 Deuxième piste : parallélisation de la recherche dans la matrice

La deuxième piste d'amélioration se base sur l'avantage de la programmation parallèle. Dans le troisième test réalisé à la section 4.6, la recherche dans une ligne de la matrice des plus courts chemins ne mobilisait qu'un seul processeur. Partager ce travail entre plusieurs processeurs disponibles devrait avoir un impact positif sur le temps d'exécution. La question est de savoir si ce partage est techniquement faisable. Une première approche de cette procédure de parallélisation serait de décomposer la recherche dans une ligne de la matrice en trois étapes :

1. Diviser la ligne de la matrice en autant de parts qu'il n'y a de processeurs disponibles
2. Chaque processeur se voit attribuer une de ces parties et y recherche les distances des nœuds inférieures à celle tirée aléatoirement pour le placement de l'activité. Lorsqu'une de ces distances est trouvée, le nœud associé est isolé.
3. Un des processeurs, après cette phase de recherche, centralise tous les nœuds candidats trouvés. Il tire alors, aléatoirement suivant une loi uniforme, dans cet ensemble le nœud du réseau qui accueillera l'activité.

Pour pouvoir mettre en place ces étapes, cette piste semble exiger des changements plus importants dans la structure de base du programme de VirtualBelgium.

## 5.4 Troisième piste : chargement de la matrice des plus courts chemins par "morceaux"

### 5.4.1 Principe

L'idée de cette troisième piste est de modifier la structure de stockage de la matrice des plus courts chemins. Au lieu de travailler avec une matrice de taille $n \times n$, on stockerait les informations dans deux fichiers différents organisés comme suit :

- le premier fichier contiendrait, pour chaque nœud, la liste des nœuds du réseau lui étant proches, et les distances associées
- le deuxième fichier contiendrait, pour chaque nœud, la liste des nœuds du réseau lui étant éloignés, et les distances associées

Grâce à ces deux fichiers, le choix d'un nœud pour l'activité considérée se ferait en parcourant d'abord le fichier des nœuds proches. Si on y découvre un nœud trop éloigné, nous aurions alors l'assurance qu'il est inutile de perdre du temps à parcourir le fichier des nœuds éloignés. En revanche, si tous les nœuds du premier fichier sont assez proches, il faudrait compléter cet échantillon de nœuds en parcourant le

second fichier.

Ainsi, si nous comparons le temps d'exécution d'une recherche avec celui de la méthode actuelle, on aurait soit une réduction dans le cas où seule la lecture du premier fichier suffit, soit une similitude dans le cas où la lecture du deuxième fichier est nécessaire.

### 5.4.2 Paramètres

Il y a un nombre important de paramètres à prendre en compte dans cette troisième piste. Nous en détaillons quelques-uns mais cette liste n'est pas exhaustive.

#### Nombre de fichiers utilisés

Dans notre explication, nous avons séparé la matrice en deux fichiers. Or, nous pouvons aussi envisager de la séparer en plus de deux parties et donc utiliser plus de deux fichiers. Ainsi, en partitionnant davantage les données, on optimise le gain de temps pour tous les cas où les premiers fichiers suffisent au choix du nœud accueillant l'activité.

#### Détermination du critère de séparation

On peut décider de couper l'ensemble des nœuds en parties de tailles égales ou en fonction des valeurs des plus courts chemins entre le nœud de départ et les autres nœuds du réseau. Dans ce cas, nous devons choisir un seuil pour partager nos deux fichiers. Remarquons que lors du choix d'un seuil, on observe la caractéristique suivante : si la distance tirée pour l'activité est supérieure au seuil choisi, alors, pour un nœud précis, tous les nœuds stockés dans le premier fichier ont leur distance par rapport à ce nœud précis inférieure à la distance tirée. En effet, soit $S$ le seuil, $d_m$ la distance entre le nœud $m$ et le nœud source et $d$ la distance tirée ; on a l'implication suivante :

$$d_m < S < d \Rightarrow d_m < d$$

De là, on déduit qu'il n'existe aucun $m$ tel que $d_m > d$. Cela nous donne l'information que le parcours du deuxième fichier sera obligatoire.

