Modélisation et optimisation de problèmes de synchronisation dans les documents hypermédia

Bruno Bachelet

N° d'ordre: D.U. 1413
EDSPIC: 273

UNIVERSITÉ BLAISE PASCAL - CLERMONT-FERRAND II

ECOLE DOCTORALE
SCIENCES POUR L'INGÉNIEUR DE CLERMONT-FERRAND

THÈSE

présentée par

Bruno BACHELET

pour obtenir le grade de

DOCTEUR D'UNIVERSITÉ

Spécialité: INFORMATIQUE

MODÉLISATION ET OPTIMISATION DE PROBLÈMES DE SYNCHRONISATION DANS LES DOCUMENTS HYPERMÉDIA

Soutenue publiquement le 24 février 2003 devant le jury

Messieurs	Alain QUILLIOT	Président
	Olivier HUDRY	Rapporteur
	Celso C. RIBEIRO	Rapporteur
	Jean-François PUGET	Examinateur
	Eric SANLAVILLE	Examinateur
	Philippe MAHEY	Directeur de thèse

REMERCIEMENTS

Pour commencer, je tiens à exprimer toute ma gratitude à Monsieur Philippe Mahey, pour m'avoir proposé ce sujet de thèse, pour la confiance qu'il m'a accordée et pour la liberté qu'il a su me laisser, ce qui m'a permis d'étudier un domaine qui me tenait à coeur, et que je présente dans la dernière partie de la thèse.

Tous mes remerciements à Messieurs Olivier Hudry et Celso Ribeiro pour avoir accepté de rapporter mon travail. Je tiens à remercier également Messieurs Eric Sanlaville et Jean-François Puget pour avoir examiné ce travail et participé au jury. J'adresse mes remerciements à Monsieur Alain Quilliot qui a accepté de présider ce jury. Les remarques de toutes ces personnes m'ont été précieuses pour améliorer le mémoire.

Je tiens à remercier Messieurs Luiz Fernando Soares et Rogério Rodrigues de la PUC, l'Université Catholique de Rio. J'ai beaucoup apprécié notre collaboration sur les problèmes de synchronisation.

Je souhaite joindre à mes remerciements toutes les personnes de l'ISIMA et du LIMOS, pour leur sympathie, leur disponibilité et leur amitié, contribuant ainsi à un environnement de travail très convivial qui m'a beaucoup apporté.

Je ne voudrais pas oublier l'équipe de thésards de l'ISIMA, sans qui il m'aurait été difficile de terminer cette thèse. Dans le désordre, un grand merci à Christophe Duhamel, Loïc Yon, Fabrice Baray, Antoine Mahul, Christophe de Vaulx, Jonas Koko, et tout particulièrement Mauricio Cardoso de Souza avec qui j'ai partagé un bureau pendant toutes ces années. Je pense également à Bruno Garcia qui m'a beaucoup aidé au début de ma thèse, et qui nous a malheureusement quittés. J'ai beaucoup apprécié la joie de vivre permanente de ces amis, les longues discussions, les soirées, et leur soutien. Ces quelques années sont pour moi inoubliables.

Je ne saurais exprimer tout l'amour et la reconnaissance que j'ai pour ma famille, mes parents et ma soeur, pour le soutien qu'ils m'ont apporté, pour m'avoir donné les moyens depuis le début de réaliser ma passion. Je tenais à leur dire que chaque jour qui passe je réalise un peu plus ma chance et tout ce qu'ils ont fait pour moi. Enfin, une pensée va à mon réveil de secours, Oréade, pour sa ponctualité matinale qui m'a évité bien des retards.

RÉSUMÉ

Les formats actuels de diffusion de documents sur Internet apportent sans conteste de nouvelles possibilités par rapport aux supports traditionnels. Mais les exigences deviennent toujours plus grandes et de nouveaux langages font régulièrement leur apparition pour tenter d'améliorer encore la structure et l'interactivité des documents. Parmi ces langages, certains offrent la possibilité d'animer et synchroniser des composants multimédia. Mais la variété de ces composants (audio, vidéo, texte, image...) font de l'animation un problème compliqué. L'auteur d'un document synchronisé fournit une liste de contraintes temporelles sur les composants de manière à décrire le déroulement de la présentation. Ces composants ont chacun une durée de présentation qui est flexible dans une certaine limite. Tout le problème consiste à trouver un bon ajustement des durées pour que la présentation se déroule au plus proche de ce que souhaite l'auteur tout en évitant les pauses.

Le problème peut se modéliser, après quelques restrictions, comme un problème de tension de coût minimal dans un graphe. Pour le résoudre avec des coûts convexes linéaires par morceaux, nous avons étudié différentes approches (programmation linéaire, mise à conformité - *out-of-kilter*, mise à l'échelle du dual - *cost-scaling*). Nous proposons également une adaptation de la mise à conformité pour des coûts convexes dérivables. Toutes ces méthodes sont comparées sur des aspects théoriques et pratiques, en considérant des graphes quelconques.

Les graphes représentant les contraintes temporelles sont en réalité très structurés et très proches de la classe des graphes appelés *série-parallèles*, et les méthodes élaborées pour une structure de graphe quelconque ne s'avèrent pas toujours très efficaces. Nous proposons une méthode polynômiale, en $O(m^3)$ opérations, plus adaptée pour résoudre le problème sur des graphes série-parallèles, et que nous appelons *agrégation*. Mais ces graphes, bien que très proches de la réalité, restent encore une idéalisation. Nous proposons de mesurer l'aspect série-parallèle d'un graphe en définissant la notion de graphe *presque série-parallèle*, basée sur la décomposition du graphe en composantes série-parallèles. En exploitant l'efficacité de la méthode d'agrégation sur cette décomposition, nous proposons une méthode dite de *reconstruction* permettant de résoudre le problème pour des graphes presque série-parallèles plus efficacement que les méthodes étudiées précédemment.

Lors de cette étude, nous avons développé une bibliothèque de composants réutilisables pour les problèmes de graphes. Nous expliquons en quoi ce type de développement ne peut pas toujours suivre les règles classiques du génie logiciel. Nous montrons comment le paradigme objet peut néanmoins être employé pour la création d'outils efficaces de recherche opérationnelle. Et nous proposons des patrons de conception pour élaborer des composants logiciels (algorithmes et structures de données) génériques, c'est-à-dire indépendants des structures de données qu'ils manipulent et des algorithmes qu'ils emploient, tout en étant fortement extensibles, et cela avec une perte d'efficacité minimale.

Mots-clé: synchronisation hypermédia, graphe temporel, tension de coût minimal, graphe série-parallèle, réutilisabilité logicielle, approche orientée objet, programmation générique.

ABSTRACT

The actual formats used to publish documents on the Internet bring, without discussion, new facilities compared to the paper material. But needs are still growing and new languages appear regularly, attempting to improve the structure and the interactivity of documents. Among these languages, some offer the possibility of animating and synchronizing media components. But the variety of these components (audio, video, text, image...) makes the animation a difficult problem. The author of a synchronized document provides a set of temporal constraints on the components to describe the progress of the presentation. Each of these components has a duration of presentation that is flexible within some boundaries. The problem consists in finding a good adjustment of the durations so the presentation progresses as close as possible to what the author wants with avoiding any pause.

The problem can be modeled, after some restrictions, as a minimal cost tension problem in a graph. To solve it with piecewise convex costs, we studied different methods (linear programming, out-of-kilter, cost-scaling on the dual). We also propose an adaptation of the out-of-kilter for convex differentiable costs. All these methods are compared on theoretical and practical aspects, considering graphs without any peculiar structure.

The graphs representing the temporal constraints are in fact very structured and very close to the class of graphs called *series-parallel*, and the methods designed for any graph appear to be sometimes inefficient. We propose a polynomial method, with $O(m^3)$ operations, more adapted to solve the problem on series-parallel graphs, and that we call *aggregation*. But these graphs, although very close to the real cases, are still an idealization. We propose to measure the series-parallel aspect of a graph by defining the notion of *almost series-parallel* graph, based on the decomposition of the graph into series-parallel components. By using the efficiency of the aggregation method on this decomposition, we propose a method called *reconstruction* that allows to solve the problem for almost series-parallel graphs more efficiently than the methods studied previously.

During this study, we developed a library of reusable components for graph problems. We explain why this kind of development can not always follow the classical rules of software engineering. We show how the object paradigm can nevertheless be used to create efficient tools for operations research. And we propose design patterns to work software components (algorithms and data structures) out that are generic, meaning independent of the data structures they manipulate and of the algorithms they use, still being highly extensible, and all this with a minimal loss of efficiency.

Key-words: hypermedia synchronization, temporal graph, minimal cost tension, series-parallel graph, software reusability, object-oriented approach, generic programming.

TABLE DES MATIÈRES

INTRODUCTION **1**

PARTIE I - SYNCHRONISATION HYPERMÉDIA **5**

Avant-Propos **7**

Chapitre 1 - Les Documents Hypermédia Synchronisés **9**
1.1. Les relations temporelles . . . 10
1.1.1. Informations qualitatives, relations d'Allen . . . 10
1.1.2. Informations quantitatives, intervalles et ensembles de tolérance . . . 12
1.1.3. Informations causales: durées et dates imprévisibles . . . 13
1.2. Problèmes de synchronisation . . . 13
1.2.1. Problèmes de réalisabilité . . . 13
1.2.2. Problèmes d'optimisation . . . 14
1.2.3. Cycle de vie d'un document synchronisé . . . 15
1.3. Méthodes de résolution existantes . . . 16
1.3.1. Cohérence et planification réalisable d'un scénario . . . 16
1.3.2. Planification optimale d'un scénario . . . 17
1.4. Conclusion . . . 17

Chapitre 2 - Les Graphes Temporels **19**
2.1. Introduction aux graphes . . . 19
2.1.1. Notions élémentaires . . . 20
2.1.2. Arbre . . . 22
2.1.3. Représentation des graphes . . . 24
2.1.4. Algorithme de Minty . . . 27
2.1.5. Conclusion . . . 29
2.2. Flot et tension . . . 29
2.2.1. Flot . . . 29
2.2.2. Tension . . . 30
2.3. Modélisation sous forme de graphe temporel . . . 32
2.3.1. Définition d'un graphe temporel . . . 32
2.3.2. Relations temporelles modélisables . . . 33
2.4. Problèmes de tension . . . 34
2.4.1. Temps et tension . . . 34
2.4.2. Problème de la tension compatible . . . 35
2.4.3. Problème de la tension de coût minimal . . . 36
2.5. Quantifier la qualité d'une présentation . . . 36
2.5.1. Pénalité, coûts convexes linéaires par morceaux . . . 36
2.5.2. Plus d'égalité, coûts convexes dérivables . . . 37

2.5.3. Comptabilisation des objets touchés par une déformation 37
2.6. Conclusion 37

PARTIE II - TENSION DE COÛT MINIMAL **39**

Avant-Propos **41**

Chapitre 3 - Tension Compatible **43**
3.1. Recherche d'une tension maximale sur un arc 43
3.1.1. Algorithme basé sur le plus court chemin 43
3.1.2. Algorithme basé sur le cocycle augmentant 45
3.2. Recherche d'une tension compatible 46
3.2.1. Algorithme basé sur le plus court chemin 46
3.2.2. Algorithme basé sur le cocycle augmentant 47
3.2.3. Variante de l'algorithme basé sur le cocycle augmentant 48
3.2.4. Un autre algorithme basé sur le plus court chemin 50
3.3. Conclusion 51

Chapitre 4 - Tension de Coût Minimal **53**
4.1. Coûts linéaires par morceaux et coûts linéaires 53
4.2. Modélisation sous forme de programme linéaire 54
4.2.1. Modèle basé sur la matrice d'incidence 55
4.2.2. Modèle basé sur une base de cycles 55
4.2.3. Conclusion 56
4.3. Conformité et optimalité 57
4.3.1. Coûts linéaires 57
4.3.2. Coûts linéaires par morceaux 58
4.3.3. Coûts dérivables 59
4.4. Méthode de mise à conformité (coûts linéaires par morceaux) 60
4.4.1. Approche directe 61
4.4.2. Avec une mise à l'échelle 65
4.4.3. Conclusion 68
4.5. Méthode de mise à conformité (coûts dérivables) 68
4.5.1. Dérivée approchée 69
4.5.2. Conformité approchée 70
4.5.3. Coloration des arcs 71
4.5.4. Tension optimale 71
4.5.5. Conclusion 72
4.6. Méthode de mise à l'échelle du dual 73
4.6.1. Transformation en un problème de flot 74
4.6.2. Flot optimal, mise à l'échelle des coûts 76
4.6.3. Tension optimale 80
4.6.4. Conclusion 80
4.7. Conclusion 82

Chapitre 5 - Tension dans un Graphe Série-Parallèle **83**
5.1. Série-parallèle . . . 83
5.1.1. Graphe série-parallèle, SP-graphe . . . 84
5.1.2. Arbre binaire de décomposition, SP-arbre . . . 85
5.1.3. Opérateurs de synchronisation série-parallèles . . . 87
5.1.4. Construction d'un SP-arbre . . . 88
5.2. Méthode d'agrégation . . . 95
5.2.1. Fonction de coût minimal . . . 95
5.2.2. Agrégation série . . . 97
5.2.3. Agrégation parallèle . . . 99
5.2.4. Tension optimale . . . 101
5.2.5. Conclusion . . . 102
5.3. Graphes presque série-parallèles . . . 103
5.3.1. Composantes série-parallèles . . . 103
5.3.2. Méthode de reconstruction . . . 104
5.3.3. Méthodes de décomposition . . . 109
5.3.4. Conclusion . . . 113
5.4. Conclusion . . . 114

PARTIE III - COMPOSANTS RÉUTILISABLES POUR LA RECHERCHE OPÉRATIONNELLE **117**

Avant-Propos **119**

Chapitre 6 - La Réutilisabilité Logicielle **121**
6.1. Qualité logicielle . . . 121
6.1.1. Fiabilité: validité et robustesse . . . 122
6.1.2. Extensibilité et maintenabilité . . . 122
6.1.3. Homogénéité: intelligibilité et compatibilité . . . 123
6.1.4. Portabilité . . . 123
6.1.5. Efficacité . . . 124
6.1.6. Conclusion . . . 124
6.2. Réutilisabilité . . . 124
6.2.1. Utilisation ou réutilisation ? . . . 125
6.2.2. Réutilisabilité et qualité . . . 125
6.2.3. Conclusion . . . 128
6.3. Niveaux d'abstraction . . . 128
6.3.1. Analyse . . . 129
6.3.2. Conception . . . 130
6.3.3. Implémentation . . . 130
6.3.4. Cadriciel . . . 131
6.3.5. Conclusion . . . 132
6.4. L'évolution vers les objets . . . 133
6.4.1. Sous-programmes . . . 133
6.4.2. Modules . . . 133
6.4.3. Types abstraits de donnée . . . 134

6.4.4. Objets . . . 135
6.4.5. Conclusion . . . 135
6.5. Conclusion . . . 136

Chapitre 7 - L'Approche Orientée Objet **139**
7.1. Objet et classe d'objet . . . 139
7.1.1. Définition d'un objet . . . 139
7.1.2. Classe d'objet . . . 140
7.1.3. Envoi de message . . . 141
7.1.4. Construction et destruction d'un objet . . . 141
7.2. Héritage . . . 142
7.2.1. Principe . . . 142
7.2.2. Méthode virtuelle . . . 143
7.2.3. Polymorphisme . . . 144
7.2.4. Réutilisabilité . . . 145
7.2.5. Coût de l'encapsulation . . . 146
7.2.6. Coût de la virtualité . . . 146
7.3. Composition, agrégation et association . . . 148
7.3.1. Différences conceptuelles . . . 148
7.3.2. Similitudes d'implémentation . . . 149
7.3.3. Délégation . . . 151
7.4. Patron de composant . . . 152
7.4.1. Patron de fonction, méta-fonction . . . 152
7.4.2. Patron de classe, méta-classe . . . 153
7.4.3. Polymorphisme statique, notion de concept . . . 155
7.4.4. Réutilisabilité . . . 156
7.5. Conclusion . . . 157

Chapitre 8 - Une Expérience de Réutilisabilité pour les Problèmes de Graphes **159**
8.1. Généricité des structures de données . . . 160
8.1.1. Modélisation d'un graphe par héritage . . . 160
8.1.2. Modélisation d'un graphe par patrons . . . 162
8.1.3. Abstraction par les itérateurs . . . 163
8.2. Généricité des algorithmes . . . 168
8.2.1. Abstraction des algorithmes . . . 168
8.2.2. Extension des algorithmes . . . 169
8.2.3. Gestion de données additionnelles . . . 172
8.3. Conception d'une bibliothèque réutilisable . . . 173
8.3.1. Organisation de la bibliothèque . . . 173
8.3.2. Portabilité de la bibliothèque . . . 175
8.3.3. Gestion des erreurs . . . 179
8.3.4. Implémentation de la notion de concept . . . 179
8.3.5. Vers un environnement de développement . . . 180
8.4. Conclusion . . . 181

CONCLUSION **185**

ANNEXE - IMPLÉMENTATION **189**
9.1. Algorithmes . . . 189
9.1.1. Arbre recouvrant . . . 189
9.1.2. Base de cycles . . . 190
9.1.3. Plus court chemin . . . 190
9.1.4. Tri topologique . . . 191
9.2. Génération aléatoire de problèmes . . . 192
9.2.1. Graphes connexes . . . 192
9.2.2. Graphes série-parallèles ou presque . . . 192
9.2.3. Problèmes de flot . . . 193
9.2.4. Problèmes de tension . . . 193
9.3. Structures de données . . . 194
9.4. Résultats numériques . . . 195

BIBLIOGRAPHIE **197**

INDEX **207**

INTRODUCTION

Grâce à la puissance des ordinateurs et au débit des communications actuels, il est possible de combiner différents médias (son, vidéo, texte, images, animations, applets...) pour produire des documents plus sophistiqués, que l'on peut diffuser aussi bien sur support fixe (CD, DVD...) qu'à travers un réseau. Cependant, malgré ces progrès, les documents hypermédia rencontrés notamment sur Internet possèdent une organisation très proche des supports papiers. Ils possèdent en effet une structure logique (décomposition en parties, chapitres, sections...) et une structure spatiale (placement précis des objets à l'écran). En revanche, ils sont très peu organisés au niveau temporel, alors que des documents par exemple éducatifs (encyclopédie, atlas, cours...) nécessitent des scénarios très planifiés prenant en compte des interactions avec l'utilisateur.

Des outils sont élaborés, à l'image de ceux permettant la structuration logique et spatiale d'un document, afin d'assister les auteurs dans la création de documents hypermédia synchronisés. Produire de tels documents consiste avant tout à spécifier les relations temporelles entre les divers objets multimédia, chacun ayant une durée fixée dite *idéale* que l'auteur accepte de modifier éventuellement dans un certain intervalle. Cela signifie qu'un objet multimédia trop long pour satisfaire les contraintes imposées peut être raccourci dans certaines limites. La seconde étape consiste alors à déterminer une planification du scénario qui soit en adéquation avec les contraintes exprimées par l'auteur, autrement dit, il faut trouver la durée de chaque objet multimédia telle qu'il n'y ait pas de temps mort dans la présentation. Au moment de la présentation du document, la planification est suivie au mieux, mais des aléas comme un retard dans le téléchargement des médias, ou bien une action de l'utilisateur, peuvent perturber la planification. Il faut alors un moyen de réajuster en temps réel cette planification.

En adoptant un modèle de graphe pour représenter les relations temporelles entre les objets multimédia, les problèmes de synchronisation adressés précédemment peuvent se traduire par des problèmes de tension dans un graphe: la recherche d'une tension compatible qui reflète la recherche d'une planification possible pour un scénario, et la recherche d'une tension de coût minimal qui modélise la recherche de la meilleure planification possible. La notion de tension dans un graphe est beaucoup moins répandue que celle de flot, peut-être plus intuitive, et permettant de modéliser de nombreux problèmes. Néanmoins, la relation de dualité qu'il existe entre le flot et la tension suggère deux stratégies pour résoudre les problèmes de tension: s'inspirer des algorithmes de flot pour élaborer des méthodes pour les problèmes de tension, ou bien transformer via la dualité un problème de tension en un problème de flot.

Nous proposons plusieurs méthodes de résolution qui tentent de prendre en compte les différentes contraintes pratiques liées à la synchronisation hypermédia. La qualité d'une planification d'un scénario est difficile à mesurer. Parmi les approches proposées, nous avons retenu celle qui attribue à chaque objet une pénalité en fonction de l'écart entre sa durée planifiée et sa durée idéale, ce coût étant naturellement une fonction convexe. Cependant la forme précise de la fonction est une question ouverte, il est nécessaire de proposer une approche suffisamment flexible pour permettre l'expérimentation de différents types de coût convexe (linéaire par morceaux, dérivable). Un autre aspect important concerne la réaction en temps réel des méthodes, il est nécessaire de proposer des méthodes qui puissent réajuster instantanément une planification qui n'est plus possible à la suite d'anomalies (retards sur le réseau, intervention de l'utilisateur...), afin de rendre la planification à nouveau réalisable et même mieux optimale. Ces besoins en flexibilité et en efficacité restreignent forcément les approches que l'on peut envisager. Les algorithmes proposés dans ce mémoire sont polynômiaux ou pseudo-polynômiaux avec une efficacité pratique établie.

Notre objectif étant de proposer des solutions pratiques pour le domaine de l'hypermédia, il nous a semblé important de fournir une implémentation des méthodes que nous proposons. Cela nous a conduit à réfléchir au projet plus ambitieux de développer des composants logiciels réutilisables pour la recherche opérationnelle, et plus spécifiquement pour les problèmes de graphes, l'objectif étant que des personnes connaissant la recherche opérationnelle puissent réutiliser et étendre les composants pour développer leurs propres solutions, et que des personnes peut-être moins averties puissent utiliser les composants comme des "boîtes noires" et les assembler pour obtenir un produit fini.

Mais la recherche opérationnelle, contrairement à d'autres domaines, ne peut pas suivre les règles classiques de développement du génie logiciel centrées sur la modélisation des données plutôt que celle des algorithmes. Outre les enjeux que suscite le développement de composants logiciels réutilisables pour les problèmes de graphes et les différences avec une conception logicielle classique, nous montrons le potentiel (et les concepts tentants à éviter) du paradigme objet pour élaborer des composants de recherche opérationnelle. Nous présentons finalement différentes techniques de conception qui permettent d'aboutir à des composants génériques réutilisables, portables, robustes et efficaces.

Ce mémoire est décomposé en trois parties. La première, formée des deux premiers chapitres, présente le domaine de la synchronisation hypermédia et comment modéliser certains de ses problèmes sous la forme de problèmes de tension dans un graphe. La seconde partie, composée des chapitres 3 à 5, propose plusieurs algorithmes pour résoudre des problèmes de tension. La dernière partie, formée des trois chapitres restants, présente les enjeux, les moyens et les solutions les mieux adaptées à l'élaboration de composants logiciels réutilisables pour la recherche opérationnelle. Voici maintenant un résumé très bref de chacun de ces chapitres.

Le **chapitre 1** présente plus en détails la problématique de la synchronisation hypermédia et les nombreux besoins des concepteurs d'outils de création et de présentation de documents hypermédia, notamment les différents types de relations temporelles à exprimer entre les objets multimédia et les possibilités de mesure de la qualité d'une planification.

Le **chapitre 2** introduit la notion de graphe et plus particulièrement celle de graphe temporel, qui permet de modéliser des contraintes temporelles entre des objets multimédia. Nous montrons que certains problèmes de synchronisation hypermédia peuvent alors être étudiés comme des problèmes de tension, en particulier ceux de la tension compatible et de la tension de coût minimal.

Le **chapitre 3** est consacré au problème de la tension compatible, pour lequel nous étudions deux approches, l'une basée sur une recherche de cocycles augmentants, l'autre sur une recherche de plus courts chemins, les deux méthodes étant fortement polynômiales.

Le **chapitre 4** se penche sur le problème de la tension de coût minimal, considérant tout d'abord des coûts convexes linéaires par morceaux, et ensuite de manière plus générale des coûts convexes dérivables. Pour le premier type de coût, deux approches sont étudiées, l'une basée sur une recherche de cocycles augmentants, elle est pseudo-polynômiale, et l'autre consistant en une transformation du problème en un problème de flot de coût minimal, elle est polynômiale. La première méthode est ensuite adaptée à des coûts convexes dérivables quelconques.

Le **chapitre 5** s'intéresse à une classe particulière de graphes, les graphes série-parallèles, qui représentent des cas idéaux de synchronisation hypermédia. Une méthode fortement polynômiale est alors proposée. Cette méthode est ensuite exploitée pour résoudre des cas plus réalistes modélisés par des graphes presque série-parallèles, la méthode est alors pseudo-polynômiale.

Le **chapitre 6** explique les enjeux de la réutilisabilité logicielle et son rôle dans le développement de logiciels de qualité, les particularités liées à la recherche opérationnelle sont évoquées.

Le **chapitre 7** présente le paradigme objet, l'une des approches les plus évoluées du génie logiciel, en expliquant en détail les forces et les faiblesses des différents concepts face à la réutilisabilité et aussi à l'efficacité qu'ils entraînent pour la recherche opérationnelle.

Le **chapitre 8** présente des techniques de conception pour élaborer des composants génériques, c'est-à-dire indépendants des structures de données qu'ils manipulent et des algorithmes qu'ils emploient, tout en étant suffisamment extensibles pour être réutilisés dans différentes situations. Enfin, nous présentons succinctement la bibliothèque que nous avons développée, en précisant quelques aspects plus pratiques tels que la portabilité, et les extensions que nous envisageons par la suite.

PARTIE I - SYNCHRONISATION HYPERMÉDIA

Cette première partie est une introduction au domaine de l'hypermédia et en particulier aux problèmes de synchronisation rencontrés lors de la conception et de la présentation d'un document hypermédia. Nous proposons de représenter ces documents synchronisés à l'aide de graphes, dont nous introduisons ici certaines notions et propriétés indispensables à notre étude. Nous discutons de la pertinence de cette modélisation et des restrictions qu'elle suggère. Enfin nous montrons que la représentation choisie permet d'aborder les problèmes de synchronisation hypermédia comme des problèmes de tension dans un graphe.

AVANT-PROPOS

Les termes *multimédia* et *hypermédia* sont devenus courants de nos jours. La puissance des ordinateurs personnels et les débits de communication actuels permettent d'imaginer l'échange de données sous des formes très évoluées. Cependant, malgré les possibilités offertes, les documents électroniques que l'on rencontre actuellement sur Internet possèdent une structure linéaire très proche des documents papiers (à l'exception de l'hyperlien qui permet de naviguer de manière très efficace). Il existe des outils pour la création de documents structurés au niveau logique (décomposition en parties, chapitres, sections...), au niveau spatial (placement d'objets multimédia à l'écran), mais pour l'instant très peu au niveau temporel (positionnement dans le temps d'objets multimédia).

A l'image des outils pour assister un auteur dans la structuration logique et spatiale d'un document, certains se sont intéressés à l'élaboration de logiciels pour faciliter la création de documents dynamiques, c'est-à-dire où les objets du document sont synchronisés dans le temps à partir de spécifications évoluées fournies par l'auteur. Cet aspect suscite de nombreux problèmes: comment distribuer les données sur un réseau afin d'obtenir un téléchargement optimal lors de la lecture du document ? comment synchroniser à bas niveau les flux de données afin d'éviter les dégradations de qualité ?... et enfin, comment synchroniser à un plus haut niveau les différents objets multimédia afin de garantir les spécifications de l'auteur du document ?

Ce mémoire se penche tout particulièrement sur la dernière question, la *synchronisation inter-média*, dont la problématique est tout à fait comparable à celle posée pour la création de documents où l'auteur spécifie une structure sous forme de paragraphes, d'images... justifiés, alignés à gauche, centrés dans la page... et un système logiciel se charge de positionner au mieux tous les éléments pour un rendu de la meilleure qualité possible, l'exemple le plus connu étant TeX (cf. [Knut81]). Dans l'aspect temporel, les relations entre les objets sont plutôt du genre: l'objet A doit être présenté avant l'objet B, l'objet A démarre en même temps que l'objet B...

De nombreuses approches sont proposées pour la synchronisation inter-média, aussi bien au niveau de la formalisation même des relations temporelles entre les objets multimédia, que des méthodes permettant la synchronisation de ces objets lors de la présentation du document au spectateur. N'étant pas experts dans ce domaine, nous proposons un premier chapitre qui positionne simplement nos travaux par rapport à toutes les problématiques et les études déjà menées, renvoyant le lecteur à des documents plus complets sur l'aspect même de la synchronisation hypermédia.

Nous nous intéressons tout particulièrement aux problèmes de recherche opérationnelle, et plus précisément d'optimisation, que la synchronisation inter-média suscite. Pour notre étude, nous avons choisi une formalisation dite *par intervalles*, très courante, permettant de représenter les relations temporelles des objets multimédia d'un document. Dans le second chapitre nous proposons une représentation sous forme de graphe de cette formalisation, et discutons des limitations que nous apportons alors dans l'expression des relations temporelles. Nous montrons enfin que la synchronisation inter-média peut être modélisée par des problèmes de tension dans un graphe.

CHAPITRE 1
LES DOCUMENTS HYPERMÉDIA SYNCHRONISÉS

Un *document multimédia* est un document électronique composé de différents médias comme du texte, du son, de la vidéo, des images, des animations, des applets... Lorsqu'il est possible de naviguer entre différents documents multimédia, grâce au *lien hypertexte*, on parle de *documents hypermédia*. Par exemple, les pages HTML que l'on trouve sur Internet et la plupart des CD-ROMs éducatifs constituent des documents hypermédia. Il est vrai cependant que, malgré les évolutions technologiques qui nous permettent notamment le transfert et la décompression quasi en temps réel de données volumineuses comme la vidéo, il n'existe toujours pas de documents qui exploitent pleinement le mélange des différents médias.

Les objets multimédia variés qui constituent un document hypermédia sont certes structurés au niveau logique, i.e. le document est décomposé en parties, chapitres, sections, sous-sections...; et au niveau spatial, en permettant le contrôle du placement des différents objets à l'écran ou dans une page (e.g. [Laya96b]). Mais au niveau temporel, très peu d'outils permettent d'organiser les objets multimédia. Bien qu'une page HTML contienne différents médias animés, ces derniers ne sont pas synchronisés entre eux et n'exploitent que de manière simpliste les capacités vidéo. Dans certains types de document comme les supports pédagogiques, il est intéressant de structurer les différents médias afin d'obtenir une présentation sophistiquée et extrêmement planifiée dans le temps, avec néanmoins une bonne interaction avec le spectateur.

Ces aspects temporels sont regroupés sous le terme de *synchronisation hypermédia*. Comme le souligne [Herm98], cette problématique soulève de nombreux problèmes à tous les niveaux du développement informatique: les technologies des réseaux, les techniques de compression des données, la synchronisation bas niveau des flux de médias (synchronisation d'une vidéo avec sa bande sonore)... Nous ne nous intéresserons ici qu'à la synchronisation entre objets multimédia à partir de spécifications fournies par l'auteur d'un document, synchronisation qualifiée d'*inter-média*. Dans la suite du mémoire, lorsque nous parlerons de synchronisation hypermédia, il sera sous-entendu ce type précis de synchronisation.

Pour concevoir un document hypermédia synchronisé, un auteur commence par collecter ou créer des données sous différentes formes (son, vidéo, texte, images...). Ces données peuvent provenir de différentes sources et n'ont, a priori, pas été conçues dans un but unique et précis de diffusion. Dans un second temps, l'auteur tente d'élaborer un scénario à partir des médias récoltés pour concevoir une présentation sophistiquée où le spectateur peut éventuellement intervenir, simplement pour ralentir, accélérer, stopper... la présentation, mais également pour déclencher des événements comme cliquer sur un bouton pour lancer une nouvelle séquence dans la présentation.

De nombreux travaux se sont penchés sur la manière de spécifier un scénario et les possibilités d'intervention du spectateur, et ont aboutis à plusieurs standards internationaux de multimédia: HyTime [Newc91], MHEG [Boud95], Premo [Herm98], SMIL [Bult98]... [Rous98] explique que les trois premiers standards cités considèrent plutôt la synchronisation bas niveau des flux multimédia. En revanche, SMIL (*Synchronized Multimedia Integration Language*), qui a été retenu comme standard par le WWW Consortium (cf. [W3CW1]) et qui devrait à terme être intégré dans les navigateurs Web, permet de spécifier des synchronisations entre objets multimédia plus sophistiquées.

Plusieurs outils ont également été élaborés pour fournir une assistance à l'auteur lors de la création d'un document hypermédia synchronisé, et pour assurer la synchronisation du document au moment de sa présentation:

CMIFed [vRos93], Firefly [Buch93a], Isis [Kim95], Madeus [Laya97], HyperProp [Soar00]... De la conception de ces outils sont issus différents types de modèle pour représenter les spécifications temporelles. Plusieurs travaux très complets discutent des possibilités de chacun: [Jour98], [Rous99], [PLuq96]. Nous nous intéressons en particulier à l'outil HyperProp, l'un des objectifs de notre étude étant de proposer des méthodes de synchronisation adaptées à ce système.

Dans un premier temps, nous présentons les besoins des auteurs concernant l'aspect temporel des documents hypermédia et plus précisément les différentes relations temporelles entre les objets multimédia qu'ils souhaitent exprimer. Cela nous permet dans le second chapitre de proposer une modélisation sous forme de graphe qui tente de prendre en compte la plupart de ces besoins. Nous exposons ensuite les principaux problèmes de recherche opérationnelle et d'optimisation qui se posent alors, aussi bien au niveau de la conception même des documents hypermédia synchronisés qu'au niveau de la synchronisation de leur présentation. Finalement, nous discutons brièvement des diverses solutions apportées actuellement à ces problèmes.

1.1. Les relations temporelles

Considérons un objet multimédia M. On note s_M l'événement, la date, de début de la présentation de M et e_M l'événement de fin de sa présentation. Une première approche pour définir un scénario consiste à affecter une date de début de présentation à chaque objet multimédia, la date de fin de présentation de chacun découlant de sa durée intrinsèque (pour une vidéo, un son...) ou d'une durée choisie par l'auteur pour les objets intemporels (une image, un texte...). Cette représentation est illustrée par la figure 1.1.

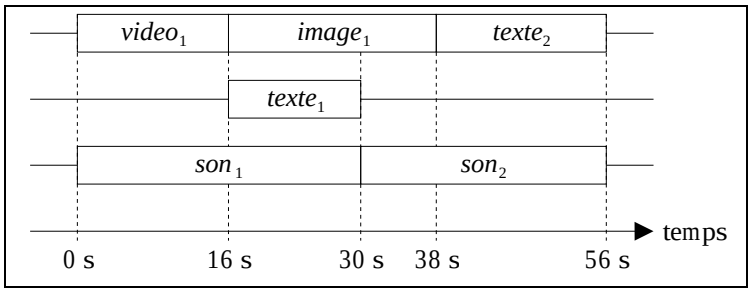


Figure 1.1: Représentation d'un scénario par lignes de temps.

Cette approche comptabilise plusieurs défauts. Tout d'abord, il est très délicat de modifier un tel scénario. Si l'on retire par exemple l'objet $video_1$, il faut revoir la synchronisation de tout le document. En outre, à cause de problèmes sur le réseau, ou même d'objets multimédia dont on ne maîtrise pas la durée (e.g. une requête dans une base de données), il faut être capable d'exprimer une incertitude quant à la durée d'un objet ou sa date de début de présentation. Il a donc fallu proposer une approche qui corrige ces défauts. Elle est issue de plusieurs travaux synthétisés notamment dans [Jour99] qui recommande, pour un système de synchronisation hypermédia, les spécifications que nous présentons maintenant.

1.1.1. Informations qualitatives, relations d'Allen

Il est important de fournir une certaine flexibilité que [Alle83] propose dans une approche descriptive des relations temporelles (totalement indépendamment de l'aspect synchronisation hypermédia à l'époque) entre les différents objets multimédia. Les figures 1.2 et 1.3 représentent les 13 relations qu'il a ainsi établies. Il y en

a en fait 7 fondamentales: *meets*, *before*, *starts*, *finishes*, *equals*, *during*, *overlaps*. Les six autres (*met-by*, *after*, *started-by*, *finished-by*, *contains*, *overlapped-by*) sont simplement les relations inverses. Toutes ces relations, comme le montre la figure 1.2, expriment des contraintes entre les événements de début et de fin des différents médias. Par la suite, nous ferons référence à ces relations par le terme ***relations d'Allen***.

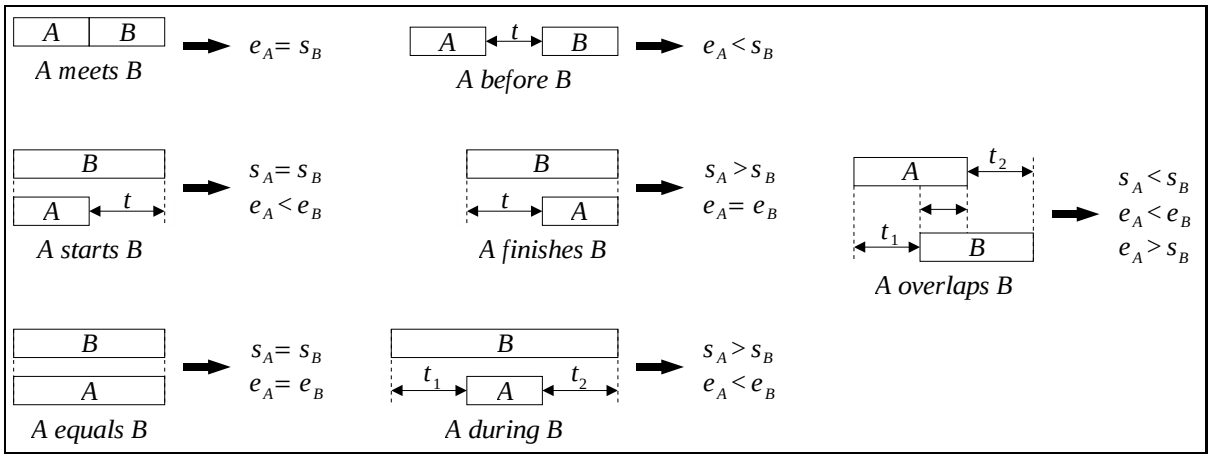


Figure 1.2: Les 7 premières relations d'Allen.

Bien que décrivant déjà un grand nombre de possibilités, ces relations sont insuffisantes pour les besoins de création de documents hypermédia synchronisés, comme le soulignent notamment [Jour99] et [Laya97]. Ils proposent de compléter ces relations avec l'opération de disjonction *or*. En particulier, la possibilité de structurer les objets multimédia sous forme de composites entraîne la nécessité d'exprimer une disjonction entre différentes contraintes (e.g. [Farg98]).

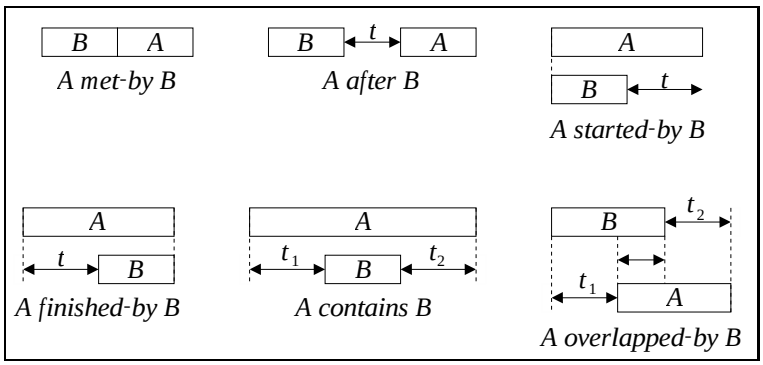


Figure 1.3: Les 6 relations inverses d'Allen.

Prenons l'exemple de la figure 1.4. L'événement s_C du composite C est défini par $s_C = \min_{X \in C}\{s_X\}$, en d'autre termes s_C coïncide avec la date la plus ancienne des objets qui composent C. A l'inverse, la date e_C est définie par $e_C = \max_{X \in C}\{e_X\}$. Dans notre exemple, cela s'exprime par des relations d'Allen combinées avec des disjonctions:

$(D\ meets\ A)\ or\ (D\ meets\ B)$
$D\ before\ A$
$D\ before\ B$
$(E\ met\text{-}by\ A)\ or\ (E\ met\text{-}by\ B)$
$E\ after\ A$
$E\ after\ B$

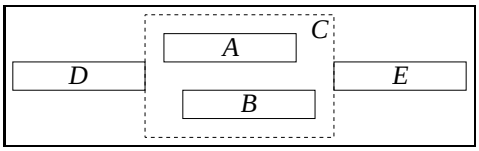


Figure 1.4: Un exemple de scénario avec un objet composite.

1.1.2. Informations quantitatives, intervalles et ensembles de tolérance

Pour permettre une expressivité plus grande, il est souhaitable de compléter les relations d'Allen par des informations quantitatives du genre: $A\ before\ B$ d'au moins 3 secondes, d'au plus 2 secondes, entre 2 et 3 secondes... Il est également nécessaire de fournir des intervalles de durées pour les objets multimédia, plutôt qu'une durée unique. Nous en expliquons maintenant les raisons, grâce à l'exemple simple illustré par la figure 1.5 (qui reprend le scénario par lignes de temps de la figure 1.1), où les flèches signifient que la fin de l'objet source coïncide avec le début de l'objet cible. La spécification textuelle équivalente à ce schéma est définie comme suit.

$video_1\ starts\ son_1$
$video_1\ meets\ texte_1$
$video_1\ meets\ image_1$
$texte_1\ finishes\ son_1$
$image_1\ meets\ texte_2$
$son_1\ meets\ son_2$
$texte_2\ finishes\ son_2$

Les objets multimédia proviennent souvent de sources différentes, il n'y a donc aucune chance de pouvoir satisfaire les contraintes exprimées précédemment. Cela signifierait par exemple que la durée de $video_1$ ajoutée à celle de $texte_1$ est égale à celle de son_1. Nous appelons ***durée idéale*** la durée intrinsèque des objets s'ils en ont une (e.g. une vidéo ou un son), ou la durée que l'auteur a choisie s'ils n'en ont pas (e.g. un texte ou une image). En effet, l'auteur peut juger du temps nécessaire à la lecture d'un texte ou à l'assimilation des différents éléments d'une image en fonction du message qu'il tente de faire passer.

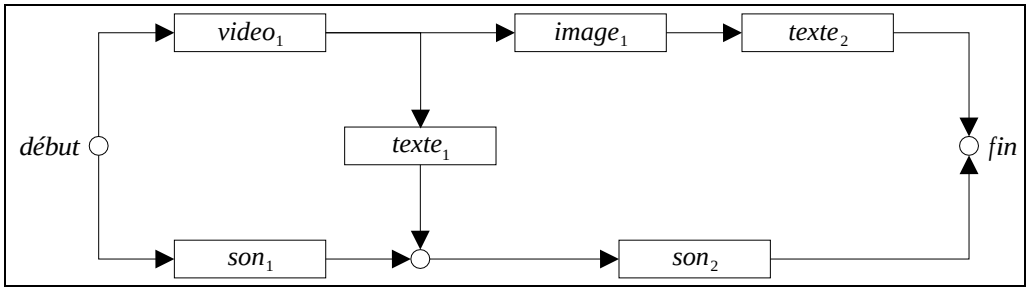


Figure 1.5: Un exemple de scénario avec les relations d'Allen.

Pour que les contraintes induites par les relations temporelles puissent être satisfaites, l'auteur doit accepter une certaine flexibilité sur la durée idéale des objets multimédia. Par exemple, pour une image dont la durée choisie était 10 secondes, il est tout à fait concevable que l'auteur accepte une durée effective entre 9 et 11 secondes. De la même manière, la durée d'un texte peut être définie dans un intervalle continu.

En revanche, pour une vidéo ou un son, il est plus délicat d'adapter la durée de manière continue. Cette durée est déterminée par le débit des données, en images par seconde pour la vidéo et en kilobits par seconde pour le son. Il est facile de s'apercevoir qu'un changement d'une seule unité dans le débit du média entraîne une modification de la durée de manière discrète. Pour ce type de média, la tolérance n'est donc pas un intervalle continu, mais un ensemble de valeurs discrètes. Néanmoins, il existe des techniques, détériorant plus ou moins la qualité du média, qui permettent d'ajuster de manière continue la durée de ces médias (cf. [Stei90]). L'aspect discret n'intervient alors finalement que lorsque l'on propose plusieurs alternatives pour un objet multimédia. Par exemple, la bande son d'une narration peut exister en plusieurs versions de durées différentes.

1.1.3. Informations causales: durées et dates imprévisibles

Nous avons vu qu'il était nécessaire de pouvoir gérer les incertitudes. Il en existe deux types, celles sur les durées de certains objets multimédia, et celles sur les dates d'occurrence de certains événements. La première peut être liée tout simplement à un objet multimédia dont la durée ne peut pas être connue à l'avance, comme une requête dans une base de données ou l'exécution d'une applet. La seconde est typiquement une action d'un utilisateur, par exemple un clic sur un bouton qui va déclencher une nouvelle séquence d'objets multimédia dans la présentation.

Pour un formalisme qui tente d'exprimer les synchronisations hypermédia, il est important de dissocier les durées et les événements prévisibles de ceux qui sont indéterminés (e.g. [Farg98]). [Laya97] préconise en outre deux relations dites *causales*. Elles confrontent des événements prévisibles à d'autres imprévisibles.

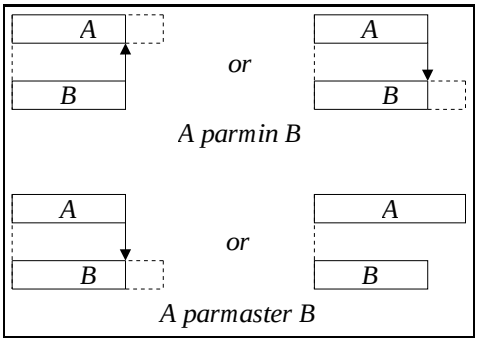


Figure 1.6: Relations causales.

La première relation, $A\ parmin\ B$, signifie que les objets multimédia A et B démarrent en même temps, et que le premier qui se termine arrête l'autre. En résumé, la durée de présentation de A et B est celle du plus petit, mais celle-ci n'est bien entendu pas connue à l'avance (les durées de A et/ou de B sont indéterminées). La seconde relation, $A\ parmaster\ B$, signifie que A et B démarrent en même temps, et que A termine B si ce dernier n'est pas déjà terminé. Là aussi, les durées de A et/ou de B sont indéterminées. La figure 1.6 illustre ces deux relations causales.

1.2. Problèmes de synchronisation

Comme nous l'avons déjà souligné, la synchronisation hypermédia suscite de nombreux problèmes de recherche opérationnelle. Ceux qui nous intéressent ici concernent la flexibilité offerte par les intervalles continus ou les ensembles discrets des durées tolérées pour les différents objets multimédia. Cette flexibilité offre un nombre important de possibilités de planification parmi lesquelles il faut choisir la meilleure. Avant de nous pencher sur ce problème d'optimisation, intéressons-nous tout d'abord à quelques problèmes de réalisabilité.

1.2.1. Problèmes de réalisabilité

La problématique générale consiste, à partir de spécifications temporelles qui décrivent un scénario, à déterminer une ***planification***, autrement dit une date pour chaque événement du scénario. Une planification sera ***réalisable*** si elle satisfait toutes les contraintes temporelles exprimées par l'auteur du scénario, i.e. les relations

temporelles et les ensembles de tolérance. S'il existe au moins une planification réalisable pour un scénario, on dira que ses contraintes temporelles sont *cohérentes*. De la littérature, il resort trois questions importantes concernant la réalisabilité d'une planification.

- La première question qui se pose est de savoir si une planification réalisable existe. Si la réponse est positive, il s'agit d'en déterminer au moins une. Si la réponse est négative, il faut déterminer la ou les contraintes qui sont responsables de ce fait, afin que l'auteur corrige les spécifications de son scénario.
- La seconde question, d'ordre plus pratique, consiste à savoir quelles sont les durées minimales et maximales de présentation d'un scénario. Comme l'explique [Kim95], il est tout à fait appréciable, aussi bien pour l'auteur du document que pour un spectateur, de connaître la durée totale d'une présentation.
- La troisième question concerne plutôt la phase de conception du document. Il peut être intéressant pour l'auteur de connaître la durée minimale (ou maximale) qu'un objet multimédia peut prendre dans une planification réalisable. Cela permet de savoir si cet objet peut poser problème dans la synchronisation. En effet, son intervalle durée minimale/durée maximale comparé à son ensemble de tolérance permet de juger des contraintes qu'il subit. Cette information est également importante pour la présentation, un objet avec peu de différence entre ses durées minimale et maximale est critique, un simple retard sur un tel objet pourrait empêcher toute planification réalisable.

Le problème de la réalisabilité prend un aspect très différent dès qu'on introduit des événements ou des durées imprévisibles. Un intervalle de temps plus ou moins précis, dans lequel un événement (ou une durée) imprévisible sera constaté, est généralement connu. Il est donc possible tout de même de détecter certaines incohérences temporelles, en fonction de la date potentielle de l'événement. Il existe des cohérences plus ou moins fortes (cf. [Vida97]).

- Cohérence forte: il existe une planification réalisable quelque soit l'occurrence des événements imprévisibles. Une planification peut donc être fixée à l'avance, on est sûr qu'elle pourra être réalisée.
- Cohérence faible: il existe toujours, pour chaque occurrence des événements imprévisibles, une planification réalisable. Il n'est pas possible de fixer à l'avance une planification réalisable.
- Cohérence dynamique: il existe une planification réalisable, qui peut être modifiée au moment de l'occurrence d'un événement imprévisible, pour donner une nouvelle planification réalisable. Une planification peut être fixée à l'avance, on est sûr que quoi qu'il arrive, il sera possible de réajuster cette planification.

La problématique peut sembler proche de celle posée en planification de projets ou de tâches (e.g. [Fold93]). Cependant, la grande différence des problèmes de synchronisation inter-média est qu'aucun temps mort implicite (i.e. non spécifié par l'auteur) n'est autorisé. Les relations temporelles sont telles qu'il est interdit, comme dans l'exemple de la figure 1.5, de jouer son_1 plus court que $video_1$ suivi de $texte_1$.

1.2.2. Problèmes d'optimisation

Une fois que l'on sait qu'une planification est possible, la réelle question est de savoir quelle est la planification qui permet une présentation du document de la meilleure qualité. Les ensembles de tolérance offrent une certaine souplesse dans la planification et généralement il existe un grand nombre de solutions réalisables.

Cela soulève le problème de pouvoir quantifier la qualité d'une planification. Cette discussion est encore ouverte à l'heure actuelle. En effet, certains proposent simplement d'attribuer un coût à chaque objet, proportionnel à la déformation (i.e. la différence entre sa durée idéale et sa durée planifiée), e.g. [Buch93b] et [Kim95]. Mais cette approche entraîne des inégalités, un objet très contraint peut se retrouver très déformé alors que d'autres très peu. Au lieu d'avoir des coûts proportionnels, des coûts quadratiques sont proposés, e.g. [Kim95]. Enfin, pour des difficultés pratiques à déformer des objets multimédia, certains auteurs préfèrent exprimer la qualité d'un document par un minimum d'objets déformés, sans distinction entre une légère ou une forte déformation, e.g. [Medi02]. Nous discutons plus en détail de cet aspect au chapitre 2 après avoir présenté une modélisation sous forme de graphe des relations temporelles et des ensembles de tolérance introduits ici.

Dans un objectif d'optimisation, une grande différence est faite entre des intervalles de tolérance continus et des ensembles discrets, puisque de manière générale il est plus facile d'optimiser dans un espace continu que dans un espace discret. Deux planifications "proches" dans le domaine continu auront des qualités similaires, alors que dans le domaine discret deux planifications quasiment identiques pourront avoir des qualités très différentes.

1.2.3. Cycle de vie d'un document synchronisé

Ces problèmes de synchronisation se posent à différentes étapes du cycle de vie d'un document hypermédia (cf. figure 1.7). En effet, il est souhaitable tout d'abord d'assister l'auteur dans la conception des documents, cette phase est appelée l'*édition*. Lors de l'élaboration du scénario, c'est-à-dire au moment d'énoncer les relations temporelles et de fixer les ensembles de tolérance des durées des objets, il faut vérifier en temps réel que l'auteur ne produit pas un système de contraintes irréalisable, et donc vérifier la cohérence d'un scénario.

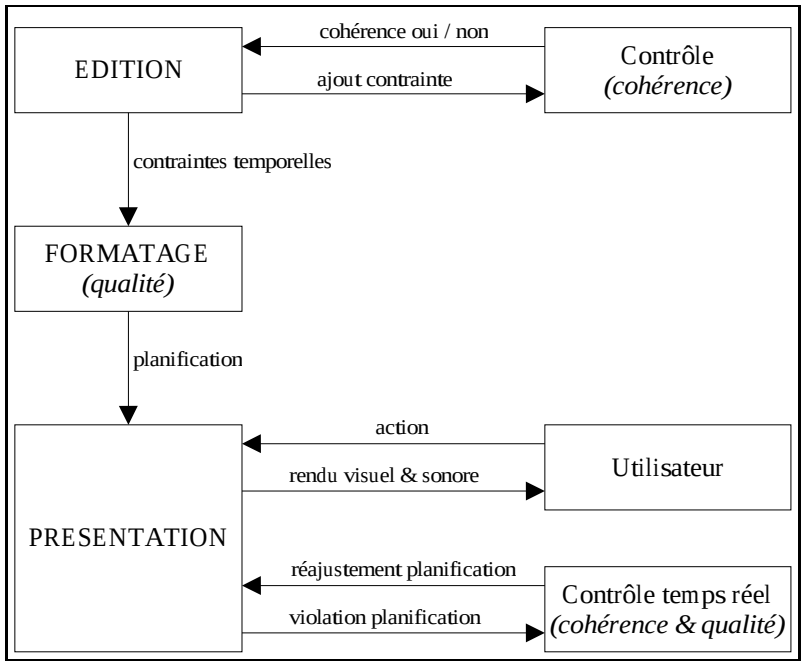


Figure 1.7: Cycle de vie d'un document hypermédia.

Il est également important d'essayer d'indiquer la ou les contraintes qui entraînent l'incohérence du scénario. Le problème de réalisabilité devient alors un peu plus délicat à traiter, il faut non seulement trouver une solution

réalisable si elle existe, mais déterminer aussi les raisons pour lesquelles éventuellement elle n'existe pas. En outre, il a été expliqué précédemment que certains événements ou durées peuvent être imprévisibles. Cet aspect doit être pris en compte dès la phase de planification du document (e.g. [Vida97]) et les différents niveaux de cohérence (forte, faible ou dynamique) ou d'incohérence établis.

La création du scénario d'un document est un processus incrémental: l'auteur crée un premier scénario et doit être sûr de suite qu'il est cohérent, il le prévisualise, revient au scénario, ajoute/supprime une relation... Il est donc nécessaire à la fois de vérifier en temps réel la cohérence du document, mais également de pouvoir en peu de temps fournir une planification optimale (ou presque) pour une prévisualisation. L'auteur refusera en effet d'attendre plusieurs secondes pour prévisualiser le résultat de son travail.

Il est également nécessaire, afin de ne pas déstabiliser l'auteur et de toujours lui permettre une approche incrémentale, de garantir une certaine robustesse dans la solution trouvée. Autrement dit, si l'auteur effectue un changement simple dans les relations temporelles ou les ensembles de tolérance, il ne faut pas que la nouvelle solution soit totalement différente de l'ancienne.

Une fois les relations et les ensembles spécifiés par l'auteur, il est possible de prendre plus de temps, dans la phase appelée *formatage*, pour déterminer la planification de meilleure qualité pour le document. Cette phase peut être lancée à chaque prévisualisation si elle est suffisamment rapide. Ensuite, vient la phase de *présentation* où la planification est respectée tant qu'il n'apparaît pas de problème, tel qu'un retard sur le réseau au cours du téléchargement d'un média. En cas d'anomalie, il faut alors être capable de réajuster en temps réel la planification, et là encore de disposer d'une méthode efficace pour réagir instantanément.

1.3. Méthodes de résolution existantes

Il est très délicat de discuter des méthodes qui permettent de déterminer une planification réalisable et/ou optimale, et encore moins de les comparer, puisqu'elles dépendent très fortement de la modélisation choisie, et donc des restrictions d'expressivité induites, pour exprimer les relations temporelles. Nous en proposons néanmoins un rapide aperçu.

1.3.1. Cohérence et planification réalisable d'un scénario

La détection de la cohérence d'un scénario et la détermination d'une planification est un problème simple lorsqu'il s'agit uniquement d'intervalles de tolérance continus sans aucun événement ou durée imprévisible. [Song99] propose par exemple une modélisation sous forme de graphe très proche de la nôtre (cf. chapitre 3), les contraintes temporelles d'un scénario se traduisent alors en conditions d'inexistence de certains cycles sur le graphe équivalent, des cycles de longueur négative plus précisément. Le problème peut alors être ramené simplement à un problème de plus court chemin dont de nombreux algorithmes polynômiaux existent. Cette approche est tout à fait adaptée au processus incrémental de la phase d'édition puisque très efficace.

Détecter la cohérence d'un scénario et déterminer une planification avec des événements imprévisibles devient plus délicat. Les solutions proposées reposent sur des systèmes d'énumération exhaustive des possibilités (e.g. RT-LOTOS [Cour96b], [Sant99], [Samp00a]), ou utilisent une simulation à événements discrets pour tester différentes planifications (réseaux de Petri [Litt90]). Ces méthodes sont tout à fait adaptées à cet aspect imprévisible, mais à cause de la combinatoire explicite qu'elle gèrent, elle deviennent tout de même assez

vite difficiles à manipuler. Dans le cas où l'aspect imprévisible est écarté, elles deviennent inefficaces faces à la méthode présentée précédemment. En revanche, elles offrent des réponses aux questions concernant la cohérence forte, faible ou dynamique du document.

Toutes ces approches ne peuvent pas être employées en temps réel pour maintenir la cohérence d'un scénario. Pour cela, [Laya02] propose un algorithme glouton qui tente au cours du temps, face aux événements imprévus tels que des retards dans le téléchargement des objets multimédia, ou des interactions du spectateur, de garantir une planification réalisable, ou tout du moins de s'en rapprocher. L'approche considère des temps morts entre certains points d'articulation du scénario afin de permettre une flexibilité supplémentaire dans la planification du scénario et essaie de les réduire, jusqu'à ce qu'ils soient nuls si possible.

A notre connaissance, il n'existe pas d'approche qui tente de déterminer une planification réalisable, ni même de vérifier la cohérence d'un scénario pour des ensembles de tolérance discrets. Une approche utilisant la programmation par contrainte semble tout à fait adaptée pour manipuler ces ensembles discrets.

1.3.2. Planification optimale d'un scénario

Les approches énoncées précédemment ne gèrent pas la qualité des planifications. La solution actuellement employée consiste à modéliser les contraintes temporelles sous forme de contraintes linéaires et de s'assurer que la fonction qui évalue la qualité d'une planification est linéaire. Il est alors possible de résoudre le problème par la programmation linéaire (e.g. [Buch93b] et [Kim95]). Mais il est impossible dans ce cas de considérer une mesure de la qualité différente d'une fonction linéaire, alors que [Kim95] explique qu'il serait intéressant de manipuler des fonctions quadratiques pour introduire une équité dans les déformations des objets multimédia.

La problématique suscitée par des événements et des durées imprévisibles est néanmoins légèrement abordée dans ces travaux, simplement en découpant le scénario sur des points d'articulation induits par les données indéterministes. Par la programmation linéaire, les portions du scénario sont ensuite optimisées, les dates trouvées pour les événements prévisibles étant relatives à celles des événements imprévisibles.

Une approche combinant l'un des modèles linéaires précédents avec une technique de *Branch & Bound* permet d'optimiser la qualité d'une planification mesurée par le nombre d'objets touchés par une déformation (cf. [Medi02]). Cette méthode peut être envisagée pour la phase de formatage, mais ne peut pas être employée pour un usage temps réel.

La programmation linéaire est efficace sur des documents de taille raisonnable (quelques centaines d'événements), et par conséquent est adaptée à la phase de formatage ou de prévisualisation. En revanche, une utilisation temps réel n'est pas possible en l'état. Il n'est pas concevable d'optimiser plusieurs fois le scénario pendant la diffusion du document, mais les contraintes linéaires pourraient être modifiées en temps réel et l'optimisation recommencée à partir de la solution existante. En résumé, aucune solution ne permet à notre connaissance un réajustement en temps réel de la planification optimale d'un scénario.

1.4. Conclusion

De cette présentation, il resort que des méthodes variées permettent de répondre à certains des problèmes liés à la synchronisation inter-média. Mais chacune se concentre sur un aspect du problème et est peu adaptée aux

autres: la cohérence ou la qualité, l'imprévisible ou le prévisible, le temps réel ou le formatage. L'objectif de notre étude est une investigation des possibilités offertes par une modélisation des contraintes temporelles et des ensembles de tolérance par l'outil de graphe possédant une théorie importante, en espérant couvrir plusieurs aspects de la problématique.

Nous proposons au cours des chapitres des méthodes permettant la détection de la cohérence d'un scénario, la recherche d'une planification réalisable, optimale, avec les principales mesures de qualité énoncées ici. L'efficacité des méthodes et leur approche permettent d'envisager une utilisation en temps réel, aussi bien pour la cohérence que pour la qualité de la planification. Cependant, nous excluons de notre étude les ensembles de tolérance discrets (comme toutes les approches présentées précédemment) et nous ne prenons pas en considération les événements imprévisibles, pour lesquels l'approche générale que nous proposons n'est pas adaptée.

Le chapitre suivant propose l'introduction de quelques notions nécessaires sur les graphes, avant de poursuivre la discussion sur la synchronisation hypermédia par la proposition d'une modélisation sous forme de graphe temporel des spécifications d'un scénario, et l'explication des limitations qu'elle impose sur les relations temporelles ainsi représentables.

CHAPITRE 2
LES GRAPHES TEMPORELS

Dans le premier chapitre, nous avons exposé quelques problèmes de synchronisation hypermédia (inter-média plus précisément) liés à la réalisabilité et à l'optimisation d'une planification pour la présentation synchronisée d'objets multimédia. Nous présentons ici une modélisation sous forme de graphe. Celle-ci ne permet pas de représenter toutes les spécifications désirées par les auteurs de documents hypermédia synchronisés, mais tente néanmoins d'en rassembler les principales. Travaillant en collaboration avec les concepteurs d'HyperProp (cf. [Soar00]), le choix des spécifications à prendre en compte dans notre modélisation a fortement été guidée par leurs besoins.

Mais avant d'étudier cette modélisation, il nous semble nécessaire de consacrer une section à l'introduction de notions et de propriétés élémentaires sur les graphes indispensables pour la compréhension du document et éventuellement pour intégrer les algorithmes des prochains chapitres dans un logiciel de conception et/ou de présentation de documents hypermédia synchronisés. Après quelques rappels très généraux sur les graphes, nous introduisons les notions de *flot* et de *tension*. Nous verrons dans les chapitres suivants qu'une relation très forte entre le flot et la tension, la *dualité*, joue un rôle très important dans les méthodes de résolution que nous présentons.

Après cette introduction, nous présentons la modélisation sous forme de graphe retenue, et précisons les restrictions qu'elle impose. Nous montrons ensuite que les problèmes de synchronisation hypermédia peuvent ainsi être considérés comme des problèmes de tension dans un graphe. En particulier, nous discutons de la pertinence des problèmes de la *tension compatible* et de la *tension de coût minimal* pour l'aspect hypermédia, et pour lesquels nous présentons des méthodes de résolution dans la seconde partie du mémoire. Nous exposons finalement différentes manières de mesurer la qualité d'une planification qui semblent significatives pour les auteurs de documents hypermédia, toutes ne seront pas abordées par la suite.

2.1. Introduction aux graphes

Cette section nous a semblé nécessaire pour rassembler les notions de base concernant les graphes. Le lecteur novice dans ce domaine y trouvera tous les éléments indispensables au suivi des chapitres suivants du mémoire. Pour le lecteur plus confirmé, il est possible d'utiliser cette section comme référence pour une définition ou un théorème, grâce à l'index situé à la fin du mémoire.

En complément, nous proposons également un annexe qui regroupe des algorithmes dont le détail est jugé peu important dans le flot de nos réflexions, mais néanmoins utile notamment pour l'interprétation et la reproduction des résultats numériques. Cette section introduit aussi quelques propriétés nécessaires pour la génération aléatoire de problèmes pour nos jeux d'essais dont les principaux algorithmes sont détaillés dans l'annexe.

Une remarque importante sur la notation employée tout au long de ce document: les fonctions dont le domaine de définition est discret et fini (e.g. un ensemble d'arcs ou de noeuds) seront souvent utilisées comme des vecteurs. Ainsi une fonction $f : I \longmapsto \mathbb{R}$ pourra être considérée comme un vecteur $f = (f_i)_{i \in I}$ dont les composantes seront notées $f_i = f(i)$.

2.1.1. Notions élémentaires

2.1.1.1. Graphe

Pour illustrer les définitions présentées ici, nous allons utiliser le graphe représenté par la figure 2.1, noté $G = (X; U)$, où X est l'ensemble des noeuds et U est l'ensemble des arcs. Nous notons $n = |X|$ le nombre de noeuds de G et $m = |U|$ le nombre d'arcs.

Ce graphe est ***orienté***, cela signifie qu'une distinction est faite entre les deux extrémités (i.e. les noeuds) d'un arc. On appelle ***noeud origine*** le noeud d'où part l'arc et ***noeud destination*** le noeud où arrive l'arc. Ainsi pour un noeud x, on appelle ***arcs entrants*** tous les arcs ayant x pour noeud origine et ***arcs sortants*** tous les arcs ayant x comme noeud destination.

Nous désignerons souvent un arc u de source x et de destination y par le couple $(x; y)$, bien que cette notation soit abusive dans la mesure où plusieurs arcs peuvent avoir même origine x et destination y. Dans un tel cas $(x; y)$ est appelé un ***arc multiple***. Un ***multigraphe*** est un graphe possédant au moins un arc multiple.

On désignera par ***sens direct*** l'utilisation d'un arc dans le sens origine-destination et par ***sens indirect*** son utilisation dans le sens destination-origine.

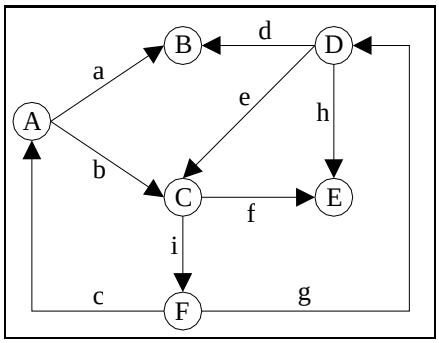


Figure 2.1: Un exemple de graphe.

Le ***degré sortant***, noté d_x^+, d'un noeud x est le nombre de ses arcs sortants et son ***degré entrant***, noté d_x^-, est le nombre de ses arcs entrants. Le ***degré*** d'un noeud est la somme de ses degrés sortant et entrant. Un noeud dont le degré entrant est nul est appelé ***noeud source*** du graphe. De même, un noeud dont le degré sortant est nul est appelé ***noeud puits*** du graphe.