Ce rapide raisonnement montre l'intérêt de choisir un seuil assez grand pour que la distance tirée aléatoirement lui soit le plus souvent possible inférieure. Mais nous devons aussi tenir compte du fait que plus le seuil est élevé, plus le nombre de nœuds dans le premier fichier est lui aussi élevé. Cela diminue donc l'intérêt de ne parcourir que le premier fichier.

Par conséquent, nous devons trouver un seuil *compromis* qui assemble au mieux ces deux contraintes contradictoires.

Par exemple, dans le cas d'une décomposition en deux fichiers, le seuil à partir duquel on effectue le classement du nœud dans le fichier un ou deux pourrait être

choisi comme la médiane de la distribution depuis laquelle les distances des activités sont tirées, majorée d'une valeur $\epsilon$. En effet, cette distribution est celle d'une loi de mixture de log-normales comme le montre l'auteur de la thèse [BJ14] grâce, entre autres, à la figure 5.1 où on y voit la répartition empirique des données.

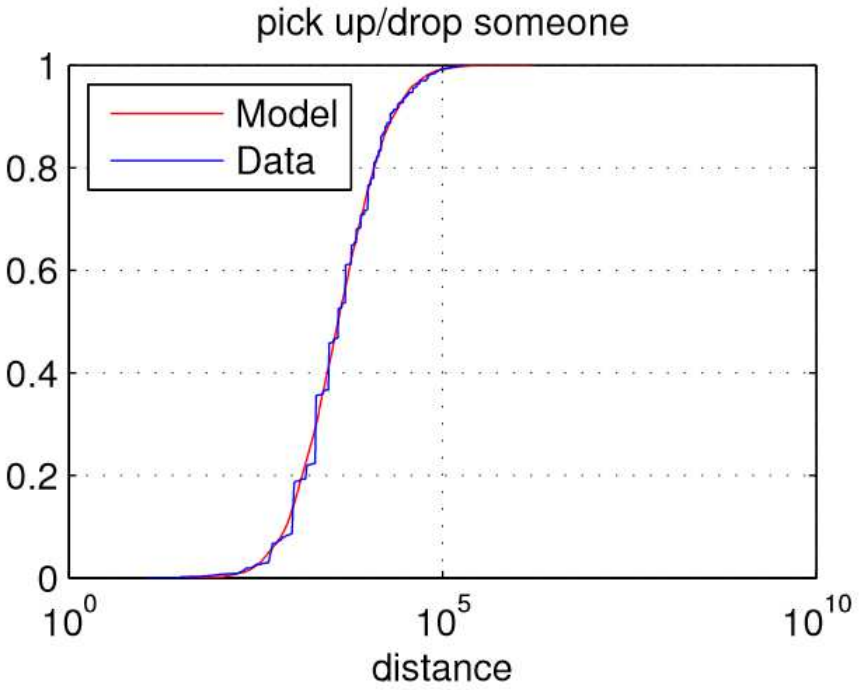



FIGURE 5.1 – Distribution générant les distances des activités

En examinant plus précisément les taux d'accroissement des fonctions de répartition des lois log-normales, on constate que l'on peut décomposer le domaine en trois zones. Dans la première, la pente de la fonction est assez faible, dans la deuxième, elle est plus importante, et elle redevient assez faible dans la troisième. De plus, on constate aussi que la valeur médiane est située dans la zone de plus forte pente.

En exploitant ces particularités, nous pouvons alors prendre comme seuil comme la valeur médiane de la distribution majorée d'un paramètre $\epsilon$. Ainsi, nous sommes garanti que les distances tirées soient inférieures au seuil dans plus de 50% des cas. La difficulté consiste à choisir judicieusement la valeur $\epsilon$ afin de trouver un compromis entre un pourcentage élevé de distances tirées inférieures au seuil et une taille du premier fichier raisonnable.

### Uniformisation des critères de séparation

Nous pouvons aussi nous demander s'il faut différencier le critère de séparation des nœuds dans les fichiers suivant le nœud de départ de l'activité. Par exemple, si nous considérons le cas de deux nœuds du réseau ayant des caractéristiques très

différentes (nombre de liens, éloignement par rapport aux autres nœuds du réseau, etc.) nous pourrions décider de séparer, pour un nœud, la liste des distances en parts de tailles égales et trouver plus judicieux pour le deuxième de suivre plutôt le critère de décomposition par rapport à un seuil.

### 5.4.3 Synthèse

Vu le nombre important de paramètres intervenant dans cette piste, il nous semble très difficile de se rendre compte exactement de son intérêt sans une étude approfondie.

# Conclusion

La première partie de la conclusion résume les résultats finaux. La suivante énonce les possibles perspectives pouvant être considérées partant de ceux-ci.

## Conclusion générale du mémoire

Après les tests de comparaison entre les algorithmes, détaillés au chapitre 4, nous arrivons aux conclusions suivantes :

- La structure du réseau n'influence pas le temps d'exécution d'une simulation.
- La comparaison entre l'algorithme de Dijkstra optimisé par les arbres de Fibonacci et l'algorithme de Floyd complet est très éloquente : l'algorithme de Floyd proposé comme alternative a un temps d'exécution beaucoup trop important, et ne constitue donc pas une amélioration.
- Par contre, la comparaison entre Dijkstra optimisé par les arbres de Fibonacci et l'algortihme de Floyd sans le calcul de la matrice des plus courts chemins semble plus prometteuse. En effet, l'équivalence relative des temps d'exécution laisse entrevoir des perspectives intéressantes pour trouver une amélioration sérieuse au projet VirtualBelgium.
- Ayant cerné de manière plus précise le passage dans l'algorithme de Floyd nécessitant le plus grand temps d'exécution, nous avons ensuite proposé trois pistes d'amélioration détaillées au chapitre 5.

Rappelons malgré tout que la version de Dijkstra utilisée par VirtualBelgium a été longuement travaillée et optimisée, notamment à l'aide des arbres de Fibonacci. En revanche, l'algorithme de Floyd implémenté pour ce travail est assez simple et n'a évidemment pas fait l'objet d'une réflexion aussi intense.

Pour finir, j'aimerais préciser que ce mémoire a été pour moi un long parcours. Le travail à fournir a été parfois assez complexe. Certains moments ont été déroutants voire même décourageants mais, d'un autre côté, très instructifs. En effet, ce mémoire m'a appris à utiliser des techniques de programmations complexes comme la programmation parallèle avec communication, à prendre possession d'un programme déjà codé par une autre personne et enfin à utiliser des outils nouveaux comme la librairie C++ MPI. En outre, le déroulement du travail m'a aussi donné l'occasion d'apprendre à réagir par rapport à des résultats inattendus, ce qui est très enrichissant sur le plan de mon évolution personnelle.

# Perspectives

Éclairé par les résultats de ce travail, nous pouvons envisager les perspectives suivantes pouvant être mises en œuvre.

Premièrement, nous pouvons travailler sur l'implémentation des trois pistes d'amélioration détaillées au chapitre 5. Ces trois pistes nous paraissent clairement les plus intéressantes à approfondir et semblent assez prometteuses.

Deuxièmement, nous avons décidé de nous passer de la construction de la matrice des plus courts chemins car cette opération demandait trop de temps d'exécution. L'optimisation de cette opération peut être prise en considération pour continuer dans une autre direction les recherches initiées avec ce travail.

Enfin, les idées suivantes pourraient aussi constituer des pistes de recherches sur le sujet. Il nous est cependant moins facile d'évaluer leur impact, étant donné qu'elles se basent sur des parties des algorithmes de calcul des plus courts chemins qui n'ont été que peu ou pas prises en compte durant ce mémoire. Il s'agit de perspectives s'appuyant sur d'autres considérations pour répondre à la problématique de départ. Ces pistes sont :

- Modifier le nombre d'individus dans la population synthétique. Nous avons vu que le nombre de fois qu'un nœud doit être choisi pour accueillir une activité était plus élevé si le nombre d'individu augmentait. Cela pourrait jouer un rôle en fonction de la méthode d'obtention des plus courts chemins utilisée.
- Modifier le nombre d'activités par individu. Les conséquences sont similaires au point précédent.
- Grouper les nœuds du réseau par entité géographique et relier ces entités par un nombre restreint de liens. Ce qui pourrait peut-être avoir un impact sur le temps d'exécution.
- Fixer le nœud d'arrivée de toute activité pour chaque individu et ainsi éviter une grande partie des calculs de plus courts chemins et donc limiter l'étape la plus longue d'une simulation.
- Assigner aux nœuds du réseau un nombre réduit de rôles. Cela permettrait par la suite de pouvoir les filtrer en fonction d'un rôle et ainsi de limiter les recherches sur le réseau.
- Repenser le système de stockage de la matrice des plus courts chemins pour limiter sa taille dans la mémoire de l'ordinateur.