2.1.1.2. Chaîne et chemin

On dit que deux arcs sont ***adjacents*** s'ils ont au moins une extrémité (le noeud origine ou le noeud destination) en commun. Un noeud est ***adjacent*** à un arc s'il est au moins l'une des deux extrémités de l'arc.

On désigne par ***chaîne*** une succession d'arcs $(u_1; u_2 \ldots u_p)$ dans un graphe telle que deux arcs consécutifs dans la chaîne sont adjacents dans le graphe. Si l'on considère une chaîne C, on notera C^+ l'ensemble des arcs qui sont dans le sens de parcours et C^- l'ensemble des arcs qui sont dans le sens opposé.

On appelle ***chemin*** une chaîne dont les arcs sont tous orientés dans le même sens, i.e. pour deux arcs consécutifs, le noeud destination du premier est le noeud source du second.

On désigne par ***chaîne élémentaire*** (respectivement ***chemin élémentaire***) une chaîne (respectivement un chemin) qui ne passe qu'une seule fois par un même noeud.

D'après la figure 2.1, voici quelques exemples de chaînes et de chemins.

- $(a; d; e; i; g; h)$ est une chaîne (pas élémentaire),
- $(a; d; h)$ est une chaîne élémentaire,
- $(b; i; g; e; f)$ est un chemin (pas élémentaire),
- $(b; i; g; h)$ est un chemin élémentaire.

2.1.1.3. Cycle et circuit

On appelle ***cycle*** une chaîne non vide telle que le noeud de départ et le noeud d'arrivée sont identiques (e.g. le cycle $(a; d; g; c)$ dans la figure 2.2). Si l'on considère un cycle γ et qu'on lui choisit arbitrairement un sens de parcours, on notera γ^+ l'ensemble des arcs qui sont dans ce sens de parcours et γ^- l'ensemble des arcs qui sont dans le sens opposé. Si tous les arcs sont dans le même sens, le cycle est appelé un ***circuit***.

Comme les chaînes et les chemins, un ***cycle élémentaire*** (respectivement un ***circuit élémentaire***) est un cycle (respectivement un circuit) qui ne passe qu'une seule fois par un même noeud. Pour des facilités d'écriture, un cycle élémentaire noté γ pourra être considéré comme un vecteur d'entiers, ses composantes seront notées γ_u pour tout arc u du graphe G, telles que:

$$\gamma_u = \begin{cases} 0, \text{ si } u \notin \gamma \\ +1, \text{ si } u \in \gamma^+ \\ -1, \text{ si } u \in \gamma^- \end{cases}$$

Dans la figure 2.2, le cycle $(a; d; g; c)$ peut être représenté par le vecteur $(+1; 0; +1; -1; 0; 0; -1; 0; 0)$, en supposant les arcs dans l'ordre $a, b, c, d, e, f, g, h, i$.

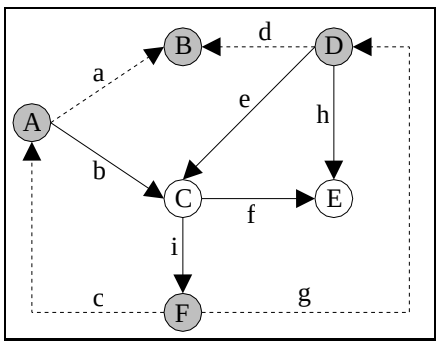


Figure 2.2: Un exemple de cycle.

Si un graphe est sans cycle, alors $m \leq n - 1$ (cette propriété se démontre facilement par récursivité sur la taille du graphe).

2.1.1.4. Cocycle et cocircuit

Soit A un sous-ensemble des noeuds de G. On note $\omega(A)$ le ***cocycle*** de A. Il s'agit de l'ensemble des arcs de G qui ont une extrémité dans A et l'autre dans $X \setminus A$, i.e. les noeuds qui ne sont pas dans A (e.g. le cocycle $(d; e; f; g)$ dans la figure 2.3).

L'ensemble $\omega(A)$ peut être séparé en deux sous-ensembles: $\omega^+(A)$ qui contient les arcs du cocycle qui ont leur source dans A, et $\omega^-(A)$ qui contient ceux qui ont leur destination dans A. Si tous les arcs sont dans le même sens, le cocycle est appelé un ***cocircuit***.

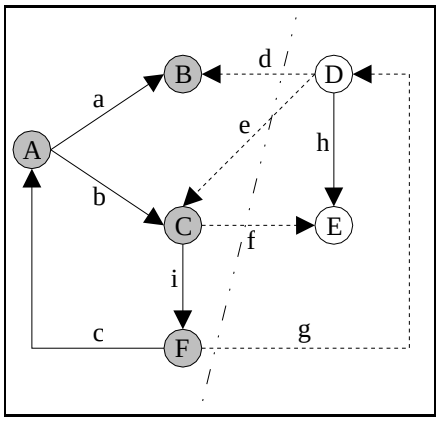


Figure 2.3: Un exemple de cocycle.

De la même manière que le cycle, un cocycle noté w pourra être considéré comme un vecteur d'entiers, ses composantes seront notées ω_u pour tout arc u du graphe G, telles que:

$$\omega_u = \begin{cases} 0, \text{ si } u \notin \omega \\ +1, \text{ si } u \in \omega^+ \\ -1, \text{ si } u \in \omega^- \end{cases}$$

Dans la figure 2.3, le cocycle $(d; e; f; g)$ peut être représenté par le vecteur $(0; 0; 0; -1; -1; +1; +1; 0; 0)$, en supposant les arcs dans l'ordre $a, b, c, d, e, f, g, h, i$.

2.1.2. Arbre

2.1.2.1. Connexité et forte connexité

Deux noeuds x et y sont ***connectés*** si et seulement s'il existe une chaîne entre x et y ou bien $x = y$. Un graphe est dit ***connexe*** si et seulement si tous ses noeuds sont connectés deux à deux.

Deux noeuds x et y sont ***fortement connectés*** si et seulement s'il existe un chemin de x à y et un chemin de y à x, ou bien $x = y$. Un graphe est dit ***fortement connexe*** si et seulement si tous ses noeuds sont fortement connectés deux à deux.

Si le graphe G est connexe, alors $m \geq n - 1$ (cette propriété se démontre aisément par récursivité sur la taille du graphe).

On appelle ***composante connexe*** (respectivement ***composante fortement connexe***) un ensemble de noeuds dont tous les noeuds sont connectés (respectivement fortement connectés) deux à deux et tout noeud extérieur à la composante n'est pas connecté (respectivement fortement connecté) à aucun noeud de la composante.

Pour exemple, le graphe de la figure 2.1 est connexe, mais pas fortement connexe. En revanche, la composante $(A; D; C; F)$ est fortement connexe.

2.1.2.2. Sous-graphe et graphe partiel

Soit un graphe $G = (X; U)$, le sous-ensemble de noeuds $X' \subseteq X$, et les sous-ensembles d'arcs $U' \subseteq U$ et $U'' = \{(x; y) \in U,\ x \in X' \text{ et } y \in X'\}$. Voici la définition de graphes qui peuvent être formés à partir de ces ensembles.

- Le graphe $(X'; U'')$ est appelé ***sous-graphe*** de G. Autrement dit, un sous-graphe de G, c'est G privé de quelques noeuds et des arcs adjacents à ces noeuds.
- Le graphe $(X; U')$ est appelé ***graphe partiel*** de G. Autrement dit, un graphe partiel de G, c'est G privé de quelques arcs.

2.1.2.3. Arbre et arbre recouvrant

Un ***arbre*** est un graphe connexe sans cycle. De cette définition découlent les propriétés suivantes.

- Tous les noeuds d'un arbre sont reliés par une chaîne (grâce à la connexité) et une seule (sinon deux chaînes formeraient un cycle).
- Un arbre à n sommets contient exactement $n - 1$ arcs (car connexe signifie $m \geq n - 1$ et sans cycle signifie $m \leq n - 1$).
- Un arbre possède au moins deux noeuds de degré 1 (cette propriété se vérifie facilement par récursivité sur la taille de l'arbre).
- L'ajout d'un seul arc dans un arbre introduit un cycle (car m devient supérieur à $n - 1$).
- La suppression d'un seul arc dans un arbre introduit deux composantes connexes (car m devient inférieur à $n - 1$).

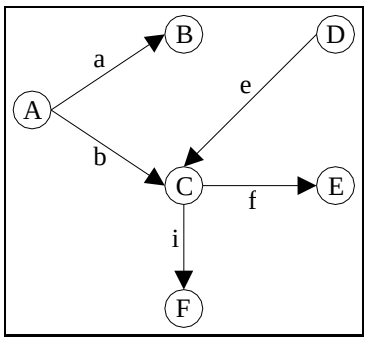


Figure 2.4: Un exemple d'arbre recouvrant.

On appelle ***arbre recouvrant*** d'un graphe $G = (X; U)$ un arbre $T = (X; U')$ tel que $U' \subseteq U$. Autrement dit, un arbre recouvrant de G est un graphe partiel connexe de G sans cycle. Le graphe de la figure 2.4 est un arbre recouvrant du graphe de la figure 2.1. Un algorithme de recherche d'arbre recouvrant dans un graphe est détaillé dans l'annexe.

2.1.2.4. Arbre binaire

Un noeud a est une ***racine*** du graphe G si pour tout noeud x de G il existe un chemin de a à x. Un arbre de racine a est appelé une ***arborescence*** de racine a.

En particulier, nous empruntons le terme ***arbre binaire*** au domaine des structures de données pour la suite de

ce document. Il désigne une arborescence pour laquelle tous les noeuds ont un degré entrant égal à 1 (exceptée la racine pour laquelle c'est 0) et un degré sortant d'au plus 2. Un exemple est montré par la figure 2.5.

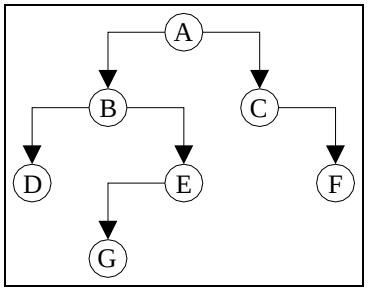


Figure 2.5: Un exemple d'arbre binaire.

En partant de la racine et en suivant les arcs dans le sens direct de gauche à droite, une arborescence représente un ordre sur les noeuds. Nous proposons ainsi une notation récursive des arbres binaires. Supposons a la racine d'un arbre binaire T, si l'on supprime la racine, on se retrouve avec deux arbres T_l et T_r, que l'on nomme ***sous-arbres***. T sera alors représenté par le triplet $(a; T_l; T_r)$. Pour chaque sous-arbre, on recommence l'opération jusqu'à avoir des sous-arbres vides qui seront symbolisés par $\emptyset$. Voici la représentation de l'arbre binaire T de la figure 2.5.

$$
\begin{aligned}
T &= (A; T_1; T_2) \\
T_1 &= (B; T_{11}; T_{12}) \\
T_2 &= (C; \emptyset; T_{22}) \\
T_{11} &= (D; \emptyset; \emptyset) \\
T_{12} &= (E; T_{121}; \emptyset) \\
T_{22} &= (F; \emptyset; \emptyset) \\
T_{121} &= (G; \emptyset; \emptyset)
\end{aligned}
$$

2.1.3. Représentation des graphes

Nous proposons ici les structures les plus populaires pour représenter un graphe. Très souvent, les performances d'un algorithme sont liées à la structure de graphe employée. En effet, chaque structure est très efficace pour un certain type d'opération, mais est souvent mauvaise pour d'autres. Ainsi, il est impossible d'avoir une structure qui soit la plus efficace pour toutes les opérations. Dans notre étude, nous avons choisi de ne pas favoriser une ou plusieurs opérations en particulier et nous avons opté pour une structure qui a une efficacité à peu près égale pour toutes les opérations de base. Notre structure n'est donc jamais la plus efficace pour un algorithme mais n'est aussi jamais la plus médiocre.

Pour commencer, nous présentons des structures sous forme de matrice qui s'avèrent très intéressantes à manipuler d'un point de vue théorique. Ensuite, nous nous intéressons à une modélisation par listes d'adjacence qui est très similaire à la structure que nous avons utilisée pour coder les algorithmes. Le détail de notre structure est proposé dans l'annexe.

2.1.3.1. Matrice d'incidence noeud-arc

Un graphe G peut être représenté par une matrice S de dimension $n \times m$, dite ***matrice d'incidence noeud-arc***, pouvant contenir uniquement les valeurs 0, $+1$ et -1. Chaque ligne de la matrice est associée à un noeud et

chaque colonne à un arc. Ainsi, une composante indique la relation qu'il existe entre un noeud x et un arc u. Pour chaque composante S_{xu} de la matrice S, on a:

$$S_{xu} = \begin{cases} +1, \text{ si } x \text{ est la source de } u \\ -1, \text{ si } x \text{ est la destination de } u \\ 0, \text{ si } x \text{ et } u \text{ ne sont pas adjacents} \end{cases}$$

Pour exemple, voici la représentation par matrice d'incidence du graphe de la figure 2.1.

	a	b	c	d	e	f	g	h	i
A	$+1$	$+1$	-1	0	0	0	0	0	0
B	-1	0	0	-1	0	0	0	0	0
C	0	-1	0	0	-1	$+1$	0	0	$+1$
D	0	0	0	$+1$	$+1$	0	-1	$+1$	0
E	0	0	0	0	0	-1	0	-1	0
F	0	0	$+1$	0	0	0	$+1$	0	-1

La structure de cette matrice est assez particulière. Seulement $2m$ des nm composantes sont non nulles. Elle occupe donc beaucoup de place en mémoire. En outre, son utilisation n'apporte que rarement (dans le cas où la matrice elle même a de l'intérêt) de bons résultats au niveau des algorithmes. En effet, rien que le parcours du graphe s'avère difficile. Cependant, cette matrice a des propriétés mathématiques très intéressantes, notamment le fait qu'elle soit *unimodulaire*.

Soit M une matrice $p \times q$. M est dite ***unimodulaire*** si les conditions suivantes sont vérifiées.

- M est à composantes entières.
- M est de rang maximal (i.e. de rang p).
- Toute sous-matrice carrée de M a un déterminant égal à $+1$, -1 ou 0.

Le résultat intéressant de cette unimodularité est le suivant. A est unimodulaire si et seulement si toute solution de base du système linéaire $Ax = b$, avec $x \geq 0$, est entière lorsque b est un vecteur d'entiers. Autrement dit, la résolution d'un programme linéaire de la forme:

$$\begin{cases} \min cx \\ Ax = b \\ x \geq 0 \end{cases}$$

avec la méthode du Simplex fournit une solution optimale à composantes entières pour tout vecteur d'entiers b, à condition bien entendu que le problème soit borné et possède au moins une solution réalisable.

2.1.3.2. Matrice d'adjacence noeud-noeud

Un graphe G peut être représenté par une matrice S de dimension $n \times n$, dite ***matrice d'adjacence noeud-noeud***, pouvant contenir uniquement les valeurs 0 et 1. Chaque ligne et chaque colonne de la matrice est associée à un noeud. Ainsi, une composante indique la relation qu'il existe entre deux noeuds x et y. Pour chaque composante S_{xy} de la matrice S, on a:

$$S_{xy} = \begin{cases} 1, \text{ si } \exists\,(x;y) \in U \\ 0, \text{ si } \nexists\,(x;y) \in U \end{cases}$$

Dans cette matrice, seulement m des n^2 composantes sont non nulles. Cette représentation sera donc efficace au niveau de l'espace mémoire utilisé lorsque le graphe est suffisamment dense (i.e. lorsqu'il y a suffisamment d'arcs). Néanmoins, cette représentation permet d'implémenter assez facilement les algorithmes. Son utilisation apporte plus souvent de bons résultats au niveau des algorithmes que la matrice d'incidence, en particulier si le graphe est dense. Malheureusement, il faut noter une simplification importante dans la modélisation: on suppose qu'il n'y a pas d'arc multiple dans le graphe. Pour exemple, voici la représentation par matrice d'incidence du graphe de la figure 2.1.

	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	0	0	0	0
C	0	0	0	0	1	1
D	0	0	1	0	1	0
E	0	0	0	0	0	0
F	1	0	0	1	0	0

2.1.3.3. Listes d'adjacence

L'un des inconvénients de la représentation sous forme de matrice est la difficulté à connaître pour un noeud tous les arcs qui lui sont adjacents, information qui est très utilisée dans les algorithmes. D'où la modélisation dite par ***listes d'adjacence***. Tous les arcs sont stockés dans une liste, tous les noeuds dans une autre et chaque noeud x possède deux listes: une pour les arcs entrants (i.e. $\omega^-(x)$), une autre pour les arcs sortants (i.e. $\omega^+(x)$). La figure 2.6 montre comment représenter le graphe de la figure 2.1.

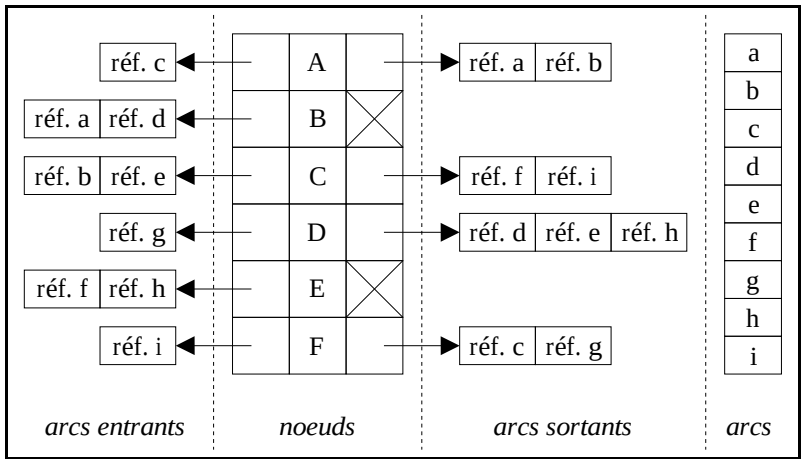


Figure 2.6: Un exemple de représentation par listes d'adjacence.

Plusieurs remarques pratiques peuvent être faites sur cette figure. Tout d'abord, pour les représentations par matrice, nous n'avons parlé que de la représentation de la structure du graphe en omettant volontairement la représentation des informations portées par les noeuds ou par les arcs, puisque nous n'allons pas utiliser ces structures en pratique.

Pour la représentation par listes d'adjacence, les informations seront stockées au niveau des identifiants, par exemple pour le noeud A (resp. l'arc a), ses données seront stockées à l'endroit noté A (respectivement a) sur la figure. Ainsi les listes des arcs entrants ou sortants ne contiennent que des références sur les arcs afin d'éviter toute redondance inutile d'information.

Il faut également noter que les listes utilisées pour cette structure peuvent être de simples tableaux (à condition que le graphe soit statique, i.e. pas d'ajout ou de suppression d'arcs ou de noeuds), des listes chaînées (à condition de limiter la suppression d'arcs ou de noeuds) ou bien des structures arborescentes (qui permettent efficacement l'ajout ou la suppression d'arcs ou de noeuds).

2.1.4. Algorithme de Minty

L'algorithme que nous présentons ici est basé sur le lemme de Minty (cf. [Mint61]). Minty propose d'associer une couleur à chaque arc d'un graphe parmi quatre (généralement vert, noir, bleu et rouge). Il démontre qu'une recherche basée sur une telle coloration conduit toujours à un cycle ou à un cocycle avec des propriétés bien particulières. L'intérêt de l'algorithme est que suivant la manière de colorer les arcs, on obtient des algorithmes pour rechercher différents types de cycle ou de cocycle.

2.1.4.1. Lemme de Minty

Soit un graphe $G = (X; U)$ avec les arcs colorés en vert, noir, bleu et rouge. Minty a démontré qu'une et une seule des propositions suivantes est vérifiée pour tout arc $u = (x; y) \in U$.

- L'arc u appartient à un cycle constitué (excepté u) seulement d'arcs verts, noirs et bleus avec les arcs noirs orientés dans la même direction que u et les arcs bleus dans la direction opposée. Les arcs verts peuvent être dans n'importe quel sens.
- L'arc u appartient à un cocycle constitué (excepté u) seulement d'arcs rouges, noirs et bleus avec les arcs noirs orientés dans la même direction que u et les arcs bleus dans la direction opposée. Les arcs rouges peuvent être dans n'importe quel sens.

La preuve de ce lemme peut être faite par la justification de l'algorithme de recherche d'un tel cycle ou cocycle exposé dans la section qui suit.

2.1.4.2. Algorithme

Nous exposons ici une méthode pour trouver un cycle ou un cocycle d'après la coloration de Minty pour un arc donné $u = (x; y)$ du graphe G. L'algorithme 2.1 parcours et marque les noeuds de G un par un en partant de y. Deux ensembles de noeuds A et B sont nécessaires pour cette opération. A contient les noeuds marqués pendant le parcours et B contient les noeuds marqués pendant le parcours susceptibles de conduire à des noeuds non marqués.

Le parcours d'un noeud à l'autre se fait en utilisant les arcs verts dans n'importe quel sens, les arcs noirs dans le sens direct et les arcs bleus dans le sens indirect. Si le noeud x est atteint, alors il existe une chaîne de x à y et donc un cycle contenant u qui correspond à la première proposition du lemme. L'algorithme nécessite alors une fonction p qui indique pour chaque noeud marqué l'arc qui a permis d'y accéder, cela permettra de construire le cycle détecté.

Si x ne peut pas être marqué, $\omega(A)$ est le cocycle qui sépare A des noeuds non marqués de G. Ce cocycle correspond à la deuxième proposition du lemme. En effet, si $\omega(A)$ contenait un arc vert, alors un autre noeud aurait dû être marqué. De manière similaire, si $\omega(A)$ contenait un arc noir dans le sens opposé à u dans le cocycle ou un arc bleu dans le même sens que u dans le cocycle, alors un autre noeud aurait dû être marqué.

```
Algorithme 2.1: cycleMinty(arc u = (x;y),graphe G,cycle γ,cocycle ω).
A ← {y};
B ← {y};
p_y ← ∅;

tant que B ≠ ∅ et x ∉ A faire
 choisir n ∈ B;
 B ← B \ {n};

 pour tout a = (n;z) ∈ U tel que a est vert ou noir et z ∉ A faire
  A ← A ∪ {z};
  B ← B ∪ {z};
  p_z ← a;
 fin pour;

 pour tout a = (z;n) ∈ U tel que a est vert ou bleu et z ∉ A faire
  A ← A ∪ {z};
  B ← B ∪ {z};
  p_z ← a;
 fin pour;
fin tant que;

si x ∉ A alors /* Cocycle trouvé. */
 construire le cocycle ω de A;
sinon /* Cycle trouvé. */
 construire le cycle γ en utilisant la fonction p à partir de x;
fin si;
```

Enfin, les deux propositions du lemme ne peuvent pas être vraies en même temps. Pour le prouver, supposons les deux propositions vérifiées. L'arc u appartient donc à la fois au cycle et au cocycle. Dans le cocycle, il existe forcément un arc v qui appartient au cycle comme u (en effet, le cycle contenant u doit forcément repasser dans le cocycle). Cet arc v ne peut être ni rouge, ni vert. En outre, s'il était bleu, il serait opposé à u dans le cycle mais aussi dans le cocycle, ce qui est impossible puisque deux arcs de même sens dans un cycle sont forcément opposés dans un cocycle. Pour la même raison, v ne peut pas être noir. D'où la contradiction.

Cet algorithme étant un simple parcours de graphe, il permet de détecter un cycle ou un cocycle de Minty en $O(m)$ opérations.

2.1.4.3. Détection de cycle, circuit, cocycle et cocircuit

L'algorithme de Minty est très intéressant puisqu'en choisissant judicieusement la coloration des arcs, il permet la recherche de différents types de cycle ou de cocycle. En général, il permet la détection d'un cycle ou d'un cocycle dont les propriétés peuvent s'exprimer indépendamment sur chaque arc. Par contre une propriété globale à tout le cycle ou le cocycle, comme par exemple la longueur d'un cycle, ne peut pas être traduite en termes de couleurs sur les arcs. Voici quelques colorations simples qui permettent les recherches les plus classiques.

- Colorer tous les arcs en vert permet de rechercher un cycle contenant un arc donné $u = (x; y)$ (tous les arcs peuvent être employés pour trouver une chaîne de y à x).
- Colorer tous les arcs en rouge permet de déterminer le cocycle du noeud source d'un arc donné $u = (x; y)$ (aucun arc ne peut être employé pour trouver une chaîne de y à x).
- Colorer tous les arcs en noir permet de rechercher un circuit contenant un arc donné u (si un cycle est trouvé, tous ses arcs sont noirs et donc dans le même sens que u). Si un tel circuit n'existe pas, alors un cocircuit est trouvé contenant l'arc u (si un cocycle est trouvé, tous ces arcs sont noirs et donc dans le même sens que u).

2.1.5. Conclusion

Dans la suite du document, les chaînes, les chemins, les cycles et les cocycles seront toujours considérés élémentaires. Nous rappelons également que nous utiliserons sans le préciser la représentation vectorielle des fonctions quand cela s'avérera nécessaire.

Nous supposons également que les graphes que nous étudions sont toujours connexes (s'ils ne le sont pas, il suffit de considérer chaque composante connexe séparément) et peuvent être des multigraphes (comme nous le verrons dans une section suivante, il est dans la nature même des graphes représentant des problèmes de synchronisation hypermédia de posséder des arcs multiples).

2.2. Flot et tension

2.2.1. Flot

Le flot est une notion très importante en théorie des graphes puisqu'elle permet de représenter des flux (e.g. l'information dans un réseau de télécommunication, les passagers dans un réseau de transport, les matières et les produits dans une chaîne de production...). De nombreux problèmes autour de ce concept ont été modélisés et étudiés, et par conséquent de nombreuses méthodes de résolution et d'importants résultats théoriques sont disponibles. Comme nous l'expliquons dans la seconde partie du document, pour résoudre des problèmes de tension, nous nous sommes fortement inspirés d'algorithmes conçus pour des problèmes de flot. En outre, la relation très particulière qui lie le flot et la tension est telle que la plupart des méthodes pour résoudre les problèmes de flot (respectivement de tension) manipulent la tension (respectivement le flot). Il semble donc important de rappeler ici quelques définitions et propriétés élémentaires sur le flot.

2.2.1.1. Définitions

On appelle ***flot*** une fonction φ qui associe à chaque arc de G une valeur (réelle ou entière). La particularité de cette fonction est que, pour chaque noeud x de G, on a la propriété suivante, appelée ***conservation des flots***.

$$\sum_{u \in \omega^+(x)} \varphi_u - \sum_{u \in \omega^-(x)} \varphi_u = 0 \tag{2.1}$$

Autrement dit, φ est un flot si et seulement si, pour chaque noeud, la somme des flots sur les arcs entrants est égale à la somme des flots sur les arcs sortants. Par exemple, si G représente un circuit électrique, l'intensité du courant est un flot sur G. Ainsi, si on note S la matrice d'incidence de G, la formule 2.1 peut s'écrire:

$$S\varphi = 0 \tag{2.2}$$

2.2.1.2. Propriétés élémentaires

Voici quelques propriétés élémentaires sur le flot très simplement vérifiables.

- Si φ est un flot et λ un réel, alors $\lambda\varphi$ est un flot.
- Si φ_1 et φ_2 sont des flots, alors $\varphi_1 + \varphi_2$ est un flot.

- Le seul flot possible sur un arbre est le flot $\varphi = 0$.
- Le vecteur qui représente un cycle est un flot (il est très facile de vérifier la conservation des flots).

2.2.1.3. Base de cycles

On dit que p vecteurs v_1, v_2 ... v_p de $\mathbb{R}^q$ sont ***dépendants*** s'il existe p coefficients non tous nuls λ_1, λ_2 ... λ_p dans $\mathbb{R}$ tels que:

$$\sum_{i=1...p} \lambda_i v_i = 0 \qquad (2.3)$$

A l'inverse, si la relation 2.3 n'est vérifiée que pour tous les λ_i nuls, alors les vecteurs v_1, v_2 ... v_p sont dits ***indépendants***.

Une ***base de vecteurs*** est un ensemble de vecteurs indépendants tel que tout vecteur v de $\mathbb{R}^q$ est une ***combinaison linéaire*** des vecteurs de la base, i.e. pour tout vecteur v il existe des coefficients λ_1, λ_2 ... λ_p tels que:

$$\sum_{i=1...p} \lambda_i v_i = v \qquad (2.4)$$

Les cycles pouvant être considérés comme des vecteurs, il est possible de construire une ***base de cycles***. Nous rappelons ici une manière simple d'obtenir une telle base. Considérons un arbre recouvrant $T = (X; U')$ du graphe G. Il ne contient pas de cycles, mais tout ajout dans l'arbre d'un arc u de $U \setminus U'$ (i.e. un arc qui n'est pas déjà dans T) engendre un cycle γ^u. Si on considère l'ensemble des cycles γ^u engendré en ajoutant séparément chaque arc u de $U \setminus U'$ dans l'arbre T, on obtient un ensemble $B_\gamma = \{\gamma^{u_1}; \gamma^{u_2} ... \gamma^{u_p}\}$ de cycles indépendants (car chaque cycle γ^u possède un arc qu'aucun autre ne possède, c'est u). Il faut s'assurer maintenant que tout cycle de G est une combinaison linéaire des cycles de B_γ.

Considérons un flot φ. Soit $\varphi' = \sum_{u \in U \setminus U'} \varphi_u \gamma^u$. φ' est une combinaison linéaire de flots donc un flot. La différence $\varphi - \varphi'$ est également un flot. Or, pour tout arc u de $U \setminus U'$, $\varphi'_u = \varphi_u$ puisque l'arc u n'apparaît que dans le cycle γ^u. Le flot $\varphi - \varphi'$ étant nul pour tout arc n'appartenant pas à T, il représente donc un flot défini strictement sur T, or tout flot sur un arbre est nul donc $\varphi = \varphi'$.

En conclusion, tout flot est une combinaison linéaire de la base B_γ. Un cycle pouvant être considéré comme un flot, tout cycle est une combinaison linéaire de B_γ. On remarque également que la base B_γ contient $m - n + 1$ cycles (car T possède $n - 1$ arcs). Enfin, l'algorithme pour construire B_γ est détaillé dans l'annexe.

2.2.2. Tension

Nous présentons maintenant la notion de tension dont nous verrons par la suite la pertinence pour modéliser des problèmes de synchronisation hypermédia. La plupart des résultats théoriques liés à ce concept proviennent du développement de méthodes de résolution pour des problèmes de flot. L'utilisation de la tension pour modéliser des problèmes dans les graphes est beaucoup moins répandue que pour le flot, cependant il existe de nombreux problèmes que la tension peut représenter (e.g. [Rock84]). Notamment dans le domaine de la planification, la tension peut être assimilée à une durée comme nous le verrons à la section suivante. Nous rappelons ici quelques définitions et propriétés élémentaires sur la tension (issues de [Berg62]) utiles pour notre étude.

2.2.2.1. Définitions

On désigne par ***potentiel*** une fonction π qui associe à chaque noeud de G une valeur (entière ou réelle). Associée à ce potentiel, on peut définir une fonction θ appelée ***tension*** qui attribue à chaque arc $u = (x; y)$ de G une valeur de la manière suivante.

$$\theta_u = \pi_y - \pi_x \tag{2.5}$$

Une fonction est donc une tension s'il est possible de lui associer un potentiel. Ces notions de tension et de potentiel peuvent être comparées à celles d'un circuit électrique. Si on note S la matrice d'incidence de G, la définition 2.5 de la tension peut se traduire:

$$\theta \textit{ est une tension} \Leftrightarrow \exists\, \pi \in \mathbb{R},\ S^t\pi = \theta \tag{2.6}$$

2.2.2.2. Propriétés élémentaires

Voici quelques propriétés élémentaires sur la tension très simplement vérifiables.

- Si θ est une tension et λ un réel, alors $\lambda\theta$ est une tension.
- Si θ_1 et θ_2 sont des tensions, alors $\theta_1 + \theta_2$ est une tension.
- Toute fonction qui associe à chaque arc d'un arbre une valeur est une tension.
- Le vecteur qui représente un cocycle est une tension (il est très facile d'y associer un potentiel).

Tout vecteur tension θ et tout vecteur flot φ sur un graphe G sont orthogonaux, i.e. le produit scalaire $\theta^t\varphi = 0$ (d'après la définition 2.6, il existe un potentiel π tel que $\theta = S^t\pi$ où S est la matrice d'incidence de G, donc $\theta^t\varphi = (S^t\pi)^t\varphi = \pi^t(S\varphi) = 0$). Il est alors possible d'en déduire une nouvelle définition de la tension.

$$\theta \textit{ est une tension} \Leftrightarrow \forall \textit{ cycle } \gamma \textit{ de } G,\ \gamma^t\theta = 0 \tag{2.7}$$

Preuve:
($\Rightarrow$) θ est orthogonale à tout flot sur G et donc en particulier à tout cycle (le vecteur cycle représentant un flot).
($\Leftarrow$) Si l'égalité est vérifiée, un potentiel π peut être associé à θ. Dans le cas contraire, cela signifierait qu'il existe pour un arc $(x; y)$ une chaîne C de x à y pour laquelle $\sum_{u\in C^+}\theta_u - \sum_{u\in C^-}\theta_u \neq \theta_{(x;y)}$. Or si l'on considère le cycle γ formé de C et de $(x; y)$, on doit avoir $\sum_{u\in\gamma^+}\theta_u - \sum_{u\in\gamma^-}\theta_u = 0$, d'où la contradiction. □

Il n'est pas nécessaire de vérifier l'égalité pour tous les cycles du graphe pour confirmer une tension. En effet, si l'égalité est vérifiée pour une base de cycles B_γ, i.e. $\forall\, \gamma_i \in B_\gamma,\ \gamma_i^t\theta = 0$, alors toute combinaison linéaire, donc tout cycle $\gamma = \sum_{\gamma_i\in B_\gamma}\lambda_i\gamma_i$, vérifie $\gamma^t\theta = \sum_{\gamma_i\in B_\gamma}\lambda_i(\gamma_i^t\theta) = 0$. La définition suivante suffit donc à caractériser une tension.

$$\theta \textit{ est une tension } \Leftrightarrow \forall \textit{ cycle } \gamma \in B_\gamma,\ \gamma^t\theta = 0 \tag{2.8}$$

2.2.2.3. Base de cocycles

De manière analogue aux cycles, nous rappelons ici une manière simple d'obtenir une ***base de cocycles***. Considérons un arbre recouvrant $T = (X; U')$ du graphe G. En supprimant un arc u de l'arbre, on fait apparaître deux composantes connexes. Notons C_u la composante qui contient le noeud source de u et notons $\omega^u = \omega(C_u)$ le cocycle de C_u dans le graphe G. Si on considère l'ensemble des cocycles ω^u engendré en supprimant sé-

parément chaque arc u de T, on obtient un ensemble $B_\omega = \{\omega^{u_1}; \omega^{u_2} \ ... \ \omega^{u_p}\}$ de cocycles indépendants (car chaque cocycle ω^u possède un arc qu'aucun autre ne possède, c'est u). Il faut s'assurer maintenant que tout cocycle de G est une combinaison linéaire des cocycles de B_ω.

Considérons une tension θ. Soit $\theta' = \sum_{u \in U'} \theta_u \omega^u$. θ' est une combinaison linéaire de tensions donc une tension. La différence $\theta - \theta'$ est également une tension. Or, pour tout arc u de U', $\theta'_u = \theta_u$ puisque l'arc u n'apparaît que dans le cocycle ω^u. La tension $\theta - \theta'$ étant nulle pour tout arc de T, tous les noeuds de l'arbre et donc du graphe ont le même potentiel. Donc $\theta = \theta'$.

En conclusion, toute tension est une combinaison linéaire de la base B_ω. Un cocycle pouvant être considéré comme une tension, tout cocycle est une combinaison linéaire de B_ω. On remarque également que la base B_ω contient $n - 1$ cocycles (car T possède $n - 1$ arcs).

2.3. Modélisation sous forme de graphe temporel

2.3.1. Définition d'un graphe temporel

Un graphe $G = (X; U)$ est dit ***temporel*** lorsque ses noeuds représentent des événements et ses arcs des contraintes de précédence entre deux noeuds, i.e. l'arc $u = (x; y)$ signifie que l'événement modélisé par x doit se produire avant l'événement modélisé par y.

Un graphe temporel est également accompagné d'un vecteur d, appelé *durée*, qui associe à chaque arc $u \in U$ une durée d_u et d'un vecteur t, appelé *date*, qui associe à chaque noeud $x \in X$ une date t_x. Un graphe temporel peut également être muni d'un vecteur I, appelé *capacité*, qui associe à chaque arc $u \in U$ un ensemble I_u, indiquant les valeurs autorisées pour la durée d_u.

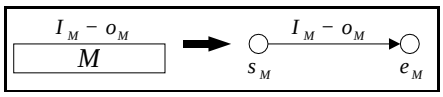


Figure 2.7: Modélisation d'un simple objet multimédia par un graphe temporel.

Un objet multimédia M de durée idéale o_M et d'ensemble de tolérance I_M sera donc représenté dans un graphe temporel par ses événements de début s_M et de fin e_M reliés par un arc indiquant la relation de précédence entre s_M et e_M (cf. figure 2.7) et portant toutes les informations associées à l'objet telles que la durée idéale o_u et l'ensemble de tolérance I_M.

En outre, un graphe temporel doit posséder un seul noeud source s et un seul noeud puits p, chacun représentant respectivement les événements de début et de fin du scénario complet modélisé. Il est possible de fixer la date de l'événement s à 0, la date de p représentera ainsi la durée totale de présentation du scénario. Et si l'on connaît le vecteur durée, on peut en déduire le vecteur date, et réciproquement.

A partir du principe qu'un arc représente une contrainte de précédence entre deux objets multimédia, la figure 2.8b modélise sous forme de graphe temporel un exemple de scénario illustré par la figure 2.8a (une flèche entre deux objets signifie que la fin de l'objet source coïncide avec le début de l'objet cible). Des simplifications peuvent être apportées à ce graphe temporel: deux événements liés par une contrainte de précédence de capacité $[0; 0]$ signifient qu'ils sont confondus (cf. figure 2.8c).

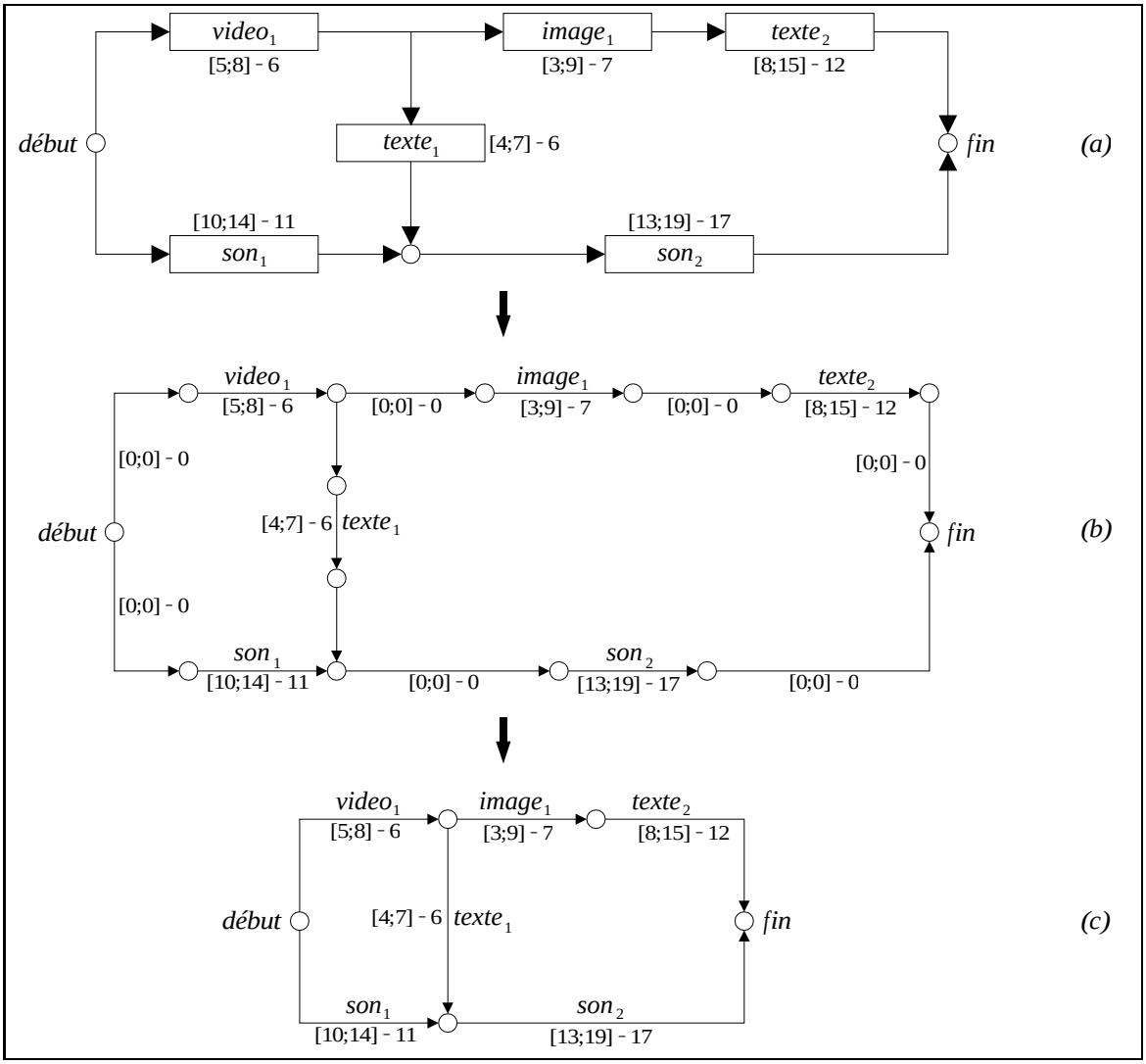


Figure 2.8: Un exemple de modélisation d'un scénario par un graphe temporel.

2.3.2. Relations temporelles modélisables

Il est relativement simple d'exprimer les relations d'Allen par un graphe temporel (cf. figure 2.9). Nous ne prenons pas en compte dans cette représentation les durées et les événements imprévisibles. Nous supposons que la planification doit établir une durée pour tous les objets multimédia et une date relative au début de la présentation du document pour tous les événements. En outre, la disjonction qui est préconisée sur les relations d'Allen n'est pas toujours représentable. Soulignons deux disjonctions modélisables importantes, nous les nommons $share\text{-}start$ et $share\text{-}end$. $A\ share\text{-}start\ B$ signifie que les objets A et B démarrent en même temps, ce qui s'exprime par $(A\ starts\ B)\ or\ (B\ starts\ A)$. $A\ share\text{-}end\ B$ signifie de façon similaire que A et B se terminent en même temps, et s'exprime par $(A\ finishes\ B)\ or\ (B\ finishes\ A)$. Comme le souligne [Alle91], les possibilités de disjonctions entre deux relations d'Allen sont au nombre de 2^{13}, mais il explique également que seulement 181 sont modélisables par une représentation sous forme d'événements comme la nôtre.

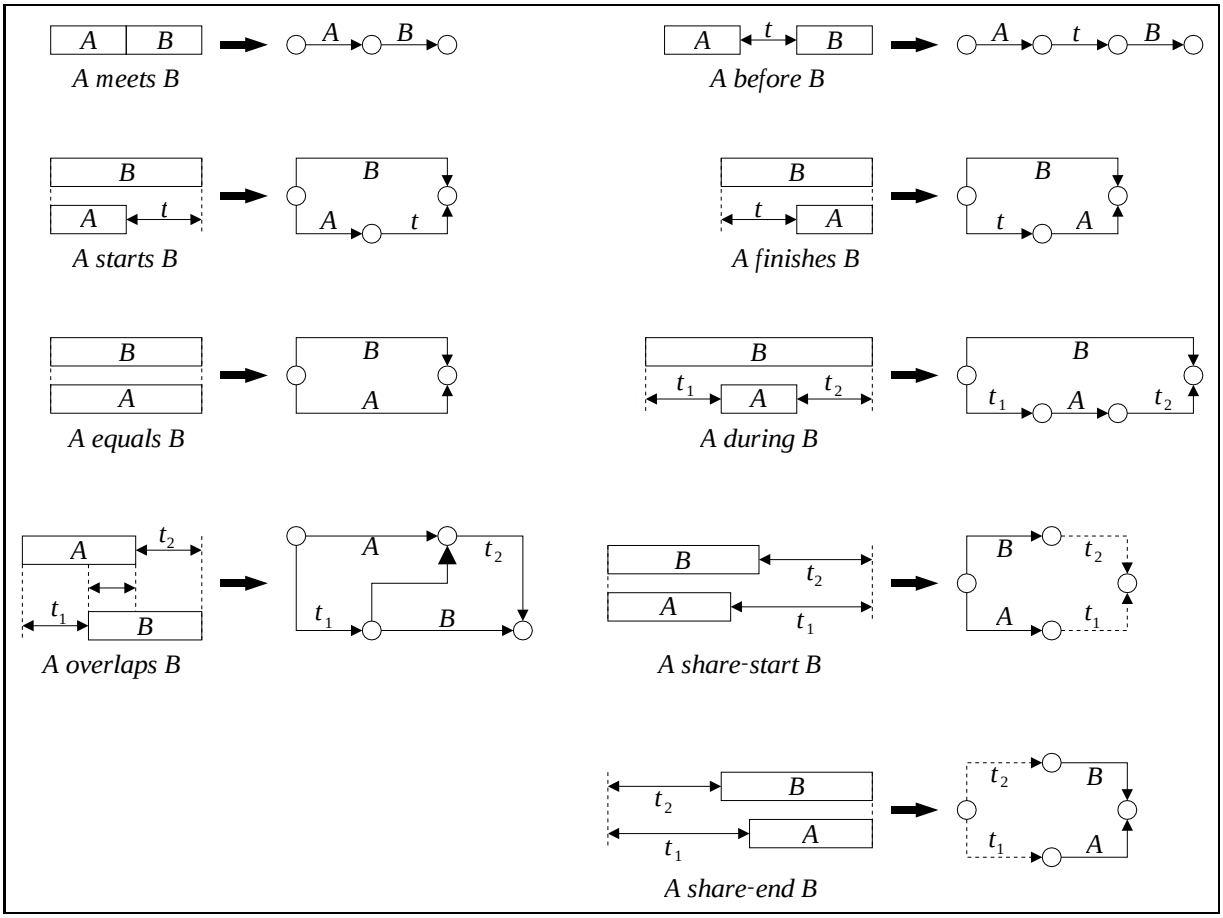


Figure 2.9: Modélisation des relations d'Allen par un graphe temporel.

2.4. Problèmes de tension

Nous allons montrer ici que les durées d_u portées par les arcs correspondent à la notion de tension définie à la section 2.2. Une fois établi le fait que les problèmes que nous tentons de résoudre sont des problèmes de tension, nous en exposons les deux principaux, la *tension compatible* et la *tension de coût minimal*, qui modélisent les problématiques de synchronisation hypermédia soulignées au chapitre 1.

2.4.1. Temps et tension

Supposons un graphe $G = (X; U)$ muni d'un vecteur durée d et d'un vecteur date t. La durée d_C d'une chaîne C entre deux noeuds x et y est définie par la somme $d_C = \sum_{u \in C^+} d_u - \sum_{u \in C^-} d_u$. Pour un chemin P du noeud x au noeud y, sa durée se résume à $d_P = \sum_{u \in P} d_u = t_y - t_x$. Notons qu'un chemin dans un graphe temporel représente une succession d'objets multimédia et la durée du chemin représente la durée totale de présentation de ces objets. La satisfaction des contraintes temporelles par une planification (i.e. un vecteur durée ou un vecteur date sur G) peut être exprimée de la manière suivante: pour chaque paire de noeuds x et y, tous les chemins entre x et y doivent avoir la même durée.

En d'autres termes, $\forall\, x, y \in X^2, \exists\, d_{(x;y)} \in \mathbb{R}$ telle que, pour tout chemin P de x à y, $d_P = d_{(x;y)}$. Tentons alors de prouver la propriété suivante.

Soit un graphe $G = (X;U)$ et un vecteur durée d. $\forall\, x, y \in X^2, \exists\, d_{(x;y)} \in \mathbb{R}$ telle que, pour tout chemin P de x à y, $d_P = d_{(x;y)}$, si et seulement si d est une tension. (2.9)

Preuve:
($\Leftarrow$) Soit Γ_G l'ensemble des cycles de G. d est une tension dans G, donc $\forall\, \gamma \in \Gamma_G$, $d_\gamma = \sum_{u\in\gamma^+} d_u - \sum_{u\in\gamma^-} d_u = 0$. Ainsi, un cycle γ formé de deux chemins P_1 et P_2 de x à y, e.g. $\gamma^- = P_1$ et $\gamma^+ = P_2$, vérifie $d_\gamma = d_{P_2} - d_{P_1} = 0$. Donc deux chemins de mêmes extrémités sont de même durée.
($\Rightarrow$) Notons d_x la durée des chemins de la source x_0 au noeud x (tous les chemins de mêmes extrémités sont de même durée). Soit γ un cycle de G, considérons i et $i+1$ deux noeuds consécutifs dans ce cycle. Il y a deux cas possibles:
- L'arc $(i;i+1) \in \gamma^+$. De la définition d'un graphe temporel, il est toujours possible de trouver un chemin P_{i+1} de x_0 à $i+1$ avec la durée d_{i+1}. Le chemin de x_0 à $i+1$ passant par i en utilisant l'arc $(i;i+1)$ du cycle γ est également de durée d_{i+1}, et vaut aussi $d_i + d_{(i;i+1)}$ (cf. figure 2.10). D'après l'hypothèse de départ, $d_{i+1} = d_i + d_{(i;i+1)}$.
- L'arc $(i+1;i) \in \gamma^-$. De manière analogue au premier cas, on trouve que $d_i = d_{i+1} + d_{(i+1;i)}$.
En résumé, on obtient l'égalité E_i: $d_i - d_{i+1} = \begin{cases} -d_u, \text{ si } u = (i;i+1) \in \gamma^+ \\ d_u, \text{ si } u = (i+1;i) \in \gamma^- \end{cases}$.
Numérotons les arcs du cycle γ consécutivement $1..k$. En sommant les égalités E_i, on obtient: $d_1 - d_{k+1} = \sum_{i=1..k} d_i - \sum_{i=1..k} d_{i+1} = \sum_{u\in\gamma^-} d_u - \sum_{u\in\gamma^+} d_u = d_\gamma$. Or les noeuds 1 et $k+1$ sont confondus, donc $d_\gamma = 0$. Le vecteur d est donc une tension. □

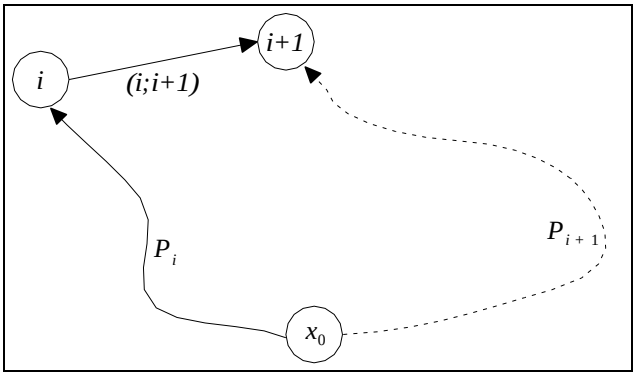


Figure 2.10: Illustration de la preuve.

En résumé, le vecteur durée d est une tension. En outre, pour tout arc $u = (x;y)$, la durée $d_u = t_y - t_x$, le vecteur date t est donc un potentiel. Etudions maintenant la relation entre certains problèmes de tension et la synchronisation hypermédia.

2.4.2. Problème de la tension compatible

On dit qu'une tension θ d'un graphe $G = (X;U)$ est ***compatible*** (ou réalisable) avec une capacité I si et seulement si, pour tout arc $u \in U$, $\theta_u \in I_u$. Le ***problème de la tension compatible*** consiste à déterminer une telle tension. Dans le contexte de la synchronisation hypermédia, cela signifie chercher une durée d_u pour chaque objet hypermédia u qui soit dans l'ensemble de tolérance I_u associé. La résolution de ce problème

permet également de répondre à la question: est-il possible de présenter le document en k secondes ? Il suffit pour cela d'ajouter un arc de capacité $[k;k]$ entre le noeud source et le noeud puits du graphe, et de tenter de déterminer une tension compatible sur ce nouveau graphe.

Au cours de notre étude sur des méthodes de résolution de ce problème de réalisabilité, nous nous intéresserons au ***problème de la tension maximale (ou minimale)***. Il s'agit, pour un arc donné u d'un graphe G muni d'une capacité I, de trouver la tension maximale (ou minimale) θ_u qu'il peut atteindre, θ restant compatible avec I. Là aussi, en ajoutant un arc de capacité $[0;+\infty]$ entre le noeud source et le noeud puits du graphe, il est possible de répondre à la question: quelle est la durée maximale (ou minimale) de présentation d'un document hypermédia ? Il suffit pour cela de déterminer la tension maximale (ou minimale) du nouvel arc.

2.4.3. Problème de la tension de coût minimal

L'un des principaux problèmes liés à la synchronisation hypermédia est tout de même de déterminer une planification de la meilleure qualité possible. Mais avant de tenter une résolution, il faut être capable de formaliser la notion de qualité d'une planification. Pour cela, nous avons choisi d'attribuer un coût (pour l'instant nous le supposerons quelconque) pour chaque arc en fonction de sa tension. Ce coût étant naturellement le plus faible lorsque la tension de l'arc est égale à sa ***tension idéale*** (i.e. sa durée idéale). Nous discutons à la section suivante des différents types de fonction de coût pertinents pour la synchronisation hypermédia.

En d'autres termes, pour chaque arc $u \in U$, on définit un ***coût*** sur un graphe $G = (X;U)$ comme étant un vecteur c, qui attribue à chaque arc u une fonction de coût $c_u : \mathbb{R} \longmapsto \mathbb{R}$. Le problème d'optimisation qui se pose alors est de trouver une tension compatible θ qui minimise la somme totale des coûts, i.e. $\sum_{u \in U} c_u(\theta_u)$. Ce problème est appelé le ***problème de la tension de coût minimal***.

2.5. Quantifier la qualité d'une présentation

Comme il l'a déjà été souligné, afin de déterminer la meilleure planification d'un document hypermédia, il est nécessaire d'en mesurer la qualité. Pour cela, nous retenons la solution qui consiste à attribuer à chaque arc u une fonction de coût c_u dépendante de la valeur de la tension θ_u, telle que $c_u(o_u) = 0$, i.e. le coût de la tension θ_u est nul si celle-ci est à sa valeur idéale o_u (i.e. l'objet multimédia est planifié à sa durée idéale). Naturellement, ce coût augmente à mesure que la tension θ_u s'éloigne de o_u. A travers la littérature, nous avons constaté trois types de coût pertinents pour représenter la qualité de la planification d'un document hypermédia.

2.5.1. Pénalité, coûts convexes linéaires par morceaux

La première idée consiste à affecter à un arc u un coût proportionnel à l'écart entre la tension θ_u et sa tension idéale o_u. Ce type de coût a été proposé et traité dans [Buch93b] et [Kim95]. Alors que [Kim95] propose un coût unitaire identique selon que la tension soit inférieure ou supérieure à la durée idéale (cf. figure 2.11a), [Buch93b] propose de différencier les deux cas (cf. figure 2.11b). [Kim95] considère même simplement la fonction $c_u(\theta_u) = |\theta_u - o_u|$. En résumé, ces approches similaires expriment la fonction de coût d'un arc u par un coût unitaire c_u^1 de diminution de la tension et un coût unitaire c_u^2 d'augmentation de la tension d'un arc u. Cette fonction s'écrit de la manière suivante.

$$c_u(\theta_u) = \begin{cases} c_u^1(o_u - \theta_u), \text{ si } a_u \leq \theta_u \leq o_u \\ c_u^2(\theta_u - o_u), \text{ si } o_u < \theta_u \leq b_u \end{cases}$$

2.5.2. Plus d'égalité, coûts convexes dérivables

La tension optimale obtenue avec le type de coût présenté précédemment peut produire une inégalité entre les objets multimédia (cf. [Kim95]). Autrement dit, certains objets peuvent subir une importante déformation alors que d'autres n'en subissent qu'une légère.

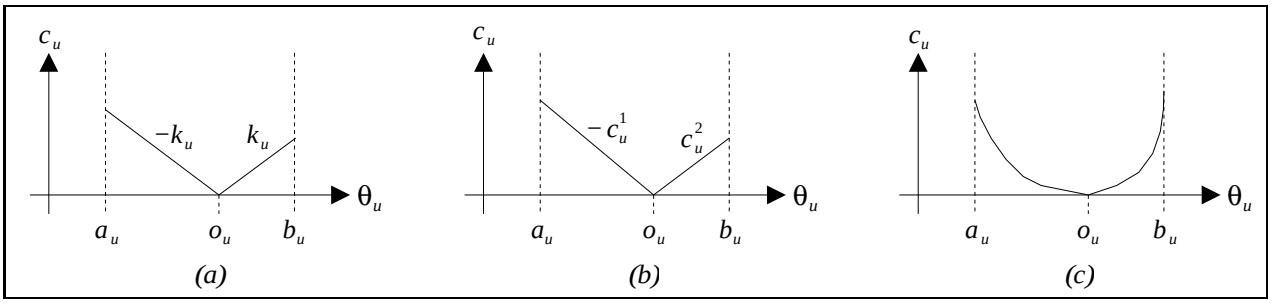


Figure 2.11: Exemples de coûts convexes pour mesurer la qualité d'une planifi cation hypermédia.

Afin d'équilibrer ces altérations, i.e. que chacun est à peu près la même déformation, [Kim95] propose une fonction de la forme $c_u(\theta_u) = (\theta_u - o_u)^2$ (cf. figure 2.11c). Nous nous intéresserons donc par la suite à déterminer une tension optimale avec des fonctions de coût convexes dérivables quelconques.

2.5.3. Comptabilisation des objets touchés par une déformation

Un dernier type de coût présenté dans [Medi02] semble également illustrer la qualité d'un document hypermédia. La déformation de la durée d'un objet multimédia peut entraîner un temps d'exécution important au moment de la présentation du document. Une mesure de qualité qui intéresse donc les auteurs de documents hypermédia est simplement le nombre d'objets qui ne sont pas planifiés à leur durée idéale. Autrement dit, la fonction de coût c_u est de la forme suivante.

$$c_u(\theta_u) = \begin{cases} 1, \text{ si } \theta_u \neq o_u \\ 0, \text{ si } \theta_u = o_u \end{cases}$$

2.6. Conclusion

Nous avons exposé ici une modélisation possible des relations temporelles d'objets multimédia à synchroniser. Nous avons supposé les ensembles de tolérance quelconques. Dans notre étude, nous nous limiterons pour l'instant à des intervalles de tolérance continu (pour tout arc u, un intervalle $[a_u; b_u] \in \mathbb{R}$), simplement parce que les problèmes sont plus faciles à aborder dans le domaine continu que dans le domaine discret, où la combinatoire peut rapidement devenir trop importante. Nous choisissons également d'aborder l'étude du problème d'optimisation avec des fonctions convexes linéaires par morceaux ou simplement dérivables. Nous n'étudierons pas pour l'instant la minimisation du nombre d'objets touchés par une déformation dans une planification hypermédia. Cette modélisation est également de nature discrète.

PARTIE II - TENSION DE COÛT MINIMAL

Les problèmes de tension de coût minimal sont des modélisations possibles des problèmes de synchronisation hypermédia. Nous proposons ici des algorithmes pour résoudre certains de ces problèmes. L'étude est effectuée tout d'abord sur des graphes quelconques, puis sur une classe particulière de graphes, les graphes série-parallèles, qui représentent des situations idéales pour les problèmes de synchronisation hypermédia. Nous proposons pour ces algorithmes et ces classes de graphes un comparatif à la fois théorique (étude de la complexité) et pratique (résultat de jeux d'essais).

AVANT-PROPOS

Le flot et la tension sont des composantes fortement couplées dans les problèmes de graphes. Cependant, alors que beaucoup d'attention est portée au flot, peu de travaux semblent s'intéresser aux problèmes de tension. Le flot permet en effet de modéliser de nombreux problèmes liés aux flux: télécommunication, transport de marchandises / de personnes... La tension est une notion beaucoup moins naturelle, mais permet néanmoins de modéliser quelques problèmes, plutôt liés à la planification de projets, au placement de dispositifs... (e.g. [Hadj96]). En outre, les méthodes d'optimisation de flot reposent très souvent sur un couplage des composantes flot et tension. Nous verrons au cours de notre étude que le flot joue le même rôle dans les problèmes de tension que la tension dans les problèmes de flot.

Dans la première partie, nous avons vu que les principaux problèmes de synchronisation hypermédia peuvent être modélisés comme des problèmes de tension, et plus précisément comme des problèmes de tension compatible et de tension de coût minimal. Nous allons donc tenter dans cette seconde partie de proposer différents algorithmes pour résoudre ces problèmes, discutant des avantages et défauts éventuels de chacun. Il ne faut pas perdre de vue que les méthodes que nous proposons doivent être rapides et ne doivent pratiquement pas dépasser la seconde pour permettre une réponse quasiment en temps réel.

Cet aspect est très important puisqu'il limite notre champs d'investigation. En effet, les graphes traités en synchronisation hypermédia sont relativement petits et ne dépassent pas à l'heure actuelle la centaine de noeuds. Il est alors impossible d'employer des algorithmes qui ont une bonne complexité théorique, mais qui révèlent leur efficacité sur des graphes de grande taille. Des algorithmes considérés moins efficaces en théorie peuvent tout à fait convenir, et s'avérer plus efficaces, sur des graphes de petite taille.

Les auteurs de documents hypermédia affirment actuellement qu'une centaine de noeuds suffit amplement pour les documents qu'ils souhaitent créer. Mais il est très difficile, même pour des experts, d'envisager les besoins et les possibilités futures, l'histoire de l'informatique est remplie d'exemples dans ce sens (le plus célèbre étant la phrase *"640 Ko of RAM is enough"*). Il est donc raisonnable de penser que si des outils efficaces de synchronisation hypermédia se répandent, la taille des documents manipulés augmentera et dépassera largement les besoins actuels. Il en existe déjà un exemple présenté dans [Vazi98] qui tente de synchroniser 10^4 objets dans un film de synthèse de 90 minutes. Il est donc important de s'intéresser tout de même au comportement théorique et pratique de nos algorithmes sur des graphes de taille conséquente.

Dans les deux premiers chapitres de cette seconde partie, nous nous intéressons aux problèmes de la tension compatible et de la tension minimale sur des graphes quelconques. Cependant, il est facile de s'apercevoir, en regardant notamment les relations d'Allen, que des graphes temporels issus d'une synchronisation hypermédia sont très organisés. Une classe de graphes semble très proche de cette structure: les graphes *série-parallèles*. Nous consacrons donc un dernier chapitre à l'étude du problème de tension de coût minimal sur cette classe de graphes. Cependant, les graphes issus d'une synchronisation hypermédia sont légèrement moins structurés que les graphes série-parallèles. Nous proposons une approche qui permet d'optimiser la tension d'un graphe *presque série-parallèle* en profitant tout de même de sa structure série-parallèle.

CHAPITRE 3
TENSION COMPATIBLE

Avant de chercher à optimiser une tension dans un graphe, il faut déjà être capable de trouver efficacement une ***tension compatible***. Nous rappelons tout d'abord ce problème, il s'agit de trouver une tension θ sur un graphe $G = (X; U)$, avec $m = |U|$ et $n = |X|$, telle que pour tout arc u, $a_u \leq \theta_u \leq b_u$, où a_u et b_u sont des valeurs réelles ou entières. Dans sa thèse, M. Hadjiat (cf. [Hadj96]) propose deux manières de résoudre ce problème. En fait, les algorithmes reposent sur deux façons différentes de trouver la tension maximale pour un arc donné, l'une se base sur des plus courts chemins, l'autre sur la recherche de cocycles. Nous proposons ici de réviser ces approches en abordant directement le problème de la tension compatible.

3.1. Recherche d'une tension maximale sur un arc

Nous nous intéressons tout d'abord au ***problème de la tension maximale***. Soit un arc v donné du graphe G, il faut trouver la valeur maximale de la tension θ_v de l'arc, sachant que θ doit être compatible. En d'autres termes, nous cherchons:

$$\hat{\theta}_v = \max_{\substack{\theta \text{ tension} \\ \text{compatible}}} \theta_v$$

3.1.1. Algorithme basé sur le plus court chemin

Nous rappelons ici l'algorithme proposé dans [Hadj96]. Il implique plusieurs propriétés très intéressantes et utiles pour la suite. Soit θ une tension compatible. Pour tout cycle γ, on a $\gamma^t\theta = 0$. Donc, pour tout cycle γ contenant l'arc v, on peut affirmer (en prenant comme sens de parcours du cycle le sens de l'arc v):

$$\theta_v = \sum_{u\in\gamma^-} \theta_u - \sum_{u\in\gamma^+\setminus\{v\}} \theta_u \leq \sum_{u\in\gamma^-} b_u - \sum_{u\in\gamma^+\setminus\{v\}} a_u$$

D'où la proposition suivante.

Pour toute tension θ compatible et pour tout cycle γ contenant l'arc v tel que $v \in \gamma^+$, on a: $\theta_v \leq \sum_{u\in\gamma^-} b_u - \sum_{u\in\gamma^+\setminus\{v\}} a_u$ *(3.1)*

La tension maximale de l'arc v est donc bornée par:

$$\hat{\theta}_v \leq \min_{\substack{\text{cycle } \gamma \\ v\in\gamma^+}} \{ \sum_{u\in\gamma^-} b_u - \sum_{u\in\gamma^+\setminus\{v\}} a_u \}$$

L'algorithme 3.2 présenté dans la section 3.1.2 démontre que cette borne est accessible, d'où la proposition suivante (une démonstration non constructive peut également être trouvée dans [Berg62]).

La tension maximale d'un arc v est égale à: $\min_{\substack{\text{cycle } \gamma \\ v\in\gamma^+}} \{ \sum_{u\in\gamma^-} b_u - \sum_{u\in\gamma^+\setminus\{v\}} a_u \}$ *(3.2)*

Considérons un cycle γ contenant $v = (x; y)$. Il est possible de former une chaîne C de x à y telle que $C = \gamma \setminus \{v\}$. Autrement dit, C est la chaîne formée en enlevant v du cycle γ. Appelons *capacité* de C la valeur $\sum_{u \in C^+} b_u - \sum_{u \in C^-} a_u$ (attention, dans la chaîne les arcs sont dans le sens opposé à celui qu'ils ont dans le cycle). Ainsi, la proposition 3.2 peut s'énoncer de la manière suivante.

La tension maximale sur l'arc $(x; y)$ est égale à la capacité minimale des chaînes allant de x à y. (3.3)

Trouver la tension maximale d'un arc consiste donc à trouver une chaîne de capacité minimale entre x et y. L'idée ici est de transformer le graphe G en un graphe G' de manière à ce que la recherche d'une chaîne de capacité minimale dans G se traduise par une recherche d'un plus court chemin dans G' pour laquelle de nombreux algorithmes existent déjà. La transformation est illustrée par la figure 3.1. Il s'agit de dédoubler chaque arc $u = (x; y)$ en deux arcs $u_1 = (x; y)$ et $u_2 = (y; x)$ où u_1 porte la distance b_u et u_2 la distance $-a_u$.

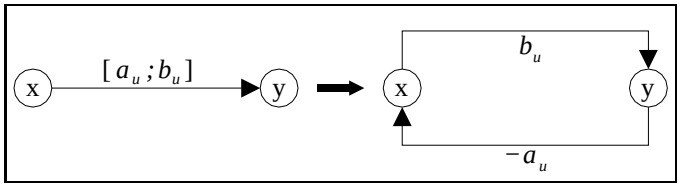


Figure 3.1: Transformation d'une chaîne maximale en un plus court chemin.

En résumé, G' se construit de la manière suivante, d étant la fonction de distance sur les arcs.

$$G' = (X; U')$$
$$U' = U \cup \{(y; x),\ (x; y) \in U\}$$
$$\forall\, (x; y) \in U',\ d_{(x;y)} = \begin{cases} b_{(x;y)},\ si\ (x; y) \in U \\ -a_{(y;x)},\ si\ (y; x) \in U \end{cases}$$

Certaines distances dans G' sont négatives. Il serait donc possible de rencontrer des circuits de longueur négative, i.e. la somme des distances des arcs sur le cycle serait négative. Il serait alors impossible de déterminer un plus court chemin. Dans notre cas, nous pouvons démontrer la proposition suivante.

Il existe une tension compatible sur le graphe G si et seulement si:
$\forall\, \gamma$ *cycle de* G, $\displaystyle\sum_{u \in \gamma^-} b_u - \sum_{u \in \gamma^+} a_u \geq 0$ (3.4)

Preuve:
($\Rightarrow$) Soit θ une tension compatible. D'après la proposition 3.1, pour tout cycle γ de G et un arc $v \in \gamma^+$, on a: $\theta_v \leq \sum_{u \in \gamma^-} b_u - \sum_{u \in \gamma^+ \setminus \{v\}} a_u$. Comme θ est compatible, on a $a_v \leq \theta_v$. D'où $a_v \leq \sum_{u \in \gamma^-} b_u - \sum_{u \in \gamma^+ \setminus \{v\}} a_u$, donc $\sum_{u \in \gamma^-} b_u - \sum_{u \in \gamma^+} a_u \geq 0$.
($\Leftarrow$) Supposons qu'il n'existe pas de tension compatible. En utilisant la définition de la tension à partir de la matrice d'incidence S et du potentiel π, cela signifie que le système suivant n'a pas de solution.

$$(S) \begin{cases} \theta = S^t \pi \\ a_u \leq \theta_u \leq b_u,\ \forall\, u \in U \end{cases}$$

D'après le théorème de Fourier-Motzkin (cf. [Mang69]), un système linéaire n'a pas de solution si et seulement s'il existe une dépendance linéaire des équations du système donnant une conséquence fausse (i.e. une relation linéaire des équations qui ne peut jamais être satisfaite). Pour notre système (S), cela signifie qu'il existe un flot φ et deux vecteurs $x \in (\mathbb{R}^+)^m$ et $y \in (\mathbb{R}^+)^m$ tels que $\varphi = y - x$ et $\sum_{u \in U} b_u y_u - \sum_{u \in U} a_u x_u < 0$. En utilisant la décomposition d'un flot sur une base de cycles Γ_G, il existe donc un cycle γ de G tel que $\sum_{u \in \gamma^-} b_u - \sum_{u \in \gamma^+} a_u < 0$. $\square$

La détection d'un cycle négatif lors de la recherche d'un plus court chemin indique alors qu'il n'existe pas de tension compatible dans le graphe G. Pour déterminer la tension maximale d'un arc, M. Hadjiat propose donc l'algorithme 3.1.

```
Algorithme 3.1: tensionMaximaleChemin(arc v=(x;y),graphe G=(X;U),réel θ̂_v).
U_1 ← ∅; U_2 ← ∅;

pour tout u=(s;t) ∈ U faire
 u_1 ← (s;t); d_{u_1} ← b_u; U_1 ← U_1 ∪ {u_1};
 u_2 ← (t;s); d_{u_2} ← −a_u; U_2 ← U_2 ∪ {u_2};
fin pour;

G' ← (X;U_1 ∪ U_2);
plusCourtChemin(x,y,G',d,C); /* Au retour C est un plus court chemin. */
θ̂_v ← longueur(C);
```

Cette procédure nécessite un algorithme de plus court chemin qui manipule des longueurs négatives et prend en compte les circuits, ce qui exclut les algorithmes de *label-setting* (cf. [Ahuj93]) comme celui de Bellman (à cause des circuits) et celui de Dijkstra (à cause des valeurs négatives). Dans [Hadj96], M. Hadjiat opte donc pour un algorithme de *label-correcting* comme celui de Bellman et Ford qui s'exécute en $O(nm)$ opérations (cf. [Ahuj93]) et qui détecte les cycles négatifs. Cet algorithme peut être trouvé dans l'annexe.

3.1.2. Algorithme basé sur le cocycle augmentant

L'algorithme présenté ici est extrait de [Hadj96]. Il introduit un mécanisme intéressant pour manipuler et modifier une tension. Considérons un cocycle $\omega(A)$ qui sépare l'ensemble de noeuds A du reste du graphe. Si on diminue de λ la tension des arcs du cocycle dans le sens A vers $X \setminus A$ et si on augmente de la même valeur λ la tension des arcs du cocycle dans le sens $X \setminus A$ vers A, les valeurs obtenues définissent toujours une tension. En effet, au niveau des potentiels, cela revient à augmenter de λ tous les potentiels de A: les tensions des arcs dans le sous-graphe engendré par A ne changent pas, seules les tensions sur le cocycle changent. (Notons qu'un phénomène similaire peut être constaté en modifiant le flot des arcs d'un cycle γ: en augmentant par exemple le flot des arcs de γ^+ et en diminuant le flot des arcs de γ^-, la conservation des flots est toujours assurée dans le graphe.)

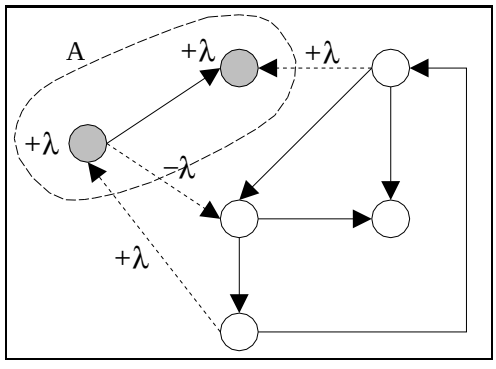


Figure 3.2: Modifi cation de la tension sur un cocycle.

Ainsi, de manière générale, si l'on veut augmenter la tension d'un arc v, il suffit de rechercher un cocycle pour lequel tous les arcs u du cocycle dans le même sens n'ont pas atteint leur capacité maximale (i.e. b_u) et tous les arcs u du cocycle dans le sens opposé n'ont pas atteint leur capacité minimale (i.e. a_u). [Hadj96] propose donc de rechercher un tel cocycle en utilisant l'algorithme basé sur le lemme de Minty avec la coloration suivante.

- Tout arc u tel que $a_u = \theta_u < b_u$ est noir.
- Tout arc u tel que $a_u < \theta_u = b_u$ est bleu.
- Tout arc u tel que $a_u < \theta_u < b_u$ est rouge.
- Tout arc u tel que $a_u = \theta_u = b_u$ est vert.

On s'aperçoit alors que si l'algorithme détecte un cocycle avec cette coloration, la tension de v peut être augmentée. Si un cycle γ est obtenu, alors il vérifie $\forall\, u \in \gamma^+ \setminus \{v\},\ \theta_u = a_u$ et $\forall\, u \in \gamma^- \setminus \{v\},\ \theta_u = b_u$, ce qui signifie que ce cycle correspond à une chaîne de capacité minimale pour v. [Hadj96] propose donc l'algorithme 3.2 pour déterminer une tension maximale pour un arc v donné.

Algorithme 3.2: **tensionMaximaleCocycle(tension** θ**,arc** v**,graphe** $G=(X;U)$**,réel** $\hat{\theta}_v$**).**

```
/* La tension θ doit être compatible. */
colorer les arcs de G;
cycleMinty(v,G,γ,ω);

tant que cocycle ω ≠ ∅ faire
 λ ← min{min_{u∈ω+}(b_u − θ_u), min_{u∈ω−}(θ_u − a_u)};
 θ ← θ + λω;
 colorer les arcs de G;
 cycleMinty(v,G,γ,ω);
fin tant que;

θ̂_v ← θ_v;
```

Après une augmentation de la tension sur un cocycle ω, au moins un arc u devient ***saturé***, i.e. $\theta_u = a_u$ ou $\theta_u = b_u$. Cet arc devient alors noir ou bleu et permet au prochain appel à la recherche d'un cocycle de marquer un noeud de plus. Ainsi un cycle est détecté au plus en n itérations. La partie principale de l'algorithme s'exécute donc en $O(nm)$ opérations, la partie qui établit une tension compatible n'étant pas comptée.

3.2. Recherche d'une tension compatible

[Hadj96] propose deux algorithmes pour trouver une tension compatible qui reposent sur les deux méthodes de recherche de tension maximale exposées précédemment. Nous présentons ici ces deux algorithmes avant de proposer deux variantes qui abordent directement le problème de la tension compatible.

3.2.1. Algorithme basé sur le plus court chemin

Ce premier algorithme repose sur la proposition suivante.

> *Soit θ une tension compatible pour laquelle $\theta_v = \hat{\theta}_v$ pour un arc donné v. Si C est la chaîne de capacité minimale reliant les deux extrémités de cet arc, on a:* (3.5)
> $\forall\, u \in C^+,\ \theta_u = b_u$ *et* $\forall\, u \in C^-,\ \theta_u = a_u$

Preuve:
θ étant compatible, $\theta_v = \sum_{u\in C^+} b_u - \sum_{u\in C^-} a_u$. Si v forme à lui seul la chaîne minimale, la preuve est évidente. Sinon, on a $\theta_v = \sum_{u\in C^+} \theta_u - \sum_{u\in C^-} \theta_u$. La soustraction des deux équations entraîne $\sum_{u\in C^+}(\theta_u - b_u) - \sum_{u\in C^-}(a_u - \theta_u) = 0$. θ étant compatible, tous les termes de la somme sont positifs ou nuls, et donc avec l'équation ils ne peuvent être que nuls. □

Cette proposition permet d'affirmer que si l'on restreint l'intervalle de tension sur les arcs d'une chaîne minimale C à $[a_u; a_u]$ pour tout arc $u \in C^-$ et à $[b_u; b_u]$ pour tout arc $u \in C^+$, alors il existe quand même une tension compatible sur le graphe. D'où l'algorithme 3.3 qui détecte une chaîne minimale, restreint les intervalles sur la chaîne et recommence jusqu'à ce que tous les intervalles de tension soient des singletons.

Algorithme 3.3: **tensionCompatibleChemin(graphe $G=(X;U)$,tension θ).**

```
tant que ∃v ∈ U tel que a_v ≠ b_v faire
 sélectionner un tel arc v = (x;y);
 trouver chaîne minimale C de x à y; /* Cf. algorithme 3.1. */

 pour tout u ∈ C+ faire a_u ← b_u;
 pour tout u ∈ C- faire b_u ← a_u;
 α ← Σ_{u∈P+} b_u − Σ_{u∈P-} a_u;
 a_v ← α;
 b_v ← α;
fin tant que;

pour tout u ∈ U faire θ_u ← a_u;
```

Si un cycle négatif est détecté lors de la première recherche de chaîne minimale, alors il n'existe pas de tension compatible. A chaque itération, au moins un intervalle est restreint à un singleton. Ainsi l'algorithme effectue au plus m recherches de chaîne minimale. Il s'exécute donc en $O(nm^2)$ opérations.

3.2.2. Algorithme basé sur le cocycle augmentant

Voici le second algorithme proposé dans [Hadj96]. Soit θ une tension quelconque. Considérons un arc v non compatible. Soit $\theta_v < a_v$, soit $\theta_v > b_v$. Dans le premier cas, on va tenter d'augmenter au maximum la tension θ_v et dans le second cas, on va tenter de diminuer au maximum θ_v.

Algorithme 3.4: **tensionCompatibleCocycle(graphe $G=(X;U)$,tension θ).**

```
θ ← 0;

tant que ∃v ∈ U tel que θ_v ∉ [a_v;b_v] faire
 sélectionner un tel arc v = (x;y);

 /* Modification des capacités pour rendre θ virtuellement compatible. */
 pour tout u ∈ U \ {v} faire
  si a_u ≤ θ_u ≤ b_u alors Γ_u ← [a_u;b_u];
  sinon si θ_u > b_u alors Γ_u ← [a_u;θ_u];
  sinon Γ_u ← [θ_u;b_u];
 fin pour;

 si θ_v < a_v alors /* La tension va être maximisée. */
  Γ_v ← [θ_v;a_v];
  tensionMaximaleCocycle(θ,v,G); /* Avec les intervalles Γ. */
  si θ_v < a_v alors arrêter; /* Pas de tension compatible. */
 sinon /* La tension va être minimisée. */
  Γ_(y;x) ← [−θ_v;−b_v];
  G' ← (X;U \ {v} ∪ {(y;x)});
  θ_(y;x) ← −θ_v;
  tensionMaximaleCocycle(θ,v,G'); /* Avec les intervalles Γ. */
  θ_v ← −θ_(y;x);
  si θ_v > b_v alors arrêter; /* Pas de tension compatible. */
 fin si;
fin tant que;
```

Pour cela, on utilise l'algorithme 3.2 qui maximise la tension d'un arc. Dans le premier cas, la méthode est appliquée directement sur $v = (x; y)$ dans le graphe G. Dans le second cas, G nécessite une modification: l'arc

v est inversé et ses valeurs de tension sont remplacées par leur opposé. Ainsi, l'algorithme de maximisation est appliqué sur $(y;x)$ pour maximiser $\theta_{(y;x)} = -\theta_v$ et donc minimiser θ_v.

La tension sera arbitrairement choisie nulle au début. Cependant, l'algorithme 3.2 suppose une tension θ compatible, ce qui est justement le but de l'algorithme présenté ici. Pour satisfaire à cette hypothèse, une transformation des bornes de tension est proposée qui rend θ virtuellement compatible à chaque itération et qui, lors de la maximisation ou de la minimisation, n'altère pas la compatibilité déjà acquise par d'autres arcs.

A chaque itération, un arc est rendu compatible. Ainsi l'algorithme exécute $O(m)$ fois l'algorithme de maximisation de la tension. L'algorithme s'exécute donc en $O(nm^2)$ opérations.

3.2.3. Variante de l'algorithme basé sur le cocycle augmentant

Les algorithmes précédents réduisent le problème de la tension compatible à des problèmes de tension maximale sur un arc, nous proposons ici d'aborder directement le problème de la tension compatible. Mais avant de parler de l'algorithme, nous démontrons la proposition suivante.

> *Soit γ un cycle, si $\sum_{u\in\gamma^-} b_u - \sum_{u\in\gamma^+} a_u = 0$ alors les seules tensions compatibles θ qui peuvent exister sur le graphe G vérifient $\theta_u = b_u$ pour tout $u \in \gamma^-$ et $\theta_u = a_u$ pour tout $u \in \gamma^+$.* (3.6)

Preuve:
D'après la proposition 3.1, pour tout arc $v \in \gamma^+$ on a: $\theta_v \leq \sum_{u\in\gamma^-} b_u - \sum_{u\in\gamma^+\setminus\{v\}} a_u = a_v$. Donc la seule tension compatible pour l'arc v est a_v. De la même manière, pour tout arc $v \in \gamma^-$, la seule tension compatible pour l'arc v est b_v. □

L'idée de l'algorithme ici est de considérer une tension quelconque (on peut choisir la tension nulle) qui n'est généralement pas compatible. Cela signifie que l'on peut trouver un arc v tel que $\theta_v < a_v$ ou $\theta_v > b_v$. Dans le premier cas, on cherchera un cocycle qui permet une augmentation de θ_v. Voici la coloration que nous proposons pour rechercher un tel cocycle.

- Tout arc u tel que $\theta_u \leq a_u$ et $a_u \neq b_u$ est noir.
- Tout arc u tel que $\theta_u \geq b_u$ et $a_u \neq b_u$ est bleu.
- Tout arc u tel que $a_u < \theta_u < b_u$ est rouge.
- Tout arc u tel que $a_u = \theta_u = b_u$ est vert.

Avec cette coloration, soit on trouve un cocycle qui permet d'augmenter la tension de v, soit on trouve un cycle, ce qui signifie qu'il n'existe pas de tension compatible.

Preuve:
Si aucun cocycle n'existe avec cette coloration, alors il existe un cycle γ contenant v. γ^+ contient des arcs noirs, avec $\theta_u \leq a_u$, et des arcs verts, avec $\theta = a_u$, donc tous les arcs de γ^- vérifient $\theta_u \leq a_u$. De la même manière, les arcs bleus et verts de γ^- vérifient tous $\theta_u \geq b_u$. Donc $0 = \sum_{u\in\gamma^-} \theta_u - \sum_{u\in\gamma^+} \theta_u \geq \sum_{u\in\gamma^-} b_u - \sum_{u\in\gamma^+} a_u$. Ainsi, soit $\sum_{u\in\gamma^-} b_u - \sum_{u\in\gamma^+} a_u < 0$, auquel cas il n'existe pas de tension compatible, soit $\sum_{u\in\gamma^-} b_u - \sum_{u\in\gamma^+} a_u = 0$. La proposition 3.6 induit alors que $\theta_v = a_v$, or $\theta_v < a_v$ pour l'instant et il n'existe pas de cocycle pour augmenter la tension de v. Donc cela signifie qu'il n'existe pas de tension compatible. □

Dans le cas où $\theta_v > b_v$, on cherchera un cocycle qui permet une diminution de θ_v. Voici la coloration (notée C_{sup}) que nous proposons pour rechercher un tel cocycle. Seulement les couleurs bleu et noir sont inversées par rapport à la première coloration (notée C_{inf}).

- Tout arc u tel que $\theta_u \geq b_u$ et $a_u \neq b_u$ est noir.
- Tout arc u tel que $\theta_u \leq a_u$ et $a_u \neq b_u$ est bleu.
- Tout arc u tel que $a_u < \theta_u < b_u$ est rouge.
- Tout arc u tel que $a_u = \theta_u = b_u$ est vert.

Avec cette coloration, soit on trouve un cocycle qui permet de diminuer la tension de v, soit on trouve un cycle, ce qui signifie qu'il n'existe pas de tension compatible (la preuve est similaire à celle du premier cas). Voici donc l'algorithme complet qui permet de déterminer une tension compatible sur le graphe G.

```
Algorithme 3.5: tensionCompatibleCocycleBis(graphe G=(X;U),tension θ).
θ ← 0;

tant que ∃v ∈ U tel que θ_v ∉ [a_v; b_v] faire
 sélectionner un tel arc v;

 si θ_v < a_v alors
  cycleMinty(v,G,γ,ω); /* Avec la coloration C_inf. */
  si ω = ∅ alors arrêter; /* Pas de tension compatible. */
  λ ← min{min_{u∈ω+}(b_u − θ_u), min_{u∈ω−}(θ_u − a_u)};
  θ ← θ + λω;
 sinon
  cycleMinty(v,G,γ,ω); /* Avec la coloration C_sup. */
  si ω = ∅ alors arrêter; /* Pas de tension compatible. */
  λ ← min{min_{u∈ω+}(θ_u − a_u), min_{u∈ω−}(b_u − θ_u)};
  θ ← θ − λω;
 fin si;
fin tant que;
```

Si on ne considère pas une méthode particulière de sélection de l'arc à traiter à chaque itération, l'algorithme s'exécute en $O(m^2A)$ opérations dans le cas où les bornes de tension sont entières et en $O(m^2A/\Delta)$ dans le cas où elles sont réelles, A étant la plus grande borne de tension en valeur absolue, i.e. $A = \max_{u \in U}\{|a_u|; |b_u|\}$, et Δ une borne inférieure de λ pour toute itération de l'algorithme. Une valeur possible de Δ est:

$$\Delta = \min_{\substack{\mu \in \mathbb{Z}^m \\ \nu \in \mathbb{Z}^m}} \{s = |\sum_{u \in U} \mu_u a_u + \sum_{u \in U} \nu_u b_u|,\ s > 0\}$$

Preuve:
Dans le cas entier, il est évident qu'à chaque itération, la tension de l'arc v est améliorée d'au moins 1. Donc, au maximum en A itérations, θ_v aura atteint soit a_u, soit b_u. L'algorithme appelle donc $O(mA)$ fois l'algorithme de recherche de cocycle.
Dans le cas réel, on s'aperçoit que la tension d'un arc est toujours une combinaison linéaire (avec des coefficients entiers) des bornes des intervalles de tension. Autrement dit, pour tout arc v et à toute itération de l'algorithme, il existe $\mu \in \mathbb{Z}^m$ et $\nu \in \mathbb{Z}^m$ tels que $\theta_v = \sum_{u \in U} \mu_u a_u + \sum_{u \in U} \nu_u b_u$. C'est très facile à vérifier dans la mesure où toutes les tensions sont à zéro au départ et qu'à chaque itération la valeur ajoutée ou retranchée à une tension est une différence entre une tension et une borne. Cela veut dire que la valeur λ à chaque itération est une combinaison linéaire. La plus petite valeur de λ s'exprime donc $\Delta = \min_{\mu \in \mathbb{Z}^m,\ \nu \in \mathbb{Z}^m}\{s = |\sum_{u \in U} \mu_u a_u + \sum_{u \in U} \nu_u b_u|,\ s > 0\}$. Cette valeur existe puisque l'ensemble dans lequel est recherché le minimum est dénombrable et majoré par 0. A chaque itération, la tension de l'arc v est améliorée d'au moins

Δ. Donc, au maximum en A/Δ itérations, θ_v aura atteint soit a_u, soit b_u. L'algorithme appelle ainsi mA/Δ fois l'algorithme de recherche de cocycle. □

En revanche, si on choisit dans l'algorithme de sélectionner un arc et de l'améliorer tant qu'il n'est pas compatible (cf. algorithme 3.6), on se ramène à l'algorithme 3.4 qui maximise la tension de chaque arc. Dans ce cas, au maximum n opérations sont nécessaires pour rendre un arc compatible (l'algorithme de recherche de cocycle marque un nouveau noeud à chaque itération), aussi bien dans le cas réel que dans le cas entier. L'algorithme s'exécute donc dans cette version en $O(nm^2)$ opérations.

Algorithme 3.6: **tensionCompatibleCocycle(graphe** $G=(X;U)$**,tension** θ**).**

```
θ ← 0;

tant que ∃v ∈ U tel que θ_v ∉ [a_v;b_v] faire
 sélectionner un tel arc v;

 tant que θ_v < a_v alors
  cycleMinty(v,G,γ,ω); /* Avec la coloration C_inf. */
  si ω = ∅ alors arrêter; /* Pas de tension compatible. */
  λ ← min{min_{u∈ω+}(b_u − θ_u), min_{u∈ω−}(θ_u − a_u)};
  θ ← θ + λω;
 fin tant que;

 tant que θ_v > b_v alors
  cycleMinty(v,G,γ,ω); /* Avec la coloration C_sup. */
  si ω = ∅ alors arrêter; /* Pas de tension compatible. */
  λ ← min{min_{u∈ω+}(θ_u − a_u), min_{u∈ω−}(b_u − θ_u)};
  θ ← θ − λω;
 fin tant que;
fin tant que;
```

3.2.4. Un autre algorithme basé sur le plus court chemin

Reprenons ici quelques résultats sur les ***problèmes de plus court chemin*** que l'on peut retrouver dans [Ahuj93]. Supposons que l'on cherche la plus courte distance entre un noeud s et tous les autres noeuds du graphe G. Notons d la fonction qui associe à chaque arc u une longueur. Voici les conditions nécessaires et suffisantes, appelées ***conditions d'optimalité du plus court chemin***, qui prouvent qu'une distance est la plus courte.

> *Soit π une fonction qui associe une valeur à chaque noeud du graphe G. Si pour tout noeud x, π_x est la distance d'un chemin de s à x, alors π_x représente la plus courte distance entre s et x si et seulement si $\forall\,(x;y) \in U,\ \pi_y - \pi_x \leq d_{(x;y)}$.* (3.7)

Preuve:
($\Rightarrow$) Considérons un arc $(x;y)$. Soit le plus court chemin entre s et y passe par $(x;y)$, soit le plus court chemin entre s et y est un chemin C ne contenant pas $(x;y)$. Dans le premier cas, la plus courte distance de s à y est π_y qui vaut également $\pi_x + d_{(x;y)}$ si on considère de passer par x. La condition d'optimalité est alors vérifiée. Dans le second cas, le plus court chemin de s à y passant par x a une longueur de $\pi_x + d_{(x;y)}$ qui est forcément supérieure à la longueur du chemin C, autrement dit π_y. Là aussi, la condition d'optimalité est vérifiée.
($\Leftarrow$) Considérons un chemin C de s à un noeud x. Supposons que ce chemin ait une longueur inférieure à π_x. Cela signifie que $\sum_{(x;y)\in C} d_{(x;y)} < \pi_x$. Cependant, les conditions d'optimalité assurent que $\forall\,(x;y) \in U,\ \pi_y - \pi_x \leq d_{(x;y)}$. Donc, $\sum_{(x;y)\in C} d_{(x;y)} \geq \sum_{(x;y)\in C}(\pi_y - \pi_x) = \pi_x - \pi_s = \pi_x$. D'où la contradiction. □

Une petite remarque: s'il n'existe pas de chemin entre le noeud s et un noeud x, alors $\pi_x = +\infty$. Revenons maintenant à notre problème de tension compatible. Nous proposons de transformer le graphe G en un graphe

$G' = (X \cup \{s\}; U_1 \cup U_2 \cup U_s)$ où chaque arc u est dédoublé en deux arcs $u_1 = (x; y)$ et $u_2 = (y; x)$, u_1 porte la distance $d_{u_1} = b_u$ et u_2 porte la distance $d_{u_2} = -a_u$ (cf. figure 3.1). Notons U_1 l'ensemble des arcs u_1 et U_2 l'ensemble des arcs u_2. Un noeud source s est également ajouté, il est connecté à tous les autres noeuds par un chemin (pour satisfaire à la première partie des conditions d'optimalité). Pour cela il faut ajouter un ensemble U_s d'arcs de longueur infinie qui relient s à tous les noeuds de degré entrant nul.

Si on cherche maintenant la plus courte distance entre le noeud s et tous les autres noeuds, on obtient le vecteur π des plus courtes distances qui satisfait les conditions d'optimalité 3.7 qui peuvent aussi s'écrire:

$$\forall (x; y) \in U_1, \pi_y - \pi_x \leq b_{(x;y)}$$
$$\forall (y; x) \in U_2, \pi_x - \pi_y \leq -a_{(x;y)}$$

Autrement dit, $a_{(x;y)} \leq \pi_y - \pi_x \leq b_{(x;y)}$. Si on pose la tension θ associée au potentiel π, on a alors $a_{(x;y)} \leq \theta_{(x;y)} \leq b_{(x;y)}$, θ est donc une tension compatible.

Algorithme 3.7: **tensionCompatibleCheminBis(graphe** $G = (X; U)$**,tension** θ**)**.

```
U1 ← ∅; U2 ← ∅; Us ← ∅;

pour tout u = (x; y) ∈ U faire
 u1 ← (x; y); du1 ← bu; U1 ← U1 ∪ {u1};
 u2 ← (y; x); du2 ← −au; U2 ← U2 ∪ {u2};
fin pour;

pour tout x ∈ X tel que d⁻x = 0 faire
 d(s;x) ← +∞;
 Us ← Us ∪ {(s; x)};
fin pour;

G' ← (X ∪ {s}; U1 ∪ U2 ∪ Us);
plusCourtChemins(s,G',d,π);
/* Au retour π associe à chaque noeud sa plus courte distance à s. */
pour tout u = (x; y) ∈ U faire θu ← πy − πx;
```

Pour résoudre ce problème des plus courtes distances, nous proposons l'algorithme 3.7 qui utilise la même méthode de résolution de plus court chemin que l'algorithme 3.1, c'est-à-dire la méthode de *label-correcting* de Bellman et Ford que l'on retrouve dans [Ahuj93] et qui est également disponible dans l'annexe. De la même manière, la détection d'un circuit de longueur négative dans G' signifie qu'il n'y a pas de tension compatible dans G (grâce à la proposition 3.4 et au fait que les arcs de U_s rajoutés n'induisent pas de nouveau circuit). L'algorithme de Bellman et Ford s'exécute en $O(nm)$ opérations donc notre algorithme s'exécute également en $O(nm)$ opérations.

3.3. Conclusion

En résumé, le tableau 3.1 récapitule les différents algorithmes pour trouver une tension compatible avec leur complexité dans le cas de bornes de tension réelles ou entières.

Méthode	Approche	Complexité	
		Bornes réelles	Bornes entières
Cocycle augmentant (3.5)	Directe (sans sélection particulière)	$O(m^2\, A/\varepsilon)$	$O(m^2\, A)$
Plus court chemin (3.3)	Tension maximale	$O(nm^2)$	$O(nm^2)$
Cocycle augmentant (3.4 ou 3.6)	Tension maximale ou directe (avec sélection spécifique)	$O(nm^2)$	$O(nm^2)$
Plus court chemin (3.7)	Directe	$O(nm)$	$O(nm)$

Tableau 3.1: Complexité des algorithmes de tension compatible.

Les algorithmes sont classés en fonction de leur efficacité théorique, de la moins bonne à la meilleure. Nous proposons également une campagne de tests numériques afin d'évaluer l'efficacité pratique des différentes méthodes. Pour connaître en détail comment ont été dirigés ces essais (méthode de génération des problèmes, compilateur utilisé...), le lecteur peut consulter l'annexe. Nous précisons seulement ici que les problèmes générés ont des bornes de tension entières. Pour les méthodes de cocycle augmentant, le nombre d'itérations est le nombre de recherches de cocycle effectuées. Les temps de calcul sont exprimés en secondes sur une machine RISC-6000 à 160 MHz.

Dimension graphe		Cocycle augmentant (3.5)		Plus court chemin (3.3)	Cocycle augmentant (3.4 ou 3.6)		Plus court chemin (3.7)
Noeuds	Arcs	Itérations	Temps	Temps	Itérations	Temps	Temps
50	200	576	0,15	0,16	96	0,04	0,01
50	400	760	0,50	0,55	127	0,14	0,03
100	400	1326	0,64	0,65	204	0,19	0,03
100	800	1907	2,4	2,6	273	0,65	0,07
500	2000	7819	19	28,4	1065	5,6	0,22
500	4000	7756	57,7	134,5	980	16,4	0,53
1000	4000	16363	95,7	153,7	2078	25	0,54
1000	8000	14649	220,4	656	1465	54,2	1,2

Tableau 3.2: Résultats numériques pour les algorithmes de tension compatible, influence de la dimension du graphe.

Le tableau 3.2 montre le temps de calcul de chaque méthode pour différentes tailles de graphe (avec $A = 1000$). On s'aperçoit que la méthode de cocycle augmentant 3.5 est plus efficace que la méthode de plus court chemin 3.3, ce qui n'est pas très étonnant car sa complexité est fonction de A qui est très faible dans cette première série de tests.

Tension maximale	Cocycle augmentant (3.5)		Plus court chemin (3.3)	Cocycle augmentant (3.4 ou 3.6)		Plus court chemin (3.7)
	Itérations	Temps	Temps	Itérations	Temps	Temps
1000	9292	42,7	70,9	972	10,8	0,37
10000	23844	113,4	72,2	1708	19,8	0,37
100000	29367	138,4	72,2	1596	19,5	0,36

Tableau 3.3: Résultats numériques pour les algorithmes de tension compatible, influence de l'échelle des tensions.

Il est donc intéressant de voir le comportement des méthodes en fonction de la valeur de A. Le tableau 3.3 montre le temps de calcul de chaque méthode pour différentes valeurs de la borne maximale de tension A sur les arcs (avec $n = 500$ et $m = 3000$). Comme prévu, la méthode de cocycle augmentant 3.5 varie en fonction de A alors que les autres méthodes sont stables. Le décalage pour $A = 1000$ est simplement dû au fait que A est trop petit et qu'il "facilite" la résolution du problème (en effet, en valeurs entières, les possibilités de tension compatible sont trop réduites voire uniques pour certains arcs).

Classement pratique		Classement théorique	
Algorithme	Vitesse	Algorithme	Complexité
Cocycle augmentant (3.5)	384,4	Cocycle augmentant (3.5)	$O(nmA)$
Plus court chemin (3.3)	200,6	Plus court chemin (3.3)	$O(nm^2)$
Cocycle augmentant (3.4 ou 3.6)	54,2	Cocycle augmentant (3.4 ou 3.6)	$O(nm^2)$
Plus court chemin (3.7)	1	Plus court chemin (3.7)	$O(nm)$

Tableau 3.4: Classement théorique et pratique des algorithmes de tension compatible.

Enfin le tableau 3.4 classe les méthodes de la moins efficace à la plus efficace, d'un point de vue pratique, et rappelle le classement d'un point de vue théorique. Un rapport des vitesses d'exécution est effectué par rapport à l'algorithme le plus rapide. Il est calculé à partir de la dernière ligne du tableau 3.3 qui nous semble la situation la plus extrême (dimension du graphe et échelle de tension importantes). Bien évidemment, ces résultats numériques dépendent énormément de la manière de programmer les méthodes, mais nous nous sommes efforcés d'implémenter au mieux et de la même manière chaque méthode afin de réduire le plus possible ce genre de biais.

CHAPITRE 4
TENSION DE COÛT MINIMAL

Nous nous intéressons ici au problème de la ***tension de coût minimal***. Il s'agit de trouver une tension θ compatible ($\forall\, u \in U,\ a_u \leq \theta_u \leq b_u$) dans un graphe $G = (X;U)$, avec $m = |U|$ et $n = |X|$. A chaque arc u est associée une fonction de coût $c_u : \mathbb{R} \longmapsto \mathbb{R}$ qui, en fonction de sa tension θ_u, impute un coût à l'arc u. L'objectif est de minimiser la somme de ces coûts, i.e. $\min \sum_{u \in U} c_u(\theta_u)$. Dans un premier temps, nous considérons des coûts convexes linéaires par morceaux (cf. figure 2.11b). Ensuite, nous nous intéressons à des coûts convexes dérivables (cf. figure 2.11c). Dans ce chapitre nous ne considérons pas de structure particulière du graphe. Le chapitre suivant sera consacré à une classe particulière de graphes, les graphes *série-parallèles*, beaucoup plus proche des problèmes de synchronisation hypermédia.

Dans un premier temps, nous montrons comment le problème avec des coûts linéaires par morceaux peut être ramené très simplement à un problème avec des coûts linéaires. Cependant, la taille du graphe résultant devient trop importante pour que l'utilisation d'un algorithme connu sur cette transformation soit efficace en pratique.

Dans le cas de coûts linéaires par morceaux, nous proposons tout d'abord de modéliser le problème sous la forme d'un programme linéaire. Ensuite, nous étudions les conditions d'optimalité du problème et rappelons la notion de *conformité* (*kilter*). Deux approches se présentent naturellement pour résoudre le problème. La première consiste à s'inspirer d'un algorithme de flot de coût minimal et à l'adapter à la tension de coût minimal. Pour cela, nous reprenons la méthode proposée dans [Hadj96] pour concevoir deux algorithmes de *mise à conformité* (*out-of-kilter*), l'un pour des coûts linéaires par morceaux, l'autre pour des coûts dérivables. La seconde approche consiste à transformer le problème de tension de coût minimal en un problème de flot de coût minimal et à résoudre ce dernier avec un algorithme connu très efficace, en l'occurrence un algorithme de *mise à l'échelle des coûts* (*cost-scaling*), nous exploitons ici les résultats de l'article [Ahuj99a] pour des coûts linéaires par morceaux.

4.1. Coûts linéaires par morceaux et coûts linéaires

Nous nous intéressons tout d'abord à des coûts linéaires par morceaux, comme exposé dans le chapitre 2 (cf. figure 2.11b). La fonction de coût d'un arc u est définie dans un intervalle $[a_u; b_u]$ et est nulle pour la tension o_u, un coût unitaire c_u^1 est imputé si $\theta_u < o_u$ et de la même manière, un coût unitaire c_u^2 est imputé si $\theta_u > o_u$. La fonction de coût c_u d'un arc u s'écrit donc:

$$c_u(\theta_u) = \begin{cases} c_u^1(o_u - \theta_u), \text{ si } a_u \leq \theta_u \leq o_u \\ c_u^2(\theta_u - o_u), \text{ si } o_u < \theta_u \leq b_u \end{cases}$$

Le problème de trouver une tension de coût minimal avec ce genre de fonction de coût peut être transformé en un problème de tension de coût minimal avec des coûts linéaires $c'_u(\theta_u) = \lambda_u \theta_u$ où $\lambda_u \in \mathbb{R}$ et c'_u est définie sur un intervalle $[a'_u; b'_u]$ pour tout arc u du nouveau problème. La transformation est illustrée par la figure 4.1.

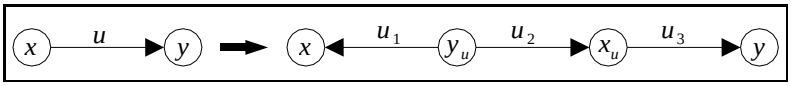


Figure 4.1: Transformation de coûts linéaires par morceaux en coûts linéaires.

Elle consiste à transformer le graphe $G = (X; U)$ en un graphe $G' = (X'; U')$ en représentant simplement chaque arc $u = (x; y)$ de G (avec un coût linéaire par morceaux) par trois arcs $u_1 = (y_u; x)$, $u_2 = (y_u; x_u)$ et $u_3 = (x_u; y)$ de G' (avec des coûts linéaires) de la manière suivante.

$$\begin{array}{l} u_{u_1}\text{: } a'_{u_1} = 0,\ \ b'_{u_1} = o_u - a_u,\ \lambda_{u_1} = c^1_u \\ u_{u_2}\text{: } a'_{u_2} = o_u,\ b'_{u_2} = o_u,\ \ \ \ \lambda_{u_2} = 0 \\ u_{u_3}\text{: } a'_{u_3} = 0,\ \ b'_{u_3} = b_u - o_u,\ \lambda_{u_3} = c^2_u \end{array}$$

Ainsi, trouver une tension de coût minimal dans le graphe G revient à trouver une tension de coût minimal dans le graphe G'.

Preuve:
D'après la figure 4.1, pour tout arc $u = (x; y)$ de G, on a, dans le graphe G', $\theta_{(x;y)} = -\theta_{u_1} + \theta u_2 + \theta_{u_3}$. D'après les intervalles de tension définis précédemment sur G', il est facile de vérifier que $a_u \leq \theta_{(x;y)} \leq b_u$ dans G', qui est le même intervalle que dans G.
Soit le coût $c_{(x;y)} = c_{u_1}(\theta_{u_1}) + c_{u_2}(\theta_{u_2}) + c_{u_3}(\theta_{u_3}) = c^1_u\theta_{u_1} + c^2_u\theta_{u_3}$ dans G'. Posons $\alpha = \theta_{u_1}$ et $\beta = \theta_{u_3}$, $c_{(x;y)} = c^1_{(x;y)}\alpha + c^2_{(x;y)}\beta$. Si la tension est optimale, θ_{u_1} et θ_{u_3} ne peuvent pas être tous les deux non nuls. En effet, supposons que $\theta_{(x;y)} < o_u$, $\theta_{u_1} \neq 0$ et $\theta_{u_3} \neq 0$, alors $\alpha > \beta > 0$. Maintenant si on choisit $\theta_{u_1} = \alpha - \beta$ et $\theta_{u_3} = 0$, alors $\theta_{(x;y)}$ ne change pas et la tension sur G' reste compatible, mais le coût $c'_{(x;y)} = c_{u_1}(\theta_{u_1}) + c_{u_2}(\theta_{u_2}) + c_{u_3}(\theta_{u_3}) = c^1_{(x;y)}(\alpha - \beta) < c_{(x;y)}$, ce qui signifie que l'ancienne tension n'était pas optimale. Le même raisonnement peut être fait dans le cas où $\theta_{(x;y)} > o_u$. On s'aperçoit alors que le coût $c_{(x;y)}$ est défini de la même manière dans G et dans G' quand le coût est minimal. □

Donc les algorithmes proposés pour des coûts linéaires, dans [Hadj96] par exemple, peuvent être utilisés directement pour résoudre le problème. Cependant, pour un graphe G à m arcs et n noeuds, le graphe G' associé a $3m$ arcs et $n + 2m$ noeuds. En regardant de plus près la complexité des algorithmes connus pour des coûts linéaires, cette transformation n'est pas exploitable directement en pratique: les graphes deviennent trop grands et les algorithmes perdent rapidement de leur efficacité. Nous cherchons donc à adapter ces algorithmes pour qu'ils manipulent directement des coûts linéaires par morceaux tout en gardant leur efficacité.

4.2. Modélisation sous forme de programme linéaire

En considérant des coûts linéaires par morceaux comme exposé dans le chapitre 2, il est possible de modéliser le problème sous la forme d'un programme linéaire. De manière générale, le problème peut s'écrire sous la forme suivante.

$$\begin{cases} \min \sum_{u \in U} c_u(\theta_u) \\ \theta \text{ une tension} \\ a_u \leq \theta_u \leq b_u,\ \forall\, u \in U \end{cases}$$

Bien entendu, cette forme n'est a priori pas linéaire. Dans la section précédente, nous avons vu comment exprimer la fonction objectif sous forme linéaire, en introduisant pour chaque arc u, deux variables θ^1_u et θ^2_u telles que:

$$\begin{array}{l} 0 \leq \theta^1_u \leq o_u - a_u \\ 0 \leq \theta^2_u \leq b_u - o_u \\ \theta_u = o_u - \theta^1_u + \theta^2_u \end{array}$$

Le coût s'exprime alors:

$$\min \sum_{u \in U} (c_u^1 \theta_u^1 + c_u^2 \theta_u^2)$$

Il reste donc maintenant à exprimer la tension sous une forme linéaire. Dans le chapitre 2, nous avons vu deux définitions lors de l'introduction aux graphes: l'une basée sur la matrice d'incidence et l'autre sur une base de cycles. Toutes les deux s'expriment sous forme linéaire. Ces approches ont déjà été abordées, la première dans [Buch93b] et [Hadj96], et la seconde dans [Kim95]. Ce sont d'ailleurs les principales solutions traitées dans la littérature sur la synchronisation hypermédia. Nous exposons ici les deux modèles en proposant une comparaison pratique des deux programmes.

4.2.1. Modèle basé sur la matrice d'incidence

La première définition de la tension s'exprime à l'aide de la matrice d'incidence S de G: $S\pi = \theta$ ou $\forall\, u = (x;y) \in U,\ \pi_y - \pi_x = \theta_u$. Le problème peut donc être représenté par le programme suivant.

$$(P_1) \begin{cases} \min \sum_{u \in U} (c_u^1 \theta_u^1 + c_u^2 \theta_u^2) & \\ \\ \pi_j - \pi_i = o_{(i;j)} - \theta^1_{(i;j)} + \theta^2_{(i;j)},\ \forall\, (i;j) \in U & \textit{(a)} \\ \theta_u^1 \leq o_u - a_u,\ \forall\, u \in U & \textit{(b)} \\ \theta_u^2 \leq b_u - o_u,\ \forall\, u \in U & \textit{(c)} \\ \\ \theta_u^1 \geq 0,\ \theta_u^2 \geq 0,\ \forall\, u \in U & \\ \pi_x \geq 0,\ \forall\, x \in X & \end{cases} \quad (4.1)$$

Ce programme contient $2m + n$ variables et $3m$ contraintes.

4.2.2. Modèle basé sur une base de cycles

La seconde définition de la tension s'exprime à l'aide d'une base de cycles Γ_G de G: $\Gamma_G \theta = 0$ ou $\forall\, \gamma \in \Gamma_G,\ \sum_{u \in \gamma^+} \theta_u - \sum_{u \in \gamma^-} \theta_u = 0$. Le problème peut donc être représenté par le programme suivant.

$$(P_2) \begin{cases} \min \sum_{u \in U} (c_u^1 \theta_u^1 + c_u^2 \theta_u^2) & \\ \\ \theta_u = o_u - \theta_u^1 + \theta_u^2,\ \forall\, u \in U & \textit{(a)} \\ \sum_{u \in \gamma^+} \theta_u - \sum_{u \in \gamma^-} \theta_u = 0,\ \forall\, \gamma \in \Gamma_G & \textit{(b)} \\ \theta_u^1 \leq o_u - a_u,\ \forall\, u \in U & \textit{(c)} \\ \theta_u^2 \leq b_u - o_u,\ \forall\, u \in U & \textit{(d)} \\ \\ \theta_u^1 \geq 0,\ \theta_u^2 \geq 0,\ \forall\, u \in U & \end{cases}$$

Les variables θ_u peuvent être éliminées en utilisant la contrainte *(a)* pour les substituer dans la contrainte *(b)*. Ce qui donne le programme suivant.

$$(P_2)\begin{cases} \min \sum_{u\in U}(c_u^1\theta_u^1 + c_u^2\theta_u^2) & \\ \\ \sum_{u\in\gamma^+}(\theta_u^2-\theta_u^1) - \sum_{u\in\gamma^-}(\theta_u^2-\theta_u^1) = \sum_{u\in\gamma^-} o_u - \sum_{u\in\gamma^+} o_u,\ \forall\,\gamma\in\Gamma_G & \textit{(a)} \\ \theta_u^1 \le o_u - a_u,\ \forall\, u\in U & \textit{(b)} \\ \theta_u^2 \le b_u - o_u,\ \forall\, u\in U & \textit{(c)} \\ \\ \theta_u^1\ge 0,\ \theta_u^2\ge 0,\ \forall\, u\in U & \end{cases} \qquad (4.2)$$

Ce programme contient $2m$ variables et $3m - n + 1$ contraintes puisque la base de cycles contient $m - n + 1$ cycles.

4.2.3. Conclusion

Les deux programmes proposés ici ont quasiment le même nombre de variables et de contraintes. Il est donc très difficile de déterminer a priori lequel est le plus efficace. Cependant il faut noter que le second modèle nécessite de déterminer une base de cycles, algorithme qui s'effectue en $O(mn)$ opérations (cf. chapitre 2 et annexe). Nous proposons donc maintenant une comparaison pratique de la résolution des deux modèles par la méthode du Simplex (cf. [Werr90]).

Pour connaître en détail comment ont été dirigés ces essais (méthode de génération des problèmes, compilateur utilisé...), le lecteur peut consulter l'annexe. Nous précisons seulement que l'outil CPLEX 6.0 a été utilisé avec ses paramètres par défaut pour résoudre les programmes linéaires et que les problèmes générés ont des bornes de tension et des coûts entiers. Le nombre d'itérations est le nombre d'itérations de la méthode du Simplex. Les temps de calcul sont exprimés en secondes sur une machine RISC-6000 à 160 MHz.

Dimension graphe		Programme P_1		Programme P_2	
Noeuds	Arcs	Itérations	Temps	Itérations	Temps
50	200	215	0,42	171	0,47
50	400	374	0,87	308	1
100	400	499	0,97	406	1,2
100	800	791	1,9	724	2,5
500	2000	2896	12,7	3270	32,4
500	4000	4393	37,8	5829	100,8
1000	4000	6799	56,8	8976	266,4
1000	8000	10330	219,7	14278	747,1

Tableau 4.1: Résultats numériques pour les programmes linéaires de tension de coût minimal, influence de la dimension du graphe.

Le tableau 4.1 montre le temps de génération et de résolution des deux programmes pour différentes tailles de graphe (avec $A = 1000$). Pour le programme P_2, ce temps inclut le temps de génération d'une base de cycles. Il est surprenant de constater que le programme P_1 se résout beaucoup plus rapidement que le programme P_2, bien que la comparaison des itérations pour les deux programmes n'explique pas cet écart. On ne peut pas non plus l'expliquer par le fait que la génération de P_2 nécessite la construction d'une base de cycles, cette dernière s'effectuant en un temps négligeable par rapport au temps de résolution (environ $2\ \%$ du temps de résolution).

La programmation linéaire est la principale solution implémentée dans les systèmes de synchronisation hypermédia. Il nous paraît donc intéressant de comparer les méthodes que nous proposons avec la résolution du programme linéaire équivalent, en occurrence le modèle P_1 qui est le plus performant.

4.3. Conformité et optimalité

Dans cette section, nous étudions les conditions suffisantes pour qu'une tension ait un coût minimal. Ces conditions sont très similaires aux conditions d'optimalité d'un flot de coût minimal. Elles sont associées à la notion de *conformité* (*kilter*). Ces conditions sont connues depuis longtemps pour le flot (cf. [Fulk61]). Elles ont ensuite été introduites pour la tension par J.M. Pla (cf. [Pla71]). Nous proposons un rappel de ces conditions et de la notion de conformité tout d'abord pour des coûts linéaires, puis pour des coûts convexes linéaires par morceaux et enfin pour des coûts convexes dérivables.

4.3.1. Coûts linéaires

Nous considérons ici une fonction de coût de la forme $c_u(\theta_u) = \lambda_u \theta_u$ pour chaque arc u du graphe. Dans [Pla71], la ***conformité*** d'un arc est définie de la manière suivante.

Soit θ une tension et φ un flot dans le graphe G. Un arc u est dit conforme par rapport à θ et φ si l'une des affirmations suivantes est vérifiée.

- $\varphi_u < \lambda_u$ *et* $\theta_u = a_u$. *(4.3)*
- $\varphi_u = \lambda_u$ *et* $a_u \leq \theta_u \leq b_u$.
- $\varphi_u > \lambda_u$ *et* $\theta_u = b_u$.

La figure 4.2 illustre cette notion de conformité. La courbe qu'elle représente est appelée ***courbe de conformité***. Quand un arc se trouve sur la courbe, il est conforme. En dehors, il ne l'est plus.

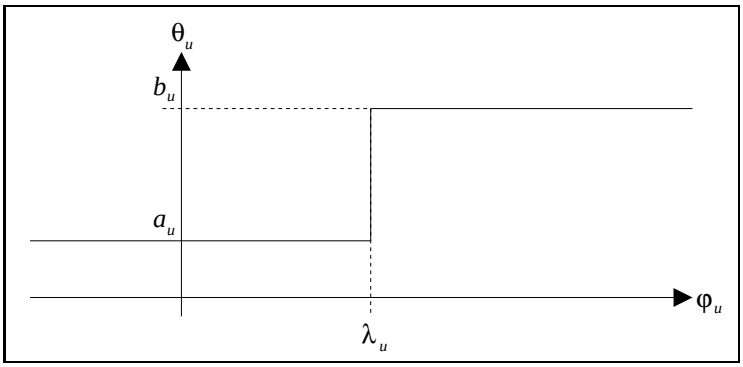


Figure 4.2: Courbe de conformité (coût linéaire).

J.M. Pla propose le théorème suivant qui associe l'optimalité d'une tension à sa conformité pour chaque arc du graphe.

Soit θ une tension dans le graphe G. S'il existe un flot φ pour lequel tout arc de G est conforme par rapport à θ et φ, alors θ est une tension de coût minimal. *(4.4)*

Preuve:
θ est optimale si et seulement si pour toute tension θ', on a $\sum_{u \in U} c_u(\theta_u) \leq \sum_{u \in U} c_u(\theta'_u)$, i.e. $\sum_{u \in U} \lambda_u(\theta_u - \theta'_u) \leq 0$ *(*)*. La tension et le flot étant orthogonaux, alors pour tout flot φ on a $\varphi^t(\theta - \theta') = 0$. L'inégalité *(*)* s'écrit alors $\sum_{u \in U}(\lambda_u(\theta_u - \theta'_u) - \varphi_u(\theta_u - \theta'_u)) \leq 0$ ou encore $\sum_{u \in U}(\lambda_u - \varphi_u)(\theta_u - \theta'_u) \leq 0$. La conformité par rapport à θ et φ de chaque arc induit que les termes de la somme sont tous négatifs (cf. définition 4.3). □

4.3.2. Coûts linéaires par morceaux

Nous considérons maintenant pour chaque arc u du graphe une fonction de coût convexe linéaire par morceaux de la forme suivante (cf. figure 2.11b).

$$c_u(\theta_u) = \begin{cases} c_u^1(o_u - \theta_u), \text{ si } a_u \leq \theta_u \leq o_u \\ c_u^2(\theta_u - o_u), \text{ si } o_u < \theta_u \leq b_u \end{cases}$$

En reprenant la transformation de la figure 4.1, un arc u du graphe G peut être remplacé par trois arcs u_1, u_2 et u_3 dans le graphe G'. Soit θ' une tension dans G' et θ la tension associée dans G telle que $\theta_u = -\theta'_{u_1} + \theta'_{u_2} + \theta'_{u_3}$. De la même manière, soit φ' un flot dans G' et φ le flot associé dans G telle que $\varphi_u = -\varphi'_{u_1} = \varphi'_{u_2} = \varphi'_{u_3}$.

La figure 4.3 montre les courbes de conformité des trois arcs u_1, u_2 et u_3. La première courbe est volontairement inversée pour faciliter la compréhension de la construction de la courbe de conformité de u (cf. figure 4.4).

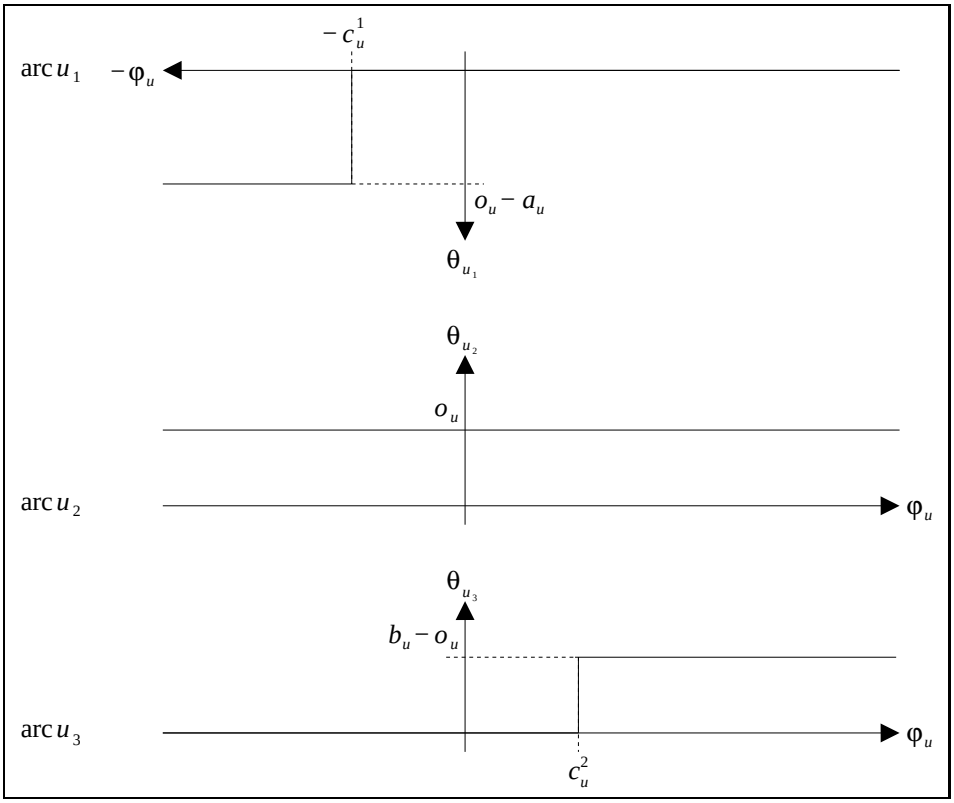


Figure 4.3: Courbe de conformité (transformation en coûts linéaires d'un coût linéaire par morceaux).

Nous proposons de définir la conformité d'un arc u de la manière suivante. Un arc u dans le graphe G est dit conforme par rapport à θ et φ si ses arcs associés dans G' sont tous les trois conformes par rapport à θ' et φ'. Autrement dit, u est conforme si l'une des affirmations suivantes est vérifiée.

- $\varphi_u < -c_u^1$ et $\theta'_{u_1} = o_u - a_u$, $\theta'_{u_2} = o_u$, $\theta'_{u_3} = 0$, i.e. $\theta_u = a_u$.
- $\varphi_u = -c_u^1$ et $0 \leq \theta'_{u_1} \leq o_u - a_u$, $\theta'_{u_2} = o_u$, $\theta'_{u_3} = 0$, i.e. $a_u \leq \theta_u \leq o_u$.
- $-c_u^1 < \varphi_u < c_u^2$ et $\theta'_{u_1} = 0$, $\theta'_{u_2} = o_u$, $\theta'_{u_3} = 0$, i.e. $\theta_u = o_u$.

- $\varphi_u = c_u^2$ et $\theta'_{u_1} = 0$, $\theta'_{u_2} = o_u$, $0 \leq \theta'_{u_3} \leq b_u - o_u$, i.e. $o_u \leq \theta_u \leq b_u$.
- $\varphi_u > c_u^2$ et $\theta'_{u_1} = 0$, $\theta'_{u_2} = o_u$, $\theta'_{u_3} = b_u$, i.e. $\theta_u = b_u$.

En résumé, voici la définition d'un arc conforme.

Soit θ une tension et φ un flot dans le graphe G. Un arc u est dit conforme par rapport à θ et φ si l'une des affirmations suivantes est vérifiée.

- $\varphi_u < -c_u^1$ *et* $\theta_u = a_u$.
- $\varphi_u = -c_u^1$ *et* $a_u \leq \theta_u \leq o_u$.
- $-c_u^1 < \varphi_u < c_u^2$ *et* $\theta_u = o_u$.
- $\varphi_u = c_u^2$ *et* $o_u \leq \theta_u \leq b_u$.
- $\varphi_u > c_u^2$ *et* $\theta_u = b_u$.

(4.5)

Ce qui se traduit par la courbe de conformité illustrée dans la figure 4.4. Comme $\theta_u = -\theta'_{u_1} + \theta'_{u_2} + \theta'_{u_3}$, la courbe se construit simplement en sommant les courbes de conformité des trois arcs u_1, u_2 et u_3 (cf. figure 4.3).

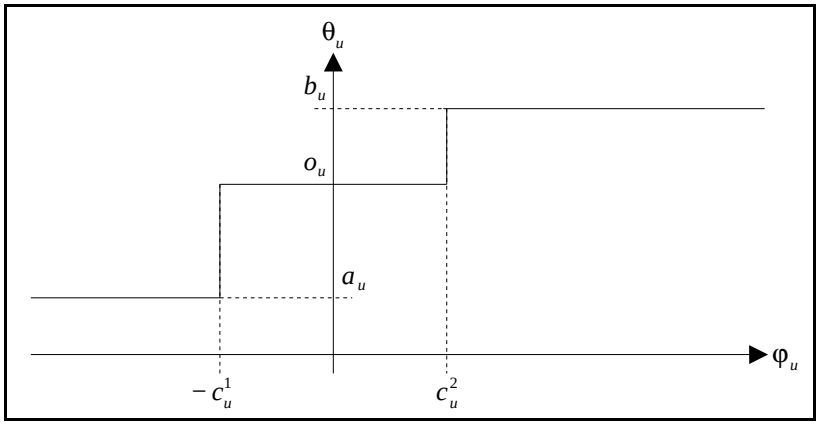


Figure 4.4: Courbe de conformité (coût linéaire par morceaux).

Comme trouver une tension optimale dans G' est équivalent à trouver une tension optimale dans G, une proposition similaire à celle de J.M. Pla peut être établie.

Soit θ une tension dans le graphe G. S'il existe un flot φ pour lequel tout arc de G est conforme par rapport à θ et φ, alors θ est une tension de coût minimal. (4.6)

Nous avons considéré ici une fonction convexe avec deux morceaux linéaires, mais l'étude peut s'appliquer à n'importe quelle fonction convexe linéaire par morceaux. La convexité de la fonction de coût implique que la courbe de conformité sera toujours croissante. La seule différence réside dans le nombre de "paliers" dans la courbe. En fait, il y en a autant que de morceaux linéaires dans la fonction de coût.

4.3.3. Coûts dérivables

Nous considérons ici une fonction de coût convexe dérivable quelconque pour chaque arc u du graphe. Nous proposons de définir la conformité d'un arc de la manière suivante.

Soit θ une tension et φ un flot dans le graphe G. Un arc u est dit conforme par rapport à θ et φ si l'une des affirmations suivantes est vérifiée.

- $\varphi_u < c'_u(a_u)$ *et* $\theta_u = a_u$.
- $\varphi_u = c'_u(\theta_u)$ *et* $a_u \leq \theta_u \leq b_u$.
- $\varphi_u > c'_u(b_u)$ *et* $\theta_u = b_u$.

(4.7)

La figure 4.5 est un exemple de courbe de conformité associée à cette définition. Comme la fonction de coût est convexe, cette courbe est toujours croissante.

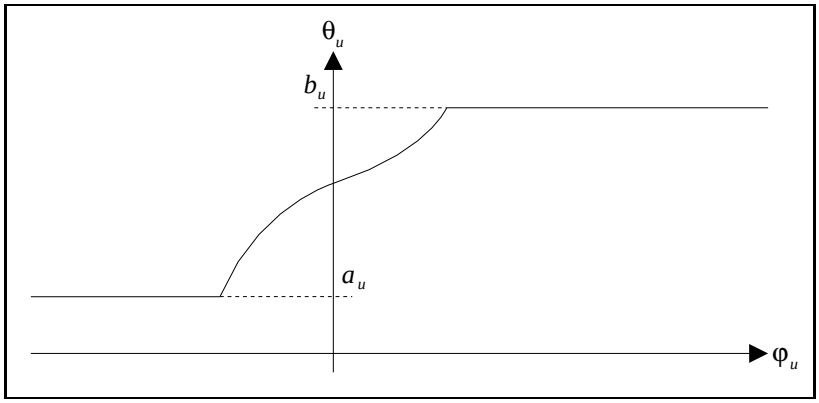


Figure 4.5: Courbe de conformité (coût dérivable).

Comme pour les coûts linéaires, il est possible d'associer l'optimalité d'une tension à sa conformité pour chaque arc du graphe grâce à la proposition suivante.

Soit θ une tension dans le graphe G. S'il existe un flot φ pour lequel tout arc de G est conforme par rapport à θ et φ, alors θ est une tension de coût minimal.

(4.8)

Preuve:
Soit θ une tension et φ un flot dans G pour lesquels tous les arcs du graphe sont conformes. Quelque soit la tension θ', il est facile de vérifier que $\forall\, u \in U,\ (c'_u(\theta_u) - \varphi_u)(\theta_u - \theta'_u) \leq 0$ ***(*)***. La fonction c_u étant convexe, on a $c_u(\theta_u) - c_u(\theta'_u) \leq c'_u(\theta_u)(\theta_u - \theta'_u)$ ***(**)***. ***(*)*** et ***(**)*** induisent $\forall\, u \in U,\ c_u(\theta_u) - c_u(\theta'_u) \leq \varphi_u(\theta_u - \theta'_u)$. Donc $\sum_{u\in U}(c_u(\theta_u) - c_u(\theta'_u)) \leq \sum_{u\in U} \varphi_u(\theta_u - \theta'_u)$. La tension et le flot étant orthogonaux, $\varphi^t(\theta - \theta') = 0$, d'où $\sum_{u\in U}(c_u(\theta_u) - c_u(\theta'_u)) \leq 0$, i.e. $\sum_{u\in U} c_u(\theta_u) \leq \sum_{u\in U} c_u(\theta'_u)$, donc θ est de coût minimal. □

4.4. Méthode de mise à conformité (coûts linéaires par morceaux)

D'après la section précédente, quand tous les arcs sont conformes, la tension est optimale. Une idée simple, proposée par J.M. Pla pour des coûts linéaires, consiste à partir d'une tension compatible et d'un flot quelconque, et à amener progressivement tous les arcs sur leur courbe de conformité. Cette méthode, dite de ***mise à conformité*** (*out-of-kilter*) et proposée tout d'abord dans [Fulk61] pour le problème du flot de coût minimal, a été adaptée dans [Pla71] pour le problème de la tension de coût minimal dans le cas linéaire. Cette méthode a été également reprise dans [Hadj96], toujours pour des coûts linéaires, et propose des variantes usant d'une *mise à l'échelle* des coûts et des capacités. Nous proposons donc une adaptation relativement immédiate de l'algorithme pour des coûts convexes linéaires par morceaux et reprenons l'étude menée dans [Hadj96].

4.4.1. Approche directe

Considérons un arc u qui n'est pas conforme. Le problème ici est de trouver un moyen de rapprocher cet arc de sa courbe de conformité. Supposons par exemple qu'il soit au dessus. Pour le rapprocher de sa courbe, il suffit soit d'augmenter son flot, soit de diminuer sa tension. Dans le chapitre sur la tension compatible (cf. section 3.1.2), nous avons vu que pour modifier la tension de l'arc u, il faut trouver un cocycle contenant u dont tous les arcs acceptent soit l'augmentation, soit la diminution de tension. De la même manière, si l'on veut modifier le flot de l'arc u, il faut trouver un cycle contenant u dont tous les arcs acceptent soit l'augmentation, soit la diminution de flot. L'idée ici est de trouver une coloration des arcs qui permet d'exploiter le lemme de Minty pour obtenir de tels cycles et cocycles.

4.4.1.1. Coloration des arcs

Voici la coloration proposée dans [Pla71] et [Hadj96] pour des coûts linéaires. Nous conservons cette coloration pour des coûts linéaires par morceaux.

- L'arc u est coloré en vert si une augmentation et une diminution de son flot sont possibles sans qu'il ne s'éloigne de sa courbe de conformité.
- L'arc u est coloré en rouge si une augmentation et une diminution de sa tension sont possibles sans qu'il ne s'éloigne de sa courbe de conformité.
- L'arc u est coloré en noir si une diminution de son flot et une augmentation de sa tension sont possibles sans qu'il ne s'éloigne de sa courbe de conformité.
- L'arc u est coloré en bleu si une augmentation de son flot et une diminution de sa tension sont possibles sans qu'il ne s'éloigne de sa courbe de conformité.

La figure 4.6 illustre cette coloration. Les arcs verts et rouges sont forcément conformes. Les arcs verts se trouvent sur les parties horizontales de la courbe (exceptés les angles) et les arcs rouges se trouvent sur les parties verticales de la courbe (exceptés les angles). Les arcs noirs et bleus ne sont pas forcément conformes (seuls les angles de la courbe le sont). Les arcs noirs non conformes sont en dessous de la courbe et les arcs bleus non conformes au dessus. Les arcs bleus et noirs conformes sont les angles de la courbe.