# Table des figures

# Bibliographie


[AGK03] George Karypis Ananth Grama, Anshul Gupta and Vipin Kumar. Introduction to Parallel Computing. Pearson, 2003.

[BJ10] Toint Ph. Barthelemy J. Synthetic population generation in presence of data inconsistencies. 23 December 2010.

[BJ14] Cornelis E. (promoteur) Barthélemy J., Toint Ph (promoteur). A parallelized micro-simulation platform for population and mobility behaviour : application to Belgium. Unamur : Faculté des sciences, Namur, Presses universitaires de Namur, 2014.

[def] Futura-sciences : High-tech, consulté le 17/04/2014. http ://www.futura-sciences.com/magazines/high-tech/.

[Del09] Jean-Charles Delvenne. Math appliquées : Théorie des graphes. Bac 2 Facultés des sciences, Département de mathématique, 2009.

[doc] Documentation du réseau au format xml de virtualbelgium, consulté le 24/04/2014. http ://www.matsim.org/files/dtd/network_v1.dtd.

[Ede05] Herbert Edelsbrunner. The design and analysis of algorithms. Technical report, Duke University Department of Computer Science, September 2005.

[E.W59] E.W.Dijkstra. A note of problems in connexion with graphs. 1959.

[HJP02] Toint Ph Hubert J-P. La mobilité quotidienne des Belges. Presses universitaires de Namur, 2002.

[inf] Dico info, consulté le 5/11/2013. http ://dictionnaire.phpmyvisites.net/.

[ins] Notre belgique.be, la belgique, ses communes et ses dépendances, consulté le 25 mai 2014. http ://www.notrebelgique.be/fr.

[Lec11] J.P. Leclercq. Theorie des graphes. Bac 3 Facultés des sciences informatiques, 2011.

[MPI] Documentation mpi. Argonne National Laboratory.

[mul] Agents et systèmes mutliagents, consulté le 5/11/2013. http ://www.damas.ift.ulaval.ca/ coursMAS/ComplementsH10/Agents-SMA.pdf.

[nxP] Documentation du module networkx, consulté le 24/04/2014. http ://networkx.github.io.

[ope] Openstreetmap, la carte coopérative libre, consulté le 13/04/2013. http ://www.openstreetmap.org/.

[PP12] Perry P.O. and Wolfe P.J. Null models for network data. ArXiv e-prints, 2012.

[Tar83] Robert Endre Tarjan. Data Structures And Network Algorithms. Society for industrial and applied mathematics, 1983.

[Way07] Kevin Wayne. Theory of algorithms. Princeton University, 2007.

[wik] Wikipédia : l'encyclopédie libre, consulté le 5/11/2013. http ://fr.wikipedia.org/wiki/Wikip

[xml] Documentation du module xml minidom, consulté le 24/04/2014. https ://docs.python.org/2/library/xml.dom.minidom.html.