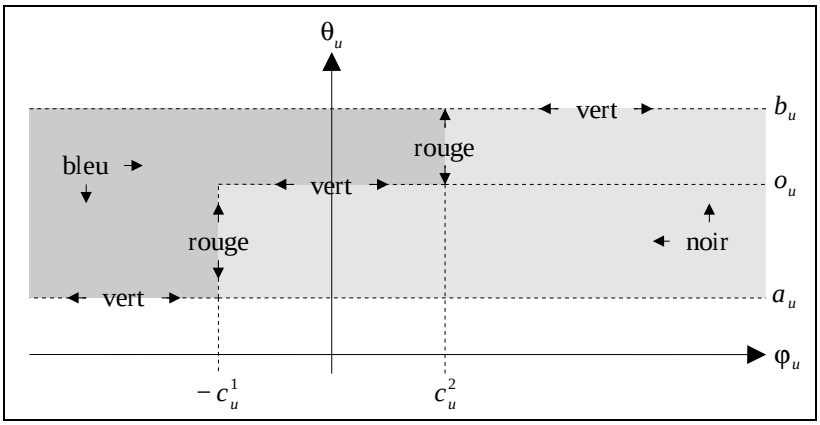


Figure 4.6: Coloration des arcs (coût linéaire par morceaux).

D'un point de vue pratique, divers cas particuliers comme par exemple $c_u^1 = c_u^2 = 0$, $o_u = a_u$... rendent fastidieuse l'expression algorithmique de la coloration. Cela se complique encore plus s'il y a plus de deux

morceaux dans la fonction de coût. Nous présentons donc ici la coloration sous forme algorithmique, uniquement dans le cas général.

- Si $\varphi_u < -c_u^1$ et $\theta_u = a_u$ alors u est vert.
- Si $\varphi_u < -c_u^1$ et $\theta_u > a_u$ alors u est bleu.
- Si $\varphi_u = -c_u^1$ et $\theta_u = a_u$ alors u est noir.
- Si $\varphi_u = -c_u^1$ et $a_u < \theta_u < o_u$ alors u est rouge.
- Si $\varphi_u = -c_u^1$ et $\theta_u \geq o_u$ alors u est bleu.
- Si $-c_u^1 < \varphi_u < c_u^2$ et $\theta_u < o_u$ alors u est noir.
- Si $-c_u^1 < \varphi_u < c_u^2$ et $\theta_u = o_u$ alors u est vert.
- Si $-c_u^1 < \varphi_u < c_u^2$ et $\theta_u > o_u$ alors u est bleu.
- Si $\varphi_u = c_u^2$ et $\theta_u \leq o_u$ alors u est noir.
- Si $\varphi_u = c_u^2$ et $o_u < \theta_u < b_u$ alors u est rouge.
- Si $\varphi_u = c_u^2$ et $\theta_u = b_u$ alors u est bleu.
- Si $\varphi_u > c_u^2$ et $\theta_u < b_u$ alors u est noir.
- Si $\varphi_u > c_u^2$ et $\theta_u = b_u$ alors u est vert.

4.4.1.2. Amélioration de la conformité d'un arc

A partir de cette coloration que nous noterons C_{noir}, le lemme de Minty nous permet d'affirmer que pour un arc noir u non conforme, il existe:

- soit un cycle contenant u et des arcs verts, noirs (dans le même sens que u) ou bleus (dans le sens opposé à u), ce qui signifie qu'il est possible de diminuer le flot sur ce cycle, autrement dit de diminuer le flot de u sans éloigner aucun arc de sa courbe de conformité;
- soit un cocycle contenant u et des arcs rouges, noirs (dans le même sens que u) ou bleus (dans le sens opposé à u), ce qui signifie qu'il est possible d'augmenter la tension sur ce cocycle, autrement dit d'augmenter la tension de u sans éloigner aucun arc de sa courbe de conformité.

Cela signifie qu'il est toujours possible de rapprocher un arc noir non conforme de sa courbe de conformité sans altérer la conformité des autres arcs. Considérons maintenant la coloration C_{bleu} où tout simplement les couleurs noir et bleu sont inversées par rapport à la coloration C_{noir}. Le lemme de Minty permet d'affirmer que pour un arc bleu u (avec la coloration C_{noir}) non conforme, il existe:

- soit un cycle contenant u et des arcs verts, noirs (dans le même sens que u) ou bleus (dans le sens opposé à u), ce qui signifie qu'il est possible d'augmenter le flot sur ce cycle, autrement dit d'augmenter le flot de u sans éloigner aucun arc de sa courbe de conformité;
- soit un cocycle contenant u et des arcs rouges, noirs (dans le même sens que u) ou bleus (dans le sens opposé à u), ce qui signifie qu'il est possible de diminuer la tension sur ce cocycle, autrement dit de diminuer la tension de u sans éloigner aucun arc de sa courbe de conformité.

Ce qui signifie qu'il est toujours possible de rapprocher un arc bleu (avec la coloration C_{noir}) non conforme de sa courbe de conformité sans altérer la conformité des autres arcs.

En utilisant les deux colorations (C_{noir} pour un arc en dessous de la courbe, C_{bleu} pour un arc au dessus de la courbe), il est alors possible de rapprocher n'importe quel arc non conforme de sa courbe de conformité. Tout ceci permet d'écrire l'algorithme 4.1 proposé en premier lieu dans [Pla71] pour des coûts linéaires et que nous adaptons ici au cas des coûts linéaires par morceaux.

Algorithme 4.1: améliorerConformité(arc v,graphe $G=(X;U)$,tension θ,flot φ).

si u est noir avec C_{noir} alors
 cycleMinty(u,G,γ,ω); /* Recherche d'un cycle ou d'un cocycle avec C_{noir}. */

 si $\gamma \neq \emptyset$ alors
  /* Trouver la diminution maximale du flot sur γ. */

$$\lambda \leftarrow \min\left\{ \min_{\substack{u\in\gamma^+\\ \theta_u\leq o_u}}(\varphi_u + c_u^1),\ \min_{\substack{u\in\gamma^+\\ \theta_u> o_u}}(\varphi_u - c_u^2),\ \min_{\substack{u\in\gamma^-\\ \theta_u< o_u}}(-c_u^1 - \varphi_u),\ \min_{\substack{u\in\gamma^-\\ \theta_u\geq o_u}}(c_u^2 - \varphi_u) \right\};$$

  $\varphi \leftarrow \varphi - \lambda\gamma$;
 sinon
  /* Trouver l'augmentation maximale de la tension sur ω. */

$$\lambda \leftarrow \min\left\{ \min_{\substack{u\in\omega^+\\ \varphi_u< c_u^2}}(o_u - \theta_u),\ \min_{\substack{u\in\omega^+\\ \theta_u\geq c_u^2}}(b_u - \theta_u),\ \min_{\substack{u\in\omega^-\\ \varphi_u\leq -c_u^1}}(\theta_u - a_u),\ \min_{\substack{u\in\omega^-\\ \varphi> -c_u^1}}(\theta_u - o_u) \right\};$$

  $\theta \leftarrow \theta + \lambda\omega$;
 fin si;
sinon /* u est bleu avec C_{noir}. */
 cycleMinty(u,G,γ,ω); /* Recherche d'un cycle ou d'un cocycle avec C_{bleu}. */

 si $\gamma \neq \emptyset$ alors
  /* Trouver l'augmentation maximale du flot sur γ. */

$$\lambda \leftarrow \min\left\{ \min_{\substack{u\in\gamma^+\\ \theta_u< o_u}}(-c_u^1 - \varphi_u),\ \min_{\substack{u\in\gamma^+\\ \theta_u\geq o_u}}(c_u^2 - \varphi_u),\ \min_{\substack{u\in\gamma^-\\ \theta_u\leq o_u}}(\varphi_u + c_u^1),\ \min_{\substack{u\in\gamma^-\\ \theta_u> o_u}}(\varphi_u - c_u^2) \right\};$$

  $\varphi \leftarrow \varphi + \lambda\gamma$;
 sinon
  /* Trouver la diminution maximale de la tension sur ω. */

$$\lambda \leftarrow \min\left\{ \min_{\substack{u\in\omega^+\\ \varphi_u\leq -c_u^1}}(\theta_u - a_u),\ \min_{\substack{u\in\omega^+\\ \varphi> -c_u^1}}(\theta_u - o_u),\ \min_{\substack{u\in\omega^-\\ \varphi_u< c_u^2}}(o_u - \theta_u),\ \min_{\substack{u\in\omega^-\\ \theta_u\geq c_u^2}}(b_u - \theta_u) \right\};$$

  $\theta \leftarrow \theta - \lambda\omega$;
 fin si;
fin si;

L'algorithme consiste tout d'abord en une recherche d'un cycle ou d'un cocycle basé sur une coloration, ce qui s'effectue en $O(m)$ opérations. Ensuite, les arcs du cycle (respectivement du cocycle) trouvé sont parcourus pour déterminer l'augmentation ou la diminution maximale de flot (respectivement de tension) pouvant être appliquée sur le cycle (respectivement le cocycle). Tout ceci nécessite $O(m)$ opérations. L'algorithme complet s'exécute donc en $O(m)$ opérations.

4.4.1.3. Tension optimale

L'idée de l'algorithme consiste à sélectionner un arc non conforme et à lui appliquer la procédure d'amélioration pour le rapprocher de sa courbe. Cette opération est répétée jusqu'à ce que tous les arcs soient conformes. Nous commençons tout d'abord par présenter l'algorithme 4.2 dans un contexte général, sans précision du mode de sélection des arcs.

```
Algorithme 4.2: miseConformitéGénérique(graphe G=(X;U),tension θ).
tensionCompatibleCheminBis(G,θ); /* Recherche d'une tension compatible. */
φ ← 0;

tant que ∃u ∈ U non conforme faire
 sélectionner un tel arc u non conforme;
 améliorerConformité(u,G,θ,φ);
fin tant que;
```

L'algorithme s'exécute en $O(m^2(A+B))$ opérations dans le cas où les bornes de tension et les coûts sont entiers, et en $O(m^2(A+B)/\Delta)$ opérations dans le cas où ils sont réels, A étant la plus grande borne de tension en valeur absolue, i.e. $A = \max_{u\in U}\{|a_u|;|b_u|\}$, B étant le plus grand coût en valeur absolue, i.e. $B = \max_{u\in U}\{|c_u^1|;|c_u^2|\}$, et Δ une borne inférieure de λ (cf. algorithme 4.1). Une valeur possible de Δ est:

$$\Delta = \min\{\Delta_1;\Delta_2\}$$
$$\Delta_1 = \min_{\substack{\kappa\in\mathbb{Z}^m\\ \mu\in\mathbb{Z}^m\\ \nu\in\mathbb{Z}^m}}\{s = |\sum_{u\in U}\kappa_u a_u + \sum_{u\in U}\mu_u o_u + \sum_{u\in U}\nu_u b_u|,\ s>0\}$$
$$\Delta_2 = \min_{\substack{\mu\in\mathbb{Z}^m\\ \nu\in\mathbb{Z}^m}}\{s = |\sum_{u\in U}\mu_u c_u^1 + \sum_{u\in U}\nu_u c_u^2)|,\ s>0\}$$

Preuve:
Dans le cas entier, il est évident qu'à chaque amélioration, la tension ou le flot de l'arc u est amélioré d'au moins 1. Donc, au maximum en $A+B$ itérations, u aura atteint sa courbe de conformité. L'algorithme appelle donc $O(m(A+B))$ fois la procédure d'amélioration.
Dans le cas réel, on s'aperçoit que la tension d'un arc est toujours une combinaison linéaire (avec des coefficients entiers) des bornes de tension. Autrement dit, pour tout arc u et à toute itération de l'algorithme, il existe $\kappa\in\mathbb{Z}^m$, $\mu\in\mathbb{Z}^m$ et $\nu\in\mathbb{Z}^m$ tels que $\theta_u = \sum_{v\in U}\kappa_v a_v + \sum_{v\in U}\mu_v o_v + \sum_{v\in U}\nu_v b_v$. La justification de cette affirmation est similaire à celle présentée pour l'algorithme 3.5 de tension compatible. De même, le flot d'un arc est toujours une combinaison linéaire (avec des coefficients entiers) des coûts. Autrement dit, pour tout arc u et à toute itération de l'algorithme, il existe $\mu\in\mathbb{Z}^m$ et $\nu\in\mathbb{Z}^m$ tels que $\varphi_u = \sum_{v\in U}\mu_v c_v^1 + \sum_{v\in U}\nu_v c_v^2$. Cela veut dire que la valeur λ à chaque amélioration est une combinaison linéaire. La plus petite valeur de λ s'exprime donc $\Delta = \min\{\Delta_1;\Delta_2\}$ où $\Delta_1 = \min_{\kappa\in\mathbb{Z}^m,\ \mu\in\mathbb{Z}^m,\ \nu\in\mathbb{Z}^m}\{s = |\sum_{u\in U}\kappa_u a_u + \sum_{u\in U}\mu_u o_u + \sum_{u\in U}\nu_u b_u|,\ s>0\}$ et $\Delta_2 = \min_{\mu\in\mathbb{Z}^m,\ \nu\in\mathbb{Z}^m}\{s = |\sum_{u\in U}\mu_u c_u^1 + \sum_{u\in U}\nu_u c_u^2|,\ s>0\}$. Les valeurs Δ_1 et Δ_2 existent puisque l'ensemble dans lequel est recherché le minimum est dénombrable et majoré par 0. A chaque itération, la tension ou le flot de l'arc u est améliorée d'au moins Δ. Donc, au maximum en $(A+B)/\Delta$ améliorations, u aura atteint sa courbe de conformité. L'algorithme appelle donc $m(A+B)/\Delta$ fois la procédure d'amélioration. □

Nous nous intéressons maintenant à la méthode de sélection de l'arc à améliorer. Dans [Hadj96] et [Pla71], le même arc est sélectionné jusqu'à ce qu'il soit conforme (cf. algorithme 4.3).

```
Algorithme 4.3: miseConformitéLocale(graphe G=(X;U),tension θ).
tensionCompatibleCheminBis(G,θ);
φ ← 0;

tant que ∃u ∈ U non conforme faire
 sélectionner un arc u non conforme;
 tant que u non conforme faire améliorerConformité(u,G,θ,φ);
fin tant que;
```

Nous proposons plutôt de considérer les arcs dans un certain ordre et de sélectionner chaque arc non conforme dans cet ordre en n'effectuant qu'une procédure d'amélioration à la fois (cf. algorithme 4.4).

```
Algorithme 4.4: miseConformitéGlobale(graphe G=(X;U),tension θ).
tensionCompatibleCheminBis(G,θ);
φ ← 0;

tant que ∃u ∈ U non conforme faire
 pour tout arc u non conforme faire
  améliorerConformité(u,G,θ,φ);
 fin pour;
fin tant que;
```

4.4.1.4. Conclusion

Les deux algorithmes proposés ont la même complexité théorique. Il est donc intéressant de les comparer sur le plan pratique. Pour connaître en détail comment ont été dirigés ces essais (méthode de génération des problèmes, compilateur utilisé...), le lecteur peut consulter l'annexe. Nous précisons seulement ici que les problèmes générés ont des bornes de tension et des coûts entiers. Le nombre d'itérations est le nombre de recherches de cycle et de cocycle effectuées. Les temps de calcul sont exprimés en secondes sur une machine RISC-6000 à 160 MHz.

Dimension graphe		Approche locale (4.3)		Approche globale (4.4)	
Noeuds	Arcs	Itérations	Temps	Itérations	Temps
50	200	315	0,13	310	0,12
50	400	534	0,4	523	0,35
100	400	646	0,66	622	0,54
100	800	1053	1,5	1040	1,2
500	2000	3175	20,7	3051	15,9
500	4000	5451	65,9	5287	50,1
1000	4000	6249	108,2	5921	78,7
1000	8000	10970	382,7	10548	280,5

Tableau 4.2: Résultats numériques pour la mise à conformité (approche directe), influence de la dimension du graphe.

Le tableau 4.2 montre le temps de résolution des deux algorithmes pour différentes tailles de graphe (avec $A = 1000$). L'algorithme 4.4, qui n'effectue qu'une amélioration à la fois sur chaque arc, est le plus performant. La raison intuitive qui nous a fait choisir cette méthode de sélection des arcs est simplement qu'au lieu de se concentrer sur un arc et de l'amener sur sa courbe de conformité sans se soucier des autres arcs, il est peut être plus judicieux d'amener petit à petit tous les arcs sur leur courbe, la convergence semblant plus globale. On constate également que les deux approches effectuent à peu près le même nombre de recherches de cycle et de cocycle, mais il semblerait que dans la seconde approche la recherche soit plus rapide. Notons enfin que ces résultats sont du même ordre de grandeur que ceux obtenus par la programmation linéaire (cf. tableau 4.1).

4.4.2. Avec une mise à l'échelle

La méthode de mise à conformité telle qu'elle a été présentée s'exécute en $O(m^2(A+B))$ opérations pour des bornes de tension et des coûts entiers. Cela signifie que le temps de calcul est fortement dépendant de l'échelle des coûts et des bornes de tension, que l'on nomme également ***capacités***. Une méthode dite ***mise à l'échelle*** (*scaling*) est souvent utilisée pour réduire cette dépendance avec l'échelle des données.

Cette méthode consiste à représenter certaines données sous la forme d'un polynôme. Pour un entier $b > 0$, tout entier positif x peut s'écrire sous la forme $x = \sum_{i=0..k} \lambda_i b^i$ où $\forall\, \lambda_i,\ 0 \leq \lambda_i < b$, autrement dit les λ_i représentent les chiffres de x en base b. Pour un ensemble de données X, k doit vérifier:

$$b^k \leq \max_{x \in X} |x| < b^{k+1}$$

Autrement dit, $k = \lfloor \log(\max_{x \in X} |x|) / \log b \rfloor$, $\lfloor a \rfloor$ représentant la partie entière par défaut de a.

Notons P_i le problème P où des données x sont remplacées par $x_i = \sum_{j=i..k} b^{j-i} \lambda_j = \lfloor x / b^i \rfloor$, autrement dit les entiers x sont remplacés par leur $k - i$ derniers chiffres. La méthode de mise à l'échelle consiste à résoudre le problème P_i en partant de la solution de P_{i+1}. Voici la structure générale de la méthode de mise à l'échelle.

```
Algorithme 4.5: miseEchelle().
k ← ⌊log(max_{x∈X} |x|)/ log b⌋;
établir une solution S_{k+1} réalisable pour le problème P_k;

tant que k ≥ 0 faire
 trouver une solution optimale S_k pour le problème P_k en partant de la solution S_{k+1};
 k ← k − 1;
 adapter la solution S_{k+1} pour qu'elle soit solution de P_k;
fin tant que;
```

Pour que la méthode de mise à l'échelle est un intérêt, il faut que l'algorithme pour résoudre le problème P_k à partir d'une solution P_{k+1} soit très efficace. Souvent on choisit $b = 2$. Dans ce cas, en passant d'une itération à une autre, $-1 \leq x_i - 2x_{i+1} \leq 1$, ce qui confère généralement des propriétés qui améliore l'efficacité de l'algorithme *(*)*.

4.4.2.1. Mise à l'échelle des coûts

Nous reprenons ici un algorithme présenté dans [Hadj96] (pour le cas linéaire) qui effectue une mise à l'échelle des coûts. L'adaptation à des coûts linéaires par morceaux est immédiate. Voici donc l'algorithme de mise à l'échelle qui effectue à chaque itération une mise à conformité de tous les arcs, en considérant à l'itération k les coûts $\lfloor c_u^1 / 2^k \rfloor$ et $\lfloor c_u^2 / 2^k \rfloor$ pour chaque arc u du graphe au lieu des coûts c_u^1 et c_u^2.

```
Algorithme 4.6: miseEchelleCoût(graphe G=(X;U),tension θ).
tensionCompatibleCheminBis(G,θ);
φ ← 0;
k ← ⌊log(max_{u∈U}{|c_u^1|;|c_u^2|})/ log 2⌋;

tant que k ≥ 0 faire
 pour tout u ∈ U faire d_u^1 ← ⌊c_u^1/2^k⌋; d_u^2 ← ⌊c_u^2/2^k⌋;

 pour tout u ∈ U faire
  tant que u non conforme faire
   améliorerConformité(u,G,θ,φ); /* Avec ∀u ∈ U, c_u^1 = d_u^1, c_u^2 = d_u^2. */
  fin tant que;
 fin pour;

 φ ← 2φ;
 k ← k − 1;
fin tant que;
```

Dans cet algorithme, la mise à conformité d'un arc s'effectue en $O(nm)$ opérations.

Preuve:
D'après la remarque *(*)* énoncée dans l'introduction de cette section, il est facile de vérifier qu'à chaque itération k, tous les arcs se trouvent à une unité de leur conformité sur la composante flot. En d'autres termes, les arcs se trouvent à une unité de flot de la partie verticale de leur courbe de conformité. Dans [Hadj96], il est prouvé que si l'on répète la procédure d'amélioration sur un arc, le nombre de fois consécutives où l'on améliore la conformité sur la composante tension, i.e. le nombre de cocycles trouvés, est fini et est borné par $n - 1$. Autrement dit, avant de trouver un cycle pour modifier le flot (ce qui rendra l'arc conforme puisqu'il se trouve à une unité de flot de sa courbe de conformité), il faudra au pire $n - 1$ améliorations. La procédure

d'amélioration s'effectuant en $O(m)$ opérations, la mise à conformité d'un arc s'effectue en $O(nm)$ opérations. □

A chaque itération, $O(nm^2)$ opérations sont donc effectuées. L'algorithme de mise à l'échelle des coûts s'exécute alors en $O(nm^2 \log B)$ opérations.

4.4.2.2. Mise à l'échelle des capacités

Nous reprenons ici un algorithme présenté dans [Hadj96] (pour des coûts linéaires) qui effectue une mise à l'échelle des capacités (i.e. les bornes de tension). L'adaptation à des coûts linéaires par morceaux est immédiate. Voici donc l'algorithme de mise à l'échelle qui effectue à chaque itération une mise à conformité de tous les arcs, en considérant à l'itération k les bornes de tension $\lfloor a_u/2^k \rfloor$, $\lfloor o_u/2^k \rfloor$ et $\lceil b_u/2^k \rceil$ pour chaque arc u au lieu des bornes a_u, o_u et b_u ($\lceil x \rceil$ représente la partie entière par excès de x).

```
Algorithme 4.7: miseEchelleCapacité(graphe G=(X;U),tension θ).
k ← ⌊log(max_{u∈U}{|a_u|;|b_u|})/log 2⌋;
pour tout u ∈ U faire α_u ← ⌊a_u/2^k⌋; σ_u ← ⌊o_u/2^k⌋; β_u ← ⌈b_u/2^k⌉;
tensionCompatibleCheminBis(G,θ); /* Avec ∀u∈U, a_u=α_u, o_u=σ_u, b_u=β_u. */
φ ← 0;

tant que k ≥ 0 faire
 pour tout u ∈ U faire
  tant que u non conforme faire
   améliorerConformité(u,G,θ,φ); /* Avec ∀u∈U, a_u=α_u, o_u=σ_u, b_u=β_u. */
  fin tant que;
 fin pour;

 k ← k − 1;
 pour tout u ∈ U faire α_u ← ⌊a_u/2^k⌋; σ_u ← ⌊o_u/2^k⌋; β_u ← ⌈b_u/2^k⌉;
 θ ← 2θ;
fin tant que;
```

Dans cet algorithme, la mise à conformité d'un arc s'effectue en $O(nm^2)$ opérations.

Preuve:
D'après la remarque *(*)* énoncée dans l'introduction de cette section, à chaque itération k, tous les arcs se trouvent à une unité de tension de la partie horizontale de leur courbe de conformité. Dans [Edmo72], il est prouvé que si l'on répète la procédure d'amélioration sur un arc, le nombre de fois consécutives où l'on améliore la conformité sur la composante flot, i.e. le nombre de cycles trouvés, est fini et est borné par $\frac{1}{2}nm$. Autrement dit, avant de trouver un cocycle pour modifier la tension (ce qui rendra l'arc conforme puisqu'il se trouve à une unité de tension de sa courbe de conformité), il faudra au pire $\frac{1}{2}nm$ améliorations. La procédure d'amélioration s'effectuant en $O(m)$ opérations, la mise à conformité d'un arc s'effectue en $O(nm^2)$ opérations. □

A chaque itération, $O(nm^3)$ opérations sont donc effectuées. L'algorithme de mise à l'échelle des capacités s'exécute alors en $O(nm^3 \log A)$ opérations.

Il faut noter que d'une itération k à l'itération $k-1$, la tension de certains arcs peut devenir incompatible (seulement d'une unité de tension). Il faut donc prévoir ce cas dans la coloration des arcs. Par exemple, dans la coloration C_{noir}, pour un arc u, si $\theta_u < a_u$ alors u est noir et si $\theta_u > b_u$ alors u est bleu. Ainsi, la procédure d'amélioration aura tendance à rendre ces arcs compatibles. Si la tension d'un arc u à une itération k ne peut pas être rendue compatible, alors il n'existe plus de cocycle pour diminuer ou augmenter la tension d'un arc u pour le rendre compatible. Le nombre de cycles trouvés consécutivement étant fini, la procédure finira par trouver un cycle dont le pas d'amélioration est infini.

4.4.3. Conclusion

En résumé, le tableau 4.3 récapitule les trois algorithmes utilisant la mise à conformité pour trouver une tension de coût minimal, avec leur complexité dans le cas de bornes de tension et de coûts réels ou entiers. Les algorithmes sont classés en fonction de leur efficacité théorique, de la moins bonne à la meilleure.

Méthode	Complexité	
	Bornes réelles	Bornes entières
Directe (4.4)	$O(m^2\ (A+B)/\Delta)$	$O(m^2\ (A+B))$
Mise à l'échelle des capacités (4.7)		$O(nm^3\ \log A)$
Mise à l'échelle des coûts (4.6)		$O(nm^2\ \log B)$

Tableau 4.3: Complexité des algorithmes de mise à conformité pour la tension de coût minimal (coûts linéaires par morceaux).

D'un point de vue théorique, les deux algorithmes de mise à l'échelle sont plus efficaces que l'approche directe de mise à conformité. Cependant, nous proposons ici une comparaison sur le plan pratique de ces trois méthodes. Pour connaître en détail comment ont été dirigés ces essais (méthode de génération des problèmes, compilateur utilisé...), le lecteur peut consulter l'annexe. Nous précisons seulement ici que les problèmes générés ont des bornes de tension et des coûts entiers. Le nombre d'itérations est le nombre de recherches de cycle et de cocycle effectuées. Les temps de calcul sont exprimés en secondes sur une machine RISC-6000 à 160 MHz.

Echelle données		Approche directe (4.4)		Mise échelle coûts (4.6)		Mise échelle capacités (4.7)	
Capacités	Coûts	Itérations	Temps	Itérations	Temps	Itérations	Temps
1000	1000	4186	30,6	4802	32,8	8972	82,8
1000	10000	4166	30	6500	43,1	8979	82,4
1000	100000	4190	30,1	7937	49,0	8983	85,1
10000	1000	4817	42	6081	63,7	9006	81,1
10000	10000	4778	40,6	7973	80,2	9017	81,9
10000	100000	4802	41,9	9208	84	8890	82,5
100000	1000	5110	45,8	7062	87,2	9138	83,3
100000	10000	5122	46,7	9059	108,4	9102	82,7
100000	100000	5101	45	10090	101,1	9134	81

Tableau 4.4: Résultats numériques des algorithmes de mise à conformité pour la tension de coût minimal (coûts linéaires par morceaux), influence de l'échelle des données.

Le tableau 4.4 montre le temps de résolution des algorithmes pour différentes valeurs de la borne maximale de tension A et de la borne maximale des coûts B (avec $n = 500$ et $m = 3000$) sur un arc. On s'aperçoit tout d'abord que l'approche directe n'est pas sensible du tout à l'échelle des coûts, ce qui explique que l'algorithme de mise à l'échelle des coûts soit si peu efficace. En outre, cette mise à l'échelle rend naturellement la méthode sensible à B. En revanche, l'approche directe est assez sensible à l'échelle des capacités. La mise à l'échelle des capacités, bien que moins performante pour les échelles testées que l'approche directe, est très stable, ce qui laisse présager un très bon comportement, probablement une meilleure efficacité que l'approche directe, pour des échelles beaucoup plus grandes qu'il nous est assez difficile de tester pour des raisons de précision dans la représentation des entiers en machine.

4.5. Méthode de mise à conformité (coûts dérivables)

La section 4.3 a montré que pour tous les types de coûts convexes que nous avons choisis d'étudier, si tous les arcs sont conformes alors la tension est optimale. Ainsi, l'idée de la méthode précédente reste valide pour des

coûts dérivables: amener progressivement tous les arcs sur leur courbe de conformité. La différence avec les coûts linéaires par morceaux réside dans la définition de la courbe de conformité des arcs.

En effet, dans le cas de coûts linéaires par morceaux, la courbe est formée uniquement de segments verticaux et horizontaux, ce qui signifie qu'un arc qui se trouve sur sa courbe de conformité peut être déplacé sans être écarté de sa courbe, en modifiant soit sa composante flot, soit sa composante tension (cf. figure 4.7). Pour un coût dérivable, la partie principale de la courbe est continue, autrement dit un arc qui se trouve sur la partie croissante de sa courbe de conformité ne peut être déplacé sans être écarté de sa courbe qu'à condition de modifier à la fois sa composante flot et sa composante tension (cf. figure 4.8).

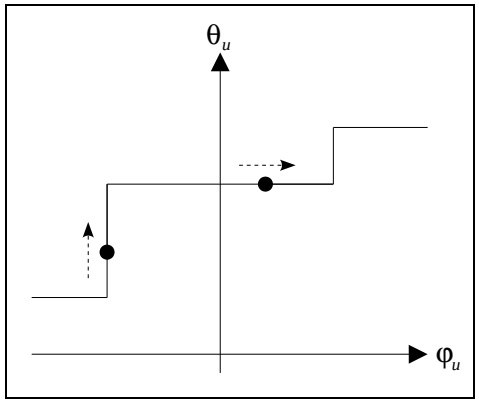


Figure 4.7: Déplacement le long d'une courbe de conformité (coût linéaire par morceaux).

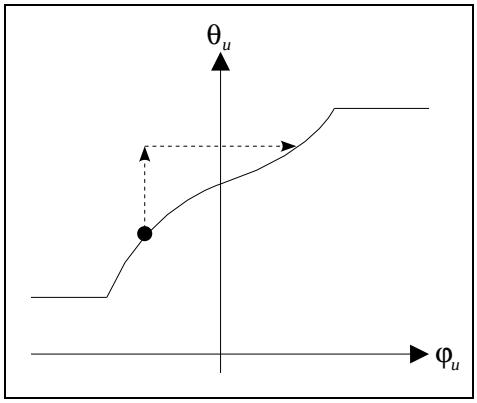


Figure 4.8: Déplacement le long d'une courbe de conformité (coût dérivable).

Il est très difficile d'effectuer un tel déplacement tout en conservant à la fois les propriétés de flot et de tension et tout en n'éloignant aucun arc de sa courbe de conformité. Nous choisissons donc de garder l'approche de la méthode de mise à conformité présentée précédemment et qui consiste à modifier soit le flot d'un cycle, soit la tension d'un cocycle séparément.

Nous proposons alors d'approcher la courbe de conformité pour des coûts dérivables par une courbe en escalier, avec une précision plus ou moins importante, ce qui nous garantit un bon fonctionnement de l'algorithme de mise à conformité. Cette méthode est fortement inspirée d'un algorithme proposé dans [Berg62] pour résoudre un problème de transport avec des coûts convexes dérivables. Pour formaliser cette approximation, nous introduisons la notion d'ε-*conformité*, à partir de laquelle nous proposons et discutons de différentes adaptations de la méthode de mise à conformité aux coûts dérivables.

4.5.1. Dérivée approchée

Lorsque la forme algébrique d'une fonction n'est pas connue, le calcul en pratique de sa dérivée est généralement approché. Afin de préciser cette approximation, nous introduisons ici la notion d'ε-***dérivée***, ou ***dérivée approchée***.

On appelle ε-dérivée d'une fonction $f : \mathbb{R} \longmapsto \mathbb{R}$ la fonction notée f'^{ε} telle que, pour une précision $\varepsilon \in \mathbb{R}$ donnée:

- $f'^{\varepsilon} : \mathbb{R} \longmapsto \mathbb{R}$ *(4.9)*
- $f'^{\varepsilon}(x) = \dfrac{f((n+1)\varepsilon) - f(n\varepsilon)}{\varepsilon}$ *où* $n \in \mathbb{Z}$*, tel que* $x \in [n\varepsilon; (n+1)\varepsilon[$

Dans le cas d'une fonction convexe, on a $f'(n\varepsilon) \leq f((n+1)\varepsilon) - f(n\varepsilon))/\varepsilon \leq f'((n+1)\varepsilon)$, ce qui entraîne $f'(n\varepsilon) \leq f'^{\varepsilon}(x) \leq f'((n+1)\varepsilon)$ pour tout $x \in [n\varepsilon; (n+1)\varepsilon[$ (cf. figure 4.9).

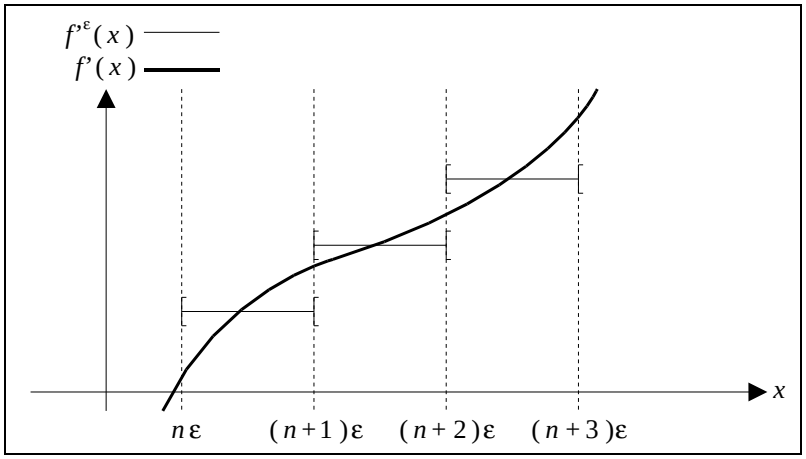


Figure 4.9: Dérivée approchée d'une fonction convexe.

4.5.2. Conformité approchée

La définition de la conformité d'un arc ayant un coût dérivable a été introduite à la section 4.3. Mais nous avons vu que celle-ci, à cause de sa continuité, empêche le bon fonctionnement de la méthode de mise à conformité. Il nous faudrait une fonction en escalier.

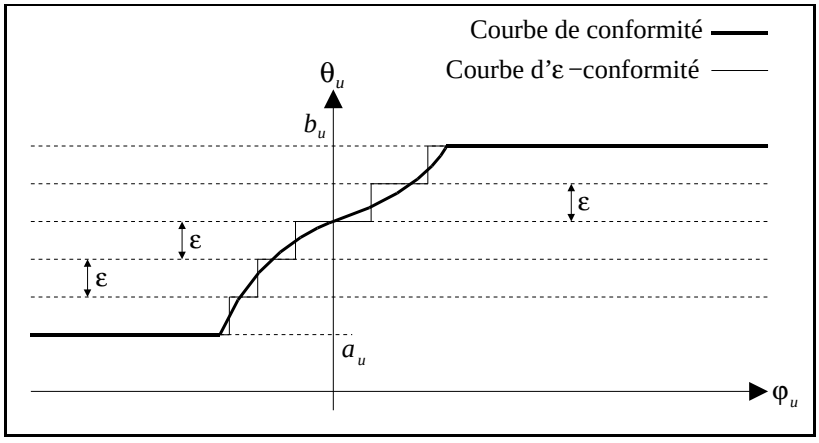


Figure 4.10: Courbe de conformité approchée (mise à conformité).

Nous introduisons donc la notion d'ε-***conformité***, ou ***conformité approchée***, qui définit la conformité d'un arc non pas par rapport à sa fonction de coût exacte, mais par rapport à une approximation définie par une ε-dérivée.

Soit θ une tension et φ un flot dans le graphe G. Un arc u est dit ε-conforme par rapport à θ et φ si l'une des affirmations suivantes est vérifiée.

- $\varphi_u < c'^{\varepsilon}_u(a_u)$ *et* $\theta_u = a_u$. *(4.10)*
- $\varphi_u = c'^{\varepsilon}_u(\theta_u)$ *et* $a_u \leq \theta_u \leq b_u$.
- $\varphi_u > c'^{\varepsilon}_u(b_u)$ *et* $\theta_u = b_u$.

La figure 4.10 illustre cette notion de conformité approchée. Nous nous retrouvons dans une configuration équivalente à celle d'un coût linéaire par morceaux, ce qui est tout à fait normal puisque l'approximation effectuée sur la dérivée revient à linéariser la fonction de coût.

4.5.3. Coloration des arcs

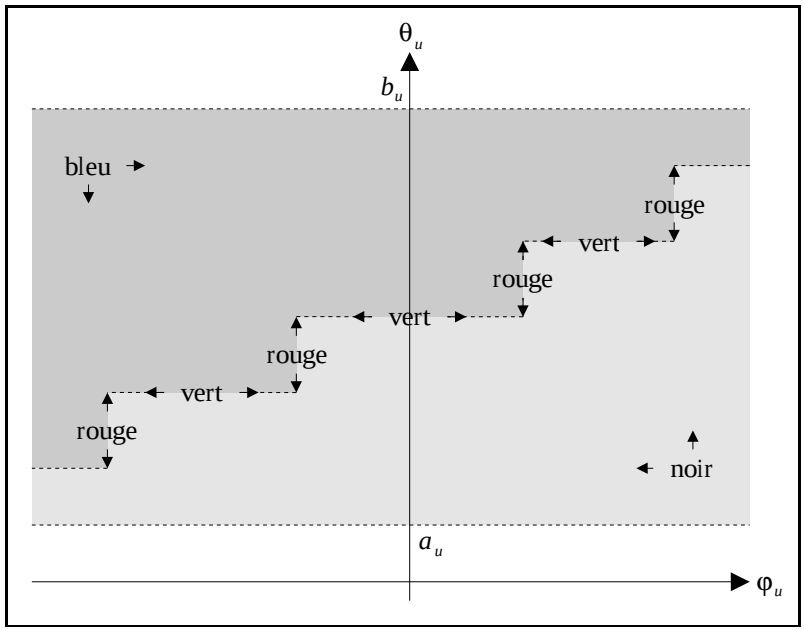


Figure 4.11: Coloration des arcs (coût dérivable).

Nous pouvons alors utiliser la même coloration que pour les coûts linéaires par morceaux. Celle-ci est illustrée par la figure 4.11 et se formule de la manière suivante pour une précision ε donnée.

- Si $\exists\, n \in \mathbb{Z}$ tel que $\theta_u = n\varepsilon$ et ${c'_u}^{\varepsilon}(\theta_u) \neq {c'_u}^{\varepsilon}(\theta_u + \varepsilon)$ alors
  - si $\varphi_u \leq {c'_u}^{\varepsilon}(\theta_u)$ alors u est bleu;
  - si ${c'_u}^{\varepsilon}(\theta_u) < \varphi_u < {c'_u}^{\varepsilon}(\theta_u + \varepsilon)$ alors u est vert;
  - si $\varphi_u \geq {c'_u}^{\varepsilon}(\theta_u + \varepsilon)$ alors u est noir;
- sinon
  - si $\varphi_u < {c'_u}^{\varepsilon}(\theta_u)$ alors u est bleu;
  - si $\varphi_u = {c'_u}^{\varepsilon}(\theta_u)$ alors u est rouge;
  - si $\varphi_u > {c'_u}^{\varepsilon}(\theta_u)$ alors u est noir.

4.5.4. Tension optimale

La première idée pour trouver une tension optimale avec la méthode de mise à conformité serait de fixer la précision ε très petite et de rendre les arcs ε-conformes avec l'algorithme 4.3 ou 4.4. Cependant, en exécutant ces méthodes on voit très rapidement leur inefficacité face à des courbes de conformité avec autant de paliers.

Intuitivement, au cours des itérations, de plus en plus d'arcs deviennent conformes. Mais un arc conforme est beaucoup plus difficile à déplacer qu'un arc non conforme puisqu'il doit à tout moment rester sur sa courbe de conformité. Autrement dit pour déplacer un arc d'un point de la courbe à un autre, il doit suivre tous les paliers de la courbe qui séparent les deux points, ce qui se traduit par une recherche de cycle ou de cocycle à chaque palier. Il est alors facile de comprendre que pour une précision très petite, le nombre de paliers devient beaucoup trop important pour que la méthode de mise à conformité fonctionne efficacement.

C'est pourquoi nous proposons plutôt une approche qui améliore progressivement l'approximation de la courbe de conformité (cf. algorithme 4.8). La méthode présentée démarre avec une valeur très importante pour ε. Elle est choisie pour qu'il n'y ait pas plus de dix paliers dans chaque courbe de conformité. Puis, à chaque itération, tous les arcs sont rendus ε-conformes en partant de la solution réalisable obtenue à l'itération précédente, ensuite la précision est descendue. L'arrêt s'effectue tout naturellement quand la précision p souhaitée est atteinte.

Algorithme 4.8: **miseEpsilonConformité(graphe** $G=(X;U)$**,tension** θ**,réel** p**).**

```
tensionCompatibleCheminBis(G,θ);
ε ← 0.1 × max_{u∈U}{b_u − a_u};

tant que ε > p faire
 rendre tous les arcs ε-conformes; /* Avec l'algorithme 4.3 ou l'algorithme 4.4. */
 ε ← max{0.1ε, p};
fin tant que;
```

L'algorithme s'exécute en $O(m^2(A+B)/\Delta \log pA)$ opérations. En effet, la méthode de mise à conformité pour des coûts linéaires par morceaux est appelée $O(\log pA/\log 10)$ fois (sa complexité étant $O(m^2(A+B)/\Delta)$). Il faut noter que la valeur de Δ est différente de celle des coûts linéaires par morceaux. Sa valeur est plus complexe à exprimer mais représente toujours la plus petite augmentation possible d'un flot ou d'une tension à une itération donnée (cf. algorithme 4.2).

4.5.5. Conclusion

La méthode employée ici appelle un certain nombre de fois la méthode de mise à conformité, avec chaque fois une précision de plus en plus petite. Il est intéressant de voir si d'une itération à l'autre la précision a un impact sur le temps de calcul ou sur le nombre de recherches de cycle et de cocycle. Nous proposons donc quelques résultats numériques. Pour connaître en détail comment ont été dirigés ces essais (méthode de génération des problèmes, compilateur utilisé...), le lecteur peut consulter l'annexe. Nous précisons seulement ici que les problèmes générés ont des bornes de tension entières et que la fonction de coût pour chaque arc u est de la forme $\lambda_u(\theta_u - o_u)^2$ avec λ_u choisi aléatoirement entre 0 et 100. Le nombre d'itérations considéré est le nombre de recherches de cycle et de cocycle effectuées. Les temps de calcul sont exprimés en secondes sur une machine RISC-6000 à 160 MHz.

Le tableau 4.5 montre le temps de résolution de l'algorithme pour différentes tailles de graphe (avec $A = 1000$ et $p = 0.001$). On aurait pu s'attendre à de meilleurs résultats. En effet, pour une précision de 0.001, la méthode de mise à conformité est appelée 6 fois donc on pourrait supposer un temps 6 fois plus élevé que la méthode pour des coûts linéaires par morceaux alors que l'on se retrouve avec un rapport de 15. Le tableau 4.6 apporte des éléments de réponse, il montre l'impact de la précision sur le temps de résolution et le nombre d'itérations. On s'aperçoit que le nombre d'itérations reste à peu près le même pour chaque appel de la méthode de mise à conformité. Par contre, le temps de calcul augmente quand on commence à avoir une bonne précision et diverge pour une précision de 100000. Il semblerait donc qu'il soit de plus en plus difficile à mesure que la précision

diminue de rechercher un cycle ou un cocycle. La divergence peut s'expliquer par le fait que la précision que l'on contrôle se rapproche de la précision de la machine et qu'il devient très difficile d'exécuter la méthode.

Dimension graphe		Mise ε-conformité (4.8)	
Noeuds	Arcs	Itérations	Temps
50	200	2541	4,8
50	400	5235	15,6
100	400	5632	18,7
100	800	11612	66,7
500	2000	32684	566,1
500	4000	71095	2399,5

Tableau 4.5: Résultats numériques de l'algorithme de mise à ε-conformité pour la tension de coût minimal, influence de la dimension du graphe.

Précision	Mise ε-conformité (4.8)	
(1/p)	Itérations	Temps
1	2670	7,6
10	3748	10,6
100	4573	13,1
1000	5598	17,7
10000	6391	44,4

Tableau 4.6: Résultats numériques de l'algorithme de mise à ε-conformité pour la tension de coût minimal, influence de la précision de la courbe.

Cet algorithme est en fait très sensible à la précision de la machine et le fait que l'algorithme puisse fonctionner pour une précision très petite dépend énormément de son implémentation. Il nous a donc semblé important d'éclairer l'éventuel programmeur sur certains problèmes liés à la représentation de l'ensemble des réels par un ensemble dénombrable en machine, ce qui induit forcément des approximations pouvant devenir importantes dans des cas bien précis.

Généralement, un indice sur l'approximation effectuée par l'ordinateur est donné par une variable δ qui est la plus petite valeur supérieure à zéro représentable par la machine. Ainsi, pour vérifier qu'une variable a est égale à une variable b, on choisira le test: $a < b + \delta$ et $a > b - \delta$. La relation d'égalité devient alors non transitive: si $a = b$ alors $c = b$ n'implique pas forcément $c = a$. Cela peut s'avérer problématique pour notre algorithme. Prenons un arc sur un palier de sa courbe de conformité et cherchons à augmenter son flot sans le sortir de sa courbe et sans modifier sa tension. L'arc est donc déplacé vers la droite le long du palier. Une fois arrivé au palier suivant, si la "marche" est inférieure à δ, le déplacement de l'arc peut continuer et ainsi de suite tant que les marches sont inférieurs à δ. Bien entendu, la tension elle aura changée d'une valeur considérée différente de zéro, ce qui nuit au bon fonctionnement de l'algorithme.

Un autre problème classique lié à la représentation des réels en machine. Si a est une valeur assez grande et b une valeur très petite, il y a de grande chance que $a + b = a$. Pour notre algorithme, cela peut se traduire par une boucle infinie. Imaginons qu'un cycle soit détecté, l'augmentation ou la diminution de flot associée est effectuée, mais le phénomène précédent se produit, autrement dit aucune valeur de flot n'est modifiée. A la prochaine itération, on risque de détecter le même cycle dans les mêmes conditions et ainsi entrer dans une boucle dont on ne pourra jamais sortir.

4.6. Méthode de mise à l'échelle du dual

La méthode proposée ici se limite à des coûts linéaires par morceaux comme définis au chapitre 2 et fournit une tension entière. Pour garantir cette intégrité et le bon fonctionnement de l'algorithme, les bornes de tension et les coûts doivent tous être entiers. Au lieu d'adapter un algorithme de flot à notre problème de tension comme nous l'avons fait avec les méthodes précédentes, nous transformons ici notre problème de tension en un problème de flot. Cette approche a été proposée dans [Ahuj99a] pour résoudre un problème intitulé par les auteurs le *dual du problème de flot à coûts convexes entiers*. Ce problème est une généralisation de notre problème de tension, et par conséquent, l'élaboration de la méthode est plus fastidieuse que celle que nous proposons ici, mais elle aboutit exactement au même résultat.

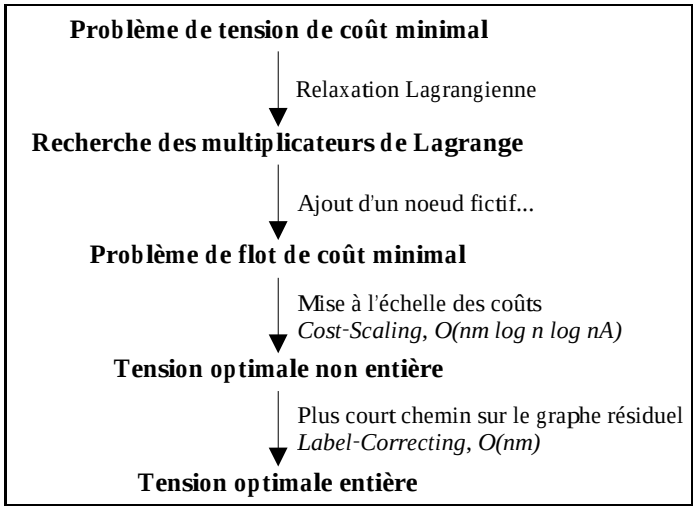


Figure 4.12: Etapes de la mise à l'échelle du dual.

Les grandes étapes de la transformation et de la résolution proposées dans [Ahuj99a] sont illustrées par la figure 4.12. Le problème de tension de coût minimal est relâché par la technique de la relaxation Lagrangienne (cf. [Rock84]). Le problème de la recherche des multiplicateurs de Lagrange est alors simplifié en ajoutant notamment un noeud source (i.e. sans aucun prédécesseur) dans le graphe. Le problème correspond alors à un problème de flot de coût minimal que [Ahuj99a] propose de résoudre à l'aide d'une adaptation de l'algorithme de mise à l'échelle des coûts (*cost-scaling*) proposé dans [Gold87] pour des coûts linéaires et qui opère en $O(nm \log n \log nA)$ opérations dans sa meilleure implémentation. Les coûts sur les arcs doivent alors être entiers comme nous l'avons précisé en début de section. Une fois le flot optimal, la tension obtenue n'est pas forcément entière. La recherche d'un plus court chemin sur le graphe résiduel est alors effectuée. Cela peut être fait par l'algorithme de Bellman en $O(nm)$ opérations (cf. [Ahuj93]). Les potentiels ainsi obtenus forment une tension optimale entière pour le problème initial.

Le détail de toutes ces étapes pour notre problème spécifique de tension de coût minimal est disponible dans [Bach01b], mais nous proposons ici une approche plus directe de la transformation du problème de tension de coût minimal en un problème de flot de coût minimal. Il nous a semblé également important de détailler ici l'adaptation de la méthode de mise à l'échelle pour des coûts linéaires par morceaux afin de fournir les éléments nécessaires à l'implémentation de la méthode et de permettre une discussion plus facile sur la méthode par la suite.

4.6.1. Transformation en un problème de flot

Il ne faut pas oublier que la notion de conformité introduite pour les problèmes de tension de coût minimal dans [Pla71] a d'abord été introduite pour les problèmes de flot de coût minimal dans [Fulk61] avec les mêmes conditions d'optimalité: lorsque tous les arcs sont conformes, le flot est de coût minimal. En outre, les courbes de conformité sont très semblables. Pour les problèmes de tension, la courbe d'un arc est définie par la dérivée du coût de sa tension, et pour les problèmes de flot, la courbe d'un arc est définie par la dérivée du coût de son flot de la même manière. Ainsi, une courbe de conformité peut être associée au coût d'une tension aussi bien qu'au coût d'un flot. Rechercher une tension de coût minimal dans un graphe revient alors à chercher un flot de coût minimal dans le même graphe, le coût du flot étant déduit de la courbe de conformité décrivant l'optimalité de la tension.

Considérons le problème de tension de coût minimal avec des coûts linéaires par morceaux. Nous rappelons que la tension θ_u d'un arc u est définie dans l'intervalle $[a_u; b_u]$ avec un coût comme suit (cf. figure 4.13a).

$$c_u(\theta_u) = \begin{cases} c_u^1(o_u - \theta_u), \text{ si } a_u \leq \theta_u \leq o_u \\ c_u^2(\theta_u - o_u), \text{ si } o_u < \theta_u \leq b_u \end{cases}$$

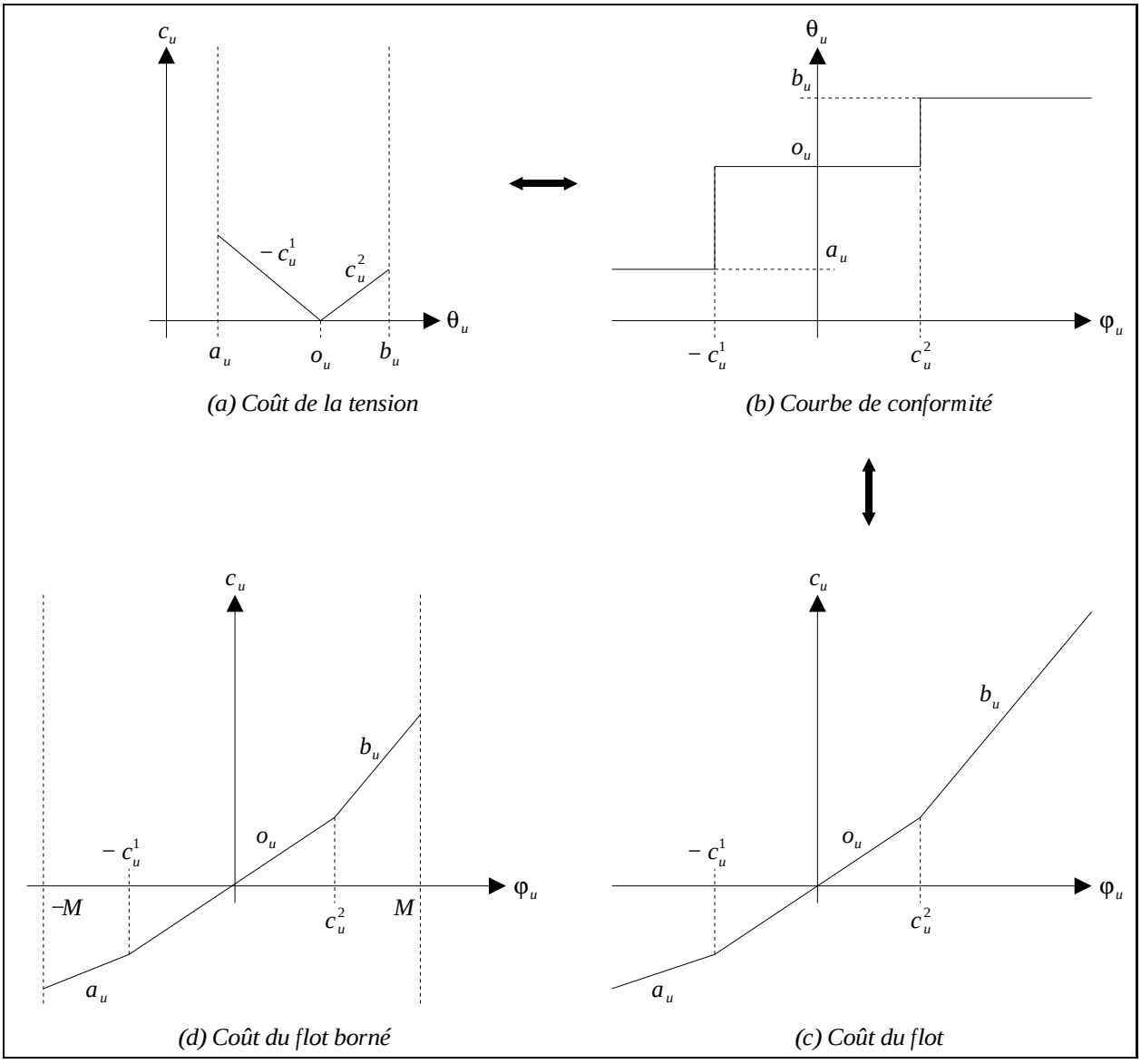


Figure 4.13: Courbe de conformité et fonctions de coûts associées pour le flot et la tension.

A partir de la courbe de conformité traduisant la condition d'optimalité de la tension (cf. figure 4.13b), il est facile d'associer une fonction de coût au flot (cf. figure 4.13c) telle que la courbe de conformité traduise également la condition d'optimalité du flot. Voici l'expression d'une fonction de coût possible.

$$c_u(\varphi_u) = \begin{cases} a_u(\theta_u + c_u^1) - o_u c_u^1, \text{ si } \varphi_u \leq -c_u^1 \\ o_u(\theta_u), \text{ si } -c_u^1 < \theta_u \leq c_u^2 \\ b_u(\theta_u - c_u^2) + o_u c_u^2, \text{ si } \theta_u > c_u^2 \end{cases} \quad (4.11)$$

En observant cette transformation, on s'aperçoit que le flot sur chaque arc n'est pas borné, ce qui empêche le bon fonctionnement de l'algorithme de mise à l'échelle des coûts pour le problème de flot. Pour cela, [Ahuj99a] propose de remplacer les bornes de la tension de chaque arc par un coût très élevé M dès que l'on sort de ces bornes (cf. figure 4.14a).

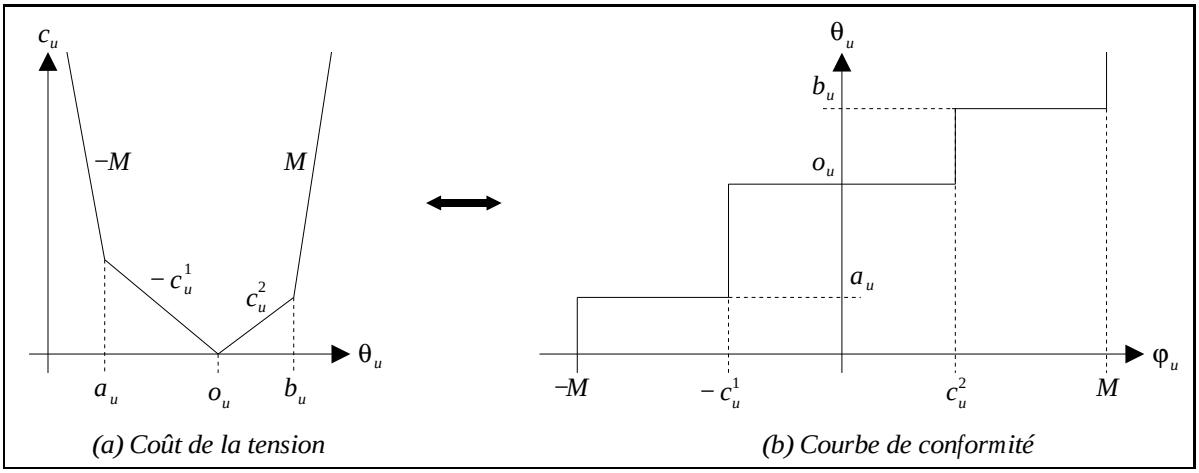


Figure 4.14: Courbe de conformité d'une tension non bornée (coût linéaire par morceaux).

M doit être choisi suffisamment grand pour qu'une tension en dehors des bornes ne soit jamais optimale. Pour cela il est possible d'affecter une valeur à M supérieure au coût d'une tension réalisable θ_f quelconque. Ainsi, une tension non réalisable aura un coût forcément supérieur à la tension θ_f et ne sera donc jamais optimale. La courbe de conformité des arcs pour une tension minimale est alors légèrement modifiée, mais on s'aperçoit que le flot associé est compris dans l'intervalle $[-M; M]$ (cf. figure 4.14b et figure 4.13d).

Pour résumer, un problème de tension de coût minimal se transforme en un problème de flot de coût minimal en supprimant les bornes de l'intervalle de tension (ce qui nécessite le calcul de M et donc la recherche d'une tension réalisable) et en construisant le coût du flot de chaque arc comme exprimé dans 4.11 et illustré par la figure 4.13d.

Une petite remarque pratique: certaines formulations du problème de flot de coût minimal exigent de n'avoir qu'un seul noeud sans prédécesseur et qu'un seul noeud sans successeur dans le graphe, ce qui est important si l'algorithme utilisé pour résoudre le problème exploite cette particularité. Les deux noeuds sont alors rajoutés en les connectant au graphe par des arcs dont le flot et le coût sont toujours nuls.

4.6.2. Flot optimal, mise à l'échelle des coûts

La méthode de ***mise à l'échelle des coûts*** (*cost-scaling*) pour le problème du flot de coût minimal a été introduite tout d'abord dans [Gold87] pour des coûts linéaires. Une analyse détaillée de l'algorithme est également proposée dans [Ahuj93]. L'article [Ahuj99a] détaille une adaptation de l'algorithme aux coûts linéaires par morceaux avec son analyse de complexité. La présentation proposée ici de l'algorithme ne fait que résumer les différentes références citées précédemment, le but étant de ne conserver que les détails importants pour notre problème de tension de coût minimal tout en fournissant les éléments essentiels à une éventuelle implémentation. Mais avant toute chose, quelques définitions et propriétés doivent être introduites. Nous invitons également le lecteur à se rapporter à l'ouvrage [Ahuj93] pour toutes les notions de base sur le problème de flot de coût minimal.

4.6.2.1. Définitions et propriétés

Soit α_u et β_u les valeurs minimales et maximales du flot φ_u qui traverse l'arc u. On dit que φ est un ***pseudo-flot*** si pour tout arc u, $\alpha_u \leq \varphi_u \leq \beta_u$, mais ne satisfait pas forcément la conservation des flots à tous les noeuds.

Il a été prouvé (cf. [Fulk61]) que si tous les arcs sont conformes, alors le flot est optimal. Ici est introduite une notion d'*optimalité approchée* différente de celle présentée pour le problème de la tension de coût minimal avec des coûts dérivables. On dira qu'un flot est ε***-optimal*** si tous les arcs du graphe sont à une distance inférieure ou égale à ε de leur courbe de conformité sur la composante tension. La figure 4.15 illustre cette notion, les arcs situés dans les parties grisées ou directement sur la courbe sont ε***-conformes***.

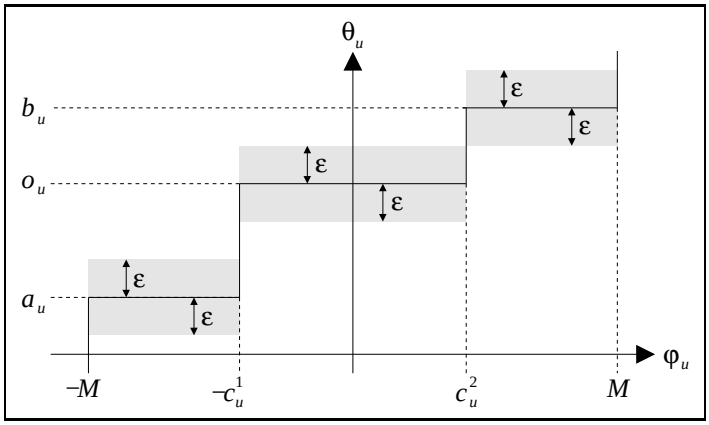


Figure 4.15: Courbe de conformité approchée (mise à l'échelle du dual).

Notons c_u^f la courbe de conformité d'un arc u définie seulement sur $]-M;M[\,\setminus\{-c_u^1;c_u^2\}$. Lorsque le flot est ε-optimal, pour tout arc u dont le flot peut être modifié sans quitter la courbe (i.e. $\varphi_u \in]-M;M[\,\setminus\{-c_u^1;c_u^2\}$), on peut affirmer que $\varepsilon \leq c_u^f(\varphi_u) - \theta_u \leq -\varepsilon$. Ainsi, pour tout cycle augmentant, i.e. dont on peut augmenter le flot, on peut calculer le coût unitaire d'augmentation C_ω^φ et affirmer:

$$
\begin{aligned}
C_\omega^\varphi &= \sum_{u\in\omega^+} c_u^f(\varphi_u) - \sum_{u\in\omega^-} c_u^f(\varphi_u) \\
C_\omega^\varphi &= \sum_{u\in\omega^+} c_u^f(\varphi_u) - \sum_{u\in\omega^-} c_u^f(\varphi_u) + \sum_{u\in\omega^+} \theta_u - \sum_{u\in\omega^-} \theta_u \\
C_\omega^\varphi &= \sum_{u\in\omega^+} (c_u^f(\varphi_u) - \theta_u) - \sum_{u\in\omega^-} (\theta_u - c_u^f(\varphi_u)) \geq -n\varepsilon
\end{aligned}
$$

Ainsi, si $\varepsilon < 1/n$, alors $C_\omega^\varphi > -1$. Comme les coûts sont entiers, cela signifie que $C_\omega^\varphi > 0$. En outre, il est prouvé que si pour tout cycle ω, $C_\omega^\varphi > 0$, alors φ est optimal (cf. [Ahuj93]). En résumé, un flot $1/n$-optimal est également optimal.

4.6.2.2. Procédure principale

L'algorithme de mise à l'échelle des coûts consiste donc à partir d'une tension θ (associée au potentiel π) nulle et d'un flot φ compatible (dans notre cas, le flot nul est compatible). On choisit un ε de départ suffisamment grand pour que le flot soit ε-optimal. Il est facile de vérifier que prendre $\varepsilon = A = \max_{u\in U}\{|a_u|;|b_u|\}$ suffit. Ensuite, la méthode consiste à diviser ε par deux à chaque itération. Chaque fois, une procédure dite d'*amélioration* tente de construire un flot $\varepsilon/2$-optimal à partir du flot ε-optimal obtenu à l'itération précédente. L'algorithme s'arrête lorsque ε est inférieur à $1/n$ (le flot est alors optimal) et exécute donc $O(\log nA)$ fois la

procédure d'amélioration (cf. algorithme 4.9).

```
Algorithme 4.9: flotMinimal(graphe G=(X;U),flot φ).
π ← 0;
ε ← A;
trouver un flot φ compatible;

tant que ε ≥ 1/n faire
 rendreFlotEpsilonOptimal(G,π,φ,ε);
 ε ← ε/2;
fin tant que;
```

4.6.2.3. Procédure d'amélioration

La procédure décrite ici consiste à passer d'un flot ε-optimal à un flot $\varepsilon/2$-optimal. Tout d'abord, le flot est transformé en un pseudo-flot $\varepsilon/2$-optimal. Pour cela, les arcs sont plaqués sur la courbe en modifiant seulement la composante flot. Ainsi, si un arc se trouve à gauche de la courbe (i.e. au dessus à $\varepsilon/2$ près) alors son flot est augmenté jusqu'à toucher la courbe. A l'inverse, si un arc se trouve à droite de la courbe (i.e. en dessous à $\varepsilon/2$ près) alors son flot est diminué jusqu'à toucher la courbe (cf. figure 4.16).

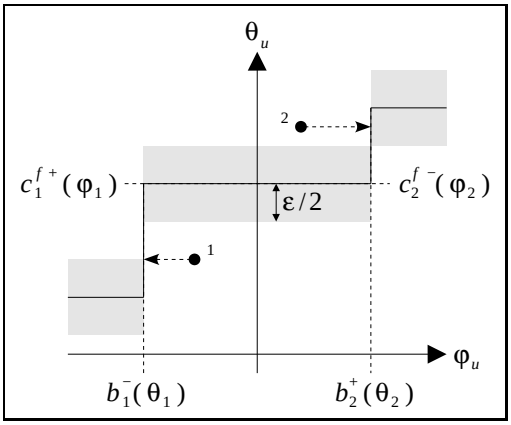


Figure 4.16: Construction d'un pseudo-flot.