# Annexes

## Code de l'algorithme de Floyd parallélisé

```
// Compute the Matrix OD by floyd algorithm
void ComputeMatrixOD(mpi::communicator world, std::vector<float> A, int n
  cout << "Computing matrix OD ..." << endl;

  // Number of ele load in a processor
  int size = (int)matSize % sqrt(world.size);

  for (int k = 0; k < matSize; ++k)
  {
    // Processor position in the grid (rows and columns)
    int rankR = world.rank() \ sqrt(world.size());
    int rankC = world.rank() % sqrt(world.size());

    // Coordinates of the senders
    int coord = k / (size);

    // Test if the processor contain the k-th row or column
    bool ownerR = (rankR == coord);
    bool ownerC = (rankC == coord);

    // Share the information if owner
    if ownerR
    {
      int klocal = k % size;

      // Create data to send
      std::vector<float> kthRow(size);
      int ii = rankR * size + klocal;
      for (int j = 0; j < size; j++)
      {
        kthRow[j] = A[ii,rankC * size + j];
      }

      // Loop on every processor of the column to send kthRow
      for (int i = 0; i < sqrt(world.size()); ++i)
      {
        // To not send to itself
        if i == rankC
         continue;

        int rankToSend = i * sqrt(world.size() + rankC
        mpi::send(rankToSend,k,kthRow);
      }
    }
```

```
if ownerC
{
  int klocal = k % size;

  // Create data to send
  std::vector<float> kthCol(size);
  int jj = rankC * size + klocal;
  for (int i = 0; i < size; ++i)
  {
    kthCol[i] = A[rankR * size + i,jj];
  }
  // Loop on every processor to send kthCol
  for (int i = 0; i < sqrt(world.size()); ++i)
  {
    // To not send to itself
    if i == rankR
     continue;

    int rankToSend = rankC * sqrt(world.size()) + i
    mpi::send(rankToSend,k,kthRow);
  }
}

// Receive information if not owner
if !ownerR
{
  // Rank of the sender
  int rankToReceiv = rankR * sqrt(world.size() + coord;

  // Receive info
  std::vector<float> kthRow(size);
  world.recv(rankToReceiv,k,kthRow);
}

if !ownerC
{
  // Rank of the sender
  int rankToReceiv = coord * sqrt(world.size() + ownerC;

  // Receive info
  std::vector<float> kthCol(size);
  world.recv(rankToReceiv,k,kthCol);
}

//gather data to compute
```

```
    for (int i = size * rankR; i < size * (rankR + 1); ++i)
    {
      for (int j = size * rankC; j < size * (rankC + 1); ++j)
      {
        if (i == j || i == k || j == k )
          continue;

        if (kthCol[i - size * rankR] + kthRow[j - size * rankC] < A[i * r
          A[i * matSize + j] = kthCol[i - size * rankR] + kthRow[j - size
      }
    }
  }
  cout << "Saving Matrix" << endl;
  // Saving in file
  ofstream saveFile("../data/MatrixOD.dat");
  if (saveFile){
    // save matrix size
    saveFile << size  << endl;

    //save entire matrix
    for (int i = 0; i < size; ++i)
    {
      for (int j = 0; j < size; ++j)
      {
         saveFile << A[i * size + j] << " ";
      }
      saveFile << "\n";
    }
  }
  cout << "done saving" << endl;
  saveFile.close();
}
```

## Code de création des fichiers null model

```
from xml.dom import minidom
import networkx as nx
import random

# Parsing du fichier XML
tree = minidom.parse('chemin_vers_le_reseau_XML')
root = tree.documentElement
print len(root.childNodes)

# Creation du graph
```

```
G = nx.DiGraph()

# lecture de tous les noeuds
for element in root.getElementsByTagName('node'):
    id = element.getAttribute('id')
    x = element.getAttribute('x')
    y = element.getAttribute('y')

    # Ajout au reseau
    G.add_node(id,{"id" : id,"x" : x,"y" : y})

# lecture de tous les liens
for element in root.getElementsByTagName('link'):
    id = element.getAttribute('id')
    depart = element.getAttribute('from')
    arrive = element.getAttribute('to')
    length = element.getAttribute('length')

    # Ajout au reseau
    G.add_edge(depart,arrive,{"id" : id,"length" : length})

# Suppression de certains noeuds
for node in G.nodes():
    if (random.random() < 0.99):
        G.remove_node(node)

# Inversion des liens
liens = G.edges()
for i in range(len(liens)):
    l1 = random.choice(liens)
    l2 = random.choice(liens)
    if l1 == l2:
        continue
    G.remove_edge(l1)
    G.remove_edge(l2)
    G.add_edge(l1[0],l2[1])
    G.add_edge(l2[0],l1[1])
    liens = G.edges()

# creation du nouveau document XML
newDoc = minidom.Document()
network = newDoc.createElement('network')
newDoc.appendChild(network)

# Ajout des arbres "nodes" et "links"
nodes = newDoc.createElement('nodes')
```

```
links = newDoc.createElement('links')
network.appendChild(nodes)
network.appendChild(links)

# Ajout des noeuds
for vertice in G.nodes(data=True):
    node = newDoc.createElement('node')
    node.setAttribute('id', vertice[0])
    node.setAttribute('x', vertice[1]['x'])
    node.setAttribute('y', vertice[1]['y'])
    nodes.appendChild(node)

# Ajout des liens
for edge in G.edges(data=True):
    link = newDoc.createElement('link')
    link.setAttribute('aid', edge[2]['id'])
    link.setAttribute('from', edge[0])
    link.setAttribute('to', edge[1])
    link.setAttribute('zlength', edge[2]["length"])
    links.appendChild(link)

# Maj INS
restant = [n[0] for n in G.nodes(data=True)]
insFile = open('chemin_vers_les_codes_INS','r')
outIns = open('chemin_vers_nouveau_fichier_codes_INS','w')
for line in insFile:
    if line.split(";")[0] in restant:
        outIns.write(line)

insFile.close()
outIns.close()

# Enregistrement du reseau
fileOut = open('nullModel.xml','w')
fileOut.write(newDoc.toprettyxml())
fileOut.close()
```