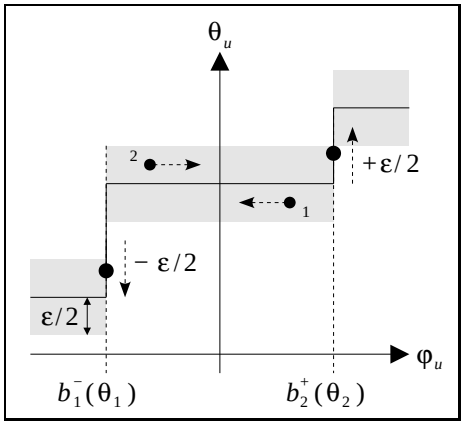


Figure 4.17: Equilibrage d'un noeud.

La courbe de conformité c_u^f n'est pas définie pour $-M$, $-c_u^1$, c_u^2 et M. Il est donc nécessaire de définir une ***courbe de conformité à droite*** c_u^{f+} et une ***courbe de conformité à gauche*** c_u^{f-} de la manière suivante, avec $\delta < 1$.

$$c_u^{f+}(\varphi_u) = \begin{cases} c_u^f(\varphi_u), \text{ si } \varphi_u \in]-M; M[\setminus \{-c_u^1; c_u^2\} \\ c_u^f(\varphi_u + \delta), \text{ si } \varphi_u \in \{-M; -c_u^1; c_u^2\} \\ b_u, \text{ si } \varphi_u = M \end{cases}$$

$$c_u^{f-}(\varphi_u) = \begin{cases} c_u^f(\varphi_u), \text{ si } \varphi_u \in]-M; M[\setminus \{-c_u^1; c_u^2\} \\ c_u^f(\varphi_u - \delta), \text{ si } \varphi_u \in \{M; -c_u^1; c_u^2\} \\ a_u, \text{ si } \varphi_u = -M \end{cases}$$

Comme nous le verrons par la suite, il est également nécessaire de pouvoir déterminer l'augmentation ou la diminution maximale du flot d'un arc sans que ce dernier ne s'éloigne de sa courbe (soit il est déjà dessus et il n'en sort pas, soit il n'y est pas encore et alors il s'en rapproche). De cette manière, des modifications du flot sur le graphe peuvent être effectuées sans altérer la conformité des arcs. Pour cela, nous définissons une ***borne à droite*** notée b_u^+ et une ***borne à gauche*** notée b_u^-.

$$b_u^+(\theta_u) = \begin{cases} M, \text{ si } \theta_u \geq b_u \\ c_u^2, \text{ si } o_u \leq \theta_u < b_u \\ -c_u^1, \text{ si } a_u \leq \theta_u < o_u \\ -M, \text{ si } \theta_u < a_u \end{cases}$$

$$b_u^-(\theta_u) = \begin{cases} M, \text{ si } \theta_u > b_u \\ c_u^2, \text{ si } o_u < \theta_u \leq b_u \\ -c_u^1, \text{ si } a_u < \theta_u \leq o_u \\ -M, \text{ si } \theta_u \leq a_u \end{cases}$$

Voici donc la procédure qui transforme un flot ε-optimal en un pseudo-flot $\varepsilon/2$-optimal d'après les définitions précédentes.

Algorithme 4.10: **construirePseudoFlot(graphe** $G=(X;U)$**,potentiel** π**,flot** φ**,réel** ε**).**

```
pour tout arc u = (x;y) ∈ U faire
 si π_y − π_x < c_u^{f−}(φ_u) − ε/2 alors φ_u ← b_u^−(π_y − π_x);
 sinon si π_y − π_x > c_u^{f+}(φ_u) + ε/2 alors φ_u ← b_u^+(π_y − π_x);
fin pour;
```

Il est évident que la conservation des flots n'est plus respectée. Il s'agit maintenant de modifier le pseudo-flot pour qu'il devienne un flot tout en maintenant l'$\varepsilon/2$-optimalité. Pour cela, on cherche un noeud x pour lequel il y a un excédent de flot (i.e. $\sum_{u\in\omega^+(x)}\varphi_u - \sum_{u\in\omega^-(x)}\varphi_u > 0$) et on évacue le surplus de flot à travers les arcs adjacents au noeud. Cette opération est effectuée jusqu'à ce que tous les noeuds soient équilibrés, i.e. la conservation des flots est respectée (cf. algorithme 4.11).

Algorithme 4.11: **rendreFlotEpsilonOptimal(graphe** $G=(X;U)$**,potentiel** π**,flot** φ**,réel** ε**).**

```
construirePseudoFlot(G,π,φ,ε);

tant que ∃x ∈ X excédentaire faire
 sélectionner un noeud x excédentaire;
 équilibrerNoeud(x,π,φ,ε);
fin tant que;
```

Pour équilibrer un noeud, il faut donc trouver des arcs sortants (respectivement entrants) pour lesquels on peut augmenter (respectivement diminuer) le flot sans quitter la courbe de conformité (à $\varepsilon/2$ près) (cf. algorithme 4.12). Pour éviter tout cyclage de l'algorithme (cf. [Ahuj93]), on n'augmentera un flot sur un arc que si l'arc se trouve au dessus de la courbe et à l'inverse on ne diminuera un flot sur un arc que si l'arc se trouve en dessous de la courbe (cf. figure 4.17). Les arcs remplissant toutes ces conditions sont dits ***admissibles***.

Algorithme 4.12: **équilibrerNoeud(noeud** x**,potentiel** π**,flot** φ**,réel** ε**).**

```
e ← Σ_{u∈ω+(x)} φ_u − Σ_{u∈ω−(x)} φ_u;

tant que e > 0 faire
 si ∃u = (x;y) ∈ ω^−(x) tel que c_u^{f+}(φ_u) < π_y − π_x ≤ c_u^{f+}(φ_u) + ε/2 et λ = b^+(π_y − π_x) − φ_u > 0 alors
  φ_u ← φ_u + min{e;λ};
  e ← e − min{e;λ};
 sinon si ∃u = (y;x) ∈ ω^+(x) tel que c_u^{f−}(φ_u) − ε/2 ≤ π_y − π_x < c_u^{f−}(φ_u) et λ = φ_u − b^−(π_y − π_x) > 0 alors
  φ_u ← φ_u − min{e;λ};
  e ← e − min{e;λ};
 sinon π_x ← π_x − ε/2;
fin tant que;
```

Si tout le flot n'a pas pu être évacué, on diminue alors le potentiel du noeud de $\varepsilon/2$, ainsi la tension des arcs sortants augmente (respectivement la tension des arcs entrants diminue), rapprochant ainsi ces arcs d'une possibilité d'augmentation de leur flot (cf. figure 4.17) tout en n'éloignant aucun arc de sa courbe de conformité. Ces opérations d'évacuation de flot et de baisse de potentiel sont répétées alternativement jusqu'à ce que le

noeud soit équilibré (i.e. la conservation des flots à son niveau est vérifiée). L'algorithme 4.12 détaille la procédure d'équilibrage d'un noeud.

Dans [Ahuj99a], il est prouvé que la procédure d'amélioration effectue $O(n^2)$ diminutions de potentiel, $O(nm)$ évacuations saturantes et $O(n^2m)$ évacuations non saturantes, ***saturante*** signifiant que l'arc impliqué dans l'évacuation est saturé, i.e. son flot a atteint sa limite ($b^+(\theta_u)$ ou $b^-(\theta_u)$). La procédure d'amélioration s'exécute donc en $O(n^2m)$ opérations.

4.6.3. Tension optimale

Toutes les données du problème sont entières, il est alors facile de vérifier que le flot obtenu est entier. Cependant, la tension n'est pas entière, puisque les augmentations ou les diminutions se font avec un pas ε rarement entier. Il faut donc maintenant rendre la tension obtenue entière. Pour cela, il suffit de remplacer pour chaque arc u du graphe G l'intervalle de tension $[a_u; b_u]$ par $[c_u^{f-}(\varphi_u); c_u^{f+}(\varphi_u)]$. Ensuite, on cherche une tension réalisable qui existe forcément puisque la solution déjà obtenue par la mise à l'échelle du dual est réalisable. On utilisera par exemple l'algorithme 3.7 (cf. section 3.2) qui garantit une solution entière. Ainsi, on obtient une tension et un flot parfaitement sur la courbe de conformité, et tous les deux entiers. En résumé, voici la procédure pour trouver une tension de coût minimal par l'approche duale.

Algorithme 4.13: **miseEchelleDual(graphe** $G = (X; U)$**,tension** θ**)**.

```
tensionCompatibleCheminBis(G,θ);
M ← Σ_{u∈U} c_u(θ_u);
construire les coûts du problème de flot équivalent; /* En utilisant M. */
flotMinimal(G,φ);
pour tout arc u ∈ U remplacer les bornes de tension [a_u;b_u] par [c_u^{f-}(φ_u); c_u^{f+}(φ_u)];
tensionCompatibleCheminBis(G,θ);
```

Nous avons vu au cours de cette présentation que la complexité de l'algorithme de mise à l'échelle pour le problème de flot s'exécute en $O(n^2m \log nA)$ opérations (en utilisant la structure d'arbre dynamique introduite dans [Slea83], [Ahuj99a] explique que l'algorithme s'exécute en $O(nm \log n \log nA)$). L'étape de transformation du problème de tension en un problème de flot nécessite la recherche d'une tension compatible (en $O(nm)$ opérations avec l'algorithme 3.7), la transformation elle-même est linéaire (avec $O(n + m)$ opérations). Enfin, l'étape de mise en valeur entière de la tension nécessite également une recherche de tension compatible (en $O(nm)$ opérations). La méthode proposée ici, que nous nommerons par la suite ***mise à l'échelle du dual***, nécessite $O(n^2m \log nA)$ opérations dans sa version générique qui est celle implémentée pour notre étude.

4.6.4. Conclusion

D'un point de vue théorique, la méthode de mise à l'échelle du dual, avec $O(n^2m \log nA)$ opérations, est plus efficace que la méthode de mise à conformité, avec $O(m^2(A + B))$ opérations. Cependant, les complexités sont trop proches pour être catégorique. Nous proposons donc une comparaison sur le plan pratique des deux méthodes. Pour connaître en détails comment ont été dirigés ces essais (méthode de génération des problèmes, compilateur utilisé...), le lecteur peut consulter l'annexe. Nous précisons seulement ici que les problèmes générés ont des bornes de tension et des coûts entiers. Le nombre d'itérations pour la méthode de mise à conformité est le nombre de recherches de cycle et de cocycle effectuées. Pour la méthode de mise à l'échelle du dual, le nombre d'itérations est le nombre d'évacuations de flot effectuées. Les temps de calcul sont exprimés en secondes sur une machine RISC-6000 à 160 MHz.

Le tableau 4.7 montre le temps de résolution des algorithmes pour différentes tailles de graphe (avec $A = 1000$). On s'aperçoit que la mise à l'échelle du dual est nettement plus rapide que la mise à conformité, avec en plus un temps de calcul qui croît beaucoup moins vite.

Dimension graphe		Mise conformité (4.4)		Mise échelle dual (4.13)	
Noeuds	Arcs	Itérations	Temps	Itérations	Temps
50	200	311	0,13	8712	0,11
50	400	514	0,31	9129	0,21
100	400	609	0,5	23171	0,3
100	800	1046	1,2	27266	0,58
500	2000	3046	15,8	220234	3,3
500	4000	5278	49,8	237786	6,6
1000	4000	5914	81,2	1715132	18,3
1000	8000	10531	249,2	498130	18,5

Tableau 4.7: Résultats numériques de l'algorithme de mise à l'échelle pour la tension de coût minimal, influence de la dimension du graphe.

Le tableau 4.8 montre le temps de résolution des algorithmes pour différentes valeurs de la borne maximale de tension A et de la borne maximale des coûts B (avec $n = 500$ et $m = 3000$). Comme prévu, la méthode duale est influencée par l'échelle des capacités puisque ce sont ces dernières qui deviennent les coûts mis à l'échelle dans le problème de flot. Cette sensibilité est du même ordre que celle constatée avec l'approche directe de la mise à conformité. En revanche, l'échelle des coûts (qui devient, par l'intermédiaire de M, l'échelle des capacités dans le problème de flot) n'a aucun impact sur la méthode duale.

Echelle données		Mise conformité (4.4)		Mise échelle dual (4.13)	
Capacités	Coûts	Itérations	Temps	Itérations	Temps
1000	1000	4180	30,8	210156	4,9
1000	10000	4182	31,6	213195	5
1000	100000	4183	31	232728	5,3
10000	1000	4752	42,6	292236	6,6
10000	10000	4806	41	297778	6,6
10000	100000	4750	41,1	298184	6,8
100000	1000	4993	45,1	357030	7,7
100000	10000	5056	45,7	370619	7,8
100000	100000	5104	47	363933	7,9

Tableau 4.8: Résultats numériques de l'algorithme de mise à l'échelle pour la tension de coût minimal, influence de l'échelle des données.

Il faut rappeler que la méthode utilisée pour résoudre le problème de flot n'est pas la plus efficace. Dans [Ahuj93] sont décrits des algorithmes dont l'efficacité théorique est bien meilleure, cependant ces algorithmes utilisent tous une mise à l'échelle des capacités (i.e. les coûts pour le problème de tension) et introduisent donc dans l'expression de leur complexité un terme fonction de B. Dans l'étude de la méthode de mise à conformité, nous nous sommes aperçu qu'introduire une sensibilité à B n'était pas efficace en pratique. Nous ne savons pas si cela se produirait avec l'approche duale, mais vus les très bons résultats fournis par la mise à l'échelle des coûts, nous n'avons pas désiré investir dans le développement d'une mise à l'échelle des capacités sans être sûrs du gain de performance.

Notre version de la mise à l'échelle ne précise pas l'ordre de parcours des noeuds pour rendre un flot ε-optimal (cf. algorithme 4.11). Une meilleure stratégie, l'*implémentation par vague* (*wave implementation*) proposée dans [Ahuj93], permet de réduire la complexité de la mise à l'échelle à $O(n^3 \log nA)$ opérations. Elle propose d'équilibrer les noeuds suivant un ordre topologique: en considérant le sous-graphe de G composé uniquement des arcs admissibles, un noeud est classé après tous ses prédécesseurs. L'existence de cet ordre est garanti (le sous-graphe est sans circuit), mais est remis en cause à chaque modification de potentiel (heureusement, l'ordre peut être rétabli en $O(1)$ opérations). Nous avons implémenté cette stratégie sans réel succès: le nombre

d'itérations et le temps de calcul ne changent pas de manière significative. Toutes les comparaisons et les discussions futures se feront donc sur notre version de la méthode.

4.7. Conclusion

Nous proposons le tableau 4.9 récapitulant, par ordre d'apparition, les méthodes présentées dans ce chapitre. La complexité de chacune est rappelée pour différentes conditions. Si aucune complexité n'est indiquée, cela signifie que la méthode telle que nous l'avons présentée ne peut pas être appliquée avec les conditions données.

Méthode	Complexité (coûts par morceaux)		Complexité (coûts dérivables)
	Données réelles	Données entières	
Mise à conformité directe (4.4)	$O(m^2\ (A+B)/\Delta)$	$O(m^2\ (A+B))$	
Mise à conformité avec échelle coûts (4.6)		$O(nm^2\ \log B)$	
Mise à conformité avec échelle capacités (4.7)		$O(nm^3\ \log A)$	
Mise à ε-conformité (4.8)			$O(m^2\ (A+B)/\Delta \log pA)$
Mise à l'échelle du dual (4.13)		$O(mn^2\ \log nA)$	

Tableau 4.9: Complexité des algorithmes de tension de coût minimal.

Nous proposons également un classement pratique des méthodes pour des coûts linéaires par morceaux et des données entières. Les algorithmes sont classés du moins efficace au plus efficace. Un rapport des vitesses d'exécution est effectué par rapport à l'algorithme le plus rapide. Il est calculé à partir de la dernière ligne des tableaux 4.8 et 4.4 qui nous semble être la situation la plus extrême (dimension du graphe et échelle de tension importantes). Bien évidemment, ce classement est discutable, notamment sur le fait que les résultats numériques dépendent énormément de la manière de programmer les méthodes, mais nous nous sommes efforcés d'implémenter au mieux et de la même manière chaque méthode afin de réduire ce genre de biais. Le classement théorique des méthodes est également rappelé.

Classement pratique		Classement théorique	
Algorithme	Vitesse	Algorithme	Complexité
Mise à conformité avec échelle coûts (4.6)	13,4	Mise à conformité directe (4.4)	$O(m^2\ (A+B))$
Mise à conformité avec échelle capacités (4.7)	10,7	Mise à conformité avec échelle capacités (4.7)	$O(nm^3\ \log A)$
Mise à conformité directe (4.4)	5,9	Mise à conformité avec échelle coûts (4.6)	$O(nm^2\ \log B)$
Mise à l'échelle du dual (4.13)	1	Mise à l'échelle du dual (4.13)	$O(mn^2\ \log nA)$

Tableau 4.10: Classement théorique et pratique des algorithmes de tension de coût minimal.

Indiscutablement, l'algorithme de mise à l'échelle est le plus efficace. Mais les tests effectués ici ont porté sur des graphes totalement aléatoires et sur la recherche d'une tension optimale sans solution de départ spécifique. Nous verrons dans le chapitre suivant que les graphes utilisés pour la synchronisation hypermédia ont une structure bien particulière et que l'algorithme de mise à l'échelle a un comportement moins satisfaisant dans cette situation. Il serait également intéressant de se pencher sur l'aspect temps réel de la synchronisation hypermédia et donc d'étudier le comportement des méthodes sur le changement tout simplement des données d'un arc. Pour la méthode de mise à conformité, repartir de la tension anciennement optimale pour trouver la nouvelle tension optimale ne semble pas difficile puisque tous les arcs seraient conformes sauf un, donc en $O(m(A+B))$ opérations, l'optimalité est atteinte. Pour la mise à l'échelle du dual, il est également possible de repartir de la tension optimale. Un seul arc n'est pas conforme, mais tout le processus de mise à l'échelle doit être fait et il est possible durant une phase d'amélioration d'altérer la conformité d'autres arcs à cause d'un ε trop grand. Une étude plus approfondie est donc nécessaire pour évaluer le nombre d'opérations effectuées par la méthode de mise à l'échelle dans une configuration temps réel.

CHAPITRE 5
TENSION DANS UN GRAPHE SÉRIE-PARALLÈLE

Au cours de notre introduction aux problématiques de synchronisation, nous avons souligné les différentes contraintes que les auteurs de documents hypermédia souhaitent utiliser (cf. section 1.1). Mais dans la première partie de notre étude, nous avons complètement éludé la nature même de ces contraintes et avons travaillé sur des graphes complètement aléatoires.

La raison de cette démarche est que nous ne pouvons pas nous limiter à l'étude de petits graphes, bien que les utilisateurs de ces systèmes synchronisés nous garantissent actuellement qu'une centaine de noeuds c'est déjà beaucoup. En effet, si des systèmes efficaces de synchronisation hypermédia sont mis en place dans les années à venir, les utilisateurs ne se contenteront plus de petits documents mais tenteront de synchroniser des documents pouvant se modéliser par des graphes avec probablement plusieurs milliers de noeuds. Cependant, nous ne disposons que de très peu d'exemples et aucun de grande taille. Il nous faut donc les générer. Mais ne voulant pas créer des graphes plus simples que ceux que proposeraient les concepteurs de documents synchronisés, nous avons pensé que générer des graphes totalement quelconques serait la meilleure manière d'éviter l'introduction d'un biais dans nos résultats.

Dans ce chapitre, nous proposons de considérer la nature particulière des contraintes de synchronisation. Nous verrons qu'une classe de graphes, les *graphes série-parallèles*, modélise des synchronisations très proches des exigences des auteurs de documents hypermédia, dans la limite de ce qu'un graphe temporel peut modéliser. Il faut noter que la résolution d'un problème d'optimisation sur ce genre de graphe est souvent plus simple que sur un graphe quelconque (e.g. [Datt99], [Baio97], [Bern87], [Taka82]).

La première partie de ce chapitre sera tout naturellement consacrée à la présentation de ces graphes très particuliers, aux limitations qu'ils apportent par rapport aux cas réels de synchronisation hypermédia et comment les détecter. Ensuite, nous proposons une méthode appelée *agrégation* qui exploite pleinement la structure particulière des graphes série-parallèles. Nous comparons alors cet algorithme avec les méthodes présentées au chapitre précédent pour la résolution du problème de tension de coût minimal sur des graphes série-parallèles (nous nous limitons à des coûts convexes linéaires par morceaux, cf. figure 2.11b).

Aux vues de l'efficacité de la méthode d'agrégation, nous avons pensé l'exploiter pour résoudre le problème de tension de coût minimal sur des graphes quelconques. Pour cela, nous proposons de décomposer les graphes en sous-graphes série-parallèles, appelés *composantes série-parallèles* et de résoudre le problème sur ces sous-graphes, avant de rassembler les composantes. L'idée ici étant que pour des cas réels de synchronisation hypermédia, les graphes seront *presque série-parallèles*.

5.1. Série-parallèle

Dans cette section, nous présentons la classe des graphes *série-parallèles* et leurs principales caractéristiques. Nous verrons que la structure récursive de ces graphes peut être représentée par un arbre binaire qui nous facilitera par la suite la conception d'algorithmes. Enfin, nous discutons et justifions quelques méthodes pour identifier un graphe série-parallèle et construire sa représentation sous forme d'arbre.

5.1.1. Graphe série-parallèle, SP-graphe

Il est possible de définir un graphe série-parallèle de plusieurs manières. Nous avons choisi une définition couramment employée (e.g. [Duff65], [Epps92], [Vald82]) qui repose sur une construction récursive du graphe très intuitive et très proche de la manière de créer des contraintes de synchronisation dans un document hypermédia.

5.1.1.1. Définition récursive

Un graphe est dit ***série-parallèle*** s'il est obtenu à partir d'un graphe à deux sommets reliés par un seul arc, en appliquant récursivement les deux opérations suivantes.

- L'***opération série***, appelée aussi ***sérialisation***, consiste à insérer un nouveau sommet de degrés entrant et sortant égaux à 1 en divisant un arc donné du graphe.

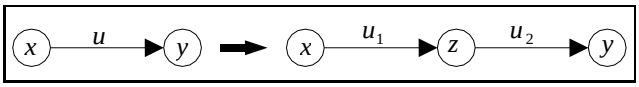


Figure 5.1: Opération série.

  Cette opération, que l'on notera S_u quand elle est appliquée à un arc u, associe une relation notée $\oplus$ entre deux arcs nouvellement créés. Ainsi, l'opération S_u remplace u par deux arcs, u_1 et u_2 par exemple, et elle associe u_1 et u_2 dans une ***relation série*** notée $u_1 \oplus u_2$.

- L'***opération parallèle***, appelée aussi ***parallélisation***, consiste à dupliquer un arc donné entre deux sommets.

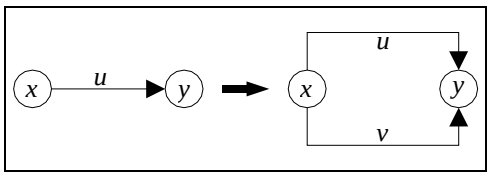


Figure 5.2: Opération parallèle.

  Cette opération, que l'on notera P_u quand elle est appliquée à un arc u, associe une relation notée $/\!/$ entre l'arc u et l'arc dupliqué. Ainsi, l'opération P_u crée un nouvel arc, v par exemple, et elle associe u et v dans une ***relation parallèle*** notée $u /\!/ v$.

Par la suite, un graphe série-parallèle sera appelé ***SP-graphe***, comme le propose par exemple [Chin99] (à ne pas confondre avec le même terme proposé dans [Scho95] pour des graphes série-parallèles sans arcs multiples, i.e. sans arcs avec la même origine et la même destination).

5.1.1.2. SP-relations

Par la suite, nous regroupons les relations $\oplus$ et $/\!/$ définies précédemment sous le terme ***SP-relations***. La propriété suivante montre la corrélation qu'il existe entre le nombre d'arcs et de noeuds d'un SP-graphe et le nombre de relations séries et parallèles qu'il contient.

> *Soit $G = (X;U)$ un SP-graphe, avec $m = |U|$ et $n = |X|$. Il contient $p = n - 2$ relations séries et $q = m - n + 1$ relations parallèles. Par conséquent, il possède globalement $p + q = m - 1$ SP-relations.* (5.1)

Preuve:
Une opération série ajoute 1 noeud à chaque application alors qu'une opération parallèle n'en ajoute aucun. Le graphe de départ contenant 2 noeuds, il faut $p = n - 2$ opérations séries pour construire le graphe final. De même, chaque opération série ou parallèle ajoute 1 arc. Comme le graphe de départ contient 1 arc et qu'il y a p arcs créés par les opérations séries, il faut $q = m - (p + 1) = m - n + 1$ opérations parallèles. □

5.1.1.3. Tension principale

D'après la définition d'un SP-graphe proposée précédemment, il est très facile de vérifier qu'il contient un seul noeud ***source*** (i.e. ne possédant aucun prédécesseur) et un seul noeud ***puits*** (i.e. ne possédant aucun successeur). Soit une tension θ sur un SP-graphe G, on appelle ***tension principale*** et on note $\overline{\theta}$ la tension entre le noeud source et le noeud puits de G.

5.1.2. Arbre binaire de décomposition, SP-arbre

Les SP-graphes ayant une structure récursive très particulière, il est intéressant, pour un développement plus aisé d'algorithmes efficaces, de représenter ces graphes sous une forme plus adaptée à leur structure. Nous montrons tout d'abord comment modéliser ces graphes sous forme d'expression de SP-relations. Ensuite, nous proposons une représentation équivalente par un arbre binaire.

5.1.2.1. Représentation sous forme d'expression

Revenons à la définition d'un SP-graphe. Ce dernier est créé en appliquant une opération série ou parallèle sur l'un des arcs du graphe à chaque itération du processus de construction. Considérons tout d'abord la première itération, le graphe se limite à un simple arc u (cf. figure 5.3a).

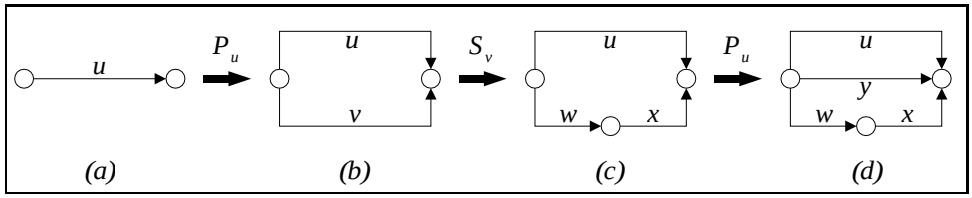


Figure 5.3: Un exemple de construction d'un SP-graphe.

L'opération S_u ou P_u est alors appliquée, à l'itération 2 le graphe possède donc deux arcs qui sont liés soit par la relation série $v \oplus w$, soit par la relation parallèle $u \mathbin{/\!/} v$. Supposons l'opération parallèle, à l'itération 2 le graphe est donc $u \mathbin{/\!/} v$ (cf. figure 5.3b). L'un des deux arcs subit ensuite une autre opération série ou parallèle, disons l'opération S_v. Le graphe est alors $u \mathbin{/\!/} (w \oplus x)$ (cf. figure 5.3c). On peut ainsi en déduire la propriété suivante.

> *Un SP-graphe peut toujours être représenté par une expression de SP-relations qui n'est pas toujours unique.* *(5.2)*

Preuve:
Nous avons vu qu'en suivant la construction d'un SP-graphe, on peut toujours écrire une expression de SP-relations le représentant. Poursuivons notre exemple en ajoutant l'opération P_u. On obtient alors le graphe $(u \mathbin{/\!/} y) \mathbin{/\!/} (w \oplus x)$ (cf. figure 5.3d). Supposons maintenant la construction $P_u; P_v; S_v$ illustrée par la figure 5.4, on obtient alors l'expression $u \mathbin{/\!/} (w \mathbin{/\!/} (x \oplus y))$ qui représente un graphe identique à celui obtenu dans notre premier exemple de construction. □

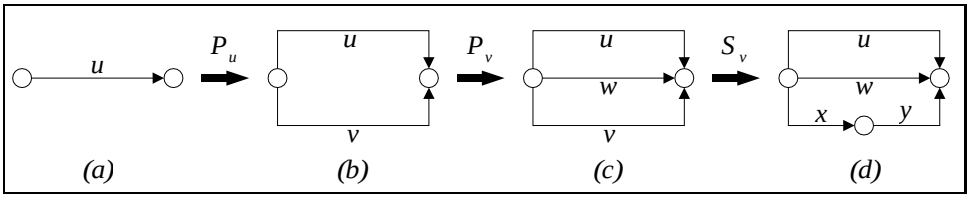


Figure 5.4: Un autre exemple de construction d'un SP-graphe.

Nous avons défini une SP-relation comme étant une relation entre deux arcs. Cependant, d'après le principe de construction d'un SP-graphe, on s'aperçoit que ces relations, au départ établies entre deux arcs, deviennent au cours du processus de construction des relations entre sous-graphes (car un arc peut être remplacé à tout moment par une SP-relation). Ainsi, on introduit la notion de ***SP-relation simple*** pour désigner une SP-relation entre deux arcs. Lors de la construction d'un SP-graphe, la dernière opération série ou parallèle crée forcément une SP-relation simple, ce qui nous permet d'affirmer la propriété suivante qui nous sera très utile par la suite.

Un SP-graphe possède au moins une SP-relation simple. *(5.3)*

5.1.2.2. Représentation sous forme d'arbre

Les SP-relations sont binaires, il est donc très facile et immédiat de représenter l'expression d'un SP-graphe par un arbre binaire que certains appellent ***arbre binaire de décomposition*** (e.g. [Vald82], [Datt99]), et d'autres ***SP-arbre*** (e.g. [Bodl96]). La figure 5.5 illustre cette représentation pour l'expression $(u \mathbin{/\!/} y) \mathbin{/\!/} (w \oplus x)$ de l'exemple précédent.

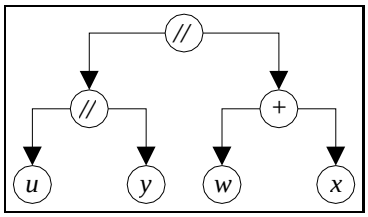


Figure 5.5: Un exemple de SP-arbre.

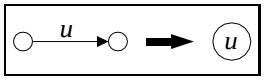


Figure 5.6: SP-arbre représentant un graphe avec un seul arc.

Nous proposons ici une définition équivalente de l'arbre binaire de décomposition qui met en évidence la relation qu'il existe entre un sous-arbre d'un SP-arbre et un sous-graphe du SP-graphe associé.

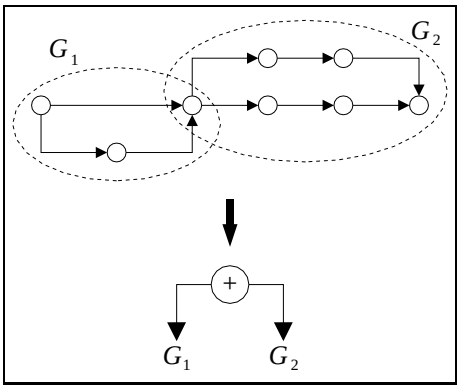


Figure 5.7: SP-arbre représentant une relation série entre deux sous-graphes.

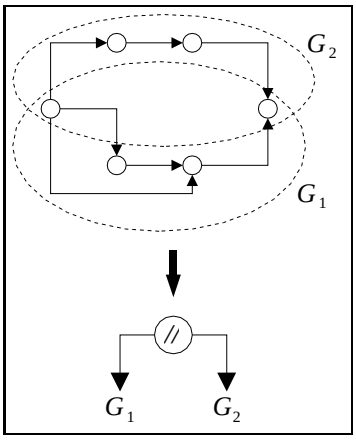


Figure 5.8: SP-arbre représentant une relation parallèle entre deux sous-graphes.

Voici la définition récursive d'un SP-arbre.

- Le SP-arbre représentant un SP-graphe avec seulement un arc reliant deux sommets est formé d'un seul noeud qui représente l'unique arc du graphe (cf. figure 5.6).
- Un SP-arbre représentant un SP-graphe avec au moins une SP-relation est formé d'un noeud racine représentant une SP-relation et de deux SP-arbres représentant les deux sous-graphes impliqués dans la relation en question (cf. figures 5.7 et 5.8).

En résumé, il est toujours possible de représenter un SP-graphe par une expression de SP-relations ou par un SP-arbre, mais partant d'un SP-graphe déjà construit, il est souvent possible de trouver plusieurs représentations. Pour obtenir une représentation unique, il suffit de considérer les SP-relations non plus binaires mais s'appliquant à un nombre quelconque d'arcs ou de sous-graphes (cf. [Bodl96]). Dans notre utilisation de cette représentation, la pluralité des représentations ne pose aucun problème.

5.1.3. Opérateurs de synchronisation série-parallèles

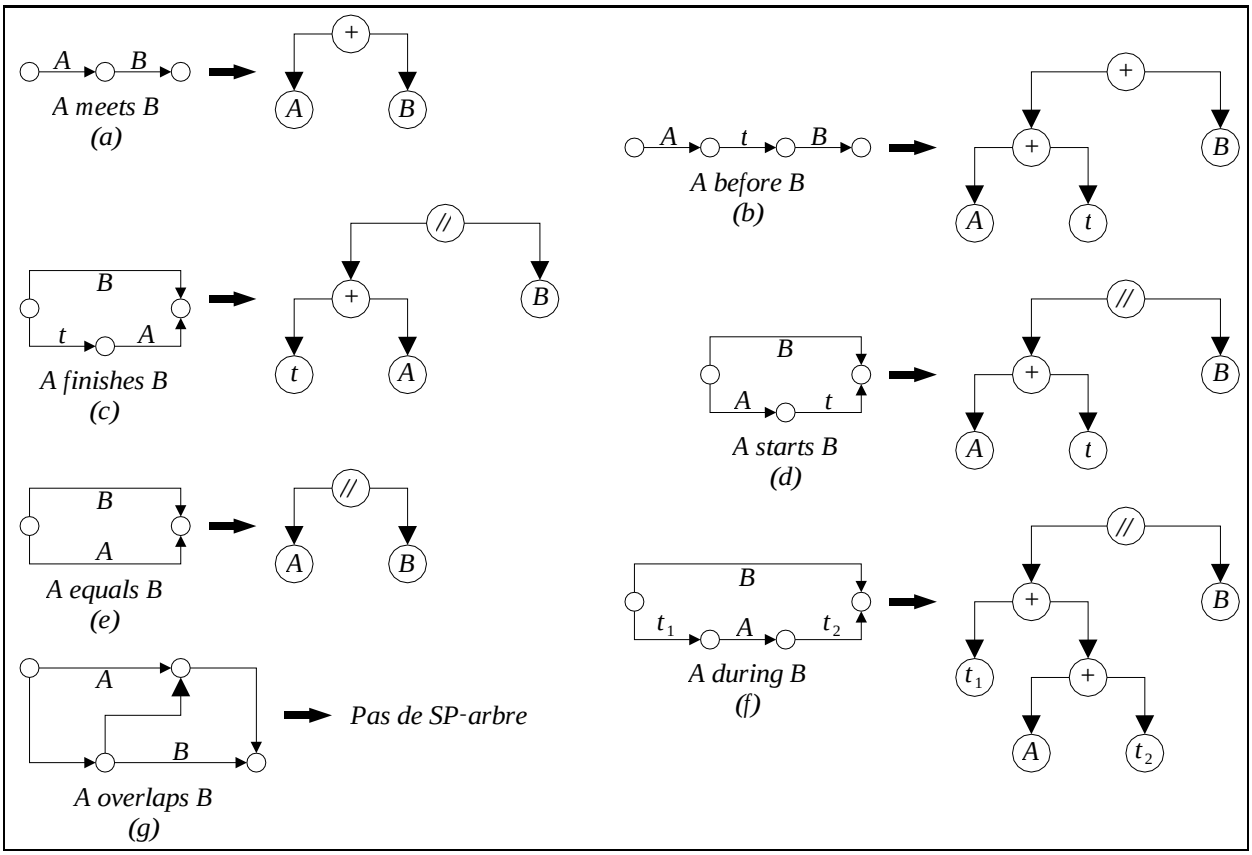


Figure 5.9: SP-arbres des opérateurs de synchronisation, partie 1.

Dans le chapitre 2, nous avons présenté les limites qu'une modélisation sous forme de graphe implique aux contraintes de synchronisation d'un document hypermédia. Parmi tous les opérateurs préconisés dans [Jour99], nous n'en avons retenu qu'une partie résumée par la figure 1.2 (les opérateurs de l'algèbre de Allen proposée dans [Alle83] et leurs principales disjonctions). Nous proposons d'identifier ici les opérateurs qui ont une représentation série-parallèle, afin de mieux juger de la pertinence d'une restriction de la modélisation des

contraintes de synchronisation à un SP-graphe. La figure 5.9 montre le SP-arbre (quand cela est possible) associé à chaque opérateur retenu au chapitre 2.

Nous avons volontairement écarté les opérateurs *share-start* et *share-end* de cette première analyse, puisqu'ils ne peuvent pas être représentés par des SP-graphes. Cependant ils peuvent tout de même être utilisés dans un graphe des contraintes série-parallèle. En effet, n'oublions pas que ce graphe est temporel, ce qui signifie qu'il possède un seul noeud source et un seul noeud puits. Ainsi, pour l'opérateur *share-start* par exemple, les scénarios A et B lancés en parallèle se rencontrent forcément à un moment ou à un autre comme le montre la figure 5.10a. En supposant que les sous-graphes G_1 et G_2 sont série-parallèles, il est facile de vérifier que le graphe est série-parallèle (cf. figure 5.10b). Le même raisonnement peut être appliqué à l'opérateur *share-end*.

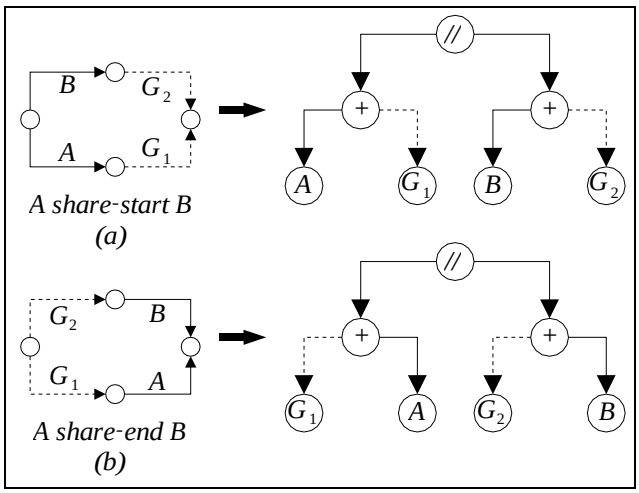


Figure 5.10: SP-arbres des opérateurs de synchronisation, partie 2.

Finalement, en n'utilisant que des opérateurs (y compris les SP-relations) avec une représentation série-parallèle pour construire le graphe des contraintes, on obtient forcément un SP-graphe. Se limiter à ces opérateurs n'est pas si restrictif, puisque seul l'opérateur *overlaps*, parmi tous ceux modélisables par un graphe temporel, n'est pas représentable par un SP-graphe, et ce n'est apparemment pas le plus important.

5.1.4. Construction d'un SP-arbre

Le problème de reconnaître un SP-graphe est connu depuis longtemps comme étant un problème facile qui peut être résolu en temps linéaire (cf. [Vald82]). Les différents algorithmes proposés dans la littérature peuvent être adaptés très facilement et sans altération de leur complexité pour construire le SP-arbre au moment de la reconnaissance. Ces méthodes étant très efficaces, notre discussion ne portera pas ici sur leur complexité, mais plutôt sur leur manière d'aborder le problème et de reconnaître un SP-graphe. Car dans la seconde partie du chapitre, nous ne nous contenterons pas seulement d'identifier un SP-graphe, mais nous tenterons d'extraire des *composantes série-parallèles* de graphes quelconques. Afin de les adapter à cet objectif, nous présentons les deux approches que nous avons retenues de la littérature et de nos propres réflexions: l'*approche par réduction* et l'*approche par chemin*.

Une petite remarque tout de même sur la complexité de ces méthodes. Les auteurs affirment que leur complexité est linéaire, c'est-à-dire en $O(m + n)$ opérations dans [Vald82]. Il faut noter que cela est vrai seulement avec une structure de graphe particulière qui est souvent passée sous silence. Cependant, vue l'efficacité des

méthodes proposées, il faut s'inquiéter du temps de construction de cette structure. Dans [Scho95] par exemple, il s'agit d'une matrice d'adjacence noeud-noeud qui nécessite donc $O(n^2)$ opérations pour être construite. Dans [Vald82], identifier si un arc possède un double (i.e. un arc avec même origine et même destination) est effectué en $O(1)$ opérations, ce qui est possible seulement après un tri des arcs entrants et sortants de chaque noeud et nécessite $O(m \log m)$ opérations. Le calcul de la complexité des méthodes que nous présentons ici (et même ailleurs dans le document) considère une structure de données raisonnablement efficace, mais sans artifice particulier.

Pour les besoins de nos algorithmes, nous avons choisi une notation récursive des arbres binaires (cf. chapitre 2). Nous rappelons seulement qu'un arbre de racine a, de sous-arbre gauche T_l et de sous-arbre droit T_r sera noté $(a; T_l; T_r)$. Nous introduisons également ci-dessous les procédures inverses des opérations série et parallèle, appelées ***SP-réductions***.

- La ***réduction série***, notée S_x^{-1}, remplace la SP-relation $(y; x) \oplus (x; z)$ par un arc $(y; z)$.
- La ***réduction parallèle***, notée P_u^{-1}, remplace la SP-relation $u \mathbin{/\!/} v$ par l'arc u.

5.1.4.1. Approche par réduction

Cette première approche est la plus répandue, car la plus intuitive. Elle repose sur un constat évident que nous avons établi précédemment: un graphe série-parallèle possède au moins une SP-relation simple. L'idée est donc d'identifier une telle relation et d'appliquer la réduction correspondant, afin de remonter le processus de construction du SP-graphe.

Cette méthode a tout d'abord été proposée dans [Vald82]. On la retrouve ensuite dans [Scho95] avec une petite amélioration: au départ tous les arcs multiples sont éliminés (il n'existe donc plus de relations parallèles simples), ensuite on détecte les relation séries et des relations *parallèles-et-séries* (composition d'une relation parallèle avec une relation série: $u \mathbin{/\!/} (v \oplus w)$). La détection est alors plus facile (on regarde uniquement les noeuds) et la réduction est plus efficace puisque dans le cas d'une relation parallèle-et-série, deux arcs et un noeud sont éliminés par rapport à une relation parallèle qui ne supprime qu'un seul arc. Cependant, nous ne retenons pas cette amélioration, puisque fondamentalement elle ne change pas l'approche et se prête difficilement à notre soucis futur qui est d'extraire les composantes série-parallèles d'un graphe quelconque. Une variante similaire a également été proposée dans [Bodl96] qui offre 18 réductions, cependant la plupart ne sont valables que pour un graphe non orienté.

Nous présentons donc ici une version générique de la méthode, en construisant en même temps un SP-arbre représentant le supposé SP-graphe. Pour cela, on introduit la fonction t qui associe un SP-arbre t_u à chaque arc u du graphe G. Au départ, chaque arc possède un SP-arbre avec un seul noeud qui est l'arc lui-même. Voici ce qui se produit lors d'une réduction.

- La réduction série S_x^{-1} supprime les deux arcs $u = (y; x)$ et $v = (x; z)$ de la relation $u \oplus v$, et crée un arc $w = (y; z)$. Le SP-arbre de w est alors $(\oplus; t_u; t_v)$.
- La réduction parallèle P_u^{-1} supprime l'arc v de la relation $u \mathbin{/\!/} v$. Le SP-arbre de u devient alors $(\mathbin{/\!/}; t_u; t_v)$.

L'algorithme 5.1 décrit toute la procédure de reconnaissance d'un SP-graphe et la construction de son SP-arbre. Il est facile de vérifier qu'à chaque réduction, si le graphe est série-parallèle, il le reste et que par conséquent il existe toujours au moins une SP-relation simple. En revanche, si le graphe n'est pas série-parallèle, il y aura une réduction après laquelle il n'y aura plus de SP-relation simple. Ainsi, si l'algorithme parvient à réduire un

graphe à un seul arc, alors le SP-arbre associé à l'arc représente le SP-graphe initial. Sinon, cela signifie que le graphe n'était pas série-parallèle.

```
Algorithme 5.1: construireSPArbre(graphe G = (X;U), arbre T).
pour tout u ∈ U faire t_u ← (u;∅;∅);
N ← X; A ← U;

tant que N ≠ ∅ faire
 /* Réductions séries. */
 tant que N ≠ ∅ faire
  choisir x ∈ N; N ← N \ {x};

  si d_x^+ = 1 et d_x^- = 1 alors
   soit u_1 = (y;x) et u_2 = (x;z) ∈ U;
   u ← (y;z);
   t_u ← (⊕;t_{u_1};t_{u_2});
   U ← U \ {u_1;u_2} ∪ {u};
   A ← A \ {u_1;u_2} ∪ ω^+(y);
   X ← X \ {x};
  fin si;
 fin tant que;

 /* Réductions parallèles. */
 tant que A ≠ ∅ faire
  choisir u = (x;y) ∈ A; A ← A \ {u};

  tant que ∃ v = (x;y) ∈ U tel que v ≠ u faire
   t_u ← (//;t_u;t_v);
   U ← U \ {v}; A ← A \ {v};
   N ← N ∪ {x}; N ← N \ {y};
  fin tant que;
 fin tant que;
fin tant que;

si |U| = 1 alors soit u ∈ U; T = t_u;
sinon arrêter; /* Le graphe n'est pas série-parallèle. */
```

L'examination d'un noeud se fait en $O(1)$ opérations et celle d'un arc en $O(m)$ (il faut regarder tous les arcs ayant la même origine). Au départ de l'algorithme, tous les noeuds sont examinés ainsi que tous les arcs. La première passe nécessite donc $O(n + m^2)$ opérations. Ensuite, un noeud ne sera examiné qu'après une réduction parallèle. De la même manière, un arc ne sera examiné que suite à une réduction série. Sachant qu'il y a $n - 2$ relations série et $m - n + 1$ relations parallèles, les passes suivantes de l'algorithme nécessitent $O(n + m^2)$ opérations. Enfin, une SP-réduction consiste en l'agrégation d'un SP-arbre, $O(1)$ opérations, et en la suppression d'un noeud ou d'un arc (le nombre d'opérations ici est fortement dépendant de la structure employée pour représenter le graphe, une structure raisonnable nécessite au pire $O(\log m)$ opérations). Sachant qu'il y a au plus $m - 1$ SP-relations, les réductions coûtent $O(m \log m)$ opérations. Au final, l'algorithme nécessite donc $O(m^2)$ opérations.

5.1.4.2. Approche par chemin

L'approche que nous proposons ici est plus globale que celle présentée précédemment. Visuellement, il est relativement facile d'identifier un SP-graphe. La raison est simplement que les chemins d'un tel graphe sont organisés de manière très particulière. Dans [Epps92], cette organisation est formalisée par le concept de *décomposition en oreille* (*ear decomposition*). Notre intérêt portera plutôt sur deux types de noeuds jouant un rôle très particulier: les ***noeuds de branchement*** (dont le degré sortant est supérieur à 1) et les ***noeuds de synchronisation*** (dont le degré entrant est supérieur à 1).

Dans un graphe série-parallèle, de tels noeuds représentent respectivement le début et la fin de SP-relations parallèles. On nomme ***branchement*** deux arcs sortants d'un noeud de branchement (i.e. le début d'une SP-relation parallèle). De même, une ***synchronisation*** représente deux arcs entrants d'un noeud de synchronisation (i.e. la fin d'une SP-relation parallèle). Par simplicité, on pourra parfois confondre le branchement (respectivement la synchronisation) avec son noeud de branchement (respectivement de synchronisation).

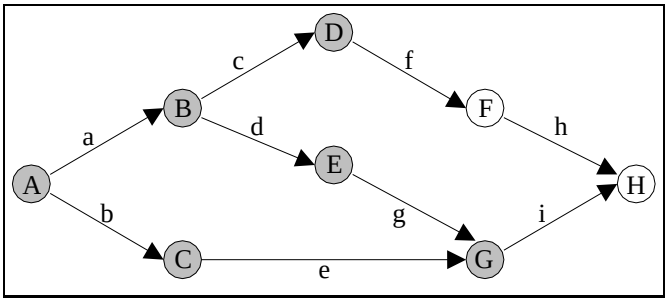


Figure 5.11: Un exemple de branchements.

L'approche proposée ici est basée sur un parcours du graphe dans un ordre topologique (i.e. un noeud est visité après tous ses prédécesseurs). Notons S_k l'ensemble des noeuds marqués par le parcours du graphe à l'itération k. Un branchement x est dit ***fermé*** à l'itération k s'il existe un noeud de synchronisation $y \in S_k$ tel qu'il existe deux chemins P_1 et P_2 entre x et y distincts (i.e. n'ayant aucun arc en commun), et contenant chacun l'un des deux arcs du branchement; les deux arcs entrants de y appartenant chacun à l'un des chemins forment alors une synchronisation y. Un branchement non fermé est dit *ouvert*. On dit également que la synchronisation y *ferme* le branchement x. A un branchement fermé, on associera une synchronisation particulière, qui est la première synchronisation qui ferme le branchement dans le parcours topologique. Les chemins qui permettent la fermeture du branchement (dans la définition P_1 et P_2) seront appelés par la suite ***chemins de fermeture***.

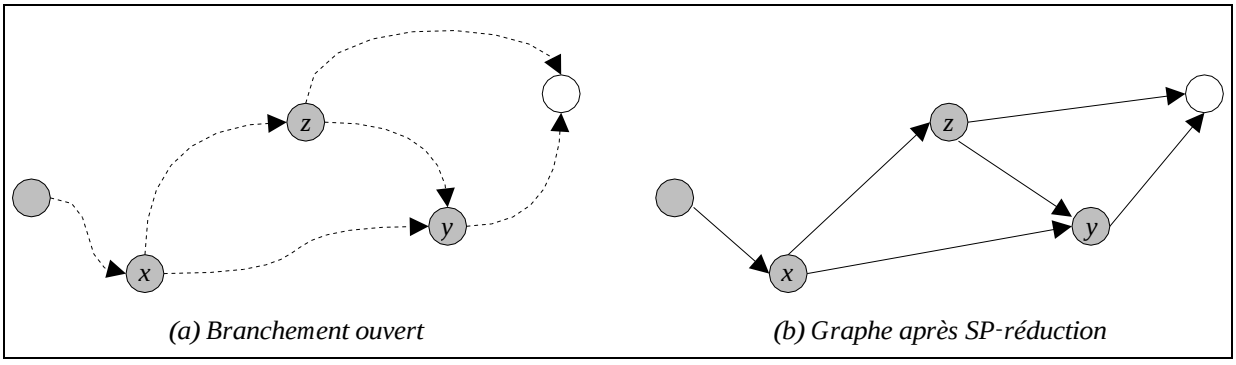


Figure 5.12: Un exemple de branchement ouvert sur un chemin de fermeture.

Pour illustrer ces définitions, considérons la figure 5.11. Les noeuds grisés représentent ceux visités à une itération donnée k. A et B sont des noeuds de branchement, G et H des noeuds de synchronisation. $(a; b)$ est un branchement fermé à l'itération k, la synchronisation associée étant $(e; g)$. En revanche, le branchement $(c; d)$ est ouvert à l'itération k. Les chemins de fermeture du branchement $(a; b)$ sont $(a; d; g)$ et $(b; e)$. Le graphe de notre exemple n'est pas série-parallèle. Intuitivement, on s'aperçoit que le problème vient du branchement ouvert $(c; d)$. Tentons alors de prouver la proposition suivante.

> *Un graphe est série-parallèle si et seulement si, à toute itération du parcours et pour tout branchement x fermé, il existe deux chemins de fermeture entre le branchement x et la synchronisation associée y qui ne possèdent pas de branchement ouvert.* (5.4)

Preuve:
($\Rightarrow$) A une itération donnée, s'il existe un branchement z ouvert sur l'un des chemins de fermeture d'un branchement fermé x associé à une synchronisation y, cela donne un graphe dont l'allure générale est présentée par la figure 5.12a (les noeuds grisés représentent ceux visités). En tentant une réduction par l'algorithme 5.1, cela aboutit au mieux au graphe illustré par la figure 5.12b. La réduction ne peut donc pas se terminer, le graphe n'est donc pas série-parallèle.
($\Leftarrow$) La preuve de cette implication peut être faite par la justification de l'algorithme 5.4 présenté dans la suite de cette section. En effet, ce dernier construit un SP-arbre associé à un graphe en vérifiant que tout branchement fermé ne possède pas de branchement ouvert dans ses chemins de fermeture. $\square$

Pour vérifier la proposition 5.4 (afin de déterminer si un graphe est série-parallèle), on propose un système de marquage qui permet d'établir, à l'arrivée sur une synchronisation, les branchements qui se ferment et si un branchement ouvert est présent sur les chemins de fermeture. On note Δ_u la signature d'un arc u, elle se résume à un noeud de branchement. $\Delta_u = x$ signifie que x est le dernier (par rapport à l'ordre topologique) branchement non fermé sur les chemins entre la source et l'arc u. On note également Δ_x la signature d'un noeud x, elle consiste en un couple $\Delta_x = (l_x; d_x)$. Si le noeud n'est pas un noeud de branchement alors $d_x = 0$, sinon d_x indique le nombre de branchements de noeud x ouverts. l_x indique le *niveau* du branchement, e.g. si $l_x = 2$, alors il existe deux branchements ouverts dans les chemins entre la source et x (x compris).

La signature d'un arc ou d'un noeud n'est pas définitive, elle peut changer au cours du parcours topologique. En effet, lorsque deux arcs u et v de même signature x (à l'itération k) se rejoignent à un noeud de synchronisation y (à l'itération $k + 1$), ils forment une synchronisation et ferment donc un branchement de noeud x (cf. figure 5.13a). La signature de leur branchement x est alors modifiée: d_x est décrémenté de 1. Si d_x atteint alors 0, cela signifie que tous les branchements de noeud x sont fermés. La signature des arcs u et v doit changer, elle devient le branchement ouvert avant x, i.e. la signature du ou des arcs entrants au branchement x (cf. figure 5.13b). Idéalement il faudrait changer la signature de tous les arcs de signature x, mais pour les besoins de l'algorithme (qui n'effectue qu'un parcours topologique et ne revient donc pas sur des arcs déjà utilisés) cette modification n'est pas nécessaire.

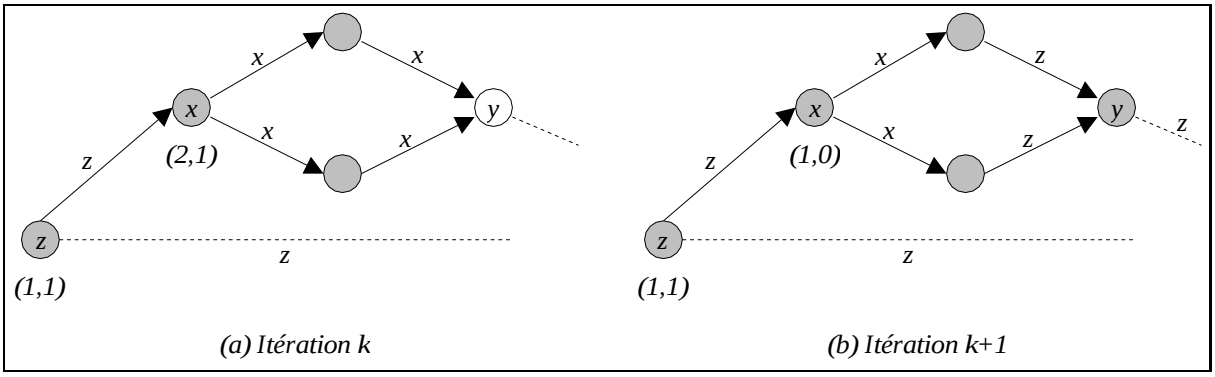


Figure 5.13: Un exemple de changement de signature.

L'algorithme 5.2 proposé effectue un parcours topologique du graphe, en marquant progressivement les arcs et les noeuds de la signature Δ. Il est supposé que le graphe n'a qu'une seule source et au moins un arc, sinon il est certain qu'il n'est pas série-parallèle. Partant de la source, l'algorithme va ainsi marquer les noeuds et les arcs du graphe. A chaque noeud de synchronisation y, il se charge de modifier certaines signatures pour représenter la fermeture de branchements associés (cf. algorithme 5.3). Si à la fin de cette étape, deux arcs entrants u_1 et u_2 n'ont pas la même signature, cela signifie que le graphe n'est pas série-parallèle.

Algorithme 5.2: **détecterSPGraphe(graphe** $G=(X;U)$**)**.

```
pour tout x ∈ X faire
 m_x ← d_x^-; /* Marque pour le parcours topologique. */
 Δ_x ← ∅; /* Permet de savoir si un noeud a été visité. */
fin pour;

S ← {x ∈ X, m_x = 0};
si |S| > 1 ou |U| = 0 alors arrêter; /* Le graphe n'est pas série-parallèle. */

tant que S ≠ ∅ faire
 choisir x dans S;
 S ← S \ {x};

 si d_x^- = 0 alors Δ_x ← (0; d_x^+ - 1);
 sinon
  si d_x^- > 1 alors fermerBranchements(x,Δ);
  soit u ∈ ω^-(x); b ← Δ_u;

  si d_x^+ = 0 et S ≠ ∅ alors /* Plusieurs noeuds puits. */
   arrêter; /* Le graphe n'est pas série-parallèle. */

  si d_x^+ ≤ 1 alors
   Δ_x ← (l_b; 0);
   Δ' ← b;
  sinon
   Δ_x ← (l_b + 1; d_x^+ - 1);
   Δ' ← x;
  fin si;

  pour tout v = (x; z) ∈ ω^+(x) faire
   Δ_v ← Δ';
   m_z ← m_z - 1;
   si m_z = 0 alors S ← S ∪ {z};
  fin pour;
 fin si;
fin tant que;

si ∃x tel que Δ_x = ∅ alors /* Un circuit a stoppé le parcours topologique. */
 arrêter; /* Le graphe n'est pas série-parallèle. */
sinon /* Le graphe est série-parallèle. */
```

Algorithme 5.3: **fermerBranchements(noeud** x**,signature** Δ**)**.

```
D ← ω^-(x);
trier D dans l'ordre décroissant des signatures;

tant que |D| > 1 faire
 soit u et v les deux premiers arcs de D;
 D ← D \ {u};
 si Δ_u ≠ Δ_v alors arrêter; /* Le graphe n'est pas série-parallèle. */
 s ← Δ_u;
 d_s ← d_s - 1;

 si d_s = 0 alors
  l_s ← l_s - 1;
  soit w = (y; s) ∈ U;
  pour tout a ∈ ω^-(x) tel que Δ_a = Δ_u faire Δ_a ← Δ_w;
  trier D dans l'ordre décroissant des signatures;
 fin si;
fin tant que;
```

En effet, il existe deux chemins P_1 et P_2 entre la source et respectivement u_1 et u_2. Ces chemins ont au moins un noeud de branchement en commun (au moins la source). Considérons x le dernier noeud (dans le parcours topologique) vérifiant cette condition. S'il n'y avait pas de branchement ouvert sur les chemins de fermeture

entre x et y, u_1 et u_2 auraient la même signature x; s'ils ne l'ont pas, cela signifie qu'au moins un des chemins de fermeture entre x et y possède un branchement ouvert dont u_1 ou u_2 portent la signature.

L'algorithme 5.3 est chargé de fermer les branchements à un noeud de synchronisation donné x, et de modifier les signatures en conséquence. Il doit donc vérifier la signature des arcs entrants de x deux à deux, et s'ils ont une signature identique, fermer le branchement correspondant s'il est ouvert. Pour un parcours efficace des arcs, nous proposons de les trier dans un ensemble D, dans l'ordre décroissant de leur signature, i.e. u avant $v \Leftrightarrow l_{\Delta_u} > l_{\Delta_v}$ ou $(l_{\Delta_u} = l_{\Delta_v}$ et $\Delta_u > \Delta_v)$ (on suppose un ordre quelconque sur les noeuds, l'important étant que dans l'ensemble D, tous les arcs de même signature soient côte-à-côte). Ainsi, en étudiant les deux premiers arcs de D, on considère le branchement de niveau le plus élevé qu'il faut absolument fermer avant tout branchement de niveau inférieur. Cette procédure effectue donc $O(m \log m)$ opérations. L'algorithme complet 5.2 nécessite $O(nm \log m)$ opérations, puisqu'il effectue un simple parcours en appelant l'algorithme 5.3 à chaque noeud visité.

L'algorithme 5.2 ne permet que la détection d'un graphe série-parallèle, il est simple ensuite de l'adapter, sans altération de la complexité, pour qu'il construise un SP-arbre associé. L'idée consiste à réduire le graphe par une opération série à chaque fois qu'un noeud de degrés entrant et sortant égaux à 1 est visité. Lorsqu'un branchement est fermé, cela signifie qu'une opération parallèle a été détectée, il suffit alors de la réduire. Le SP-arbre est construit une fois tout le graphe visité (il se réduira alors à un simple arc).

Algorithme 5.4: **construireSPArbreBis(graphe** $G=(X;U)$**,arbre** T**).**

```
pour tout u ∈ U faire t_u ← (u;∅;∅);
pour tout x ∈ X faire m_x ← d_x^-;
S ← {x ∈ X, m_x = 0};

tant que S ≠ ∅ faire
 choisir x dans S;
 S ← S \ {x};

 si d_x^- = 0 alors Δ_x ← 0;
 sinon
  si d_x^- > 1 alors fermerBranchementsBis(x,Δ,t);
  soit u = (b;x) ∈ U;

  si d_x^+ = 1 alors /* Réduction série. */
   soit v = (x;z) ∈ U;
   w ← (b;z);
   t_w ← (⊕;t_u;t_v);
   U ← U \ {u;v} ∪ {w};
   X ← X \ {x};
  sinon
   Δ_x ← Δ_b + 1;

   pour tout v = (x;z) ∈ ω^+(x) faire
    m_z ← m_z − 1;
    si m_z = 0 alors S ← S ∪ {z};
   fin pour;
  fin si;
 fin si;
fin tant que;

si |U| = 1 alors soit u ∈ U; T = t_u;
sinon arrêter; /* Le graphe n'est pas série-parallèle. */
```

Nous proposons l'algorithme 5.4 accompagné de 5.5 qui permet une telle construction. La gestion des signatures des arcs et des noeuds est plus simple avec cet algorithme, car les SP-réductions déduisent automatiquement la plupart des signatures. En effet, la signature Δ_x d'un noeud x est limitée à l_x (car d_x est toujours

égal à $d_x^+ - 1$), et la signature d'un arc $u = (x; y)$ n'est plus nécessaire, car elle est toujours égale à x. Pour l'algorithme 5.1 (construction d'un SP-arbre par réduction), nous avons vu que les réductions nécessitent $O(m \log m)$ opérations, l'algorithme 5.4 a donc la même complexité que l'algorithme 5.2 de détection, autrement dit $O(nm \log m)$ opérations.

```
Algorithme 5.5: fermerBranchements(noeud x,signature Δ,vecteur t).
D ← ω⁻(x);
trier D dans l'ordre décroissant des signatures;

tant que |D| > 1 faire
 soit u = (s;x) et v = (y;x) les deux premiers arcs de D;
 D ← D \ {u};
 si s ≠ y alors arrêter; /* Le graphe n'est pas série-parallèle. */

 /* Réduction parallèle. */
 t_v ← (Ⓟ; t_u; t_v);
 U ← U \ {u};

 si d_s^+ = 1 alors
  /* Réduction série. */
  soit a = (e;s) ∈ U;
  w ← (e;x);
  t_w ← (⊕; t_a; t_v);
  U ← U \ {a;v} ∪ {w};
  X ← X \ {s};

  D ← D \ {v} ∪ {w};
  trier D dans l'ordre décroissant des signatures;
 fin si;
fin tant que;
```

5.2. Méthode d'agrégation

Nous proposons ici une méthode pour résoudre le problème de la tension de coût minimal sur des graphes série-parallèles (avec des coûts linéaires par morceaux, cf. figure 2.11b). Cette méthode est appelée *agrégation* car elle repose sur une approche qui consiste à remplacer récursivement une SP-relation par un seul arc aux propriétés équivalentes. Cette technique d'agrégation exploite la notion de *fonction de coût minimal*, que nous présentons avant de s'intéresser à l'agrégation d'une relation série, puis d'une relation parallèle. Nous expliquons enfin la méthode complète et la comparons aux méthodes présentées dans le chapitre précédent.

5.2.1. Fonction de coût minimal

L'idée de la méthode d'agrégation consiste à remplacer un SP-graphe par un arc, dit ***agrégé***, aux propriétés équivalentes. Bien évidemment, cette opération, appelée ***agrégation***, élimine certaines informations concernant le graphe pour n'en garder que les pertinentes pour le problème qui nous intéresse, à savoir optimiser la tension du graphe. La principale information agrégée que l'on considère tout d'abord est la ***fonction de coût minimal*** d'un graphe G, ou de son arc agrégé u_G. Cette fonction, notée C_G, représente le coût minimal de la tension de G, pour une tension principale $\overline{\theta}$ forcée à une valeur donnée x.

$$C_G(x) = \min\{\sum_{u \in U} c_u(\theta_u),\ \theta \textit{ tension},\ \overline{\theta} = x\} \qquad (5.5)$$

L'intérêt de l'agrégation, avec cette notion de fonction de coût minimal, est de pouvoir manipuler un seul arc u_G au lieu de tout un SP-graphe G, comme s'il s'agissait d'un arc quelconque avec une fonction de coût linéaire par morceaux. Nous montrons maintenant que la fonction de coût minimal d'une SP-relation est toujours convexe si les fonctions de coût sur les arcs non agrégés sont convexes.

5.2.1.1. Agrégation série

Considérons deux sous-graphes série-parallèles G_1 et G_2, et supposons que leur fonction de coût minimal C_{G_1} et C_{G_2} sont connues. Si l'on considère la SP-relation $G_1 \oplus G_2$ (cf. figure 5.7), G_1 et G_2 partagent seulement un noeud, la SP-relation n'ajoute donc pas de contrainte de tension supplémentaire entre eux. Mais si l'on ajoute la contrainte que la tension principale de $G_1 \oplus G_2$ doit être égale à x, cela impose à x_1 et x_2, les tensions principales respectivement de G_1 et G_2, que $x = x_1 + x_2$. La fonction de coût minimal $C_{G_1 \oplus G_2}$ de la SP-relation $G_1 \oplus G_2$ est donc:

$$C_{G_1 \oplus G_2}(x) = \min_{x=x_1+x_2} C_{G_1}(x_1) + C_{G_2}(x_2) \tag{5.6}$$

Cela signifie que $C_{G_1 \oplus G_2}$ est l'*inf-convolution* $C_{G_1} \square C_{G_2}$. Il est prouvé que cette opération maintient la convexité (e.g. [Rock70]).

5.2.1.2. Agrégation parallèle

Considérons deux sous-graphes série-parallèles G_1 et G_2, et supposons que leur fonction de coût minimal C_{G_1} et C_{G_2} sont connues. Si l'on considère la SP-relation $G_1 \circledcirc G_2$ (cf. figure 5.8), G_1 et G_2 partagent leur source et leur puits, la seule contrainte de tension entre eux est que leur tension principale, respectivement x_1 et x_2, doivent être égales. Si l'on ajoute la contrainte que la tension principale de $G_1 \circledcirc G_2$ doit être égale à x, cela impose que $x = x_1 = x_2$. La fonction de coût minimal $C_{G_1 \circledcirc G_2}$ de la SP-relation $G_1 \circledcirc G_2$ est donc:

$$C_{G_1 \circledcirc G_2}(x) = C_{G_1}(x) + C_{G_2}(x) \tag{5.7}$$

Cela signifie que $C_{G_1 \circledcirc G_2}$ est simplement la somme $C_{G_1} + C_{G_2}$, qui est convexe si les fonctions C_{G_1} et C_{G_2} sont convexes.

5.2.1.3. Fonction de coût minimal t-centrée

De cette rapide analyse, il est possible d'écrire un algorithme récursif simple pour construire la fonction de coût minimal C_G d'un SP-graphe G. Mais nous sommes plutôt intéressés par déterminer la tension de coût minimal de G. Nous proposons maintenant une représentation particulière de la fonction de coût minimal, de manière à connaître non seulement le coût de la tension optimale d'une SP-relation, mais également comment construire cette tension optimale. Pour cela, nous définissons la ***fonction de coût minimal t-centrée*** C_G^t de G comme suit.

$$C_G^t(x) = C_G(x + t) - C_G(t) \tag{5.8}$$

Autrement dit $C_G^t(0) = 0$, et la fonction C_G^t représente le coût minimal pour augmenter ou diminuer la tension principale à partir de la valeur t. Centrer la fonction de coût minimal sur la valeur courante θ_u d'un arc agrégé u permet d'obtenir des informations de coût relatives à la valeur courante θ_u. Nous choisissons de représenter cette fonction linéaire par morceaux sous la forme de deux ensembles sh_G^t et st_G^t, sh_G^t représentant C_G^t sur l'intervalle $]-\infty; 0[$ et st_G^t représentant C_G^t sur l'intervalle $]0; +\infty[$. Ces ensembles contiennent simplement la définition de chaque morceau de la fonction. Un morceau p est défini par un triplet de la forme $(c; e; l)$, où c représente la pente de la courbe sur le morceau p, e la longueur de l'intervalle sur lequel p est défini, et l

l'ensemble des arcs dont la tension doit être augmentée ou diminuée pour adapter la tension sur le morceau p. Pour des raisons d'efficacité, les morceaux des ensembles sh_G^t et st_G^t sont triés de la plus petite pente à la plus grande. La figure 5.14 illustre un exemple de fonction de coût minimal t-centrée, définie par les ensembles suivants.

$sh_G^t = \{(-2/5; 5; \{a; b\}); (3/5; 10; \{c\}); (3; 5; \{d; e\})\}$
$st_G^t = \{(2/5; 5; \{a; b\}); (2; 7; \{c; e\})\}$

Dans cette figure, considérons la tension principale $\overline{\theta}_G$ de G égale à t. Si l'on souhaite diminuer $\overline{\theta}_G$ d'une unité, il faut diminuer la tension des arcs a et b d'une unité, et le coût global de la tension du graphe sera diminuée de $2/5$ unité.

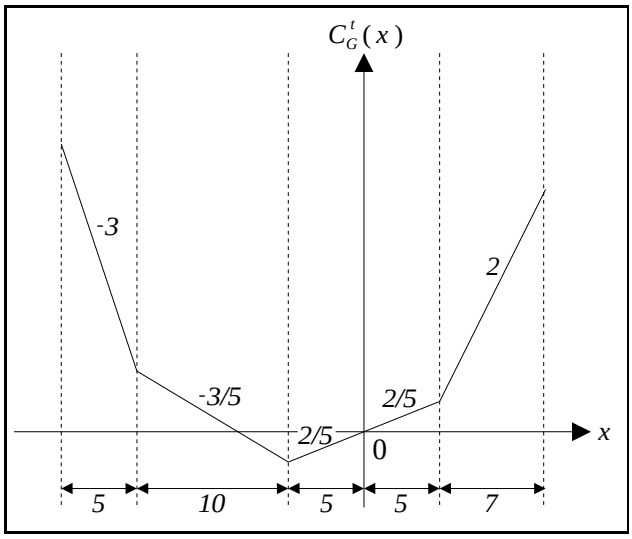


Figure 5.14: Un exemple de fonction de coût minimal t-centrée.

Notons θ_G^* la tension de coût minimal d'un graphe G et $C_G^* = C_G^{\overline{\theta}^*}$. Nous expliquons maintenant comment trouver θ_G^* et construire la fonction C_G^*. Remarquons simplement que la fonction de coût minimal C_u^* pour un arc non agrégé u est représentée par $sh_u^* = \{(c_u^1; o_u - a_u; \{u\})\}$ et $st_u^* = \{(c_u^2; b_u - o_u; \{u\})\}$ avec la tension optimale $\theta_u^* = o_u$. Intéressons-nous maintenant à trouver ces informations pour une SP-relation.

5.2.2. Agrégation série

Considérons le graphe $G = G_1 \oplus G_2$, et supposons pour les sous-graphes G_1 et G_2 que leur tension optimale, respectivement θ_1^* et θ_2^*, et leur fonction de coût minimal, respectivement C_1^* et C_2^*, sont connues. La tension θ_G^* formée des deux tensions θ_1^* et θ_2^* est optimale, puisque la relation série $G_1 \oplus G_2$ n'entraîne aucune contrainte supplémentaire entre G_1 et G_2.

Pour augmenter la tension principale $\overline{\theta}_G^*$ du graphe G, nous pouvons choisir d'augmenter celle du sous-graphe G_1, i.e. $\overline{\theta}_1^*$, ou celle de G_2, i.e. $\overline{\theta}_2^*$. Observons $p_1 = (c_1; e_1; l_1)$ et $p_2 = (c_2; e_2; l_2)$ les premiers morceaux respectivement de st_1^* et de st_2^*. Afin d'effectuer une augmentation optimale, il faut choisir d'augmenter $\overline{\theta}_1^*$ si $c_1 < c_2$, ou $\overline{\theta}_2^*$ sinon. Supposons que $\overline{\theta}_1^*$ ait été augmentée, cette opération ne peut pas dépasser e_1 unités. Ensuite il est nécessaire d'appliquer à nouveau le même raisonnement sur les nouvelles valeurs de $\overline{\theta}_1^*$ et $\overline{\theta}_2^*$. Cette analyse peut être faite également pour la diminution de $\overline{\theta}_G^*$.

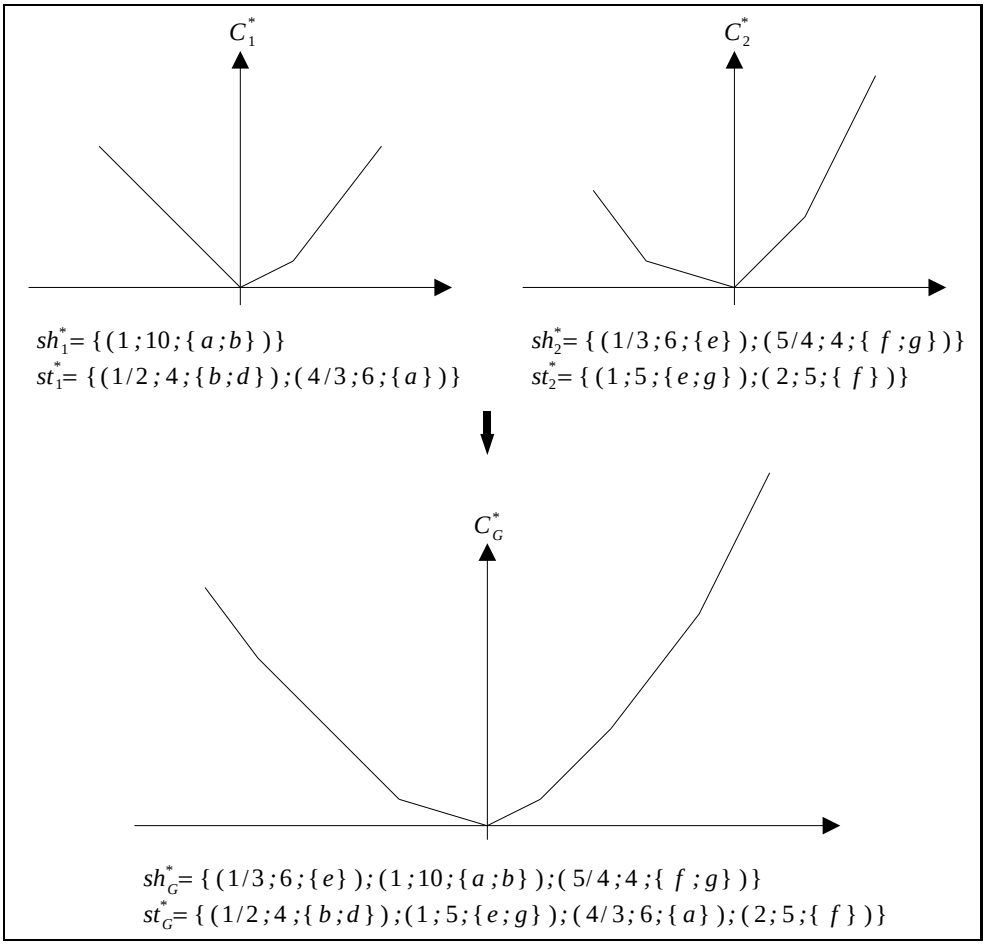


Figure 5.15: Un exemple de construction de la fonction de coût minimal d'une relation série.

En conclusion, la fonction $C_G^* = C_1^* \square C_2^*$ est représentée par les ensembles $sh_G^* = sh_1^* \cup sh_2^*$ et $st_G^* = st_1^* \cup st_2^*$ triés de la plus petite pente à la plus grande. La figure 5.15 montre un exemple de construction de la fonction de coût minimal t-centrée d'une relation série, la procédure étant résumée par l'algorithme 5.6.

Algorithme 5.6: agrégerSérie(fonction C_G^*,fonction C_1^*,fonction C_2^*).

```
tant que sh1* ≠ ∅ et sh2* ≠ ∅ faire
 soit p1 = (c1;e1;l1) premier morceau de sh1*;
 soit p2 = (c2;e2;l2) premier morceau de sh2*;

 si c1 < c2 alors sh1* ← sh1* \ {p1}; shG* ← shG* ∪ {p1};
 sinon sh2* ← sh2* \ {p2}; shG* ← shG* ∪ {p2};
fin tant que;

tant que sh1* ≠ ∅ faire
 soit p1 = (c1;e1;l1) premier morceau de sh1*;
 sh1* ← sh1* \ {p1}; shG* ← shG* ∪ {p1};
fin tant que;

tant que sh2* ≠ ∅ faire
 soit p2 = (c2;e2;l2) premier morceau de sh2*;
 sh2* ← sh2* \ {p2}; shG* ← shG* ∪ {p2};
fin tant que;

/* Procédure similaire pour stG*. */
...
```

Notons p_1 et p_2 les nombres de morceaux de C_1^* et C_2^*, C_G^* possède donc $p = p_1 + p_2$ morceaux, et la procédure de construction de la fonction de coût minimal et de la tension optimale d'une relation série nécessite $O(pm)$ opérations: $O(p)$ opérations pour parcourir les p morceaux des fonctions de coût minimal, et $O(m)$ opérations pour copier un ensemble d'au plus m arcs pour chaque morceau.

5.2.3. Agrégation parallèle

Considérons maintenant le graphe $G = G_1 \mathbin{/\!\!/} G_2$, et supposons pour les sous-graphes G_1 et G_2 que leur tension optimale, respectivement θ_1^* et θ_2^*, et leur fonction de coût minimal, respectivement C_1^* et C_2^* sont connues. La relation parallèle est possible seulement si $\overline{\theta}_1^* = \overline{\theta}_2^*$, la tension du graphe G ne sera valide qu'à cette condition. Comme nous devons trouver la tension optimale θ_G^* du graphe G, une méthode est nécessaire pour égaliser $\overline{\theta}_1^*$ et $\overline{\theta}_2^*$ de manière optimale, i.e. rendre la tension θ_G^* formée de θ_1^* et θ_2^* optimale.

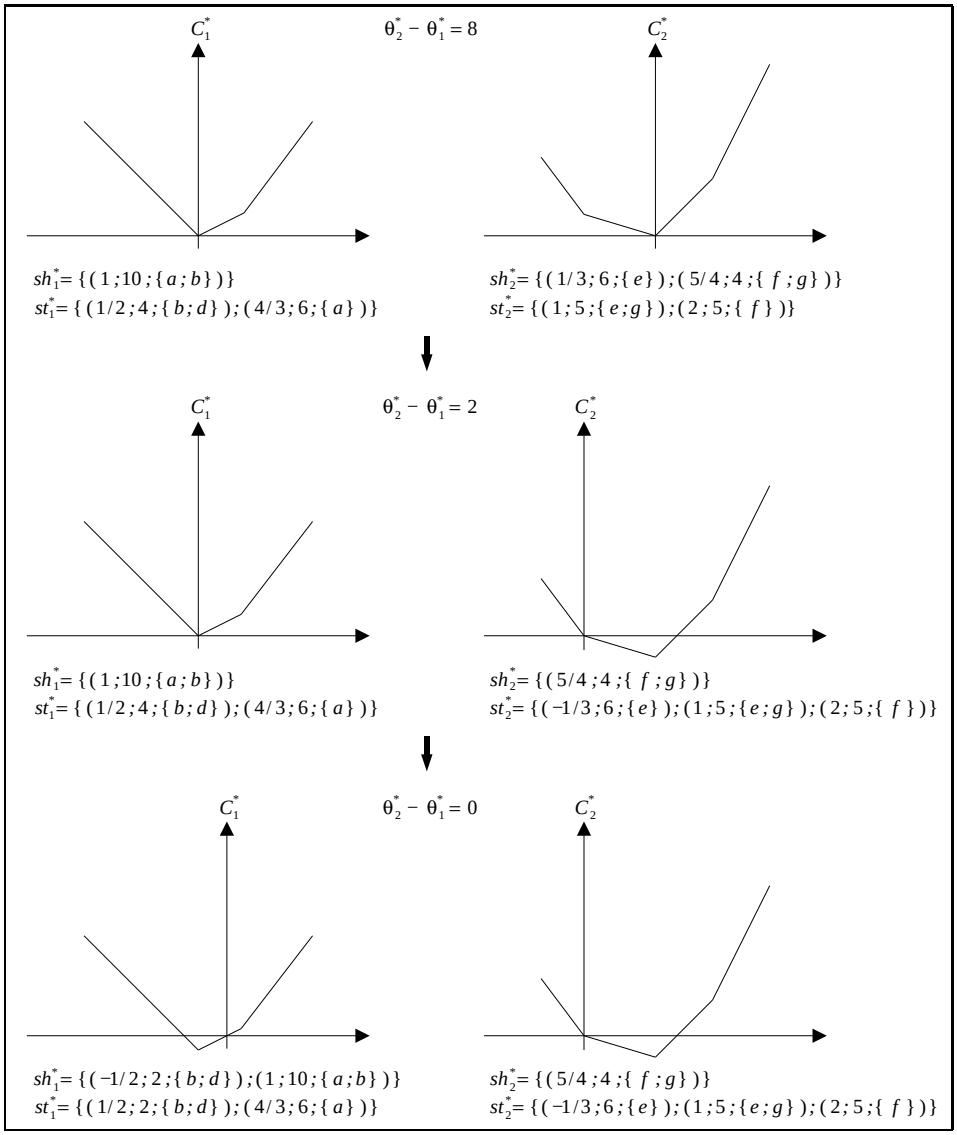


Figure 5.16: Un exemple d'égalisation des tensions lors d'une sérialisation.

Supposons que $\overline{\theta}_1^* < \overline{\theta}_2^*$, pour égaliser $\overline{\theta}_1^*$ et $\overline{\theta}_2^*$, il faut augmenter $\overline{\theta}_1^*$ et/ou diminuer $\overline{\theta}_2^*$. Observons donc $p_1 = (c_1; e_1; l_1)$ et $p_2 = (c_2; e_2; l_2)$, les premiers morceaux respectivement de st_1^* et sh_2^*. Afin d'effectuer un rapprochement des deux tensions de manière optimale, il faut choisir d'augmenter $\overline{\theta}_1^*$ si $c_1 < c_2$, ou de diminuer $\overline{\theta}_2^*$ sinon. Supposons que $\overline{\theta}_2^*$ ait été diminuée, cette opération ne peut dépasser e_2 unités. Ensuite il est nécessaire d'appliquer le même raisonnement sur les nouvelles valeurs de $\overline{\theta}_1^*$ et $\overline{\theta}_2^*$. Cette opération est répétée jusqu'à ce que $\overline{\theta}_1^* = \overline{\theta}_2^*$.

Algorithme 5.7: **égaliserTensionsParallèles(fonction C_1^*,fonction C_2^*,tension θ_1^*,tension θ_2^*).**

```
tant que θ̄₁* < θ̄₂* faire
 si st₁* = ∅ et sh₂* = ∅ alors arrêter; /* Tension non réalisable. */
 soit p₁ = (c₁; e₁; l₁) le premier morceau de st₁* si st₁* ≠ ∅;
 soit p₂ = (c₂; e₂; l₂) le premier morceau de sh₂* si sh₂* ≠ ∅;

 si sh₂* = ∅ ou c₁ < c₂ alors /* Augmentation tension. */
  λ ← min{e₁; θ̄₂* − θ̄₁*};
  pour tout u ∈ l₁ faire θ*_{1_u} ← θ*_{1_u} + λ;
  st₁* ← st₁* \ {p₁};
  si λ < e₁ alors st₁* ← st₁* ∪ {(c₁; e₁ − λ; l₁)};
  sh₁* ← sh₁* ∪ {(−c₁; λ; l₁)};
 sinon /* Diminution tension. */
  λ ← min{e₂; θ̄₂* − θ̄₁*};
  pour tout u ∈ l₂ faire θ*_{2_u} ← θ*_{2_u} − λ;
  sh₂* ← sh₂* \ {p₂};
  si λ < e₂ alors sh₂* ← sh₂* ∪ {(c₂; e₂ − λ; l₂)};
  st₂* ← st₂* ∪ {(−c₂; λ; l₂)};
 fin si;
fin tant que;

/* Procédure similaire pour le cas θ̄₁* > θ̄₂*. */
...
```

La figure 5.16 illustre un exemple d'égalisation de tensions principales entre deux sous-graphes associés par une relation parallèle. L'algorithme 5.7 résume cette procédure. Une fois les deux tensions principales égales, il est alors possible de construire $C_G^* = C_1^* + C_2^*$ centrée sur $\overline{\theta}_G^* = \overline{\theta}_1^* = \overline{\theta}_2^*$, comme le montre l'algorithme 5.8. La figure 5.17 illustre cette procédure par un exemple.

Algorithme 5.8: **agrégerParallèle(fonction C_G^*,fonction C_1^*,fonction C_2^*).**

```
tant que st₁* ≠ ∅ et st₂* ≠ ∅ faire
 soit p₁ = (c₁; e₁; l₁) le premier morceau de st₁*;
 soit p₂ = (c₂; e₂; l₂) le premier morceau de st₂*;
 λ ← min{e₁; e₂};
 st_G* ← st_G* ∪ {(c₁ + c₂; λ; l₁ ∪ l₂)};
 st₁* ← st₁* \ {p₁};
 st₂* ← st₂* \ {p₂};
 si e₁ > λ alors st₁* ← st₁* ∪ {(c₁; e₁ − λ; l₁)};
 si e₂ > λ alors st₂* ← st₂* ∪ {(c₂; e₂ − λ; l₂)};
fin tant que;

/* Procédure similaire pour sh_G*. */
...
```

Notons p_1 et p_2 les nombres de morceaux de C_1^* et C_2^*. Il est possible de vérifier que l'exécution de l'algorithme 5.7 suivi de 5.8 construit C_G^* avec au plus $p = p_1 + p_2$ morceaux (le premier algorithme crée au plus un morceau supplémentaire, et de la figure 5.17, il est évident que le second algorithme crée au plus $p_1 + p_2$ morceaux). La procédure complète de construction de la fonction de coût minimal et de la tension optimale d'une relation parallèle nécessite alors $O(pm)$ opérations: la procédure d'égalisation des tensions parcours au plus p morceaux des fonctions de coût minimal, et copie au plus m arcs pour chaque morceau; de même l'algorithme 5.8 crée au plus p morceaux et copie pour chacun au plus m arcs.

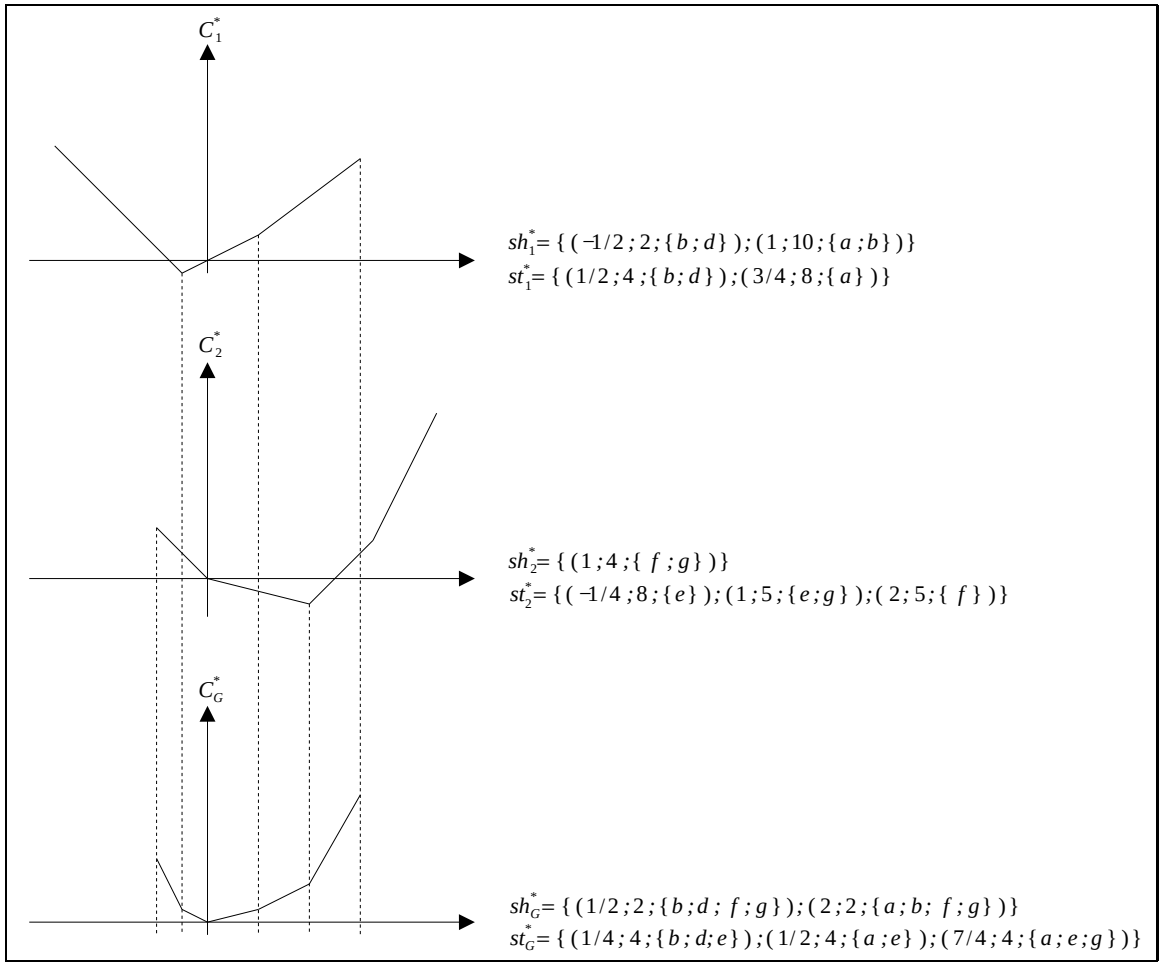


Figure 5.17: Un exemple de construction de la fonction de coût minimal d'une relation parallèle.

5.2.4. Tension optimale

A partir des algorithmes présentés précédemment, il est possible de proposer une méthode permettant de déterminer la tension optimale θ_G^* et la fonction de coût minimal C_G^* d'un SP-graphe G entier. Cet algorithme repose sur le SP-arbre T associé à G: partant de ses feuilles, il suffit d'appliquer récursivement les méthodes d'agrégation sur des SP-relations détaillées aux sections précédentes (cf. algorithme 5.9). Nous montrons maintenant que cette procédure est polynômiale.

La méthode d'agrégation nécessite $O(m^3)$ opérations. (5.9)

Preuve:
Nous avons établi que chaque agrégation effectue $O(pm)$ opérations. En début de chapitre, il a été signalé qu'un SP-graphe possède $m-1$ SP-relations. La méthode d'agrégation nécessite donc $O(pm^2)$ opérations. Nous avons également expliqué que, pour chaque SP-relation, si p_1 et p_2 sont les nombres de morceaux de C_1^* et C_2^*, alors C_G^* possède au plus p morceaux. Autrement dit, si chaque arc non agrégé possède deux morceaux dans sa fonction de coût, alors la fonction de coût minimal du SP-graphe complet ne peut pas excéder $2m$ morceaux. La méthode d'agrégation nécessite donc bien $O(m^3)$ opérations. □

```
Algorithme 5.9: agréger(arbre T = (u; T_l; T_r),tension θ*_G,fonction C*_G).
si T ≠ ∅ alors
 agréger(T_l,θ*_l,C*_l);
 agréger(T_r,θ*_r,C*_r);

 si u = ⊕ alors
  agrégerSérie(C*_G,C*_l,C*_r);
  θ*_G ← θ*_l ∪ θ*_r;
 sinon si u = // alors
  égaliserTensionsParallèles(C*_l,C*_r,θ*_l,θ*_r);
  agrégerParallèle(C*_G,C*_l,C*_r);
  θ*_G ← θ*_l ∪ θ*_r;
 sinon
  sh*_G ← {(c¹_u; o_u − a_u; {u})};
  st*_G ← {(c²_u; b_u − o_u; {u})};
  θ*_G ← o_u;
 fin si;
fin si;
```

5.2.5. Conclusion

Nous proposons ici de comparer l'efficacité pratique des méthodes de résolution du problème de tension de coût minimal présentées au chapitre précédent et de la méthode d'agrégation sur des graphes série-parallèles par rapport à des graphes quelconques. Pour connaître en détails comment ont été dirigés ces essais (méthode de génération des problèmes, compilateur utilisé...), le lecteur peut consulter l'annexe. Nous précisons seulement que l'outil CPLEX 6.0 a été utilisé avec ses paramètres par défaut pour résoudre les programmes linéaires et que les problèmes générés ont des bornes de tension et des coûts entiers. Les temps de calcul sont exprimés en secondes sur une machine RISC-6000 à 160 MHz.

Dimension graphe		Graphes quelconques			Graphes série-parallèles			
Noeuds	Arcs	Prog. linéaire	Conformité	Echelle dual	Prog. linéaire	Conformité	Echelle dual	Agrégation
50	200	0,4	0,13	0,11	0,4	0,07	0,09	0,05
50	400	0,9	0,31	0,21	0,71	0,15	0,2	0,09
100	400	1	0,5	0,3	0,73	0,2	0,31	0,09
100	800	1,9	1,2	0,58	1,4	0,38	0,63	0,18
500	2000	12,7	15,8	3,3	4,4	3	4,5	0,5
500	4000	37,8	49,8	6,6	10,7	5,3	11,3	0,99
1000	4000	56,8	81,2	18,3	11,6	9	17,2	1
1000	8000	219,7	249,2	18,5	30,8	21,5	34,5	2,1

Tableau 5.1: Comparatif des résultats numériques des algorithmes de tension de coût minimal sur des SP-graphes par rapport à des graphes quelconques.

Le tableau 5.1 présente le temps de résolution (avec $A = 1000$) de la programmation linéaire (modèle P_1), de la mise à conformité (algorithme 4.4) et de la mise à l'échelle du dual (algorithme 4.13). Les deux premières méthodes se comportent nettement mieux, même si la mise à conformité profite un peu plus de la simplification du problème. En revanche, la mise à l'échelle du dual est plus décevante. Il semblerait que la structure série-parallèle des graphes complique l'équilibrage des noeuds (cf. section 4.6). Même avec la technique d'*implémentation par vague* (*wave implementation* [Ahuj93]), qui améliore la complexité de la méthode, nous n'avons pas abouti à de meilleures performances.

Comme nous pouvions l'espérer, la méthode d'agrégation fournit des temps de calcul significativement meilleurs que les autres méthodes. En outre, elle semble beaucoup moins sensible à la taille du graphe: la mise à conformité et la mise à l'échelle offrent un rapport entre 300 et 400 entre le temps de calcul des plus petites instances et des plus grandes; l'agrégation propose elle un rapport d'environ 40; seule la programmation linéaire se rapproche de ce comportement avec un rapport de 70. L'efficacité de la méthode d'agrégation pour

des graphes série-parallèles est telle qu'il serait intéressant de l'exploiter pour résoudre le problème de la tension de coût minimal sur des graphes "presque" série-parallèles, plus proches des cas réels de synchronisation hypermédia.

5.3. Graphes presque série-parallèles

Notamment à cause de la relation *overlaps* (cf. figure 5.9), les graphes temporels issus de synchronisations hypermédia ne sont pas série-parallèles, mais restent en pratique très proches de cette structure particulière. Nous introduisons ici la notion de *graphe presque série-parallèle* pour caractériser des graphes dont la structure de base est série-parallèle mais est perturbée par l'ajout de quelques arcs. Cette dernière section propose une méthode pour résoudre le problème de la tension de coût minimal sur ce type de graphe, l'objectif étant d'exploiter la méthode d'agrégation pour fournir une méthode plus efficace que les méthodes de mise à conformité et de mise à l'échelle du dual pour des graphes presque série-parallèles. Cependant les résultats numériques montrent que cette méthode est très inefficace sur des graphes quelconques.

L'idée de la méthode de résolution consiste à décomposer un graphe temporel en sous-graphes série-parallèles, ses *composantes série-parallèles*. L'optimisation de la tension sur ces sous-graphes est effectuée par la méthode d'agrégation. Ainsi, chaque composante est agrégée et représentée par un seul arc avec un coût linéaire par morceaux. Ensuite, dans un ordre précis, les composantes sont rassemblées une à une, pour progressivement reformer le graphe original. Lors de cette phase de reconstruction, la mise à conformité est employée pour conserver une tension optimale. Dans un premier temps, nous précisons la notion de composantes série-parallèles. Nous décrivons ensuite la phase de reconstruction et présentons quelques résultats numériques, en supposant la décomposition déjà effectuée (elle est obtenue lors de la génération des jeux d'essais). Enfin, nous présentons et comparons deux méthodes simples de décomposition d'un graphe quelconque en composantes série-parallèles.

5.3.1. Composantes série-parallèles

Un sous-graphe série-parallèle d'un graphe G est appelé ***composante série-parallèle***, ou ***SP-composante***, de G. Une SP-composante est dite ***maximale*** si elle n'est pas contenue strictement dans une autre. Une partition P des arcs de G définit une ***décomposition série-parallèle***, ou ***SP-décomposition***, du graphe G où chaque ensemble d'arcs de P induit un sous-graphe série-parallèle de G. Une SP-décomposition du graphe G est donc un ensemble de SP-composantes deux à deux disjointes, en termes d'arcs, et dont l'union est égale à G.

Pour les besoins de l'algorithme de reconstruction, nous supposons un ordre partiel sur les composantes d'une SP-décomposition, telle que si la source ou le puits d'une composante S_i appartiennent à la composante S_j, alors $S_j < S_i$. Nous montrons dans la section sur les méthodes de décomposition qu'il existe toujours une décomposition avec un tel ordre pour n'importe quel graphe.

Un ***graphe presque série-parallèle*** $G = (X; U)$ est un graphe formé de l'union d'un SP-graphe $G' = (X; U')$ et d'un ensemble d'arcs U'', i.e. $U = U' \cup U''$, G' étant l'une des plus grandes SP-composantes maximales de G. Le rapport $|U''|/|U|$ est appelé la ***SP-perturbation*** du graphe G. D'après cette définition, tout graphe est presque série-parallèle, avec une SP-perturbation plus ou moins élevée, mais le terme *presque série-parallèle* est naturellement réservé aux graphes avec une SP-perturbation faible, le seuil restant à définir. Nous proposons maintenant une méthode d'optimisation pour des graphes avec une SP-perturbation faible, de l'ordre de

quelques pourcents. Des résultats numériques montrent la SP-perturbation maximale pour laquelle la nouvelle approche est efficace, ce qui sera le seuil au delà duquel les graphes ne seront plus considérés presque série-parallèles par la suite. Notons que la SP-perturbation d'un graphe ne peut être mesurée que si l'une de ses plus grandes SP-composantes maximales est connue, problème qui reste ouvert et que nous aborderons plus tard.

5.3.2. Méthode de reconstruction

Supposons D une SP-décomposition d'un graphe G triée selon l'ordre décrit à la section précédente. La procédure itérative de reconstruction du graphe G à sa tension optimale à partir de sa SP-décomposition est la suivante. La figure 5.18 illustre ce processus.

- Etape 1: Agrégation.
  Supposons G_{k-1} le graphe obtenu à l'itération $k-1$ (au départ, $G_0 = \emptyset$). Pour construire G_k, on considère S_k la $k^{\text{ième}}$ composante de l'ensemble trié D. La tension optimale θ_k^* et la fonction de coût minimal C_k^* de S_k sont obtenues par la méthode d'agrégation (cf. algorithme 5.9). On tente ensuite d'ajouter l'arc agrégé u_k représentant S_k au graphe G_{k-1}. Si l'origine et la destination de cet arc sont des noeuds du graphe G_{k-1}, alors l'arc peut être ajouté directement (cf. étape 3). En revanche, si une extrémité de l'arc n'existe pas, il faut désagréger l'arc dont la composante qu'il représente contient l'extrémité manquante (cf. étape 2). Cet arc existe forcément grâce à l'ordre partiel des composantes dans la SP-décomposition. Dans l'exemple de la figure 5.18, lorsque l'arc agrégé représentant la composante C_2 est ajouté, les noeuds 2 et 3 ne sont pas présents dans G_1, il faut donc désagréger l'arc u_1 qui représente la composante C_1 et qui contient ces deux noeuds. Il faut remarquer que dans une SP-décomposition où l'on a obtenu la plus grande composante possible, celle-ci sera souvent ajoutée la première et désagrégée dès la seconde itération.
- Etape 2: Désagrégation.
  Désagréger un arc u_i consiste à le remplacer par la SP-composante S_i qu'il représente. Cependant, le bon fonctionnement de la procédure de reconstruction repose sur la propriété qu'à chaque itération k, les arcs du graphe G_k, agrégés ou non, sont conformes. L'arc u_i est donc conforme, en se basant sur la fonction de coût C_i^*. A partir de la tension θ_i et du flot φ_i de l'arc agrégé u_i, il faut déterminer la tension et le flot équivalent sur la composante S_i, i.e. la tension principale de S_i doit être θ_i, le flot principal (i.e. entre la source et le puits) de S_i doit être φ_i, et le coût total de S_i doit être le même que celui de u_i. La tension est déterminée à partir de la fonction de coût minimal C_i^*, et un algorithme de recherche de flot compatible (cf. la sous-section suivante pour les détails) permet de déterminer un flot tel que tous les arcs sont conformes.
- Etape 3: Reconstruction.
  L'arc agrégé u_{k+1} peut alors être ajouté au graphe G_k pour former le graphe G_{k+1}. La tension de cet arc est établie à partir du potentiel de ses noeuds, et son flot est fixé à 0. De cette manière, la tension et le flot sur tout le graphe sont valides. En outre, tous les arcs sont conformes excepté celui qui vient d'être ajouté. Il faut donc le rendre conforme en appliquant par exemple la procédure d'amélioration d'un arc vue au chapitre 4. La seule différence réside dans le fait que les coûts traités ont plus de deux morceaux: les arcs désagrégés possèdent bien un coût avec deux morceaux, mais les arcs agrégés ont comme coût leur fonction de coût minimal. Cette différence ne modifie pas la complexité de la procédure (le nombre total de morceaux ne dépassant pas au pire $O(m)$ dans tout le graphe), mais en pratique elle induit, comme le montrent les résultats numériques, un ralentissement d'un facteur constant (e.g. tableau 5.2).

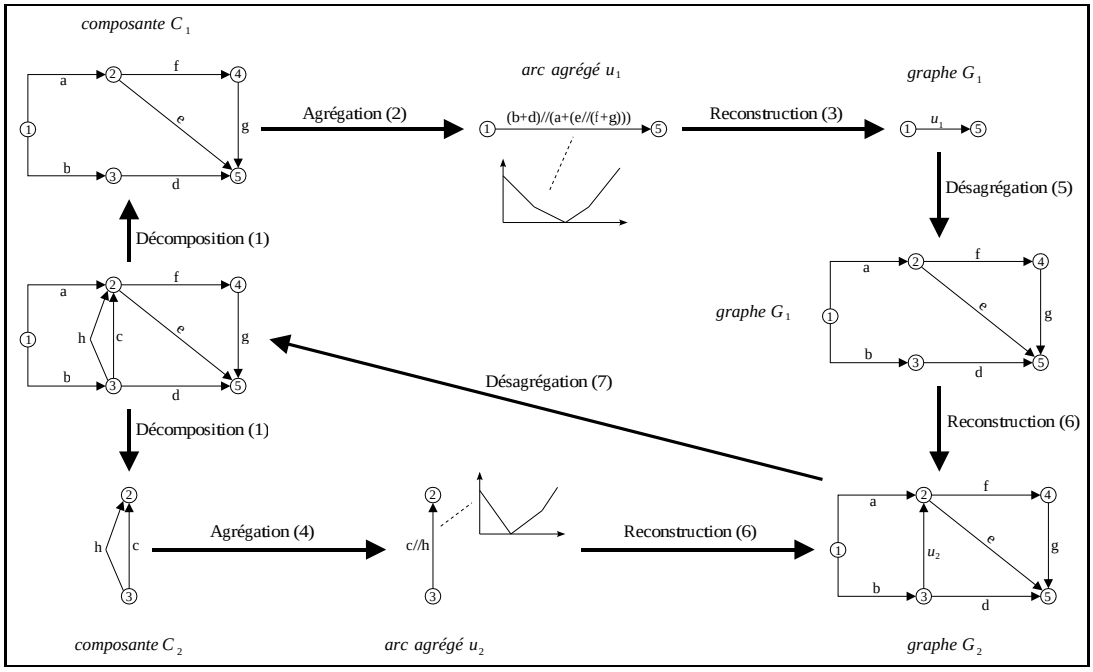


Figure 5.18: Un exemple de reconstruction.

Cette procédure est répétée jusqu'à ce que toutes les composantes soient ajoutées au graphe. A la dernière itération, au moins un arc, le dernier ajouté, est agrégé. Il faut donc terminer la reconstruction par la désagrégation des arcs qui ne l'ont pas été au cours de la procédure (cf. étape 2). Le graphe ainsi obtenu est bien l'original avec tous ses arcs conformes. L'enchaînement des étapes de cette reconstruction est résumé succinctement dans l'algorithme 5.10.

Algorithme 5.10: reconstruire(ensemble D,tension θ).

soit π potentiel associé à θ;
$k \leftarrow 0$;
$G_k \leftarrow (\emptyset;\emptyset)$;

tant que $k < |D|$ faire
 $k \leftarrow k+1$;
 soit S_k la $k^{\text{ième}}$ composante de D;
 soit s et p la source et le puits de S_k;
 soit $G_{k-1} = (X_{k-1};U_{k-1})$;
 ... /* Etape 1: Agréger S_k en l'arc u_k. */

 si $s \notin X_{k-1}$ alors
  soit u_i l'arc agrégé de G_{k-1} contenant s;
  ... /* Etape 2: Désagréger u_i. */
 fin si;

 si $p \notin X_{k-1}$ alors
  soit u_j l'arc agrégé de G_{k-1} contenant p;
  ... /* Etape 2: Désagréger u_j. */
 fin si;

 /* Etape 3: Reconstruction. */
 $G_k \leftarrow (X_{k-1};U_{k-1} \cup \{s;p\})$;
 $\theta_{(s;p)} \leftarrow \pi_p - \pi_s$;
 $\varphi_{(s;p)} \leftarrow 0$;

 tant que u_k non conforme faire améliorerConformité(u_k,G_k,θ,φ);
fin tant que;

5.3.2.1. Algorithme de construction d'un flot conforme

Au cours de la phase de reconstruction, nous avons vu qu'il était nécessaire, pour une composante série-parallèle S_k dont on connaît le flot principal φ_k (i.e. le flot entre sa source et son puits) et la tension optimale θ_k (par rapport à tout le graphe G_k), de déterminer un flot sur tous les arcs de la composante de manière à ce qu'ils soient conformes (cf. étape 2). Il est possible en regardant la courbe de conformité d'un arc u de déterminer, par rapport à la tension θ_u de l'arc, l'intervalle dans lequel son flot φ_u doit se trouver pour que l'arc soit conforme (cf. figure 5.19). De cette manière, à chaque arc u est associé un intervalle de flot $[e_u; f_u]$.

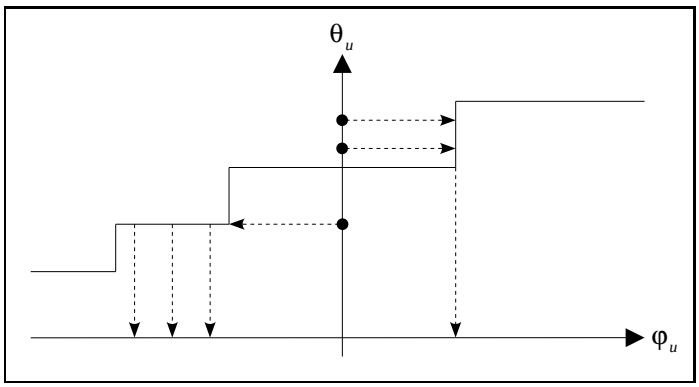


Figure 5.19: Construction d'un flot conforme, connaissant la tension optimale.

En ajoutant un arc v de capacité de flot $[\varphi_k; \varphi_k]$ entre le puits et la source de la composante série-parallèle S_k, trouver un flot qui rend tous les arcs conformes sur la composante S_k revient à déterminer un flot compatible avec les intervalles établis sur la composante S_k complétée de l'arc v. Une condition nécessaire et suffisante pour qu'un tel flot existe est, pour tout cocycle ω, $\sum_{u\in\omega^+} f_u - \sum_{u\in\omega^-} e_u \geq 0$ (cf. [Gond79]). Cette condition est toujours vérifiée, car f_u représente le coût unitaire d'augmentation de la tension de u et e_u le coût unitaire de diminution de la tension de u, un cocycle avec une telle somme négative signifierait qu'une diminution du coût global de la tension est possible.

L'une des meilleures complexités pour résoudre le problème du flot compatible est $O(n^2\sqrt{m})$ (cf. [Ahuj93]), mais nous avons choisi d'utiliser une méthode similaire à l'agrégation, i.e. basée sur une approche récursive exploitant le SP-arbre associé à la composante, car elle est très simple à implémenter et polynômiale. Cette méthode récursive tente de déterminer le flot principal $\overline{\varphi}^*_G$ (i.e. entre la source et le puits) minimal d'un SP-graphe G, en récoltant en même temps des informations permettant d'augmenter le flot $\overline{\varphi}^*_G$ efficacement par la suite. De manière similaire à l'agrégation, ces informations sont récoltées sous la forme d'un ensemble M^*_G constitué de couples $(e; l)$ signifiant que l'on peut augmenter le flot des arcs de l'ensemble l au maximum de e, le flot sur le graphe restant compatible. Ainsi, pour la composante S_k, on obtiendra le flot principal minimal et comment l'augmenter efficacement à la valeur φ_k.

```
Algorithme 5.11: égaliserFlotsSéries(fonction M1*,fonction M2*,flot φ1*,flot φ2*).
tant que φ̄1* < φ̄2* faire
 si M1* = ∅ alors arrêter; /* Flot non réalisable. */
 soit p = (e;l) ∈ M1*;
 λ ← min{e; φ̄2* − φ̄1*};
 pour tout u ∈ l faire φ*1u ← φ*1u + λ;
 M1* ← M1* \ {p};
 si λ < e alors M1* ← M1* ∪ {(e − λ; l)};
fin tant que;

/* Procédure similaire pour le cas φ̄1* > φ̄2*. */
...
```

Considérons le graphe $G = G_1 \oplus G_2$, et supposons pour les sous-graphes G_1 et G_2 que leur flot minimal, respectivement φ_1^* et φ_2^*, et leur ensemble, respectivement M_1^* et M_2^*, sont connus. La relation série est possible seulement si $\overline{\varphi}_1^* = \overline{\varphi}_2^*$, le flot du graphe G n'est valide qu'à cette condition. Il faut une procédure d'égalisation de deux flots en série, le flot principal résultant étant minimal. A partir des ensembles M_1^* et M_2^*, il est possible de proposer une approche polynômiale, similaire à la méthode d'égalisation de tensions parallèles (cf. algorithme 5.11).

```
Algorithme 5.12: minimiserSérie(fonction M*_G,fonction M*_1,fonction M*_2).
tant que M*_1 ≠ ∅ et M*_2 ≠ ∅ faire
 soit p_1 = (e_1;l_1) ∈ M*_1;
 soit p_2 = (e_2;l_2) ∈ M*_2;
 λ ← min{e_1;e_2};
 M*_G ← M*_G ∪ {(λ;l_1 ∪ l_2)};
 M*_1 ← M*_1 \ {p_1};
 M*_2 ← M*_2 \ {p_2};
 si e_1 > λ alors M*_1 ← M*_1 ∪ {(e_1 − λ;l_1)};
 si e_2 > λ alors M*_2 ← M*_2 ∪ {(e_2 − λ;l_2)};
fin tant que;
```

Une fois les deux flots principaux $\overline{\varphi}_1^*$ et $\overline{\varphi}_2^*$ égaux, il faut construire l'ensemble M_G^* de la relation série $G_1 \bigoplus G_2$. L'approche est similaire à la méthode d'agrégation de deux sous-graphes parallèles (cf. algorithme 5.12).

Considérons maintenant le graphe $G = G_1 \parallel G_2$, et supposons pour les sous-graphes G_1 et G_2 que leur flot minimal, respectivement φ_1^* et φ_2^*, et leur ensemble, respectivement M_1^* et M_2^*, sont connus. Le flot principal minimal $\overline{\varphi}_G^*$ est égal à $\overline{\varphi}_1^* + \overline{\varphi}_2^*$, et l'ensemble M_G^* est l'union $M_1^* \cup M_2^*$. La procédure complète pour déterminer le flot minimal d'un graphe G est détaillée dans l'algorithme 5.13.

```
Algorithme 5.13: minimiserFlot(arbre T = (u;T_l;T_r),flot φ*_G,fonction M*_G).
si T ≠ ∅ alors
 minimiserFlot(T_l,φ*_l,M*_l);
 minimiserFlot(T_r,φ*_r,M*_r);

 si u = ⊕ alors
  égaliserFlotsSéries(M*_l,M*_r,φ*_l,φ*_r);
  minimiserSérie(M*_G,M*_l,M*_r);
  φ*_G ← φ*_l ∪ φ*_r;
 sinon si u = ∥ alors
  M*_G ← M*_l ∪ M*_r;
  φ*_G ← φ*_l ∪ φ*_r;
 sinon
  C*_G ← {(b_u − a_u;{u})};
  φ*_G ← a_u;
 fin si;
fin si;
```

Une fois le flot minimal $\overline{\varphi}_G^*$ déterminé et l'ensemble M_G^* construit, il est possible d'augmenter le flot principal de l'arc pour atteindre $\overline{\varphi}_k^*$, en utilisant les couples restants de l'ensemble M_G^*. La méthode complète détermine, pour chaque SP-relation, son flot minimal à partir des deux sous-graphes déjà optimisés, cette opération nécessite $O(pm)$ opérations, p étant la cardinalité de l'ensemble M^*. Comme pour l'agrégation, p ne peut dépasser $O(m)$, pour chaque SP-relation il faut donc $O(m^2)$ opérations. Le graphe contient $m-1$ SP-relations, cette méthode effectue donc $O(m^3)$ opérations.

Cette complexité n'est pas la meilleure que l'on puisse obtenir pour le problème du flot compatible, mais la méthode est simple à implémenter. En outre les résultats numériques montrent que le temps passé dans cette phase de la méthode de reconstruction est négligeable, la méthode employée pour déterminer un flot compatible n'altère pas les performances globales de la méthode de reconstruction (e.g. tableau 5.6).

5.3.2.2. Complexité et résultats numériques

Notons k le nombre de composantes de la SP-décomposition du graphe G utilisée pour la reconstruction. Pour chaque composante S_i dont le nombre d'arcs est p_i, la méthode effectue une opération d'agrégation (soit $O(p_i^3)$ opérations), une opération de désagrégation (soit $O(p_i^2)$ opérations pour construire la tension et $O(p_i^3)$ opérations pour construire le flot) et une opération de reconstruction, i.e. la mise à conformité d'un arc (soit $O(m(A+B))$ opérations). Le nombre total d'opérations est $O(\sum_{S_i \in D}(p_i^3 + m(A+B)))$, or $m = \sum_{S_i \in D} p_i$, donc $m^3 > \sum_{S_i \in D} p_i^3$. La méthode complète, hors décomposition du graphe en SP-composantes, nécessite alors $O(m^3 + km(A+B)))$ opérations. Remarquons que si le graphe est décomposé en m composantes, une par arc, on retrouve la complexité de la mise à conformité.

Nous proposons maintenant une comparaison sur le plan pratique de la méthode de reconstruction par rapport aux deux méthodes les plus efficaces sur des graphes quelconques: la mise à l'échelle du dual et la mise à conformité. Pour connaître en détails comment ont été dirigés ces essais (méthode de génération des problèmes, compilateur utilisé...), le lecteur peut consulter l'annexe. Nous précisons seulement ici que les problèmes générés ont des bornes de tension et des coûts entiers. Le nombre d'itérations pour la méthode de mise à conformité et la reconstruction est le nombre de recherches de cycle et de cocycle effectuées. Les temps de calcul sont exprimés en secondes sur une machine RISC-6000 à 160 MHz.

Les résultats présentés ici ne prennent pas en compte la phase de décomposition, puisque la SP-décomposition des graphes est déterminée au moment de leur génération. Il faut savoir que la décomposition n'est pas optimale dans le sens où elle ne contient pas forcément l'une des plus grandes SP-composantes maximales, mais elle en est très proche. En effet, les graphes sont générés à partir d'un graphe série-parallèle en lui ajoutant aléatoirement les arcs manquants pour obtenir la SP-perturbation souhaitée. Mais ces ajouts peuvent tout à fait rajouter des opérations parallèles dans le graphe, et agrandir la composante série-parallèle de départ.

Dimension graphe		Mise conformité (4.4)		Mise échelle dual (4.13)	Reconstruction (5.10)	
Noeuds	Arcs	Itérations	Temps	Temps	Itérations	Temps
50	200	251	0,08	0,1	10	0,05
50	400	471	0,16	0,2	19	0,1
100	400	484	0,22	0,32	31	0,12
100	800	935	0,47	0,61	48	0,24
500	2000	2125	3	4,6	149	1,2
500	4000	4303	8	10,7	227	3,2
1000	4000	4103	11,8	17	334	4,8
1000	8000	8224	27,7	33,6	454	11,9

Tableau 5.2: Résultats numériques de la méthode de reconstruction sur des graphes presque série-parallèles, influence de la dimension du graphe.

Le tableau 5.2 montre le temps de résolution des algorithmes pour différentes tailles de graphe (avec $A = 1000$) et une SP-perturbation fixe de $1\ \%$. La méthode de reconstruction se comporte mieux que les deux autres méthodes, notamment en comparaison avec la méthode de mise à conformité, puisqu'elle effectue beaucoup moins d'itérations. Cependant, comme elle manipule des coûts linéaires avec plus de deux morceaux, chaque itération de la mise à conformité est ralentie d'un facteur constant.

Le tableau 5.3 s'intéresse au comportement des méthodes en fonction de la SP-perturbation du graphe (avec $n = 500$, $m = 3000$ et $A = 1000$). La méthode de reconstruction reste la plus efficace jusqu'à $6\ \%$ de SP-perturbation, à partir de ce seuil la mise à l'échelle du dual, la plus efficace sur des graphes quelconques, redevient compétitive. Notons la stabilité de la mise à l'échelle qui, même si elle ne donne pas ses meilleures performances ici, est totalement insensible à la SP-perturbation, ce qui veut dire qu'avec $20\ \%$ de perturbation les graphes possèdent toujours une structure particulière qui handicape la méthode. A la vue des résultats

numériques, nous décidons qu'une SP-perturbation de $10\ \%$ est un seuil raisonnable pour déclarer un graphe presque série-parallèle, c'est la valeur au delà de laquelle la mise à conformité simple est plus efficace que la méthode de reconstruction.

SP-perturbation	Mise conformité (4.4)		Mise échelle dual (4.13)	Reconstruction (5.10)	
Pourcentage	Itérations	Temps	Temps	Itérations	Temps
1	3175	5,1	7	189	2
2	3248	6,1	6,8	335	3,1
3	3306	6,7	7,2	464	4
4	3256	6,8	7	585	4,7
5	3375	8,1	6,7	702	5,9
6	3317	7,4	7,2	784	5,8
7	3408	8,6	6,6	923	7,3
8	3363	8,9	6,8	986	7,3
9	3460	9,7	6,9	1144	8,6
10	3468	9,9	6,5	1204	9,1
15	3544	11,8	6,6	1684	12,6
20	3567	12,7	6,6	2053	14,6

Tableau 5.3: Résultats numériques de la méthode de reconstruction sur des graphes presque série-parallèles, influence de la SP-perturbation du graphe.

5.3.3. Méthodes de décomposition

Dans les résultats numériques précédents, nous avons omis volontairement la phase de décomposition d'un graphe en composantes série-parallèles, afin de mettre en évidence le potentiel de la méthode de reconstruction. Les tests ont été effectués sur des graphes dont une bonne SP-décomposition était fournie (de la façon dont elle a été obtenue, il est raisonnable de penser que la décomposition contient une SP-composante très proche d'une des plus grandes composantes maximales), et ils montrent que la méthode de reconstruction est potentiellement efficace pour des graphes avec une SP-perturbation inférieure à $6\ \%$.

Il est donc nécessaire maintenant de proposer une méthode qui, pour un graphe quelconque, propose une SP-décomposition avec l'une des plus grandes composantes maximales. Le problème similaire pour les graphes planaires a été étudié et s'avère être NP-complet (cf. [Liu77]). La plupart des algorithmes à l'heure actuelle tentent de déterminer de bonnes solutions approchées. Les graphes série-parallèles sont planaires, mais il n'est pas sûr que le problème soit difficile pour cette classe de graphes.

A notre connaissance il n'existe pas de travaux pour déterminer une plus grande composante maximale série-parallèle d'un graphe quelconque. Nous ne démontrons pas ici la complexité du problème, ni même ne proposons une méthode exacte. Dans notre cas, il ne faut pas consacrer trop de temps dans l'étape de décomposition, l'efficacité de la phase de reconstruction pourrait alors être perdue. Nous avons donc choisi d'élaborer deux méthodes rapides et simples à implémenter, puisqu'elles reposent sur les deux algorithmes de reconnaissance de graphe série-parallèle vus précédemment (cf. les algorithmes 5.1 et 5.4). Ces méthodes n'offrent naturellement pas la meilleure SP-décomposition d'un graphe, mais elles permettent tout de même à la méthode de reconstruction de rester efficace jusqu'à $2\ \%$ de SP-perturbation.

Les deux approches reposent sur la même idée. Lors de la reconnaissance du graphe, c'est un blocage de la procédure qui permet de détecter que le graphe n'est pas série-parallèle. Dans la première méthode, il n'est plus possible de réduire le graphe. Dans la seconde méthode, il y a deux possibilités: soit les signatures de deux arcs entrants d'un noeud de synchronisation sont différentes, soit il y a un circuit dans le graphe et le parcours est stoppé. Dans ces trois cas, il est possible, en supprimant judicieusement un ou plusieurs arcs de continuer la reconnaissance, le graphe n'est bien entendu pas série-parallèle, mais l'identification de la structure

série-parallèle continue. Les arcs ainsi enlevés représentent chacun une SP-composante. Si l'arc est agrégé, la composante est un sous-graphe, sinon il s'agit d'un simple arc.

Il faut noter également que l'ordre imposé à la SP-décomposition, pour que la procédure de reconstruction fonctionne, est facile à obtenir avec cette approche. Les arcs sont retirés du graphe dans un certain ordre, en inversant cet ordre, la SP-décomposition est triée comme souhaité. En effet, au moment où l'on retire un arc, ses extrémités sont encore dans le graphe, la ou les composantes les contenant seront retirées par la suite, et se trouveront alors avant l'arc dans la SP-décomposition. Penchons-nous maintenant, pour les deux approches de reconnaissance, sur les stratégies intuitives élaborées pour sélectionner le ou les arcs à retirer au moment du blocage de la procédure.

5.3.3.1. Approche par réduction

Pour reconnaître un graphe série-parallèle, cette méthode réduit successivement les relations séries et parallèles simples du graphe. Lorsque la procédure s'arrête, soit il reste un seul arc dans le graphe et à ce moment le graphe est série-parallèle, soit il faut décider d'enlever un arc pour poursuivre les réductions. En supprimant un arc, il est impossible de faire apparaître une relation parallèle, en revanche des relations séries (au maximum 2) peuvent être révélées. Une manière intuitive de procéder consiste, en cas de blocage, à supprimer l'arc qui fait apparaître le plus de relations séries, et s'il n'en existe aucun, il faut alors sélectionner celui qui en favorisera le plus. Pour cela, nous avons choisi d'attribuer à chaque noeud x deux scores, l'un qui sera utilisé lorsque x est l'origine d'un arc (i.e. le score s_x^+), et l'autre lorsqu'il est la destination d'un arc (i.e. le score s_x^-). Le score d'un arc $u = (x; y)$ est la somme $s_x^+ + s_y^-$, les scores des noeuds étant calculés de la manière suivante.

$$s_x^+ = \begin{cases} -M, \text{ si } d_x^+ = 1 \\ 0, \text{ si } d_x^+ > 1 \text{ et } d_x^- = 0 \\ 1/(d_x^+ - 1) + 1/d_x^-, \text{ si } d_x^+ > 1 \text{ et } d_x^- > 0 \end{cases}$$

$$s_x^- = \begin{cases} -M, \text{ si } d_x^- = 1 \\ 0, \text{ si } d_x^- > 1 \text{ et } d_x^+ = 0 \\ 1/(d_x^- - 1) + 1/d_x^+, \text{ si } d_x^- > 1 \text{ et } d_x^+ > 0 \end{cases}$$

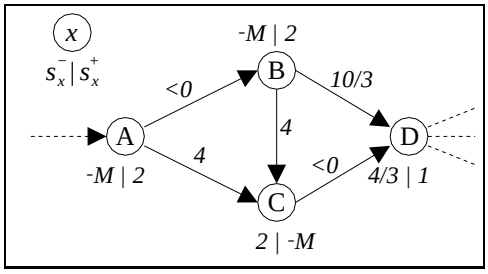


Figure 5.20: Un exemple de score pour une SP-décomposition.

Intuitivement, le noeud x avec $d_x^+ = 2$ et $d_x^- = 1$ est le meilleur candidat pour être l'origine de l'arc à supprimer, il fait apparaître une relation série et totalise un score de 2. De la même manière, le noeud y avec $d_y^- = 2$ et $d_y^+ = 1$ est le meilleur candidat pour être la destination de l'arc à supprimer. Un arc qui fait apparaître deux relations séries totalise donc un score de 4. Ce score décroît ensuite plus les extrémités de l'arc possèdent des degrés entrant et sortant élevés, puisque cela signifie de nombreuses suppressions d'arc avant qu'elles soient une relation série. Les suppressions d'arc altérant la connexité du graphe sont pénalisées d'un score négatif, ainsi un noeud qui risque d'obtenir un degré entrant ou sortant nul se voit attribué un score de $-M$, M doit être choisi de manière à ce que même le meilleur score de la seconde extrémité de l'arc ne puisse pas remonter le score de l'arc au dessus de 0, toute valeur $M > 2$ peut donc convenir. La figure 5.20 montre un exemple de scores. Les arcs $(A; C)$ et $(B; C)$ totalisent le meilleur score, et supprimer l'un ou l'autre permet bien de poursuivre la réduction du graphe par deux relations séries et une relation parallèle.

En cas d'égalité de scores, c'est l'arc qui représente le plus petit sous-graphe série-parallèle qui est éliminé, ainsi un arc agrégé sera toujours préservé face à un arc non agrégé, et l'on espère dégager de cette manière une plus grande composante série-parallèle au final. L'algorithme de décomposition ainsi formé a une complexité

identique à celle de l'algorithme de reconnaissance par réduction sur lequel il repose (i.e. $O(m^2)$ opérations), car la sélection d'un arc à éliminer nécessite $O(m)$ opérations (pour attribuer un score à tous les arcs du graphe).

5.3.3.2. Approche par chemin

Dans cette approche, l'identification d'un graphe non série-parallèle survient de deux manières. La première se produit lors du parcours des noeuds dans un ordre topologique, s'il existe un circuit dans le graphe, il arrive un moment où le parcours ne peut plus continuer, et le circuit passe forcément par l'un des noeuds visités en dernier (i.e. noeuds pour lesquels leurs prédécesseurs et eux mêmes ont été visités, mais pas tous leurs successeurs), notons S l'ensemble de ces noeuds. En effet, plus aucun noeud ne peut être visité car dans l'ensemble S, deux noeuds au moins sont mutuellement prédécesseurs l'un de l'autre, directement ou indirectement. La procédure consiste alors pour chaque noeud de S à vérifier que l'un de ses arcs entrants ne fait pas partie d'un circuit. Une fois un tel circuit trouvé, l'arc entrant impliqué est éliminé du graphe, et la procédure de reconnaissance peut reprendre. L'algorithme 5.14 précise ce processus d'élimination.

```
Algorithme 5.14: briserCircuit(graphe G=(X;U),ensemble S,arc u).
u ← ∅; /* L'arc à éliminer. */

tant que u=∅ et S≠∅ faire
 soit x un noeud de S;
 S ← S\{x};

 pour tout v=(y;x)∈U et tant que u=∅ faire
  si y=x alors u ← v;
  sinon
   /* On utilise l'algorithme de Minty avec la bonne coloration */
   /* (cf.  section 2.1.4) pour rechercher le cycle. */
   cycleMinty(v,G,γ,ω);
   si γ≠∅ alors u ← v;
  fin si;
 fin pour;
fin tant que;
```

La seconde manière d'identifier un graphe non-série parallèle survient lorsqu'à la visite d'un noeud y, il est impossible d'établir une synchronisation, i.e. un arc entrant $u = (x; y)$ possède une signature différente de tous les autres arcs entrants de y.

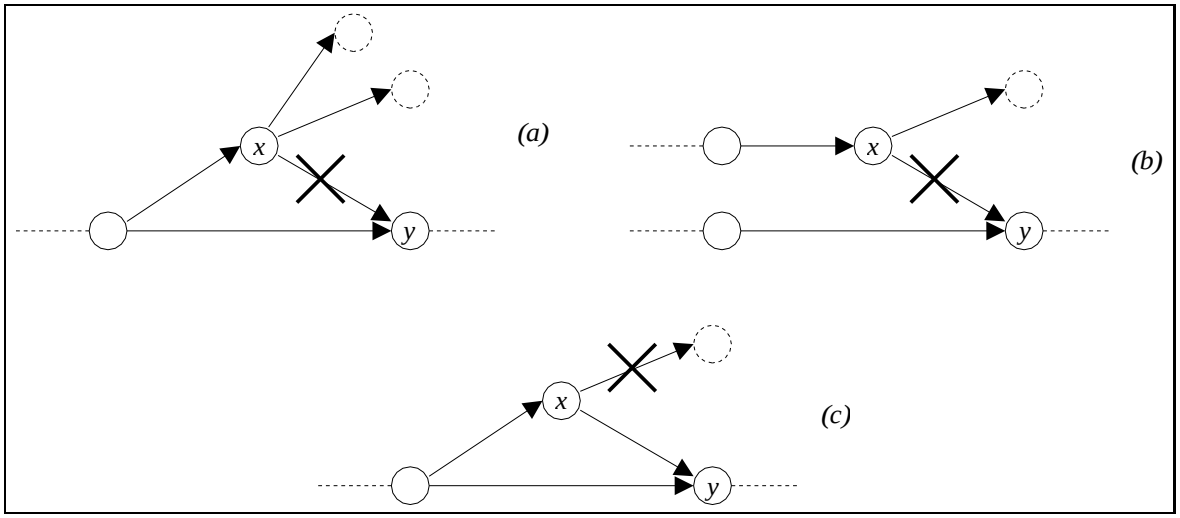


Figure 5.21: Des exemples de situations de suppression d'arc.

Deux possibilités intuitives s'offrent alors, supprimer cet arc (cf. figure 5.21a) ou bien s'il est issu d'un noeud de branchement x de degré sortant 2, vérifier que la suppression de son voisin (i.e. le second arc sortant de x) ne lui permet pas d'obtenir une signature identique à l'un des arcs entrants de y (cf. figures 5.21b et 5.21c). Pour cela, il suffit de vérifier que la signature de l'arc entrant de x, s'il existe, possède une signature identique à celle de l'un des arcs entrants de y. En résumé, quand un arc $u = (x; y)$ bloque la procédure de reconnaissance, soit on le supprime, soit on supprime son voisin au branchement x si cela permet ensuite la synchronisation de l'arc u avec l'un de ses voisins entrants de y. L'algorithme 5.15 détaille cette procédure d'élimination.

```
Algorithme 5.15: supprimerArc(graphe G = (X;U),arc u = (x;y)).
si d_x^+ ≠ 2 ou d_x^- = 0 alors /* Supprimer u. */
sinon
 soit v ∈ ω^-(x);

 si ∃w ∈ ω^-(y) tel que Δ_w = Δ_v alors /* Supprimer u. */
 sinon
  soit v ∈ ω^+(x) tel que v ≠ u;
  /* Supprimer v. */
 fin si;
fin si;
```

L'algorithme de décomposition résultant a une complexité identique à celle de l'algorithme de reconnaissance par chemin sur lequel il repose (i.e. $O(nm \log m)$ opérations), car la sélection d'un arc à éliminer nécessite $O(m)$ opérations (pour vérifier les signatures des arcs entrants du noeud considéré).

5.3.3.3. Résultats numériques

Les résultats numériques présentés ici comparent les deux approches de décomposition en termes de rapidité d'exécution et également de qualité de la décomposition. Pour évaluer le deuxième critère, la SP-décomposition obtenue au moment de la génération aléatoire des instances sera utilisée comme référence. Même si elle n'est pas optimale dans le sens où elle ne contient pas forcément la plus grande SP-composante du graphe, elle en est certainement très proche, et nous permet donc de juger tout de même de la qualité des décompositions proposées par les deux approches.