# Installation de VirtualBelgium

La section suivante, écrite en collaboration avec J. Barthelemy, décrit le processus d'installation du programme VirtualBelgium sur un ordinateur personnel. Plus précisément, ce document explique comment utiliser et installer VirtualBelgium sur une architecture GNU/linux 64 bits. Pour commencer, nous détaillerons les fichiers et librairies à télécharger. Ensuite, nous décrirons les commandes d'installation à entrer et enfin celles pour lancer le programme en lui-même.

# Téléchargements

Avant toute chose, il est évident que nous devons commencer par télécharger une distribution Linux. Toutes les commandes décrites dans les lignes suivantes ont été testées sur la distribution Linux Mint 15.

Les fichiers du code en lui-même sont hébergés à l'adresse suivante :
`http://sourceforge.net/projects/virtualbelgium/files/`
et sont organisés de la manière suivante :

| | |
|---|---|
| **./** | dossier racine, contient les scripts d'exécution ainsi que le "Makefile" |
| **./bin** | fichiers d'exécution et de configuration |
| **./data** | données d'entrées |
| **./doc** | documentation |
| **./include** | fichiers d'en-tête |
| **./licenses** | licenses de Repast HPC, tinyxml2 et VirtualBelgium |
| **./logs** | fichiers log des simulations |
| **./outputs** | sorties générées par les simulations |
| **./scripts** | scripts pour le traitement des sorties |
| **./src** | fichiers sources et le "Makefile" |

Listons à présent les différents outils obligatoires à la compilation et/ou exécution du programme.

### Compilateur C++

Le projet étant codé en C++, un compilateur de ce langage est nécessaire. Le compilateur le plus connu est g++. Il est disponible dans le gestionnaire de dépôts grâce à la commande

apt-get install g++

### Outil Make

Cet outil permet de simplifier les commandes de compilation lorsque le programme comporte beaucoup de fichiers. Il s'occupe des dépendances et automatise certains processus.

apt-get install make

La commande Make exécute les règles définies par le fichier Makefile.

### L'environnement MPI

Le Message Passing Interface (MPI) est un standard permettant à différents processus d'une même exécution du programme de communiquer ensemble. Un utilitaire bien connu est l'exécutable OpenMPI, disponible à nouveau dans le gestionnaire de dépôt par la commande :

apt-get install libopenmpi-dev openmpi-common

### La librairie Repast HPC

La librairie principale utilisée par notre programme est le framework Repast HPC 1.0.1. Elle permet de modéliser au mieux un système basé sur le c++ et utilisant un grand ensemble d'unités de calcul. À l'origine, elle a été créée par le Laboratoire National d'Argonne. Le code compilable est disponible à l'adresse suivante : `http://repast.sourceforge.net`. Cette librairie en nécessite d'autres avant d'être compilée :

- Curl ;
- Boost 1.48 (ou supérieure) : en plus de cette librairie générale, nous avons besoin des spécifiques suivantes :boost-mpi, boost-system, boost-serialization and boost-filesystem ;
- NetCDF 4.2.1 ;
- NetCDF C++ 4.2.

Les commandes de téléchargement sont simplement :

**Pour Curl :**

apt-get install libcurl-dev

**Pour Boost :**

apt-get install libboost-dev libboost-mpi-dev libboost-serialization-dev libboost-system-dev libboost-filesystem-dev

**Pour NetCDF :**

apt-get install libnetcdf-dev

# Repast HPC

### Compilation de Repast HPC

Lorsque tout est téléchargé et extrait de l'archive, la compilation s'effectue grâce aux trois commandes suivantes : (attention à utiliser les privilèges super-utilisateur).