Le tableau 5.4 montre le temps de calcul de chaque méthode avec le nombre de composantes qu'elle a obtenue pour différentes tailles de graphe (avec une SP-perturbation fixée à 1 %). Le tableau 5.5 met en évidence le comportement des méthodes en fonction de la SP-perturbation du graphe (avec $n = 500$ et $m = 3000$).

Dimension graphe		Décomposition idéale	Par réduction		Par chemin	
Noeuds	Arcs	Composantes	Composantes	Temps	Composantes	Temps
50	200	3	3	0,02	3	0,02
50	400	5	5	0,04	4	0,04
100	400	5	5	0,04	5	0,04
100	800	9	11	0,08	10	0,07
500	2000	21	25	0,32	26	0,23
500	4000	41	51	0,9	53	0,46
1000	4000	41	50	0,97	55	0,5
1000	8000	81	100	3,1	112	1

Tableau 5.4: Comparaison des méthodes de SP-décomposition, influence de la dimension du graphe.

Pour une SP-perturbation de quelques pourcents, les deux méthodes semblent finalement très proches. L'approche par réduction propose néanmoins des décompositions apparemment de meilleure qualité, puisqu'elles possèdent moins de composantes. En revanche, elle est un peu moins rapide que l'approche par chemin pour une SP-perturbation raisonnable, et devient inefficace pour une forte SP-perturbation. Les résultats numériques présentés en conclusion montreront que la vitesse de décomposition obtenue par nos approches n'est pas un

facteur décisif pour choisir entre les deux méthodes, en raison du faible degré de SP-perturbation. Le temps de décomposition est négligeable par rapport au temps de reconstruction. En revanche la qualité de la décomposition est beaucoup plus importante, car elle semble influer directement sur la vitesse de la phase de reconstruction.

SP-perturbation	Décomposition idéale	Par réduction		Par chemin	
Pourcentage	Composantes	Composantes	Temps	Composantes	Temps
1	31	38	0,56	41	0,33
2	61	75	0,81	83	0,34
5	151	190	1,6	207	0,34
10	301	369	2,7	390	0,37
20	601	697	5,2	728	0,39
30	901	1000	7,9	1036	0,45
40	1201	1309	10,8	1341	0,47
50	1501	1602	14	1637	0,5

Tableau 5.5: Comparaison des méthodes de SP-décomposition, influence de la SP-perturbation du graphe.

5.3.4. Conclusion

Pour conclure, nous présentons le comportement en pratique de la méthode de reconstruction complète, avec les deux méthodes de décomposition utilisant l'approche par réduction (A) et l'approche par chemin (B). En outre, les résultats détaillent le temps passé dans les principales étapes de la reconstruction: la décomposition (I), l'agrégation (II), la désagrégation (III) et la mise à conformité (IV).

Dimension graphe		Conformité (4.4)		Echelle dual (4.13)	Reconstruction A (5.10)					
						Temps				
Noeuds	Arcs	Itérations	Temps	Temps	Itérations	I	II	III	IV	V
50	200	251	0,08	0,1	9	0,02	0,02	0,01	0,01	0,07
50	400	471	0,16	0,2	21	0,04	0,05	0,02	0,01	0,15
100	400	484	0,22	0,32	32	0,04	0,04	0,02	0,03	0,16
100	800	935	0,47	0,61	59	0,08	0,1	0,04	0,08	0,36
500	2000	2125	3	4,6	209	0,32	0,26	0,11	1	1,9
500	4000	4303	8	10,7	339	0,9	0,61	0,23	3,3	5,4
1000	4000	4103	11,8	17	494	0,97	0,64	0,24	6,2	8,5
1000	8000	8224	27,7	33,6	739	3,1	1,7	0,48	16,4	22,5

Dimension graphe		Reconstruction B (5.10)					
			Temps				
Noeuds	Arcs	Itérations	I	II	III	IV	V
50	200	10	0,02	0,02	0,01	0,01	0,08
50	400	21	0,04	0,05	0,02	0,01	0,15
100	400	36	0,04	0,05	0,02	0,03	0,17
100	800	60	0,07	0,1	0,04	0,07	0,35
500	2000	235	0,23	0,27	0,11	1,24	2
500	4000	374	0,46	0,68	0,24	3,7	5,5
1000	4000	542	0,5	0,67	0,25	6,4	8,3
1000	8000	856	1	1,9	0,48	18,3	22,5

Tableau 5.6: Résultats numériques des méthodes de reconstruction sur des graphes presque série-parallèles, influence de la dimension du graphe.

Le temps total de la méthode de reconstruction est indiqué dans la colonne (V). Pour connaître en détails comment ont été dirigés ces essais (méthode de génération des problèmes, compilateur utilisé...), le lecteur peut consulter l'annexe. Nous précisons seulement ici que les problèmes générés ont des bornes de tension et des coûts entiers. Le nombre d'itérations présenté correspond au nombre de recherches de cycle et de cocycle lors de la phase de mise à conformité. Les temps de calcul sont exprimés en secondes sur une machine RISC-6000 à 160 MHz.

Le tableau 5.6 montre le comportement des deux versions de la méthode de reconstruction en fonction de la taille du graphe (avec une SP-perturbation de $1\ \%$), et le tableau 5.7 montre l'évolution de leurs performances par rapport à la SP-perturbation (avec $n = 500$ et $m = 3000$). On constate que le temps passé dans les phases d'agrégation et de désagrégation est négligeable. Le temps d'exécution de la phase de décomposition l'est un peu moins mais reste faible par rapport à celui de la phase de mise à conformité. Comme nous l'avions déjà remarqué, la phase de mise à conformité dans la reconstruction effectue peu d'itérations par rapport à la méthode de mise à conformité seule, mais les itérations sont plus longues à cause d'une structure de données plus lourde pour manipuler des coûts linéaires avec plus de deux morceaux. Nous constatons aussi que, moins il y a de SP-composantes, plus la reconstruction est rapide, l'approche par réduction pour la décomposition semble mieux adaptée, mais elle ne rend la reconstruction efficace que jusqu'à $2\ \%$ de SP-perturbation.

SP-perturbation	Conformité (4.4)		Echelle dual (4.13)	Reconstruction A (5.10)					
					Temps				
Pourcentage	Itérations	Temps	Temps	Itérations	I	II	III	IV	V
0,5	3245	5,2	6,8	154	0,44	0,37	0,17	1,2	2,5
1	3175	5,1	7	282	0,56	0,41	0,16	2	3,4
1,5	3204	5,4	7,2	424	0,68	0,47	0,17	2,8	4,4
2	3248	6,1	6,8	540	0,81	0,52	0,16	3,8	5,6
3	3306	6,7	7,2	894	1	0,62	0,16	6,3	8,4
5	3375	8,1	6,7	1321	1,6	0,81	0,16	8,9	11,8
6	3317	7,4	7,2	1692	1,8	0,85	0,18	10	13,2
7	3408	8,6	6,6	1845	2	0,91	0,17	12	15,5
9	3460	9,7	6,9	2414	2,6	1,1	0,16	15,2	19,4
10	3468	9,9	6,5	2748	2,7	1,1	0,17	16,8	21,1

SP-perturbation	Reconstruction B (5.10)					
		Temps				
Pourcentage	Itérations	I	II	III	IV	V
0,5	181	0,33	0,39	0,17	1,3	2,5
1	312	0,33	0,44	0,16	2,1	3,4
1,5	485	0,33	0,49	0,17	3,3	4,5
2	595	0,34	0,54	0,16	4	5,3
3	1121	0,34	0,65	0,16	7,5	9
5	1596	0,34	0,85	0,18	10,7	12,4
6	1830	0,35	0,89	0,17	10,7	12,4
7	2184	0,35	0,99	0,16	14,1	15,9
9	2789	0,37	1,1	0,16	17,8	19,8
10	3105	0,37	1,2	0,16	19,3	21,3

Tableau 5.7: Résultats numériques des méthodes de reconstruction sur des graphes presque série-parallèles, influence de la SP-perturbation du graphe.

Il ressort de ces résultats numériques deux points importants. Le premier est que la qualité de la SP-décomposition est importante pour l'efficacité de la phase de reconstruction. En outre, compte tenu du peu de temps consacré à la décomposition, il semble possible d'investir dans une méthode plus coûteuse mais donnant des SP-décompositions de meilleure qualité. Le second point est qu'il y a un réel décalage entre les performances de la mise à conformité pour des coûts linéaires avec deux morceaux et des coûts avec plus de morceaux. Un gain important de vitesse pourrait profiter à la méthode de reconstruction, en utilisant une structure de données plus performante pour représenter des coûts linéaires avec plus de deux morceaux.

5.4. Conclusion

Dans ce chapitre, une méthode d'agrégation tout à fait efficace, en $O(m^3)$ opérations, a été proposée pour résoudre le problème de la tension de coût minimal dans des graphes série-parallèles. Cette classe d'instances

étant trop idéale, il nous a semblé important de considérer des graphes avec une structure plus réaliste. Pour cela, nous proposons la notion peu formelle de graphe presque série-parallèle qui permet toutefois d'appréhender et de mesurer la complexité de la structure d'un graphe par rapport au problème de la tension de coût minimal.

Une méthode de reconstruction a été proposée pour ces graphes presque série-parallèles. Des résultats numériques, il ressort un potentiel intéressant de la méthode dont nous n'avons pas complètement réussi à profiter. Il faudrait en effet étudier des méthodes de décomposition d'un graphe quelconque en composantes série-parallèles, en expérimentant deux types d'optimalité: minimiser le nombre de composantes ou obtenir la composante maximale la plus grande.

Néanmoins, avec la méthode d'agrégation, la méthode de reconstruction et les méthodes du chapitre précédent, il est possible de proposer différents algorithmes pour le problème de la tension de coût minimal, chacun étant efficace sur différentes structures de graphe: la mise à l'échelle est très efficace sur des graphes a priori quelconques, la mise à conformité prend le relais pour des graphes plus structurés, se rapprochant des graphes presque série-parallèles et enfin l'agrégation s'avèrent très efficace pour des graphes série-parallèles. La combinaison de la mise à conformité et de l'agrégation permet de traiter plus efficacement le problème de la tension de coût minimal sur des graphes presque série-parallèles, que nous avons définis comme étant des graphes avec une SP-perturbation inférieure à $10\ \%$.

PARTIE III - COMPOSANTS RÉUTILISABLES POUR LA RECHERCHE OPÉRATIONNELLE

L'étude présentée dans ce mémoire a donné lieu à une implémentation informatique qui s'est déroulée dans l'esprit de fournir des composants logiciels fortement réutilisables et qui a aboutit à l'élaboration d'une bibliothèque logicielle pour la recherche opérationnelle. Dans cette dernière partie, nous tentons de faire partager cette expérience en espérant guider de futurs développements. Nous présentons donc, outre les enjeux d'un tel travail, nos choix technologiques et conceptuels qui ont été guidés par l'objectif à long terme de proposer une plateforme de développement d'outils de recherche opérationnelle.

AVANT-PROPOS

La recherche opérationnelle est une discipline où divers outils théoriques sont utilisés pour résoudre de nombreux problèmes pratiques. Malheureusement, la manière dont les experts de ce domaine apportent leurs réponses ne facilite pas toujours une exploitation ultérieure de leurs travaux, aussi bien par des néophytes que par des personnes averties.

En effet, la recherche opérationnelle répond généralement à un problème par une méthode de résolution dont la diffusion se fait principalement par une publication dans une revue scientifique où le principal intérêt, à juste titre, est la théorie qui a permis d'élaborer l'algorithme. La méthode est alors présentée sous une forme très abstraite et s'adresse donc à un public averti. Le degré de réutilisabilité d'un algorithme sous cette forme est alors très faible. Cela suppose de la part du réutilisateur une très grande compétence des concepts fondateurs de l'algorithme pour pouvoir implémenter à son tour la méthode. La présentation de l'algorithme est souvent très détaillée sur le plan théorique, mais est trop laxiste sur l'implémentation dont l'expérience de l'auteur ne peut alors pas profiter au réutilisateur.

A l'opposé, pour un utilisateur non expert en recherche opérationnelle, la tendance est généralement de lui développer un logiciel qui implémente la méthode de résolution élaborée avec une interface suffisante. Le programme résout alors un problème précis et satisfait le plus souvent l'utilisateur final qui n'a aucune connaissance du fonctionnement de l'algorithme implémenté, il sait juste qu'en fournissant certaines données il obtient un certain résultat. Le degré de réutilisabilité de la méthode sous cette forme est là aussi très faible: le programme est certainement très simple d'utilisation, mais il est souvent très difficile, voire impossible, de l'adapter pour résoudre un autre type de problème sans connaître précisément à la fois l'aspect théorique et l'implémentation de la méthode.

En écartant la publication qui est indispensable pour la communauté scientifique, une manière idéale de diffuser une méthode de résolution serait un moyen qui conviendrait à la fois aux personnes qui désirent un produit fini et aux personnes qui veulent réutiliser un algorithme sans en connaître les détails et surtout les fondements théoriques difficiles à appréhender, et dont l'acquisition peut être perçue comme une perte de temps et donc d'argent. Les composants logiciels peuvent répondre à ce besoin. Il est possible de fournir un algorithme sous une forme relativement simple, à partir de laquelle un produit fini peut rapidement être développé, et également sous une forme suffisamment souple, pour qu'un utilisateur plus averti, mais non nécessairement expert, puisse réutiliser l'algorithme dans un contexte différent de son utilisation première (e.g. un algorithme de plus court chemin peut être adapté pour rechercher une tension compatible dans un graphe, cf. algorithme 3.7).

Quelques bibliothèques de composants réutilisables existent déjà, nous en parlerons plus en détail par la suite, mais elles n'ont pas la célébrité que peuvent avoir certaines bibliothèques équivalentes en génie logiciel pour la conception d'interfaces (e.g. Borland C++ Builder [BorlWb], Microsoft Visual C++ [MicrWb]) ou en simulation à événements discrets (e.g. VSE [OrcaWb]). Une raison peut être que la souplesse de l'implémentation des méthodes est généralement trop faible, l'utilisateur se trouve rapidement obligé de réécrire complètement les algorithmes. Une autre raison peut être que ces bibliothèques sont souvent développées avec des langages *orientés objet* et que des composants réellement souples pêchent par un excès d'utilisation des concepts objets dans leur conception, ce qui freine toute personne non experte en programmation orientée objet à réutiliser, voire adapter, un composant à ses besoins. Cela peut également conduire à une utilisation excessive des ressources mémoire et processeur.

Au cours de notre étude, nous avons dû implémenter divers algorithmes et structures de données. Depuis le départ, notre politique a été de fournir des composants logiciels réutilisables. Cela nous a conduit progressivement au développement d'une bibliothèque portable (i.e. elle peut aisément être intégrée dans un programme existant et peut donc servir au développement d'un produit fini) et avec, nous l'espérons, des composants suffisamment réutilisables (mais pas trop) qui permettent un développement rapide de prototypes. Notre objectif à très long terme étant de fournir une plateforme de développement complète pour la recherche opérationnelle (plus précisément pour les problèmes d'optimisation dans les graphes) comportant: des composants réutilisables, une représentation visuelle des graphes, une génération et une gestion des instances de problèmes et des campagnes de tests numériques, une documentation des problèmes et des algorithmes...

Le but de la dernière partie de ce document est donc de justifier la nécessité de développer des bibliothèques de composants réutilisables pour la recherche opérationnelle, et d'éclairer les raisons qui font que les bibliothèques existantes sont si peu utilisées, en discutant notamment de certaines idées reçues sur la programmation orientée objet qui freinent très certainement l'utilisation et le développement de composants réutilisables. Dans cette présentation nous nous appuyons sur notre expérience, ne prétendant aucunement que notre bibliothèque soit meilleure qu'une autre. Nous pensons simplement que chacune a ses défauts, ses avantages et que par conséquent, chacune est adaptée à un certain type d'utilisation. Notre discussion portera donc sur les différents choix de conception possibles en fonction des besoins des réutilisateurs.

Dans un premier chapitre, nous discutons de la réutilisabilité logicielle, de manière générale mais aussi par rapport à la recherche opérationnelle, et exposons les différents enjeux qu'elle suscite. Pour les personnes peu familières avec la programmation orientée objet, le second chapitre introduit quelques concepts du paradigme objet et explique leur impact sur la réutilisabilité, ce qui nous permettra de mieux comprendre pourquoi la recherche opérationnelle ne trouve pas un intérêt pour la programmation orientée objet aussi important que d'autres disciplines. Le troisième chapitre sera consacré à des patrons de conception (i.e. *design patterns*) qui formalisent la manière d'implémenter des structures de données et des algorithmes génériques pour la recherche opérationnelle, nous discuterons de leur pertinence en fonction de différents objectifs de réutilisabilité. Enfin, nous terminerons ce chapitre par une brève présentation de notre bibliothèque, nous permettant ainsi d'exposer nos réflexions, notre état d'avancement et notre ambition de développer une plateforme de développement pour la recherche opérationnelle.

CHAPITRE 6
LA RÉUTILISABILITÉ LOGICIELLE

La volonté de réutiliser ce qui a déjà été créé a toujours existée. Dans tous les domaines, les scientifiques profitent du travail de leurs prédécesseurs pour progresser. Bien évidemment, l'informatique n'échappe pas à cette règle, mais du souhait à la réalisation il y a un réel décalage. Dans ce chapitre, nous allons avant tout parler de *réutilisabilité* au sens large et montrer que ce terme est très subjectif et qu'il dépend du réutilisateur. Il est reconnu qu'à tous les niveaux d'abstraction ces concepts existent, mais en recherche opérationnelle, c'est plus particulièrement dans les niveaux les plus concrets, là où l'aspect logiciel intervient, qu'ils sont le moins présents. Nous nous concentrerons alors sur la phase de conception logicielle.

Cette étape est guidée par le ***génie logiciel*** qui est une discipline qui apporte des méthodologies, des outils, des concepts... pour développer des logiciels de qualité. Cependant, il est notable que ce domaine se penche plutôt sur une certaine catégorie de logiciels où la problématique est la quantité et la variété des éléments qui interagissent dans le logiciel (e.g. gestion de systèmes d'information, outils de bureautique, interfaces utilisateur, simulation à événements discrets...).

La recherche opérationnelle, que nous limiterons dans la suite du document à l'étude de problèmes dans les graphes, n'est pas confrontée la plupart du temps à ce type de difficulté. En effet, comme cette discipline tente de répondre le plus souvent de manière théorique à un problème, sa formulation est naturellement relativement simple (même si elle a pu être très compliquée à obtenir, sa forme finale est souvent très synthétique). Ainsi, le génie logiciel qui apporte surtout des réponses quant à la modélisation d'un système, n'éclaire pas beaucoup sur la manière de concevoir et d'implémenter un algorithme (pour nous une méthode de résolution d'un problème dans les graphes) et encore moins sur la manière de les rendre réutilisables et structurés dans une bibliothèque.

L'un des buts de ce chapitre est justement d'expliquer la différence qu'il peut exister entre la conception d'un logiciel au sens classique du génie logiciel et la conception d'un logiciel de recherche opérationnelle, en l'occurrence une bibliothèque de composants logiciels réutilisables. Dans un premier temps, nous discutons des principaux critères de qualité d'un logiciel, des problématiques et des enjeux qu'ils entraînent dans sa conception, et le rôle important tenu par la réutilisabilité. Nous proposons ensuite une rapide description des différentes phases de conception d'un logiciel, qui traduisent en fait différents niveaux d'abstraction du logiciel et où la réutilisabilité est présente par divers moyens. Nous terminons avec un historique de l'évolution des langages de programmation qui a été guidée principalement par un besoin de réutilisabilité, et qui a conduit naturellement à l'approche objet dont nous présentons les principaux concepts et leurs apports à la réutilisabilité au chapitre suivant.

6.1. Qualité logicielle

Dans ce document, nous appelons ***composant logiciel*** tout élément logiciel: fonction / procédure, structure / enregistrement, variable, classe / type abstrait, objet, méthode, attribut, module, bibliothèque, programme... Nous rappelons les principaux critères reconnus pour juger de la qualité d'un composant logiciel dont nous empruntons les définitions à [Meye97] et [Booc87], et nous discutons de leur rôle dans le développement de composants logiciels de recherche opérationnelle.

6.1.1. Fiabilité: validité et robustesse

La ***validité*** est la capacité d'un composant logiciel à effectuer les tâches pour lesquelles il a été défini. En recherche opérationnelle, une partie de ce critère est garanti par les preuves que l'on peut apporter sur le bon fonctionnement d'un algorithme. Il s'agit ensuite de vérifier que l'implémentation correspond aux spécifications de l'algorithme formel. Cette tâche est relativement simple en comparaison avec le génie logiciel ou la simulation où souvent la complexité des modèles (notamment le nombre de composants et la diversité des interactions qu'il existe entre eux) rend toute vérification impossible. On doit alors se reposer sur des méthodes, des outils de vérification, de validation et d'accréditation (e.g. [Balc98c]) qui permettent d'apporter une certaine confiance (appelée ***accréditation***) sur la validité d'un composant logiciel.

La ***robustesse*** est la capacité d'un composant logiciel à réagir de manière appropriée à des conditions anormales. Ce critère est déjà beaucoup plus difficile à atteindre, il suppose de la part de l'auteur une clairvoyance rare pour imaginer toutes les situations anormales possibles, même si un peu de rigueur dans le développement du logiciel permet d'en détecter la plupart. En outre, comment réagir à de telles situations, quelle est la réaction appropriée ? La réponse peut se trouver dans les spécifications du composant logiciel, dans ce cas la situation n'est pas anormale et l'on revient à la notion d'exactitude. En revanche, si les spécifications n'ont pas prévu la situation, alors c'est le bon sens de l'auteur qui permettra de trouver une réaction appropriée. Une façon de fournir toujours une réponse à peu près appropriée consiste à traiter toute situation anormale comme une erreur, une *exception* (cf. chapitre 8) est alors levée et la lourde tâche de gérer la situation anormale est alors déléguée au composant logiciel appelant (celui qui a demandé au composant subissant la situation anormale d'agir).

Les notions de validité et de robustesse étant vraiment très proches, elles sont couramment regroupées sous le terme ***fiabilité*** qui est alors la capacité d'un composant logiciel à réagir selon l'attente du concepteur.

6.1.2. Extensibilité et maintenabilité

L'*extensibilité* d'après [Meye97] (ou la *modifiabilité* d'après [Booc87]) est la capacité d'un composant logiciel à être adapté aux changements de spécifications ou à des corrections d'erreurs. Il est naturel au cours de la vie d'un composant logiciel que ses spécifications changent, tout simplement à cause d'erreurs détectées dans son fonctionnement, mais très souvent aussi en raison de changements des besoins des utilisateurs du composant au cours du temps. Ce critère est d'autant plus difficile à atteindre que le nombre de composants logiciels dans le programme ou la bibliothèque est important. Il faut créer des composants les plus indépendants possible les uns des autres, afin qu'une modification locale à l'un des composants se répercute le moins possible aux autres composants. Cependant, cette indépendance se fait généralement au détriment de la vitesse d'exécution et parfois d'une simplicité de conception. [Booc91] recommande de structurer les composants en modules (cf. section 6.4) où les composants qui participent à une même fonctionnalité globale sont regroupés.

Il faut noter que cette définition de l'extensibilité est très proche de ce que communément on appelle la ***maintenabilité***. Cette notion correspond plutôt au développement d'un programme où l'utilisation des composants se limite au programme, il est alors logique dans cette simple optique que tout changement de spécification implique des modifications dans les composants. Lorsque l'on développe une bibliothèque, l'utilisation des composants n'est pas toujours limitée à la bibliothèque. De nouveaux besoins peuvent en effet entraîner la modification même de composants, mais le plus souvent elle consiste en une ***extension*** (i.e. modification de la fonctionnalité) de composants: les anciens ne sont pas modifiés, mais exploités pour créer des composants

répondant aux nouveaux besoins. Le terme ***extensibilité*** dans le développement d'une bibliothèque correspond donc plutôt (e.g. [Bert95]) à la capacité d'un composant logiciel à pouvoir être étendu sans être modifié.

6.1.3. Homogénéité: intelligibilité et compatibilité

L'***intelligibilité*** est la capacité d'un composant logiciel à être compréhensible. Aussi bien au niveau le plus concret avec un code source le plus clair possible (e.g. nom représentatif et lisible pour une variable, organisation du corps d'une fonction afin de mettre en évidence sa logique algorithmique...), mais également à un niveau plus conceptuel avec une structuration des composants logiciels. Elle nécessite généralement pour un développement à grande échelle d'établir un guide de style, des règles d'écriture et d'organisation des composants, permettant d'homogénéiser la conception des différentes parties d'un logiciel. La documentation notamment participe pleinement à l'intelligibilité d'un composant. C'est la raison pour laquelle à l'heure actuelle elle est considérée comme partie intégrante de la phase d'implémentation (cf. [dSil96]): lors de l'écriture du code source, des commentaires suivant un guide de style particulier peuvent être imposés pour permettre à des outils d'extraire automatiquement une documentation lisible, structurée et navigable des composants logiciels (e.g. Javadoc [SunW2], Doxygen [vHeeWb]).

La notion d'intelligibilité est à rapprocher très fortement de la notion de ***compatibilité*** qui est la capacité de composants logiciels à interagir les uns avec les autres. En effet, toute tentative d'homogénéisation, favorable à l'intelligibilité, va tout naturellement contribuer à développer des composants compatibles. Si l'on cherche la compatibilité avec d'autres logiciels, il faudra alors utiliser des standards: formats de fichier (e.g. XML [W3CW2]), types de donnée, protocoles de communication inter-programme (e.g. Corba [OMGW2])...

En raison des moyens communs employés pour atteindre ces deux critères, nous préférerons nous pencher sur l'***homogénéité*** que nous définissons comme la capacité de composants logiciels à posséder une même logique simple dans leur conception. Nous pensons que si le critère d'homogénéité est atteint par des composants logiciels, alors leur intelligibilité et leur compatibilité sont quasiment garanties.

6.1.4. Portabilité

La ***portabilité*** est la capacité d'un composant logiciel à être transféré d'un environnement à un autre. Le souci premier de la portabilité est avant tout de parvenir à transférer un composant logiciel d'une machine à une autre, d'un système d'exploitation à un autre. Cette problématique s'estompe à mesure que les années passent. En effet, l'enjeu dans l'industrie des logiciels est telle que les compilateurs et les interpréteurs deviennent de plus en plus standards, et les échanges avec le système d'exploitation sont pour la plupart maintenant uniformisés (e.g. la norme POSIX pour la gestion des fichiers, les représentations numériques, les *threads*... [JTC1Wb]; le système graphique X Window [XOrgWb]; le langage Java avec sa machine virtuelle [SunW1]).

Dans le cadre du développement d'un programme, la portabilité s'arrête ici. Par contre, dans la conception d'une bibliothèque de composants, il faut se soucier également de pouvoir l'intégrer dans un logiciel existant, ce qui complique sérieusement la conception: la bibliothèque peut être liée statiquement à un programme (i.e. au moment de sa compilation), dynamiquement (i.e. au moment de l'exécution du programme), utilisée dans un programme avec des threads concurrents (i.e. utilisant au même moment un même composant de la bibliothèque), avoir besoin de communiquer avec l'utilisateur (e.g. indice de progression d'un traitement long ou avertissement d'un problème)... Certains de ces problèmes sont discutés plus en détail dans le chapitre 8.

6.1.5. Efficacité

L'***efficacité*** est la capacité d'un composant logiciel à utiliser le moins de ressources possibles pour effectuer sa tâche. Ce critère est généralement source de compromis. Tout d'abord, entre le taux d'occupation mémoire et le taux d'occupation processeur, puisque souvent une baisse de l'un impose une augmentation de l'autre. Ensuite, les critères de qualité présentés précédemment ne sont pas toujours favorables à l'efficacité: la fiabilité entraîne plus de tests dans les traitements, l'extensibilité une généralisation du composant et donc des opérations supplémentaires, et l'homogénéité et la portabilité des contraintes de conception.

En général, on constate deux types d'attitude vis-à-vis de l'efficacité. Tout d'abord, il y a ceux qui cherchent à tout prix à obtenir un code très optimisé, ce qui nécessite beaucoup de temps et ne favorise en aucun cas la portabilité et l'extensibilité, voire nuit à la fiabilité. D'un autre côté, il y a ceux qui considèrent que l'efficacité n'est pas importante et qui se reposent sur les avancées des matériels informatiques, ce qui peut conduire à des logiciels qui respectent les qualités énoncées précédemment mais qui utilisent ridiculement trop de ressources par rapport à leur fonctionnalité. Il faut noter également que, outre la puissance des ordinateurs, les capacités des techniques de compilation ne font que croître et que certaines optimisations que l'on faisait il y a encore quelques années à la main sont maintenant réalisées automatiquement.

Pour la recherche opérationnelle, l'efficacité est importante et en fonction de l'utilisation à laquelle on destine un composant on favorisera souvent l'efficacité au détriment d'autres critères. Cependant, lorsque l'on cherche à concevoir une bibliothèque de composants, ces critères ont toute leur importance, ne serait-ce que pour fournir des composants fiables, ce qui semble être un prérequis à toute bibliothèque. Mais nous verrons par la suite que l'approche objet permet de bons compromis entre l'efficacité et les autres critères de qualité.

6.1.6. Conclusion

Les critères énoncés précédemment ne sont pas indépendants et très souvent ils s'influencent les uns les autres, en bien ou en mal, et tout l'art de concevoir un logiciel de qualité consiste à trouver un compromis judicieux entre eux. Volontairement la réutilisabilité a été omise de la liste des critères de qualité. Nous n'en n'avons pas encore donné une définition précise, mais le sens commun nous permet déjà de nous rendre compte que tous les critères énoncés précédemment ont un rôle à jouer dans la réutilisabilité. La section suivante se charge d'éclaircir tous ces points.

Nous avons pu voir également que la recherche opérationnelle et en particulier la conception d'une bibliothèque dans ce domaine ne rentre pas dans le cadre habituel du génie logiciel. Certains critères comme la fiabilité et la portabilité sont peut-être plus simples à aborder (quoi qu'il existe un réel problème avec la stabilité numérique qui entraîne un décalage sérieux entre la théorie et la pratique). En revanche d'autres critères comme l'extensibilité et l'efficacité semblent très délicats à combiner.

6.2. Réutilisabilité

Nous proposons ici d'éclaircir les concepts d'utilisation et de réutilisation, et de définir précisément la notion de réutilisabilité. Nous discutons ensuite de son rôle dans le développement de logiciels de qualité et plus précisément la relation qu'elle établit avec les critères de qualité présentés dans la section précédente.

6.2.1. Utilisation ou réutilisation ?

Avant tout, il faut faire la distinction entre l'***utilisation*** d'un composant logiciel qui consiste à le prendre tel quel et à s'en servir directement pour résoudre un problème, e.g. l'utilisation d'un programme, d'une structure de données ou d'une fonction dans une bibliothèque. La ***réutilisation*** est bien une utilisation d'un composant, mais sa fonctionnalité est modifiée, soit en effectuant une modification directe du composant, soit en proposant une extension qui ne modifie pas le composant même. L'utilisation est une finalité, alors que la réutilisation est une préparation d'un composant logiciel à un certain type d'utilisation.

La frontière entre l'utilisation et la réutilisation est mince, cependant beaucoup préconisent une séparation de ces deux concepts (e.g. [Meye97]) en proposant deux autres critères de qualité: la *facilité d'utilisation* et la *réutilisabilité*. La ***facilité d'utilisation*** est la capacité d'un composant logiciel à être utilisé simplement. Par utilisation on entend très souvent l'appel du composant dans un programme, mais également la manière de se le procurer, de l'installer et/ou de le compiler dans l'environnement, la clarté de sa documentation...

La ***réutilisabilité*** est la capacité d'un composant logiciel à être utilisé pour créer d'autres composants. La réutilisabilité est un concept très subjectif et dépend très fortement des personnes, les réutilisateurs, qui vont effectivement réutiliser le composant. En effet, dans notre problématique, des composants réutilisables pour des spécialistes en recherche opérationnelle ne le seront pas forcément pour des personnes moins expérimentées. Les experts souhaiteront des composants fortement extensibles, pouvant être paramétrés à volonté, alors que les plus novices voudront des composants prêts à l'emploi. Nous allons nous efforcer ici de répondre à ces deux types de réutilisateurs.

Mais avant de continuer, nous voulons apporter des précisions sur ce que nous appelons réutilisabilité dans ce document. La réutilisabilité rassemble tous les facteurs qui favorisent une utilisation ou une réutilisation d'un composant logiciel. Ainsi, la facilité d'utilisation qui est souvent mise à l'écart sera pour nous totalement intégrée au concept de réutilisabilité. Nous pensons qu'un composant trop extensible, même s'il est très réutilisable au sens premier, ne sera pas réutilisable au sens second (cf. section suivante).

Dans notre cas, il ne faut pas non plus oublier que les utilisateurs (ou réutilisateurs) auxquels sont destinés les composants ne sont pas experts en génie logiciel et en programmation orientée objet, qui sera le support de notre bibliothèque comme nous le verrons par la suite. Donc des composants paramétrables avec des concepts objets trop avancés ne seront jamais réutilisés par ces personnes.

6.2.2. Réutilisabilité et qualité

Le but de cette section est d'expliquer les relations qu'il existe entre les différents critères de qualité d'un logiciel et plus particulièrement avec la réutilisabilité. Nous tentons de montrer que la réutilisabilité intervient favorablement dans pratiquement tous les critères, et à l'opposé tous interviennent positivement dans la réutilisabilité.

6.2.2.1. Fiabilité

Tout naturellement la fiabilité d'un composant logiciel va favoriser sa réutilisabilité. En effet, si un utilisateur est convaincu du bon comportement d'un composant, il sera tenté de le réutiliser. A l'opposé, la réutilisabilité peut favoriser la fiabilité, puisque construire un nouveau composant à partir de bases fiables et réutilisables ne

peut que faciliter la conception et la vérification du composant. En outre, un composant qui est réutilisé devient fiable au fil du temps car ses défauts ont plus de chance d'être détectés et corrigés par les réutilisateurs.

Cependant, cela forme un cycle dans lequel il est souvent difficile d'entrer. En effet, un composant est réutilisé s'il est fiable et il sera fiable très souvent s'il est réutilisé. On tombe alors sur l'une des difficultés pratiques indépendantes du concepteur d'un composant réutilisable: même si ce dernier est fiable, il faut que des utilisateurs acceptent de prendre un risque à un moment donné. Celui-ci peut se traduire par une réutilisation directe du composant qui peut conduire à de catastrophiques résultats, ou bien alors par un investissement dans des tests sur le composant afin d'évaluer sa fiabilité avant de le réutiliser.

6.2.2.2. Extensibilité

L'extensibilité est un critère indissociable de la réutilisabilité. D'ailleurs dans leur usage quotidien, ces deux concepts sont confondus. Tout composant extensible a forcément un certain degré de réutilisabilité (même si par la suite nous expliquons que trop d'extensibilité peut tuer la réutilisabilité, ce n'est pas l'avis de tous, e.g. [Meye97], [Kuhl96]). A l'opposé, des composants réutilisables et donc relativement adaptables vont faciliter la conception de composants à leur tour extensibles.

Par expérience, nous pouvons affirmer qu'un composant logiciel trop difficile à paramétrer n'est pas toujours beaucoup réutilisé. Il faut fournir des composants paramétrables par rapport aux attentes naturelles des réutilisateurs, et ne pas exagérer le nombre de paramètres, afin d'éviter toute confusion.

Java est source d'exemples de ce type. Pour ouvrir un fichier texte en mode lecture avec la possibilité d'en extraire des mots, des nombres..., il faut créer tout naturellement un composant représentant le fichier, mais cette opération nécessite la création de deux composants intermédiaires. Voici la commande en Java qui permet d'ouvrir le fichier localisé à l'URL (*Unified Resource Locator*) u.

```
f = new StreamTokenizer(new BufferedReader(new InputStreamReader(u)));
```

Cependant, avec un minimum d'effort, un composant `TextFile` pourrait être conçu pour fournir une syntaxe simple: `f = new TextFile(u)`. La modélisation par défaut permet certainement beaucoup de flexibilité dans l'ouverture des fichiers, mais pour des personnes qui veulent ouvrir un simple fichier texte (et c'est très certainement le cas le plus courant) elle fournit trop de détails inutiles.

6.2.2.3. Homogénéité

L'homogénéité favorise indéniablement la réutilisabilité, puisqu'une organisation cohérente des composants facilitera leur réutilisation. En effet, après un apprentissage de la logique qui gouverne la structure des composants, il est très facile pour un utilisateur de retrouver les composants dans une bibliothèque et d'en comprendre les différents paramètres (il pourra profiter des similitudes avec d'autres composants qu'il connaît déjà). Il ne faut donc pas dénigrer cet aspect. Nous pensons même que la plupart des bibliothèques, comme les programmes, qui ont du succès ne sont pas celles qui sont les plus fiables, les plus efficaces... mais bien celles dont la prise en main est facile et intuitive. Dans toutes les disciplines, les gens détestent lire une documentation avant d'utiliser un outil.

A l'opposé, la réutilisabilité ne conduit pas forcément à une homogénéité. Ce sont, dans ce sens, des critères relativement indépendants. Mais la réutilisabilité a parfois tendance à rapprocher des composants d'horizons très différents. Un moyen de garantir une homogénéité dans ce cas est de masquer les composants derrière des composants qui eux sont compatibles avec les règles d'écriture et d'organisation. Cependant l'interfaçage

fourni par ces composants, appelés ***adapteurs*** (cf. [Gamm95]), peut nuire à l'efficacité voire à la réutilisabilité, puisque des extensions très utiles mais incompatibles seront masquées.

6.2.2.4. Portabilité

La portabilité est un facteur qui renforce la réutilisabilité. Un exemple flagrant est Java qui fournit une portabilité de ses composants quasi totale, ce qui explique en grande partie le succès de ce langage. A l'opposé, comme pour la fiabilité, construire un composant à partir de bases portables et réutilisables facilite forcément sa propre portabilité. En revanche, cette propriété nuit généralement beaucoup à l'efficacité, car l'assurer implique d'établir une interface entre l'environnement et les composants. Toute communication entre les deux se fait donc par cette interface, ce qui ralentit forcément l'exécution. Très souvent ce coût est négligeable, mais pour certaines applications critiques, il est très important. L'exemple que l'on connaît le mieux est celui des interfaces graphiques qui perdent significativement de leur efficacité lorsqu'elles sont portables (e.g. Java).

Comme nous l'avons déjà expliqué, ce critère est naturellement amélioré à mesure que les années passent. Avec certains langages, ce n'est même plus un problème puisque leurs spécifications garantissent un standard. Néanmoins pour nous, la portabilité, au même titre que la fiabilité, sont des prérequis à tout composant réutilisable. Nous avons choisi C++ pour développer notre bibliothèque, il s'agit d'un langage dont les spécifications parfois imprécises entraînent de légères ambiguïtés, ce qui rend le problème de la portabilité important (cf. chapitre 8). Néanmoins, dans le domaine de la recherche opérationnelle, la plupart des applications reposent plutôt sur des calculs et très peu d'interactions avec l'environnement sont nécessaires. La seule problématique sérieuse est la représentation des nombres qui peut changer d'une machine (voire d'un compilateur) à une autre.

6.2.2.5. Efficacité

L'efficacité est souvent le critère mis en avant contre la réutilisabilité. Il est souvent supposé que l'utilisation de composants extensibles, autrement dit génériques, par rapport à des composants dédiés est forcément plus lente. Ceci est vrai pour une certaine catégorie de composants réutilisables, en fonction des concepts qu'ils utilisent pour assurer leur extensibilité. Comme nous l'expliquerons au chapitre suivant, la notion d'*héritage* et le *polymorphisme* associé sont des concepts très puissants en termes de réutilisabilité, mais leur abus conduit généralement à un code peu performant. A l'opposé, il existe le concept de généricité qui est un compromis idéal entre l'efficacité et la réutilisabilité. A peu de perte de performance, on peut gagner beaucoup en réutilisabilité. Un exemple est la STL (*Standard Template Library* [SGIWb], [Step95]) qui fournit des structures de données et les algorithmes associés en C++ qui sont très efficaces et très réutilisables.

Néanmoins, on peut raisonnablement supposer que des composants beaucoup réutilisés sont optimisés. Il y a un réel enjeu à améliorer les performances de tels composants, la répercussion de leur évolution s'étend à de nombreux systèmes logiciels. En outre, lorsque l'on conçoit un programme, il est rare de posséder toutes les compétences requises, cela nécessite généralement des connaissances dans des domaines très variés: algorithmique, structure de données, graphisme, bases de données... Il est donc logique d'avoir recours à des composants logiciels développés par des gens compétents dans le domaine concerné, ayant consacré beaucoup de temps à la conception de ces composants. Il est alors très improbable de produire un composant plus efficace, même s'il est dédié à notre application.

Voici un exemple simple où beaucoup, dont moi, se reconnaîtront. Tout informaticien sait implémenter la structure d'une liste chaînée, et de manière très efficace pense-t-il. En effet, il s'agit d'un outil élémentaire codé par chacun des dizaines de fois. Un ajout se fait en quelques lignes: allocation de la cellule, modification des

liens dans la liste. C'est très rapide et efficace. Cependant... A chaque ajout d'un élément dans la liste, une nouvelle cellule est allouée. Cette opération prend beaucoup de temps car le système interroge la mémoire pour savoir où il reste de la place. Une technique célèbre pour améliorer de plusieurs dizaines de pourcents les performances de la liste consiste à allouer une zone mémoire pouvant recueillir un grand nombre de cellules et à gérer soi-même cet espace. Au lieu d'allouer dans la mémoire centrale une cellule, il suffit de piocher dans la zone déjà allouée, et avec une politique efficace, on obtient un gain de performance conséquent. Combien d'informaticiens font cette manipulation chaque fois qu'ils implémentent une liste chaînée ? Peu très certainement. Mais en utilisant par exemple la liste chaînée fournie par la STL, où cet artifice est déjà présent, le développeur gagne à la fois sur son temps de développement et sur l'efficacité de ses composants logiciels.

6.2.3. Conclusion

En conclusion, nous proposons la figure 6.1 qui tente de synthétiser les relations qu'il existe entre les différents critères de qualité. Une case du tableau correspond à l'influence du critère de la ligne sur le critère de la colonne. Une case vide signifie que nous n'avons pas pu décider, soit parce que le critère n'influence pas du tout l'autre, soit parce qu'il est impossible d'en tirer une règle générale.

	Fiabilité	Extensibilité	Homogénéité	Portabilité	Efficacité	Réutilisabilité
Fiabilité				☺	☹	☺
Extensibilité	☺			☺	☹	☺
Homogénéité		☺		☺	☹	☺
Portabilité	☺	☺	☺		☹	☺
Efficacité	☹	☹	☹	☹		☺
Réutilisabilité	☺	☺	☹	☺		

☺ favorise
☹ nuit

Figure 6.1: Relations entre les critères de qualité.

Ce tableau a été construit à partir des remarques que nous avons énoncées tout au long de ce chapitre. Il est difficile de définir catégoriquement l'influence d'un critère sur un autre, le tableau reflète simplement notre point de vue sur la question. Il faut également noter que nous nous sommes placés dans un contexte assez général et que des cas particuliers peuvent facilement infirmer n'importe quelle case du tableau. Nous remarquerons simplement de ce tableau que tous les critères ont une influence positive sur la réutilisabilité et que l'inverse est souvent vrai, ce qui tente à rapprocher la qualité d'un logiciel de sa réutilisabilité.

6.3. Niveaux d'abstraction

La création d'un logiciel passe par trois grandes phases: l'*analyse*, la *conception* et l'*implémentation* (cf. figure 6.2). Le cycle de développement n'est pas séquentiel et il est courant, voire obligatoire, de revenir

sur des étapes antérieures. Ces trois phases représentent différents niveaux d'abstraction. Le premier est très abstrait et consiste en une étude avant même la conception du logiciel. La seconde consiste en l'élaboration de la structure du logiciel, a priori indépendamment de l'environnement informatique sur lequel il sera développé. La dernière phase est la création même du logiciel dans un ou plusieurs langages informatiques. Depuis le début de ce chapitre, nous avons parlé principalement de réutilisabilité au niveau de l'implémentation, mais ce concept existe tout naturellement aux autres niveaux. Nous proposons donc d'en discuter ici, en se posant la question de savoir à quel niveau il est le plus judicieux d'exploiter la réutilisabilité.

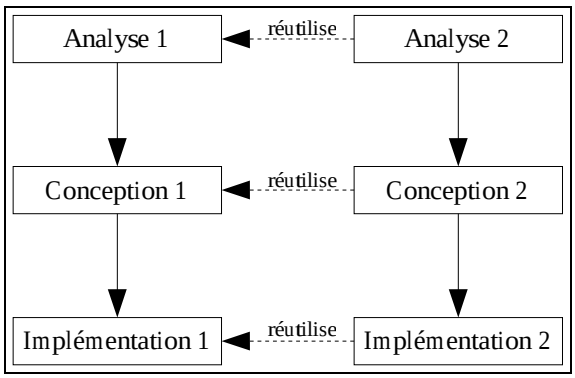


Figure 6.2: Les phases de développement d'un logiciel.

La figure 6.2 explique comment, en dehors de toute méthode favorisant la réutilisabilité, des logiciels d'un même domaine d'application sont élaborés. D'un premier besoin démarre la création du premier logiciel suivant les trois phases (*Analyse 1*, *Conception 1* et *Implémentation 1*). Lorsque le besoin d'un nouveau logiciel apparaît, comme aucune stratégie particulière de développement n'a été mise en place, les trois phases sont à nouveau lancées (*Analyse 2*, *Conception 2* et *Implémentation 2*). Ces dernières réutilisent le peu qu'elles peuvent du premier développement. Mais voyons maintenant comment améliorer la part de réutilisation.

6.3.1. Analyse

L'analyse est l'étape préliminaire à la conception de tout logiciel. Elle consiste principalement à analyser un système et à recenser les besoins des futurs utilisateurs. Le système étudié est celui dans lequel le logiciel doit s'insérer. Par exemple, un logiciel de gestion assistée par ordinateur sera intégré dans les systèmes de production et comptable d'une entreprise; pour une base de données ou un modèle de simulation, le système analysé est celui qu'ils devront représenter. Cette étude est appelée l'***analyse de système***.

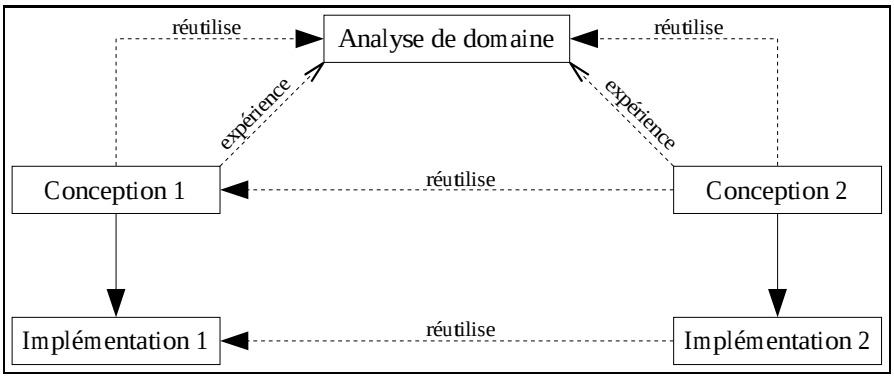


Figure 6.3: Les phases de développement d'un logiciel, avec analyse de domaine.

Une approche plus abstraite peut être menée, elle consiste à étudier plusieurs systèmes d'un même domaine, dans l'espoir de fournir une analyse commune et de conduire à un environnement logiciel unique. Cette étude est appelée ***analyse de domaine***. Cependant, il est très difficile de concevoir un seul environnement informatique qui réponde aux besoins de tous les systèmes. En revanche, il est possible d'identifier des éléments communs qui permettront de mener des analyses communes (cf. [Leac97], [Cohe98]). Ces résultats pourront alors être réutilisées lors de l'analyse d'un système précis du domaine (cf. figure 6.3). Comme le précise [Camp00], l'analyse de domaine peut s'effectuer progressivement en concevant des logiciels pour plusieurs systèmes d'un même domaine. Un retour d'expérience permet alors d'étoffer l'analyse du domaine.

6.3.2. Conception

La ***conception*** est l'étape où la structure du logiciel est élaborée. A ce niveau l'étude est menée indépendamment du ou des langages de programmation qui seront employés pour le logiciel. Cependant, il faut tout de même décider d'une approche de modélisation, e.g. faut-il employer le paradigme objet, un modèle entité-association... A ce niveau tout le logiciel est élaboré, dans un ou plusieurs formalismes.

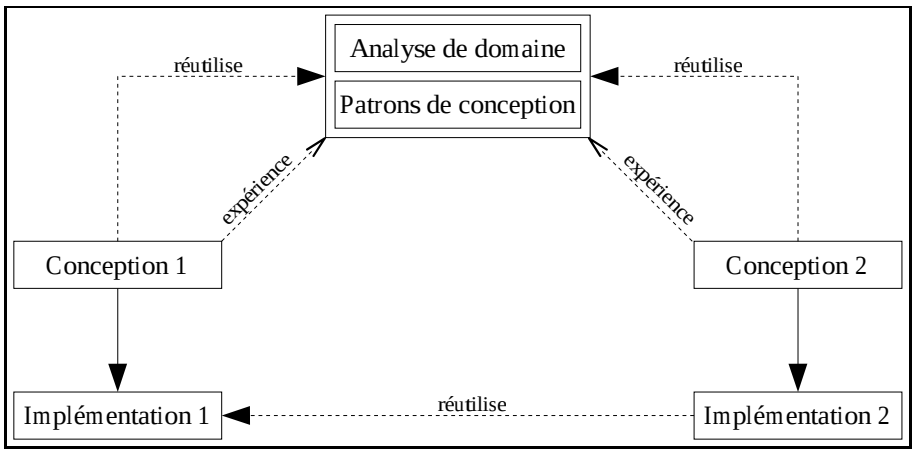


Figure 6.4: Les phases de développement d'un logiciel, avec patrons de conception.

Lors de ces phases de conception, certains problèmes de modélisation sont récurrents. Des solutions génériques peuvent alors être apportées. Afin d'en permettre une réutilisation, la notion de ***patron de conception*** (***design pattern***) a été introduite (cf. [Schm96]). Un patron de conception est un élément réutilisable de conception qui est indépendant du domaine d'application (et bien entendu de tout langage de programmation). Il propose une solution générique à un type précis de problème. Il s'agit donc d'un outil qui permet de mémoriser et de formaliser l'expérience d'experts. Bien qu'il soit indépendant du langage de programmation, un patron de conception n'en est pas moins dépendant de l'approche choisie pour la modélisation. Par exemple, les patrons désormais célèbres fournis dans [Gamm95] proposent des solutions de conception pour l'approche orientée objet. Comme dans la phase d'analyse, un retour d'expérience des différentes conceptions permet d'alimenter les collections de patrons (cf. figure 6.4).

6.3.3. Implémentation

La phase d'***implémentation*** est l'étape qui consiste à programmer proprement dit le logiciel. Notre discussion sur la réutilisabilité a porté principalement sur ce niveau d'abstraction jusqu'à cette dernière section. La fig-

ure 6.5 résume l'intérêt d'une bibliothèque de composants dans la phase d'implémentation du logiciel. Comme pour les deux phases précédentes, le retour d'expérience de chaque implémentation permet de fournir la bibliothèque logicielle en nouveaux composants.

Au travers des derniers paragraphes on pourrait penser que plus la réutilisabilité est abstraite, plus sa portée est grande puisqu'elle est indépendante de l'approche et du langage de programmation. Par conséquent, la réutilisabilité logicielle (i.e. au niveau implémentation) n'aurait que peu d'intérêt.

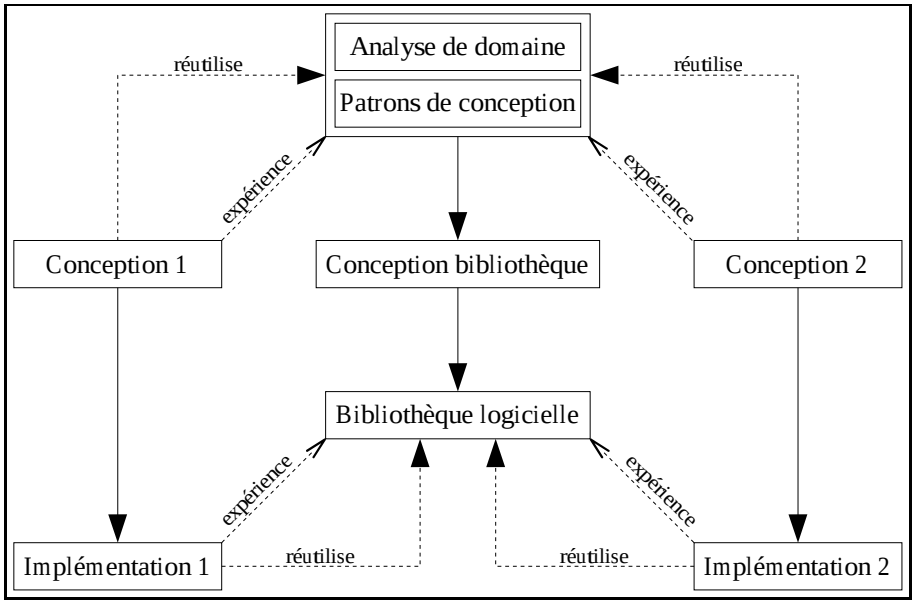


Figure 6.5: Les phases de développement d'un logiciel, avec bibliothèque logicielle.

Cependant, comme l'explique [Stro96], la réutilisabilité ne peut pas être totalement indépendante du langage de programmation, il y a trop de différences dans les concepts de chacun pour concevoir un bon composant réutilisable à un niveau trop abstrait. A vouloir exprimer quelque chose avec les seules notions communes à plusieurs langages, on en perd une puissance d'expression. L'idéal, c'est de combiner la réutilisabilité aux différentes phases d'élaboration du logiciel, c'est ce que l'on appelle un *cadriciel*.

6.3.4. Cadriciel

Plusieurs définitions du terme ***cadriciel*** (***framework***) peuvent être énoncées (e.g. [John88], [Matt96]). Nous retenons celle de [Camp00] qui s'intègre le mieux à notre discussion: un cadriciel est une collection d'éléments de conception (e.g. les patrons de conception) et d'implémentation (e.g. les composants logiciels) en coopération et réutilisables qui permettent de créer des applications ou des parties d'applications dans un domaine spécifique.

L'idée est de fournir, à partir d'une analyse de domaine, des éléments réutilisables, aussi bien au niveau de la conception que de l'implémentation (cf. figure 6.6). Associés à une assistance informatisée, cela permet à un expert d'un domaine de concevoir un logiciel pour ses besoins sans connaître tous les concepts liés au génie logiciel et en particulier à l'approche orientée objet si elle a été retenue pour la conception. La liste des cadriciels célèbres à l'heure actuelle est longue. Nous n'en citerons que deux types représentatifs: les interfaces graphiques, qui proposent au niveau conceptuel les fenêtres, les boutons... et qui offrent aussi les composants

logiciels pour les implémenter; les gestionnaires de bases de données (e.g. Microsoft Access), ils permettent de manipuler à un niveau d'abstraction élevé des bases de données, et fournissent les composants logiciels associés.

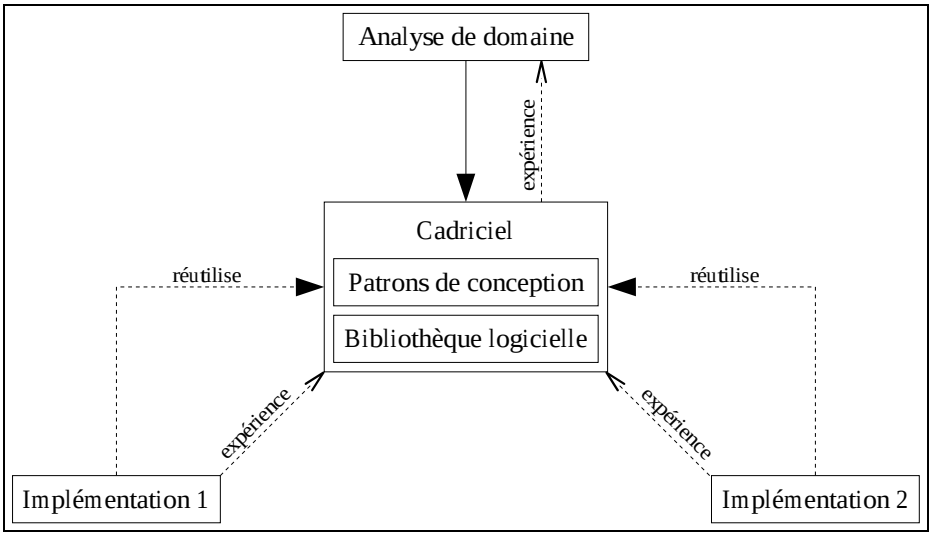


Figure 6.6: Les phases de développement d'un logiciel, avec cadriciel.

6.3.5. Conclusion

Cette discussion a porté sur l'élaboration de logiciels au sens large. Quand est-il de la recherche opérationnelle et de notre objectif de créer une bibliothèque de composants réutilisables ? En recherche opérationnelle, le processus de développement d'un logiciel est très similaire à celui d'un logiciel au sens classique du génie logiciel. La différence réside dans l'analyse de domaine où les méthodes employées sont radicalement différentes. Pour la recherche opérationnelle, la modélisation des systèmes même est très synthétique, la majeure partie de l'analyse consiste alors à élaborer des méthodes de résolution pour la problématique étudiée. En génie logiciel de manière générale, ou en simulation, la plus importante partie du travail consiste à modéliser les systèmes pour lesquels on aboutit souvent à de larges modèles, comparés à ceux que l'on rencontre en recherche opérationnelle. La résolution des problèmes modélisés est alors plus secondaire et peut concerner d'autres disciplines.

La phase de conception est en revanche très similaire, il s'agit de traduire en termes plus informatiques les résultats de l'analyse de domaine pour établir une structure encore abstraite du logiciel. Cela signifie concevoir logiciellement les structures de données et les algorithmes nécessaires aux méthodes de résolution pour la recherche opérationnelle, et modéliser avec des concepts informatiques les modèles des systèmes pour le génie logiciel ou la simulation. En résumé, la recherche opérationnelle travaille plutôt sur les algorithmes alors que le génie logiciel et la simulation se concentrent sur les données et leurs interactions, même si, en fonction des domaines d'application, les deux approches sont plus ou moins mélangées.

Il est intéressant de noter qu'au niveau de l'analyse de domaine, la recherche opérationnelle est très performante en termes de réutilisabilité. Il existe énormément d'articles qui proposent des méthodes pour résoudre un grand nombre de problèmes. En revanche, en génie logiciel ou en simulation, les articles et les études apportent plutôt des réponses générales sur la manière de modéliser un système, mais semble-t-il un peu moins sur une manière de modéliser les systèmes d'un domaine particulier. A l'inverse au niveau de la conception et

de l'implémentation, il est beaucoup plus rare de trouver soit des patrons de conception, soit des composants logiciels pour des problèmes de recherche opérationnelle, alors qu'ils prolifèrent pour le génie logiciel ou même la simulation.

Notre objectif dans ce document sera de présenter quelques patrons de conception et de discuter des problèmes liés à l'implémentation. En aucun cas nous présentons un cadriciel complet pour les problèmes d'optimisation dans les graphes. Néanmoins notre objectif à plus long terme est justement de le fournir et de concevoir l'assistance informatique associée pour disposer d'un environnement de développement d'outils de recherche opérationnelle relativement complet, et suffisamment simple d'utilisation par rapport à tous les concepts de génie logiciel et en particulier l'approche orientée objet (présentée au chapitre suivant).

6.4. L'évolution vers les objets

Avec des ordinateurs de plus en plus puissants, l'homme cherche toujours à concevoir des logiciels, modéliser, simuler et résoudre des systèmes de plus en plus complexes. Cependant, sa capacité à appréhender cette complexité est très limitée. Pour tenter de la maîtriser, l'homme a progressivement développé des concepts. Nous allons rappeler ici leur évolution, qui a conduit à la notion d'objet (le fil conducteur de cette présentation est extrait de [Hill96] et [Satz96]).

6.4.1. Sous-programmes

Les premiers langages assembleurs ne fournissaient aucun concept particulier pour concevoir un programme structuré, qui se résumait alors en une simple séquence d'instructions. Même avec relativement peu de lignes, de tels programmes sont très difficiles à maintenir. En outre, une même séquence d'instructions peut être répliquée plusieurs fois dans un même programme. C'est donc tout naturellement que les premiers langages structurés (e.g. COBOL, Fortran) sont apparus, proposant la notion de ***sous-programme***. Cette entité regroupe une séquence d'instructions pour former une opération plus sophistiquée. Un sous-programme peut être paramétré, notamment sur les données qu'il traite, afin de remplacer des séquences d'instructions similaires.

Un sous-programme est finalement écrit une seule fois dans un programme, même s'il est utilisé plusieurs fois. Il permet également à un programmeur de réutiliser simplement des parties d'un programme d'un autre développeur. Ce sont les premiers pas vers la réutilisabilité. Un autre apport des sous-programmes est de cacher de l'information. En effet, il n'est pas nécessaire de connaître tous les détails du fonctionnement interne d'un sous-programme pour l'utiliser. Cet aspect est important, puisqu'à un certain niveau du développement d'un logiciel, il est possible de le considérer comme un assemblage de sous-programmes et de se concentrer uniquement sur leur organisation, sans être troublé par des détails inutiles à ce moment de l'étude.

6.4.2. Modules

Cependant, le masquage des informations n'est que partiel. Les sous-programmes ne sont pas des entités totalement indépendantes, certains nécessitent de faire appel à d'autres pour réaliser leur propre fonctionnalité. En revanche, tous n'ont pas besoin d'avoir accès à tous les sous-programmes. De la même manière, plusieurs sous-programmes peuvent partager des données, mais il serait souhaitable que n'importe quel sous-programme

ne puisse pas y accéder. Afin de permettre la définition d'une zone où des données et des sous-programmes ne sont visibles qu'entre eux, le concept de ***module*** a été introduit. La logique veut qu'un module regroupe des données et des sous-programmes qui participent à une même fonctionnalité. Mais n'oublions pas que certains composants d'un module doivent être visibles par d'autres, tout simplement pour qu'il puisse faire profiter de ses fonctionnalités. Les composants d'un module sont donc séparés en deux ensembles: l'un est visible par tout le programme (l'***interface***), l'autre est complètement caché (l'***implémentation***).

Ce concept de module permet d'accroître l'abstraction que l'on peut faire d'un logiciel: il peut être vu comme un ensemble de modules en interaction. Cela permet notamment une étude partielle séparée de chaque module. En outre, les données se trouvant dans l'implémentation d'un module sont isolées et modifiées seulement par un nombre restreint de sous-programmes, ce qui permet d'en garantir une certaine fiabilité.

6.4.3. Types abstraits de donnée

La notion de module est intéressante, mais les données partagées par les sous-programmes d'un module sont uniques dans le programme. Imaginons un exemple simple qui consiste à concevoir un module pour gérer les éléments d'une pile. Supposons que la structure de données de la pile (par exemple un tableau) se trouve dans le module, et soit par conséquent unique dans le programme (cf. figure 6.7). Les sous-programmes du module sont dans l'interface et permettent de l'extérieur d'ajouter, de retirer, accéder... aux éléments de la pile. Mais comment obtenir différentes piles dans un même programme ? Il faudrait dupliquer autant de fois que nécessaire les données du module. Pour répondre à ce besoin, la notion de ***type abstrait de donnée*** a été introduite (e.g. le langage Ada).

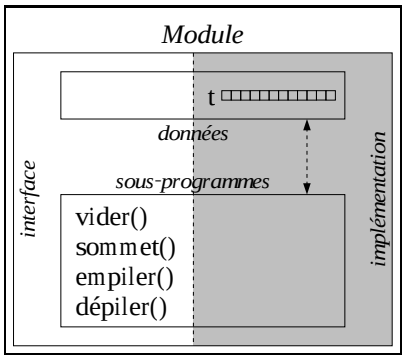


Figure 6.7: Un exemple de module.

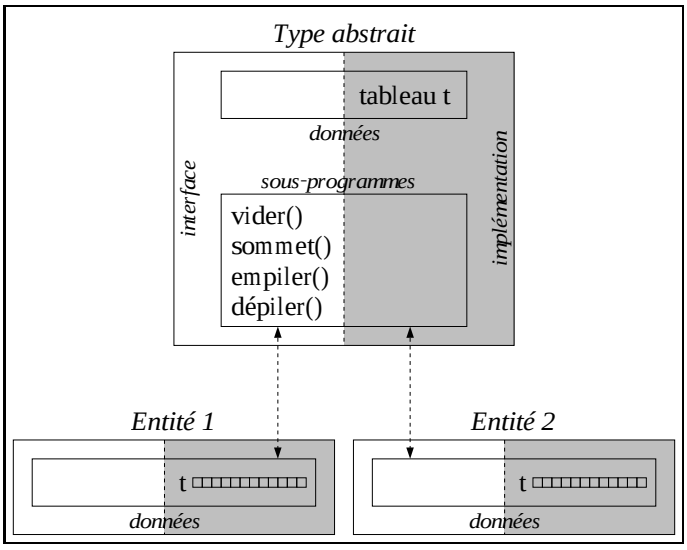


Figure 6.8: Un exemple de type abstrait de donnée.

Un type abstrait de donnée est un modèle qui permet de créer à volonté des entités semblables aux modules. Ces entités ont une interface et une implémentation, les données et les sous-programmes qu'elles abritent ont la même visibilité que dans un module, et donc seuls les sous-programmes d'une entité peuvent modifier ses données cachées. La seule différence, comme la figure 6.8 le montre, est que les sous-programmes font partie du type abstrait de donnée, ils sont donc uniques dans le programme alors que les données sont localisées dans chaque entité.

La figure 6.8 reprend notre exemple de pile: un type abstrait de donnée décrit la pile qui peut ensuite être physiquement créée en autant d'exemplaires que l'on souhaite (sur la figure *Entité 1* et *Entité 2*). En résumé, un type abstrait de donnée est un type qui représente une structure de données pour laquelle un certain nombre d'opérations (les sous-programmes) sont définies (et ce sont les seules autorisées à manipuler la structure de données). Après une première évolution de langage qui a permis de réutiliser du code (i.e. les sous-programmes) et de le cloisonner (i.e. le module), le type abstrait de donnée propose une réutilisation des structures de données.

6.4.4. Objets

La notion d'*objet* a été introduite dans le milieu des années 60 avec le langage SIMULA, dédié à la simulation à événements discrets. Il apporte un ensemble de nouveaux concepts que l'on regroupe de nos jours sous le terme d'***approche orientée objet***, ou de ***paradigme objet***. Sa principale contribution est le renforcement du concept de type abstrait de donnée par la notion de ***classe***. Les entités créées sur le modèle d'une classe sont appelées des ***objets***. En plus d'unifier des opérations et des données dans une même entité, les classes peuvent être organisées en hiérarchie par un mécanisme appelé *héritage*. Ce concept est très important pour la réutilisabilité, puisqu'il permet à une classe héritant d'une autre de s'approprier toutes les données et les opérations de cette dernière, afin de les adapter à ses propres besoins. La réutilisabilité est également renforcée par le concept de *polymorphisme* associé à l'héritage. Tous ces termes sont bien entendu introduits en détail au chapitre suivant.