1. ./configure
2. ./make
3. ./make install

Il se peut que la commande "make" génère une erreur. Pour y remédier, il faut remplacer toutes les occurrences de

```
getItems(...)
```

par

```
this->getItems(...)
```

dans les documents

./src/repast_hpc/DirectedVertex.h et ./src/repast_hpc/UndirectedVertex.h

Si une erreur se présente à nouveau lors de l'exécution du "make", le Makefile doit être édité de telle façon à supprimer toutes les références vers les modèles Zombie et Rumor.

# Compilation et exécution de VirtualBelgium

## Ordinateur personnel

Pour compiler simplement le programme sur un ordinateur personnel, il suffit d'exécuter la commande

make

Après la compilation, il faut lancer l'exécutable. Un script a été créé spécialement, il s'agit de la commande :

./run.sh NP

où NP est le nombre de processeurs désirés.

## Exécution de calculs intensifs

Pour des simulations mettant en scène un nombre important d'agents, il est recommandé d'effectuer les opérations sur un système de plusieurs ordinateurs, "cluster". Suivant les appareils utilisés, il est possible que le fichier "Makefile" doive être modifié. Si le cluster utilisé est Lemaitre du Consortium des Equipements de Calcul Intensif(`http://www.ceci-hpc.be/`), la compilation se fait simplement avec la commande

make ucl

Ensuite, l'exécution se lance avec le script run_lemaitre2.sh

sbatch run_lemaitre2.sh

Ce script permet aussi de personnaliser l'exécution en proposant à l'utilisateur les options suivantes :

- mail-user : une adresse e-mail pour les notifications ;
- time : le temps d'exécution demandé ;
- ntask : le nombre de processeurs voulus ;
- mem-per-cpu : la mémoire réquisitionnée par processeur.

### Debuggage

Si le code est modifié pour un ajout de fonctionnalité ou pour une optimisation, il est préférable de pouvoir le débugger si une erreur survient lors de la compilation ou l'exécution. À nouveau la commande "make" va nous être utile en lui ajoutant l'option "debug"

make debug

Cette commande fait appel au débugger GNU gdb dont la documentation peut être trouvée à `http://www.sourceware.org/gdb/documentation/`.

## Configuration de VirtualBelgium

VirtualBelgium est un programme de simulation par agents contenant plusieurs modèles possibles. Le choix du modèle se fait dans le fichier

\bin\model.props

Ce fichier reprend aussi le choix, entre autres, du réseau, de la population synthétique ou encore de la partie à simuler[1].

## Mise à jour du code

Les dernières versions du code de VirtualBelgium sont disponibles sur un répertoire Subversion (SVN) hébergé à l'Université de Namur. Lorsque le logiciel Subversion est installé sur notre machine, les données sont récupérables en trois étapes :

1. Demander un compte à virtualbelgium@math.unamur.be ;
2. Créer un tunnel SSH vers Gauss :

   ssh -N -f -l 5555 :localhost :3690 user@gauss.math.fundp.ac.be

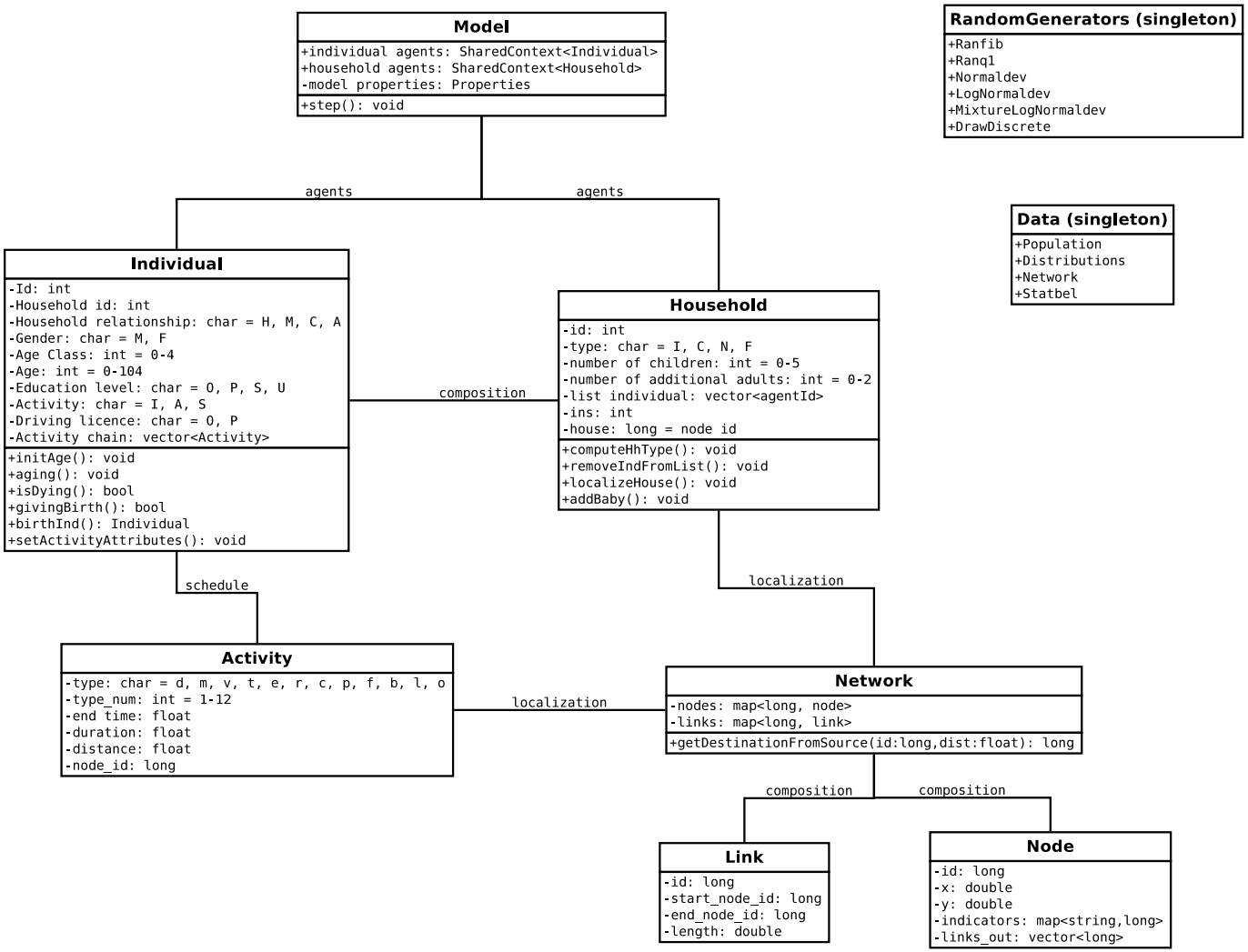
3. Se connecter à

   svn co svn ://user@localhost :5555/virtualbelgium/

   pour recevoir la dernière version du programme

Les différentes commandes de Subversion pour éditer un projet sont listées à `http://svnbook.red-bean.com/index.en.html` (disponible en français).

1. Chaines d'activités, naissances, morts ou âges


Model
+individual agents: SharedContext<Individual>
+household agents: SharedContext<Household>
-model properties: Properties
+step(): void
RandomGenerators (singleton)
+Ranfib
+Ranq1
+Normaldev
+LogNormaldev
+MixtureLogNormaldev
+DrawDiscrete
agents
agents
Data (singleton)
+Population
+Distributions
+Network
+Statbel
Individual
-Id: int
-Household id: int
-Household relationship: char = H, M, C, A
-Gender: char = M, F
-Age Class: int = 0-4
-Age: int = 0-104
-Education level: char = O, P, S, U
-Activity: char = I, A, S
-Driving licence: char = O, P
-Activity chain: vector<Activity>
+initAge(): void
+aging(): void
+isDying(): bool
+givingBirth(): bool
+birthInd(): Individual
+setActivityAttributes(): void
composition
Household
-id: int
-type: char = I, C, N, F
-number of children: int = 0-5
-number of additional adults: int = 0-2
-list individual: vector<agentId>
-ins: int
-house: long = node id
+computeHhType(): void
+removeIndFromList(): void
+localizeHouse(): void
+addBaby(): void
schedule
localization
Activity
-type: char = d, m, v, t, e, r, c, p, f, b, l, o
-type_num: int = 1-12
-end time: float
-duration: float
-distance: float
-node_id: long
localization
Network
-nodes: map<long, node>
-links: map<long, link>
+getDestinationFromSource(id:long,dist:float): long
composition
composition
Link
-id: long
-start_node_id: long
-end_node_id: long
-length: double
Node
-id: long
-x: double
-y: double
-indicators: map<string,long>
-links_out: vector<long>