D'autres langages suivirent... Dans les années 70, Smalltalk introduit le terme d'objet dans le contexte du génie logiciel et formalise leur interaction avec la notion de *message*, qui est une requête d'un objet à un autre lui demandant d'exécuter l'une de ses méthodes. Les années 80 ont vu l'évolution de langages existants vers l'approche orientée objet, e.g. C vers C++, les versions orientées objet de Pascal... A la fin de cette décennie, et au tout début des années 90, les premiers cadriciels pour les interfaces graphiques apparaissent, ce qui donne naissance très rapidement au langage Java dont les fonctionnalités multimédia font le succès. Il s'agit actuellement du plus jeune des langages généralistes orientés objet dont l'utilisation est très répandue.

6.4.5. Conclusion

Nous pouvons alors constater que l'évolution des langages de programmation a été guidée par un souci de maîtrise de la complexité des systèmes modélisés et par un besoin de réutilisabilité. Mais ces deux facteurs n'ont-ils pas finalement des objectifs communs ?

La complexité des logiciels ou des systèmes que l'on cherche à modéliser est généralement trop importante pour être appréhender clairement par l'esprit humain. En outre, ce travail est réalisé par une équipe composée de personnes de différents domaines: des analystes, des programmeurs, des experts du domaine à étudier... La nécessité de communiquer entre eux ne fait qu'accroître la difficulté d'appréhender le problème dans son ensemble. Afin de maîtriser cette complexité, des concepts ont dû être développés: l'*abstraction*, l'*encapsulation*, la *modularité* et la *hiérarchie* (cf. [Booc91], [Hill96]). Ces notions sont omniprésentes dans l'approche orientée objet.

Nous avons discuté de ces concepts sans les nommer lors de la présentation de l'évolution des langages. L'***abstraction*** est le fait de voir toute une partie d'un système comme une entité ayant un rôle défini (e.g. un

sous-programme, un module, un type abstrait de donnée, une classe). L'***encapsulation*** est le fait de cacher des informations à l'utilisateur ou au réutilisateur. Ce concept renforce la notion d'abstraction, puisqu'il permet de choisir les éléments visibles de l'extérieur, qui reflètent normalement le rôle du composant. La ***modularité*** est le fait de décomposer un système en sous-ensembles, regroupant les composants par fonctionnalité. [Booc91] préconise de rendre les modules aussi indépendants que possible les uns des autres, alors que les composants d'un module sont fortement liés entre eux par un partage de données ou de fonctionnalités.

La ***hiérarchie*** est le fait d'organiser les composants d'un système. Lorsque le nombre de composants devient trop important, il est nécessaire, à la fois pour la compréhension du système et pour une clarté indispensable au développement d'un logiciel de qualité, de structurer les composants. [Booc91] propose deux types de classification. La première est caractérisée "appartient à" (elle correspond aux notions d'*agrégation* et de *composition* dans l'approche orientée objet). Une entité peut être considérée comme appartenant à une entité plus grande. La seconde classification est caractérisée "est un" (elle correspond à l'*héritage* dans l'approche orientée objet). Elle consiste à classer les composants dans des catégories hiérarchisées de la plus générale à la plus spécifique.

Ces concepts sont bien évidemment favorables à la réutilisabilité. L'abstraction et l'encapsulation correspondent parfaitement à l'utilisation simple d'un composant: il a un rôle, mais son fonctionnement interne est inconnu. La modularité et la hiérarchie permettent d'organiser les composants, ce qui facilite la compréhension du réutilisateur. En outre, les deux types de hiérarchie exposés au paragraphe précédent autorisent respectivement l'assemblage de composants pour en former un plus grand, et la création d'une sous-catégorie afin d'adapter les fonctionnalités d'une catégorie. Tout cela favorise l'extensibilité des composants.

6.5. Conclusion

En résumé, la réutilisabilité n'entraîne pas d'approche radicalement différente dans le développement de logiciels. Au contrainte, tenter de concevoir des composants réutilisables favorise à long terme le développement de logiciels de qualité. Néanmoins, nous nous sommes concentrés dans ce chapitre sur les aspects techniques et parfois psychologiques liés à la réutilisabilité, mais en très peu d'occasions nous avons parlé des aspects économiques et organisationnels, liés plus directement à la notion de *productivité* (cf. [Coul98]). Nous proposons ici une rapide discussion autour de ces aspects, afin d'en entrevoir certaines difficultés.

Certains ouvrages (e.g. [Stro95], [dSil96]) précisent que le temps consacré à la réutilisation d'un composant inclut le temps de le récupérer (i.e. rechercher le composant le mieux adapté), de l'évaluer (i.e. le composant est-il fiable ?), de le modifier (i.e. l'étendre aux nouveaux besoins) et de l'intégrer (i.e. l'insérer dans l'environnement logiciel où il est réutilisé). Cela soulève d'importants problèmes.

Tout d'abord, comment trouver un composant adapté ? [Leac97] préconise qu'une équipe soit chargée de l'aspect réutilisabilité et réutilisation, à la fois pour examiner les composants disponibles sur le marché, mais aussi pour en produire et fournir un catalogue référençant tous ces composants, afin de permettre une consultation facile et efficace de la part des réutilisateurs.

Il faut également se préoccuper de la sûreté de ces composants. De manière intentionnelle ou non, un mauvais fonctionnement d'un composant peut entraîner un fonctionnement désastreux de tout un système. Pour sécuriser la réutilisation d'un composant, [Wald98] explique que les *systèmes distribués*, où les composants sont sur différentes machines et communiquent par un langage indépendant, empêchent que l'échec d'un seul

composant ne se répercute sur tout le système. Ce dernier reste alors opérationnel avec seulement quelques fonctionnalités manquantes liées au composant défectueux. En ce qui concerne des malfonctions intentionnelles (comme un virus embarqué dans un composant), le langage Java propose par exemple que le fournisseur encrypte ses composants en fournissant une clé publique permettant à tout le monde de vérifier qu'un composant est intègre, c'est-à-dire qu'aucune modification n'a été apportée au composant sans l'autorisation du fournisseur (e.g. [Niem97]).

Pour en revenir plus directement à la productivité, il faut noter que la réutilisabilité est un investissement à long terme, il ne faut pas espérer en tirer des bénéfices dès l'écriture des premiers composants. En effet, le temps nécessaire à la conception de composants réutilisables est bien supérieur à la création de n'importe quel composant dédié. Le bénéfice s'effectue au moment de la réutilisation du composant. Les profits de la réutilisabilité n'apparaîtront qu'après le développement de plusieurs logiciels.

En outre, des aspects plus stratégiques peuvent freiner la réutilisabilité. En effet, si une entreprise fournit à d'autres ses composants, elle ne sera peut être plus sollicitée pour créer d'autres logiciels pour cette même société (et l'on peut penser que ce sera d'autant plus vrai que ses composants seront de bonne qualité). Créer un logiciel dédié est plus intéressant pour le producteur: tout changement dans les besoins du client passera forcément par une demande de modification, en premier lieu au fournisseur du produit original. Les producteurs de logiciels n'ont donc qu'un intérêt limité dans la diffusion de composants réutilisables. En revanche, pour leur fonctionnement interne, la réutilisabilité est très intéressante, puisqu'elle permet un développement rapide de logiciels, qu'ils soient dédiés à la vente ou à un service de l'entreprise.

Ce chapitre nous a également permis de préciser les particularités de la recherche opérationnelle dans le développement de composants logiciels, par rapport à l'approche classique du génie logiciel. En effet, alors que la recherche opérationnelle se concentre principalement sur l'algorithmique, le génie logiciel se concentre plutôt sur la modélisation de systèmes. Nous avons également formalisé en termes de génie logiciel notre objectif de fournir un cadriciel pour le développement d'outils de recherche opérationnelle. Comme nous l'avons déjà précisé, nous ne proposons ici qu'une simple contribution à ce cadriciel qui s'inscrit dans une vision à plus long terme. Néanmoins, nous tenterons, au travers d'un constat sur notre propre expérience, d'apporter des éléments de réponse sur la manière de développer des composants logiciels réutilisables pour la recherche opérationnelle. Enfin, suite à notre discussion sur l'évolution des langages de programmation, l'orientation objet semble être l'approche la plus avancée et mûre actuellement pour élaborer des composants réutilisables et efficaces. Elle guidera donc la suite de notre étude.

Il faut tout de même noter que l'approche orientée objet n'est pas l'ultime progrès dans la conception de logiciels. L'expansion d'Internet avec son architecture distribuée tend à considérer, grâce aux concepts de la programmation orientée objet et aux technologies client / serveur, la conception de logiciels sous une forme séparable plutôt que monolithique (cf. [Brow96]). Cette vision d'un programme comme un assemblage de composants a donné naissance à l'***approche basée composant***. Elle insiste sur une utilisation de composants standards et sur le développement de composants suffisamment complets pour tendre vers une utilisation exclusivement *boîte noire* (i.e. sans connaître les détails) des composants (cf. [Jaza95]). Cette approche est souvent comparée à la manière actuelle de concevoir des circuits électroniques, et hérite des mêmes problèmes liés aux relations client / fournisseur (cf. [Clem96]), notamment concernant la normalisation des composants et leur catalogage systématique. Il est intéressant de constater que cette approche basée composant est très bien implantée dans le domaine de la simulation à événements discrets (e.g. DEVS, HLA, VSE, Silk [Pidd99]), qui est à l'origine de l'approche orientée objet avec SIMULA, et qu'elle est également pionnière dans l'utilisation de composants distribués sur Internet (cf. [Pidd99], [Camp00]).

CHAPITRE 7
L'APPROCHE ORIENTÉE OBJET

Ce chapitre est consacré à l'***approche orientée objet*** et à l'introduction de ses principaux concepts. De nombreux ouvrages proposent d'excellentes présentations de ce domaine (e.g. [Booc91], [Meye97], [Satz96]). Notre objectif ici n'est donc pas d'écrire une fois de plus un état de l'art des concepts du paradigme objet. Néanmoins, nous proposons une introduction suffisante pour qu'un débutant dans le domaine puisse suivre l'étude menée dans ce document (si celui-ci est désireux d'approfondir ces concepts, nous l'invitons à consulter l'un des ouvrages cités précédemment). Mais le but principal de ce chapitre est plutôt de discuter de l'apport de chacun des concepts à la réutilisabilité, et d'expliquer ainsi les différentes manières d'étendre un composant logiciel.

Dans ce document, toutes les modélisations orientées objet que nous présentons sont exprimées avec le langage UML (*Unified Modeling Language* [OMGW1]), qui est le formalisme le plus répandu. La présentation des concepts se déroule de manière à ce que la syntaxe du langage soit introduite naturellement. Tout lecteur qui souhaite plus d'information sur UML peut, par exemple, consulter l'ouvrage [Mull97]. Les exemples de code que nous proposons sont proches de la syntaxe du langage C++, dont de nombreux livres d'apprentissage sont disponibles (e.g. [Lipp92], [Stro97]).

Ce chapitre explique les avantages et les faiblesses des différents concepts de l'orientation objet, ce qui nous permet de justifier notre choix de langage pour l'implémentation de notre bibliothèque et d'entrevoir différentes possibilités de conception pour des composants logiciels réutilisables de recherche opérationnelle. Notre hésitation porte principalement entre Java et C++ qui sont les langages orientés objet les plus répandus, et donc les plus intéressants pour fournir une réutilisabilité logicielle.

7.1. Objet et classe d'objet

7.1.1. Définition d'un objet

De nombreux langages orientés objet ont été développés depuis SIMULA, chacun apportant ses nouveaux concepts et parfois une vision différente de la notion d'objet. Il est donc difficile de proposer une définition simple et univoque d'un objet. Du point de vue de la programmation, nous avons vu au chapitre précédent qu'un objet est une entité encapsulant des données et des opérations, mais la véritable notion d'objet est beaucoup plus abstraite que cela. Parmi les nombreuses définitions données à un objet, nous retenons celle proposée dans [Booc91]: *"Du point de vue de la conscience humaine, un objet correspond à l'une des définitions suivantes:*

- *Une chose palpable et/ou visible.*
- *Quelque chose qui peut être appréhendée intellectuellement.*
- *Quelque chose vers laquelle des pensées ou des actions sont dirigées."*

A ces définitions, nous ajoutons simplement que dans le contexte du génie logiciel ou de la modélisation d'un

système, un objet est considéré comme tel seulement s'il présente un intérêt pour l'étude menée. Un objet a naturellement des caractéristiques, appelées ***propriétés***, qui définissent ce qu'il est. Ces propriétés peuvent être des caractéristiques de son comportement, ses ***méthodes***, ou bien des caractéristiques de son état, ses ***attributs***. En termes informatiques, les méthodes sont donc les opérations qu'un objet peut effectuer et les attributs sont ses données propres.

7.1.2. Classe d'objet

Une ***classe d'objet*** représente une catégorie d'objets. Elle décrit un ensemble d'objets qui partagent des propriétés communes. Par partage d'attributs, on entend partage des types des attributs, et non des valeurs des attributs. Il faut distinguer le type d'un attribut, qui spécifie de quel genre de donnée il s'agit, de la valeur même de l'attribut. Pour renforcer cette différence, nous appelons ***structure*** l'ensemble des types des attributs d'un objet. En résumé, une classe décrit les opérations et la structure communes d'un ensemble d'objets (cf. figure 7.1a pour la notation UML).

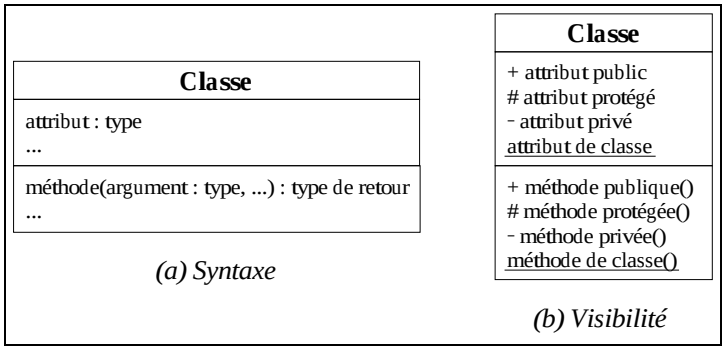


Figure 7.1: Représentation d'une classe.

Comme pour un type de donnée classique, il est possible, à partir d'un modèle qu'est une classe, de créer des objets. L'action de créer un objet à partir d'une classe est appelée ***instanciation*** de la classe (cf. figure 7.2, *(b)* montre la notation UML d'un objet), et l'objet est désigné comme une ***instance*** de cette classe.

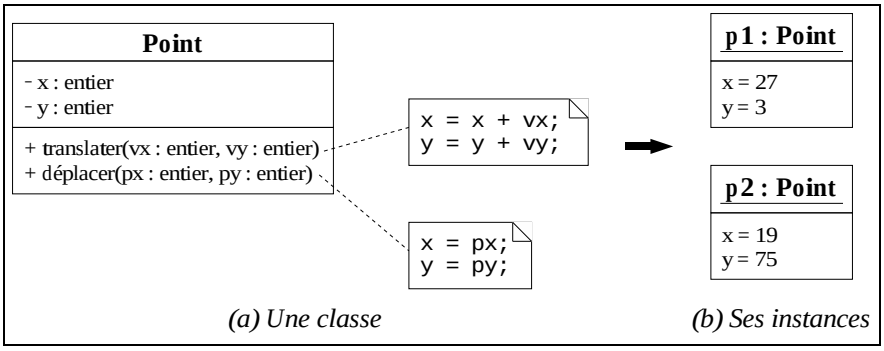


Figure 7.2: Un exemple d'instanciation d'une classe.

Une classe peut être vue comme un objet, c'est-à-dire posséder ses propres propriétés. Pour distinguer les propriétés d'une classe des propriétés de ses objets, on parle respectivement de ***propriétés de classe*** et de ***propriétés d'instance***. Pour assurer le principe d'encapsulation, certaines propriétés peuvent être cachées, soient partiellement, soit complètement. Les propriétés sont ***privées*** si elles sont cachées de tous (excepté de leur classe bien entendu), ***protégées*** si elles sont visibles seulement de certaines classes (les *sous-classes* de

leur classe, cf. section 7.2 pour les détails) et ***publiques*** si elles sont visibles par tous. La figure 7.1b précise les notations employées avec UML pour représenter les différents concepts introduits dans ce paragraphe.

Au chapitre précédent, nous avons défini l'interface et l'implémentation d'un type abstrait de donnée et par conséquent d'une classe. Voici cependant quelques précisions importantes. L'***interface*** d'une classe décrit simplement ses services publics (c'est-à-dire les types de ses attributs publics et uniquement les signatures de ses méthodes publiques). L'***implémentation*** d'une classe décrit tout ce qu'elle doit cacher des utilisateurs (ses propriétés privées ou protégées et le corps de toutes ses méthodes, quelque soit leur visibilité).

7.1.3. Envoi de message

Les méthodes d'un objet sont les seules capables de modifier ses attributs cachés. Si un objet `a` veut modifier l'un des attributs de l'objet `b`, il doit passer par les méthodes de l'interface (i.e. les propriétés publiques) de `b`. L'objet `a` doit donc effectuer une demande, appelée ***message***, auprès de `b`. La figure 7.3 illustre un tel envoi de message, `a` demande à `b` de modifier son attribut `x` en appelant la méthode `setX()`, le message étant émis à partir de la méthode `calcul()` de `a`. La figure emprunte sa notation d'un envoi de message aux langages C++ et Java: étant donné un objet `a`, l'accès à un attribut `x` ou l'appel d'une méthode `f()` se note respectivement `a.x` et `a.f()`.

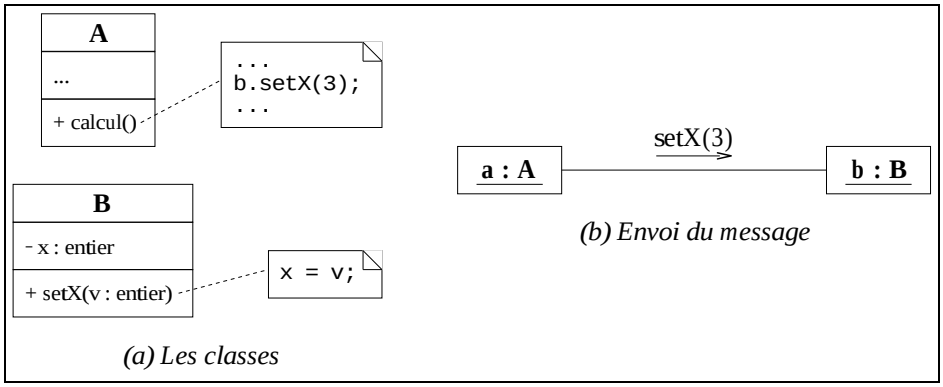


Figure 7.3: Un exemple d'envoi de message entre objets.

Le terme *message* prend plus de sens dans les systèmes où les objets "vivent" en parallèle comme dans un système *multithread*, multitâche ou même distribué, où l'envoi d'un message au sens orienté objet correspond à un véritable échange de messages entre tâches, *threads* ou ordinateurs. Un envoi de message est alors une requête qui peut ne pas aboutir ou être mise en attente si l'objet sollicité est occupé. Un envoi de message ne se résume donc pas toujours à un simple appel de méthode, même en excluant la problématique du parallélisme. Dans la suite du chapitre, nous verrons que certains concepts orientés objet nécessitent un mécanisme d'envoi de message assez complexe, entraînant un surcoût de l'encapsulation souvent incriminée pour expliquer la lenteur d'un programme orienté objet.

7.1.4. Construction et destruction d'un objet

La vie d'un objet est divisée en trois grandes étapes: sa création, sa vie proprement dite et sa suppression. La création se déroule en deux temps: l'*allocation* et la *construction*. Tout d'abord, une zone mémoire est allouée pour stocker les attributs de l'objet. Ensuite, une méthode spéciale de l'objet est appelée, le ***constructeur***,

chargée d'initialiser les attributs. Des arguments peuvent être passés à cette méthode. La figure 7.4 montre pour la classe `Point` un constructeur prenant en paramètres les deux coordonnées du point. Ainsi, l'instruction `new Point(cx,cy)` crée un objet de la classe `Point` avec les coordonnées $(cx; cy)$.

La suppression d'un objet se déroule également en deux temps: la *destruction* et la *libération*. Tout d'abord, une méthode spéciale de l'objet est appelée, le ***destructeur***, chargée de préparer l'objet à sa suppression. Ensuite, la mémoire occupée par les attributs de l'objet est libérée. Aucun argument ne peut être passé au destructeur. Le rôle classique du destructeur est de libérer la mémoire que l'objet a pu allouer au cours de sa vie. Mais il peut servir également à prévenir d'autres objets de sa suppression (e.g. un arc peut informer le graphe auquel il appartient de sa suppression). La figure 7.4 montre pour la classe `Cercle` un destructeur qui supprime l'attribut `centre` créé par le constructeur de la classe. Ainsi, l'instruction `delete centre` supprime l'objet référencé par `centre`.

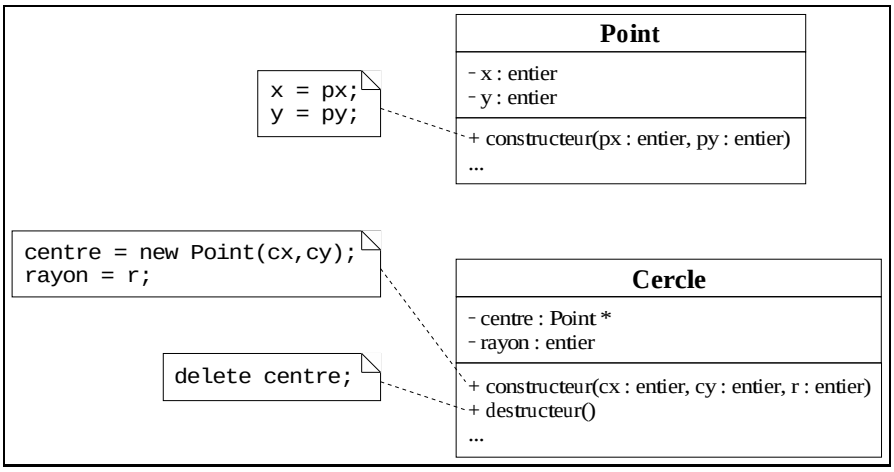


Figure 7.4: Un exemple de constructeur et de destructeur.

Dans ce document, nous ferons la différence entre une variable ou un attribut qui est un objet d'une classe `A`, dont le type sera `A`, et une référence (ou un pointeur, nous ne faisons pas ici de distinction entre les deux termes), dont le type sera `A *`, pour indiquer qu'il s'agit simplement d'un moyen d'accéder à un objet de type `A`. L'instruction `new` renvoie donc une référence sur l'objet qu'elle vient de créer.

7.2. Héritage

7.2.1. Principe

L'héritage est certainement le concept le plus novateur du paradigme objet, mais un mauvais usage peut entraîner des surconsommations mémoire ou processeur importantes. C'est la raison principale pour laquelle les langages objets sont considérés lents par rapport aux langages procéduraux. L'efficacité étant un facteur important pour la recherche opérationnelle, l'héritage mérite une présentation détaillée, aussi bien au niveau des concepts qu'au niveau de l'implémentation, afin d'identifier précisément les sources de surconsommations.

Nous avons expliqué qu'une classe décrit un ensemble d'objets ayant des caractéristiques communes. Il peut y avoir des classes très générales comme des classes très spécifiques. L'***héritage*** consiste en une spécialisa-

tion d'une classe. Lorsqu'une classe `B` hérite d'une classe `A`, cela signifie que les instances de la classe `B` appartiennent également à la classe `A`, mais avec des propriétés supplémentaires propres aux instances de `B`. La classe `A` est appelée ***super-classe*** de `B` et `B` ***sous-classe*** de `A`, en rapport avec la structure hiérarchique que forme l'héritage. La figure 7.5 montre un exemple d'héritage où la classe `Dessin` possède deux sous-classes `Rectangle` et `Point`. Lorsque l'on passe d'une classe à l'une de ses sous-classes, on parle de ***spécialisation***. A l'inverse, lorsque l'on passe d'une classe à l'une de ses super-classes (en effet, une classe peut hériter de plusieurs classes, e.g. [Hill96]), on parle de ***généralisation***.

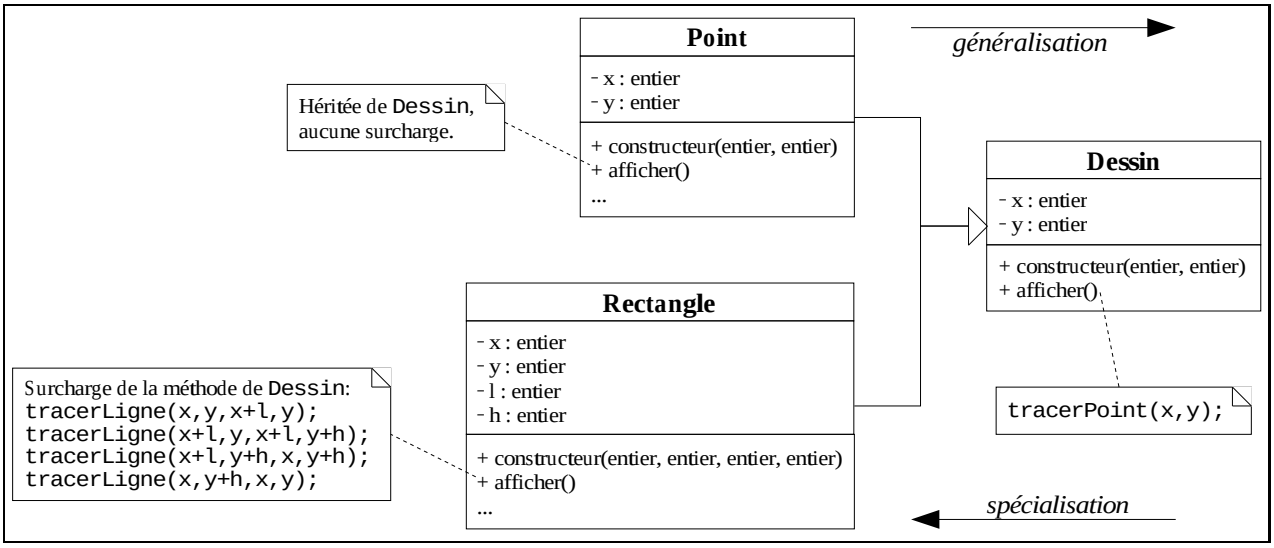


Figure 7.5: Un exemple d'héritage.

Qu'une classe `B` hérite d'une classe `A` signifie que `B` possède l'interface de `A` et profite par conséquent de l'implémentation de `A`. On dit que `B` hérite à la fois de l'interface (i.e. une instance de `B` répond aux mêmes messages que `A`) et de l'implémentation de `A` (i.e. le comportement en réponse aux messages est le même). Cependant, `B` ne peut accéder qu'aux propriétés publiques et protégées (d'où l'intérêt de ce niveau de visibilité) de sa super-classe, mais elle n'a accès en aucun cas aux propriétés privées. Enfin, pour se spécialiser, une sous-classe possède son propre complément d'interface et d'implémentation.

7.2.2. Méthode virtuelle

Une sous-classe peut également modifier le comportement d'une méthode d'instance héritée, en remplaçant l'implémentation de la super-classe par la sienne (uniquement à son niveau, car la super-classe conserve naturellement son implémentation). On parle alors de *surcharge de méthode*. Reprenons l'exemple de la figure 7.5 où les classes `Rectangle` et `Point` héritent de la classe `Dessin`. La super-classe possède une méthode `afficher()` surchargée dans les deux sous-classes. Considérons maintenant le code suivant.

```
Dessin * obj1 = new Dessin(10,20);
Dessin * obj2 = new Point(30,40);
Dessin * obj3 = new Rectangle(0,50,100,60);

obj1.afficher();
obj2.afficher();
obj3.afficher();
```

Les variables `obj1`, `obj2` et `obj3` sont trois références sur des objets de la classe `Dessin` (ou l'une de ses sous-classes). Les trois dernières lignes de l'exemple ont donc toutes les trois la même signification: appeler la méthode `afficher()` de l'objet de la classe `Dessin` référencé. Pour `obj1`, le comportement est assez clair, c'est la méthode de la classe `Dessin` qui est appelée. Pour `obj2`, là aussi, c'est la méthode de `Dessin` qui est appelée, puisque `Point` hérite de la méthode sans la surcharger. En revanche, pour `obj3`, est-ce la méthode de `Dessin` qui est appelée, ou bien est-ce la méthode surchargée de `Rectangle` ?

La réponse dépend de la nature de la méthode: si `afficher()` est *virtuelle* alors c'est la version de `Rectangle` qui est appelée, sinon c'est celle de `Dessin`. Le fait qu'une méthode soit ***virtuelle*** signifie qu'au moment de son appel, c'est sa version la plus spécialisée (en s'arrêtant bien entendu à la classe réelle de l'objet sur lequel la méthode est appliquée) qui est appelée. Si une méthode n'est pas virtuelle, nous l'appelons ***finale*** (en rapport avec le mot clé `final` de Java). Selon les langages, les méthodes sont considérées par défaut virtuelles (e.g. Java) ou finales (e.g. C++), et un mot-clé permet de les faire changer de catégorie (e.g. `final` pour Java, `virtual` pour C++). Dans nos schémas, nous considérons toujours, sauf précision de notre part, des méthodes virtuelles.

7.2.3. Polymorphisme

La ***surcharge de méthode*** consiste à fournir une nouvelle implémentation à une méthode existante. Le fait qu'une méthode puisse prendre plusieurs formes est également appelé ***polymorphisme***. La surcharge d'une méthode virtuelle est une forme de polymorphisme dite ***dynamique*** puisque l'implémentation qui est appelée est choisie au moment de l'envoi du message. Il existe également une forme de ***polymorphisme statique*** où la sélection de l'implémentation de la méthode est effectuée à la compilation. Nous verrons deux exemples de ce type de polymorphisme. L'un est abordé en détail dans la section 7.4 concernant les patrons de composant.

L'autre consiste en une manière de surcharger une méthode différente de la virtualité et indépendante de l'orientation objet. Dans la plupart des langages actuels, il est possible de surcharger une méthode (ou une fonction) en proposant des méthodes avec le même nom mais des signatures différentes. Voici un exemple simple.

```
function max(entier,entier) : entier;
function max(réel,réel) : réel;

...

max(100,200);
max(2.5,2.25);
```

Deux méthodes `max()` ont le même nom. Pour identifier quelle version est appelée, il suffit de regarder le type des paramètres fournis à la méthode, ce qui peut être effectué à la compilation. Ce type de surcharge n'est pas fondamental pour la réutilisabilité, mais il apporte une certaine simplicité dans l'utilisation des méthodes tout à fait appréciable. En outre il est très intéressant au niveau de la maintenabilité, e.g. un remplacement des variables entières par des variables réels dans un programme n'entraîne aucun changement des appels à la méthode `max()`.

En défaveur de la maintenabilité, cette surcharge statique peut entraîner de graves confusions. Imaginons l'appel `max(2,2.5)`. Dans la plupart des langages, le nombre entier est converti implicitement en un nombre réel et c'est donc la version réelle de `max()` qui est appelée. Supposons maintenant qu'au cours d'une phase de

maintenance, une version `max(entier,réel)` soit ajoutée, pour des besoins totalement indépendants de la partie de code que nous étudions. Dans ce cas, notre conversion implicite n'a plus lieu d'être et c'est la nouvelle méthode qui est appelée. Ainsi, une modification dans le programme totalement indépendante de notre partie de code entraîne un changement significatif dans ce même code, ce qui peut être catastrophique si la nouvelle version a des fonctionnalités totalement différentes des anciennes.

Nous pensons donc que cette surcharge statique peut tout à fait être utilisée lorsqu'il s'agit de méthodes dans une classe (la portée de la maintenance est relativement contrôlée: il est impossible de confondre une méthode d'une classe avec celle d'une autre), mais doit être évitée pour les fonctions (la portée de la maintenance est difficile à définir, à moins d'employer un concept comme l'espace de nommage, e.g. le *namespace* en C++).

7.2.4. Réutilisabilité

Pour promouvoir le paradigme objet, il n'est pas rare de trouver des références qui affirment que l'héritage offre un haut degré de réutilisabilité (cf. [John88]). Il est important de discuter ce point. En effet, l'héritage permet à une sous-classe de s'approprier les fonctionnalités de sa super-classe et de les adapter à ses besoins, permettant ainsi beaucoup d'extensibilité. Il s'avère cependant que pour permettre suffisamment d'extensions, la sous-classe potentielle doit voir des éléments de l'implémentation (i.e. des propriétés cachées) de sa super-classe, d'où la possibilité de définir des propriétés protégées. Malheureusement, cela introduit une brèche dans l'encapsulation de la super-classe qui peut se révéler néfaste pour la maintenabilité des composants, puisqu'une légère modification dans l'implémentation de la super-classe peut se répercuter à toutes les sous-classes.

Pour garantir une certaine maintenabilité, l'interface d'une classe est un élément qu'il faut éviter de changer (seuls des ajouts n'ont pas d'impact). Il faut donc bien réfléchir et être perspicace au moment de son élaboration. Pour les mêmes raisons, la partie protégée d'une classe, étant une interface pour ses sous-classes, ne devrait pas être changée. Malheureusement, il est très difficile de ne pas modifier l'implémentation d'une classe, c'est ce qui la fait évoluer, et l'on constate généralement que l'interface d'une classe a moins souvent besoin d'être changée que son implémentation (cf. [Meye97]).

L'héritage fournit en revanche le polymorphisme dynamique, qui est un concept très puissant. Il permet en effet une forte abstraction des objets. Des composants peuvent être développés manipulant un type d'objet très abstrait, utilisant uniquement l'interface de cette classe abstraite. Ensuite, n'importe quel objet héritant de cette classe peut être manipulé par les composants. On obtient alors un découplage fort entre les objets et les composants qui les utilisent, ce qui favorise la maintenabilité et un fort potentiel de réutilisabilité. Le polymorphisme dynamique est beaucoup utilisé pour des traitements massifs. Prenons comme exemple un tableau de références sur des objets de la classe `Dessin` (cf. figure 7.5). Un simple parcours, comme le montre l'exemple suivant, permet d'afficher tous les objets du tableau.

```
Dessin * t[] = ...;

for i = 0 to t.size() do
 t[i].afficher();
end for;
```

En résumé, la spécialisation fournie par l'héritage offre une bonne extensibilité mais peut conduire à une mauvaise maintenabilité. En revanche, le polymorphisme dynamique de l'héritage permet un fort découplage des composants logiciels entraînant une bonne réutilisabilité et maintenabilité. Un petit détail peut cependant modérer l'utilisation du polymorphisme dynamique: il faut que la personne qui conçoit la ***classe de base*** (classe

au sommet de la hiérarchie d'héritage qui nous intéresse) déclare la méthode virtuelle. Et il est toujours délicat d'anticiper les besoins des réutilisateurs. Malheureusement, comme nous l'expliquons dans une prochaine section, la virtualité a un coût et il n'est pas envisageable de considérer toutes les méthodes virtuelles.

7.2.5. Coût de l'encapsulation

Une idée généralement fausse est que l'encapsulation, i.e. le fait accéder à certains attributs par l'intermédiaire d'une méthode, a toujours un coût. Prenons l'exemple très simple d'un attribut `x` dans une classe `A`. Nous pouvons choisir de mettre `x` en public s'il peut être modifié à volonté de l'extérieur de la classe. Cependant, beaucoup préconisent une approche où `x` est privé (ou protégé) et possède deux méthodes `getX()` et `setX()` qui permettent respectivement d'obtenir et de modifier la valeur de `x`. Ce surplus d'écriture peut paraître dans un premier temps superflu, mais il est en fait très important pour la réutilisabilité. Tout d'abord, il faut éviter que l'utilisateur se pose la question de savoir si la donnée `x` est un attribut ou bien le résultat d'un calcul. Dans un simple souci d'homogénéité, il est donc important qu'une donnée soit toujours accédée par une méthode. Ensuite, il est possible qu'au cours du temps une donnée qui était un attribut devienne une donnée calculée (e.g. la longueur d'un arc peut être fixée ou calculée en fonction des coordonnées de ses extrémités). Pour renforcer la maintenabilité, nous recommandons donc de toujours encapsuler les données. Mais cette encapsulation systématique n'a-t-elle pas un impact sur le temps d'exécution ? Si la méthode permettant accéder à un attribut est virtuelle, l'appel a en effet un coût non négligeable. En revanche, lorsqu'il s'agit d'une méthode finale, le coût est identique à un appel à une fonction, voire moins. Il est généralement possible que certaines méthodes soient définies *inline* (soit par l'utilisateur, soit automatiquement par le compilateur).

C'est-à-dire que le mécanisme d'appel de fonction n'est pas utilisé, à la place le code de la méthode est directement recopié à l'endroit de l'appel. Cela revient donc exactement à ce que le programmeur réécrive le corps de la méthode directement à chaque endroit où il en a besoin. Le bénéfice direct est qu'il n'y a pas de mécanisme d'appel à une fonction, qui est relativement coûteux (sauvegarde du contexte avant l'appel, recopie des paramètres...). Le bénéfice indirect est que le compilateur peut éventuellement effectuer une optimisation qu'il n'aurait pas pu faire avec l'appel de fonction. Des expérimentations menées dans [Lipp97] (cf. sa table 3.1) montrent pour C++ que l'accès à une variable locale est tout aussi coûteux que l'accès encapsulé par une méthode *inline* à l'attribut d'un objet. L'encapsulation n'a donc aucun coût, si ce n'est un peu plus de code source à écrire, et elle renforce l'homogénéité et la maintenabilité des composants.

7.2.6. Coût de la virtualité

L'encapsulation n'a finalement un coût que lorsqu'une méthode virtuelle est appelée. Mais quel est exactement l'impact d'un appel à une méthode virtuelle plutôt qu'à une méthode finale ? Nous proposons d'expliquer succinctement le mécanisme mis en oeuvre lors d'un polymorphisme dynamique. Nous espérons que cela permettra au lecteur de mieux juger de la pertinence d'employer l'héritage et donc la virtualité par rapport à une autre technique. Le petit résumé que nous exposons ici est extrait de [Lipp97] et [Drie96] qui expliquent dans les détails, avec la prise en compte de nombreux aspects que nous passons ici sous silence, comment un appel à une méthode virtuelle est réalisé.

L'idée originale est très simple. Lorsqu'un objet appartient à une classe qui possède une méthode virtuelle ou hérite d'une méthode virtuelle, alors il possède un attribut supplémentaire, une référence, appelée ***pointeur virtuel*** ou *v-pointeur*, qui pointe sur un tableau, appelé ***table virtuelle*** ou *v-table*. Cette table, unique pour

chaque classe, contient les pointeurs des implémentations des méthodes virtuelles qu'un objet effectivement de la classe (i.e. n'appartenant pas à l'une des sous-classes) peut appeler. La figure 7.6 illustre le mécanisme d'appel à une fonction virtuelle avec cette structure. La figure considère une référence sur un objet de la classe `Dessin`. Pour appeler la méthode virtuelle `afficher()`, qui possède trois implémentations selon que l'objet appartienne à l'une des trois classes, il suffit accéder à la v-table référencée par le v-pointeur de l'objet et d'exécuter l'implémentation de la méthode `afficher()` qui s'y trouve.

Le surcoût engendré par le polymorphisme dynamique semble assez immédiat à évaluer. Au lieu d'appeler une simple fonction, il faut tout d'abord accéder à la table virtuelle de l'objet, et ensuite il faut trouver la méthode appelée dans cette table. Ces deux étapes ne se résument pas toujours à une simple indirection de pointeur, à cause de contraintes liées à des concepts que nous aborderons très peu, voire pas du tout dans ce document: héritage multiple, héritage virtuel, information de type en temps réel (e.g. RTTI en C++)...

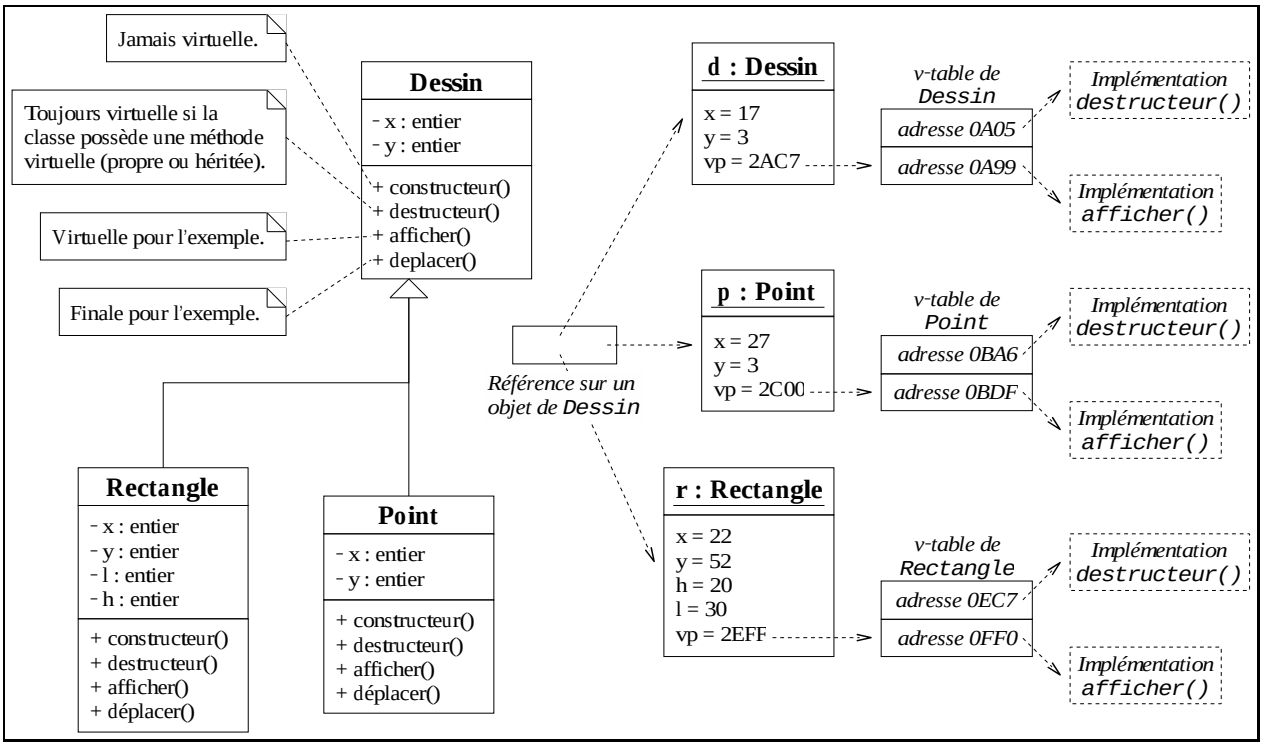


Figure 7.6: Polymorphisme dynamique et table virtuelle.

Le tableau 7.1, extrait de [Lipp97] (table 4.1) et confirmé par le rapport [ORio02] d'un groupe de standardisation ISO/IEC, fournit des résultats intéressants sur le coût d'un appel à une fonction virtuelle. Le test a consisté à appeler un nombre important de fois une même méthode (contenant seulement quelques lignes de calculs simples), cette méthode étant tout d'abord *inline*, puis une méthode de classe (techniquement équivalente à une fonction), une méthode d'instance finale et enfin une méthode virtuelle (issue d'un héritage simple comme nous l'avons défini, mais également d'un héritage multiple ou virtuel, hors de propos ici, mais dont le résultat est intéressant pour ceux qui connaissent). Les résultats sont présentés sans optimisation du compilateur, afin d'isoler le coût direct de la virtualité, et ensuite avec les optimisations de manière à entrevoir le potentiel des coûts indirects.

Ce tableau montre tout d'abord pour une méthode simple que la suppression de la propriété *inline* entraîne une augmentation de 30 % du temps d'exécution. Ce coût étant fixe, l'impact tend à être négligeable plus

les fonctions sont longues à exécuter. Il faut également noter qu'une méthode *inline* est dupliquée dans le code compilé autant de fois qu'elle est appelée. Dans un développement logiciel à grande échelle, comme une bibliothèque, cela peut entraîner un accroissement de la taille du code compilé très important, voire démesuré, qui est en relation directe avec le temps et la mémoire utilisés pour la compilation. Précisons aussi que le corps d'une méthode *inline* publique doit être dans l'interface de la classe, afin qu'il soit possible, à l'extérieur de la classe, de dupliquer le corps de la méthode à l'endroit de son appel. Enfin en mode optimisé, on remarque que le fait d'être *inline* a permis une très importante amélioration du code de la méthode impossible sinon.

Regardons maintenant le passage à une méthode virtuelle, l'impact n'est pas aussi important que la "rumeur" le laisse croire. En effet, en mode non optimisé, la virtualité introduit un ralentissement de $13\ \%$, ce qui est assez important. En mode optimisé, l'impact est pratiquement divisé par deux, $7\ \%$, ce qui est non négligeable mais pas catastrophique pour la plupart des applications. Cependant, dans certains cas d'héritage (e.g. virtuel), l'impact monte à $17\ \%$. L'inconvénient de ces résultats est qu'ils reposent sur une seule expérimentation. [Drie96] a mené une étude sur une dizaine de logiciels existants. Elle a consisté à simuler les programmes pour évaluer le coût direct de la virtualité, tout d'abord en supposant que les appels aux méthodes virtuelles étaient miraculeusement aussi performants que des appels à de simples fonctions. Ensuite, des simulations avec le mécanisme réel de virtualité ont été effectuées et les temps simulés d'exécution ont été comparés. Les résultats révèlent qu'en moyenne les programmes perdent $5,2\ \%$ de leur temps dans l'appel aux méthodes virtuelles et dans le pire des cas $29\ \%$. Une autre expérimentation a consisté à remplacer tous les appels aux méthodes par des appels virtuels, le résultat est une moyenne de $13,7\ \%$ de perte de temps avec un maximum de $47\ \%$.

Type d'accès	Temps (sans optimisation) par rapport à		Temps (avec optimisation) par rapport à	
	inline (1)	fonction (2)	*inline* (1)	fonction (2)
Méthode *inline (1)*	1	-	1	-
Fonction / méthode de classe *(2)*	1,3	1	55,38	1
Méthode d'instance fi nale	1,3	1	55,38	1
Méthode virtuelle (héritage simple)	1,46	1,13	59,5	1,07
Méthode virtuelle (héritage multiple)	1,5	1,15	61,25	1,1
Méthode virtuelle (héritage virtuel)	1,5	1,15	65	1,17

Tableau 7.1: Coût de la virtualité.

Nous verrons par la suite que l'héritage et en particulier les appels virtuels sont la cause majeure de l'inefficacité de certains programmes orientés objet par rapport à une approche procédurale. Néanmoins, les études rappelées ici tendent à le montrer, il ne s'agit pas d'un facteur de $100\ \%$ ou plus comme certaines "rumeurs" le prétendent. Cependant, il faut reconnaître qu'un abus du polymorphisme dynamique peut conduire à ce genre de mauvaises performances. La suite du chapitre a donc pour but de montrer comment contourner dans certains cas l'héritage et la virtualité, et donc de maintenir une efficacité raisonnable des composants.

7.3. Composition, agrégation et association

7.3.1. Différences conceptuelles

Nous présentons ici trois types de relation similaires entre classes: la composition, l'agrégation et l'association. Les deux premières expriment l'appartenance d'un objet à un objet plus vaste. Elles traduisent l'action "possède" ou "est composé(e)" entre un composé et ses composants. Par exemple, la relation "un graphe est com-

posé d'arcs et de noeuds" se traduit par une composition entre la classe `Graphe` et les classes `Arc` et `Noeud` (cf. figure 7.7). Dans une composition, les composants (e.g. les arcs ou les noeuds) n'ont pas d'existence en dehors de leur composé (e.g. le graphe). La vie des composants est donc liée directement à celle du composé. L'agrégation diffère de la composition sur ce point, les composants ont une vie propre et n'appartiennent à un composé que pour une durée limitée. Notamment, dans une agrégation, la destruction du composé n'entraîne pas la destruction des composants, contrairement à la composition. Par exemple, la relation "un cycle est composé d'arcs" est traduite par une agrégation entre la classe `Cycle` et la classe `Arc` (cf. figure 7.8).

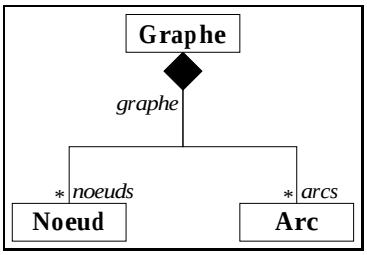


Figure 7.7: Un exemple de composition.

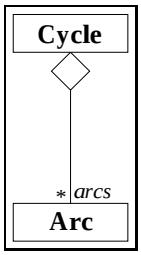


Figure 7.8: Un exemple d'agrégation.

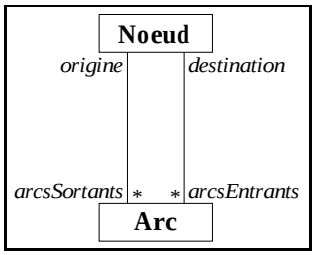


Figure 7.9: Un exemple d'association.

L'association diffère des deux autres relations par le fait qu'il n'y ait pas de dominance de l'une des deux classes impliquées. Par exemple, la relation "un arc possède un noeud origine et un noeud destination" se traduit par deux associations entre la classe `Arc` et la classe `Noeud` (cf. figure 7.9). Pour une association entre deux classes, il est possible de spécifier combien de relations il peut effectivement exister entre des instances de ces classes, simplement en apposant une cardinalité sur l'association. Dans la figure 7.9, l'arc possède un noeud origine et un noeud destination, donc la cardinalité du point de vue de l'arc est 1 pour les deux associations. En revanche du point de vue du noeud, l'association peut aller de 0 à n, d'où le symbole $*$ pour symboliser cette cardinalité. Une cardinalité explicite peut être apposée (e.g. $1..4$).

7.3.2. Similitudes d'implémentation

Ces trois relations, dont la différence est très importante au niveau conceptuel, sont quasiment identiques au niveau de l'implémentation. Toutes les trois sont en effet traduites par la présence d'attributs des classes composants dans la classe composé.

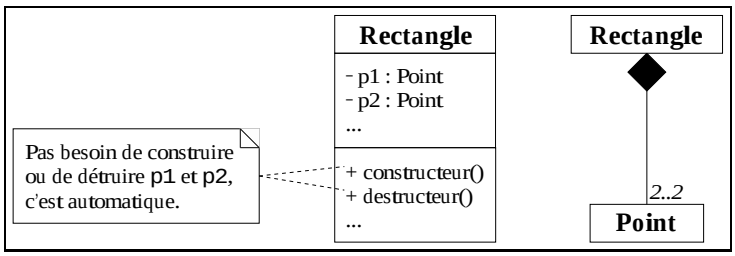


Figure 7.10: Un exemple d'implémentation (avec attributs objets) d'une composition.

Pour commencer, tout attribut objet d'une classe représente forcément une composition entre la classe possédant l'attribut et la classe indiquant le type de l'attribut. Par exemple, la figure 7.10 représente une classe `Rectangle` possédant deux attributs de la classe `Point`. Il s'agit d'une composition puisque les composants (i.e. les attributs) sont créés et supprimés en même temps que le composé (i.e. un objet de la classe). Plus précisément, lorsque le constructeur (respectivement le destructeur) de `Rectangle` est appelé, les constructeurs (respectivement les destructeurs) de `p1` et `p2` sont automatiquement exécutés.

Reprenons l'exemple de la composition de la figure 7.7, la classe `Graphe` possède deux attributs `arcs` et `noeuds`, deux listes de références respectivement d'arcs et de noeuds (cf. figure 7.11). Les attributs induits par une composition peuvent donc être des références, la classe composé doit alors gérer la création et la suppression des composants (cf. figure 7.11).

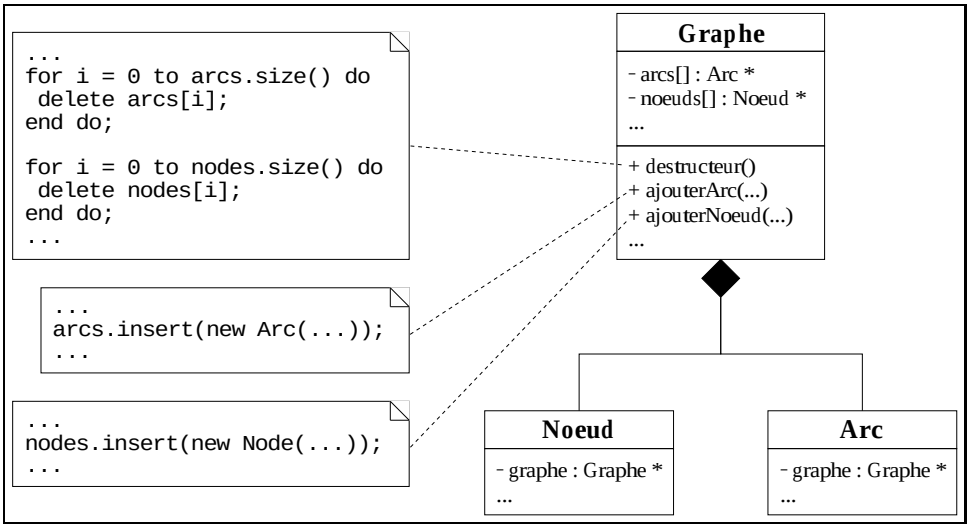


Figure 7.11: Un exemple d'implémentation (avec attributs références) d'une composition.

L'agrégation et l'association se traduisent également par des attributs références d'objets. Dans l'exemple de l'agrégation (cf. figure 7.8), la classe `Cycle` possède un attribut `arcs` qui est une liste de références sur des objets de la classe `Arc` (cf. figure 7.12). Dans l'exemple d'association (cf. figure 7.9), la classe `Arc` possède deux attributs, `origine` et `destination` qui sont des références sur des objets de la classe `Noeud` (cf. figure 7.13). Cette dernière possède deux attributs `arcsEntrants` et `arcsSortants` qui sont des listes de références d'objets de la classe `Arc`.

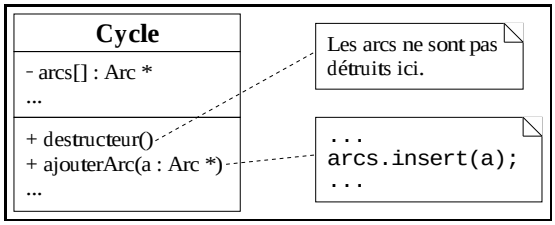


Figure 7.12: Un exemple d'implémentation d'une agrégation.

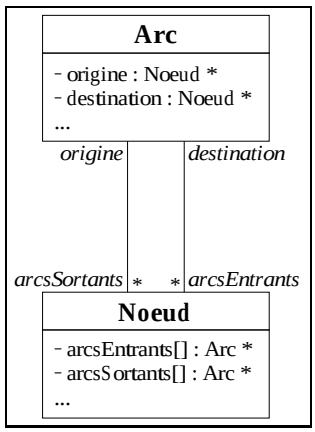


Figure 7.13: Un exemple d'implémentation d'une association.

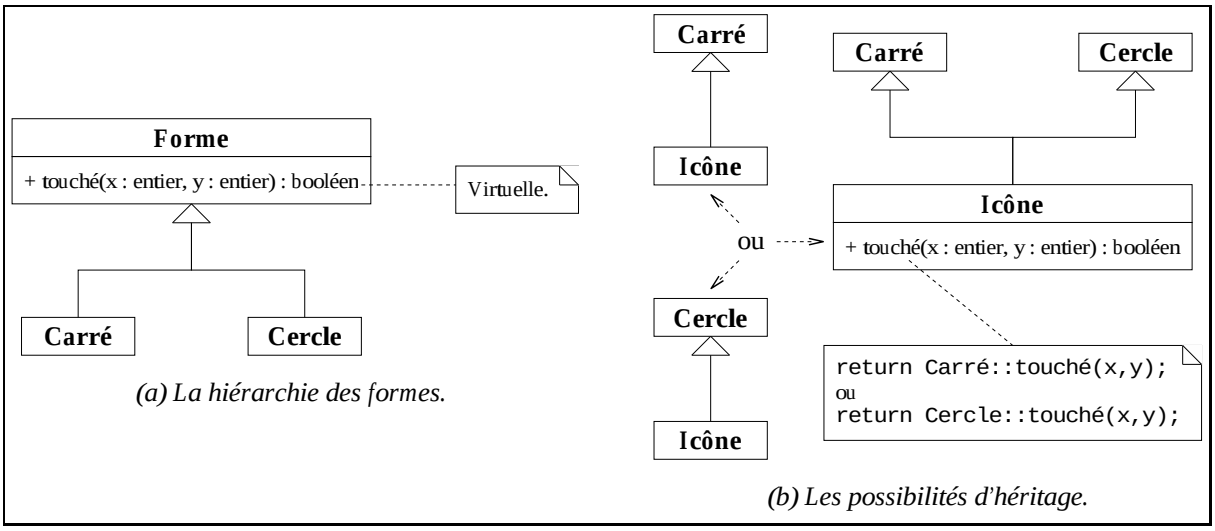
Ces trois relations, que nous résumerons par le terme *composition* dans la suite du document afin de faciliter la discussion, sont une possibilité supplémentaire de réutilisabilité. Par l'assemblage d'objets, il est possible d'en construire un plus complexe et de lui ajouter des fonctionnalités. Contrairement à l'héritage, il n'est pas possible de modifier les fonctionnalités des objets réutilisés, en revanche il est possible de les faire interagir pour produire des fonctionnalités plus complexes. Cela suppose de la part des composants fournis d'être très

complets pour pouvoir être adaptés à de nombreux besoins. C'est donc le principal défaut de ce type de réutilisation: si le concepteur de l'objet réutilisé n'a pas prévu une fonctionnalité, elle ne peut pas toujours être rajoutée (e.g. si l'accès à un élément privé est nécessaire). L'avantage majeur en revanche est la réutilisation en *boîte noire* des classes, ce qui apporte une grande indépendance entre les composants et leur composé. Une modification de l'implémentation d'un composant n'entraîne aucune modification pour le composé, ce qui est tout à fait différent de l'héritage, qui est une réutilisation en *boîte blanche* (cf. [Gamm95]), et qui apporte une dépendance forte entre la super-classe et ses sous-classes.

7.3.3. Délégation

La tentation avec l'approche objet est grande d'utiliser l'héritage chaque fois que l'on souhaite étendre un composant (cf. [John88], l'approche *programming-by-difference*). C'est la solution de facilité. On a besoin de modifier la fonctionnalité d'une classe, il suffit d'hériter, de modifier les quelques méthodes qu'il faut et c'est terminé. Outre le nombre important de classes que cela peut produire, nous avons déjà discuté des problèmes de maintenabilité qu'entraîne l'héritage. Tout cela peut conduire à long terme à une hiérarchie d'héritage complexe où le réutilisateur se perd (cf. [Gamm95]), ce qui est dommage quand on sait que l'un des objectifs majeurs du paradigme objet est de fournir une vision claire de la structure d'un logiciel.

En outre, la hiérarchie fournie par l'héritage est statique, cela veut dire qu'au cours de l'exécution du programme, il n'est pas possible de modifier l'héritage d'une classe pour qu'un objet héritant des fonctionnalités d'une classe puisse hériter de celles d'une autre. Considérons l'exemple de la figure 7.14a qui présente une hiérarchie d'héritage avec une classe de base `Forme` spécialisée en deux sous-classes `Carré` et `Cercle`. Une classe `Icône` doit hériter des propriétés d'une des classes `Forme`.



(a) La hiérarchie des formes.

(b) Les possibilités d'héritage.

Figure 7.14: Un exemple de l'intérêt de la délégation.

Comme le montre la figure 7.14b, il est possible de décider statiquement de la forme de l'icône par un héritage de `Carré`, de `Cercle` ou des deux (c'est ce que l'on appelle l'***héritage multiple***). Imaginons maintenant que l'on souhaite savoir si un clic de souris a atteint un objet `Icône`. Il suffit alors d'appeler la méthode `touché()` avec en paramètre les coordonnées de la souris. Pour l'héritage simple, la méthode héritée est appelée. Pour l'héritage multiple, il faut faire le choix d'appeler la méthode de `Carré` ou de `Cercle` au moment de l'écriture de la classe `Icône` (cf. figure 7.14b).

Mais comment faire pour permettre à un tel objet de changer sa forme en cours d'exécution et bien entendu d'hériter des caractéristiques de sa nouvelle forme ? Grâce à la composition, il est possible de changer dynamiquement la forme de l'objet `Icône`. Cette technique est appelée ***délégation*** (cf. [John91]). Elle consiste pour la classe qui hérite à agréger un objet de la classe dont elle veut les fonctionnalités. Elle fournit ensuite une interface identique à celle de l'objet agrégé. Chaque fois qu'une de ses méthodes est appelée, elle délègue l'appel, i.e. elle appelle la méthode homonyme de l'attribut qu'elle agrège. Dans notre exemple, il suffit que `Icône` agrège un objet de la classe `Forme`, pouvant être changé par la méthode `setForme()`. Ainsi, chaque fois qu'elle est appelée, la méthode `touché()` délègue son travail à la méthode `touché()` de son attribut `forme`.

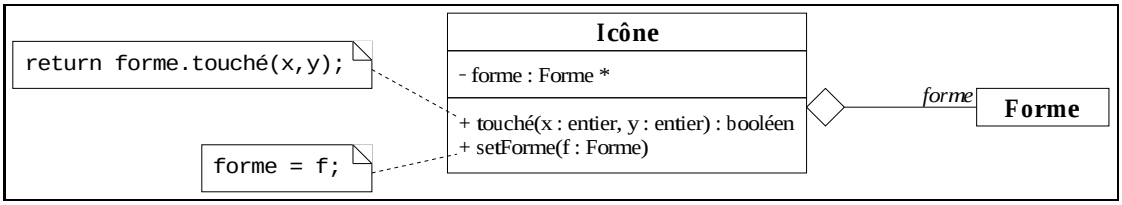


Figure 7.15: Un exemple de délégation.

La délégation est une technique intéressante pour remplacer l'héritage. Elle fournit une forme de polymorphisme dynamique gérée en partie par le concepteur. Cependant, elle nécessite toujours le polymorphisme de l'héritage, ici pour la méthode `touché()` de la classe `Forme`. L'intérêt est donc principalement le dynamisme de ce pseudo-héritage. Mais il faut se méfier de la complexité (liée à l'augmentation des paramètres de la classe) qu'apporte cette technique dans l'utilisation du composant. [Gamm95] conseille donc d'employer cette technique avec modération, quand cela est réellement nécessaire (c'est le cas de notre exemple), et surtout quand cela simplifie la conception.

7.4. Patron de composant

Au niveau de sa mise en oeuvre, le concept de *patron* est certainement l'un des plus complexes de l'orientation objet. C'est la raison pour laquelle il a fallu attendre des années avant que C++ l'exploite pleinement et que ses compilateurs fournissent un standard sur cet aspect. Cela explique aussi probablement que Java n'ait pas implémenté initialement ce concept, ce qui lui fait aujourd'hui défaut, même si des tentatives pour l'ajouter sont actuellement en cours.

7.4.1. Patron de fonction, méta-fonction

Un ***patron de fonction***, ou une ***méta-fonction***, est un moyen d'écrire un modèle d'une fonction paramétré sur certains types des données qu'elle manipule. Par exemple, considérons une fonction `max()` qui retourne le maximum de deux nombres. Nous avons vu à la section 7.2 qu'une forme de polymorphisme statique est possible, simplement en créant une fonction avec le même nom `max()` pour chaque type de nombre que l'on souhaite prendre en charge. Il faut alors créer autant de fonctions qu'il y a de types de nombre, ce qui n'est pas gênant lorsque les contenus de ces fonctions sont fondamentalement différents. Mais dans notre exemple, les corps des fonctions sont identiques:

```
if (n1 < n2) return n2;
else return n1;
```

L'idéal serait donc de pouvoir considérer le type des nombres manipulés comme un paramètre. C'est exactement ce que permet le patron de fonction. Pour notre exemple, nous proposons une méta-fonction `max()` de paramètre `T`, que l'on note `max<T>()`.

```
function max<T>(n1 : T, n2 : T) : T
 if (n1 < n2) return n2;
 else return n1;
end function;
```

Pour simplifier notre discussion, nous appelons ***arguments*** les paramètres classiques d'une fonction (e.g. `n1` et `n2`), et réservons le terme ***paramètres*** à ceux du patron (e.g. `T`). La méta-fonction `max<T>()` est un modèle qui permet de créer des fonctions `max()` où le type de `T` est connu. Une telle fonction est identifiée en remplaçant les paramètres de la méta-fonction par les types choisis pour la fonction. Par exemple, la fonction `max()` pour les réels sera identifiée `max<réel>()` et sera utilisée de la manière suivante.

```
max<réel>(2,5);
```

Il est également possible de profiter d'un polymorphisme statique et de ne pas spécifier explicitement les paramètres fournis à la méta-fonction. L'exemple suivant appelle la fonction `max<entier>()` pour la première ligne et `max<réel>()` pour la seconde.

```
max(2,5);
max(2.5,2.4);
```

Ce polymorphisme statique (identique à celui présenté à la section 7.2, i.e. avec les mêmes avantages et défauts) est possible uniquement si les types des arguments de la fonction permettent de déterminer tous les paramètres de la méta-fonction. Il est en effet tout à fait possible de paramétrer une méta-fonction sur un type qui n'est utilisé qu'à l'intérieur de la fonction (e.g. une structure de données temporaire pour un algorithme).

7.4.2. Patron de classe, méta-classe

Les patrons de fonction ouvrent une nouvelle perspective de réutilisabilité des opérations. Les patrons de classe, tout à fait similaires, offrent la même approche pour les structures de données. Un ***patron de classe***, ou une ***méta-classe*** (appelée aussi classe générique, classe paramétrée ou *template*), est un moyen d'écrire un modèle d'une classe paramétré sur certains types des données qu'elle manipule (les attributs, les arguments des méthodes ou les variables locales). Un exemple classique est une classe représentant une liste d'éléments, la classe étant paramétrée sur le type des éléments qu'elle agrège.

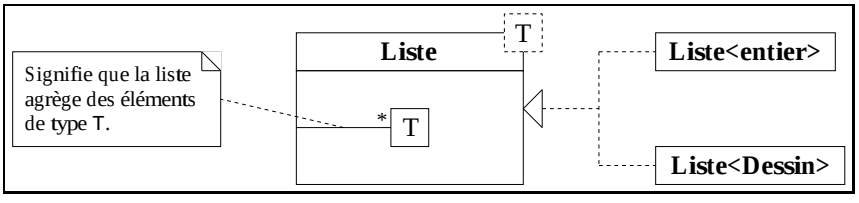


Figure 7.16: Un exemple de patron de classe.

La figure 7.16 illustre la notation d'une telle méta-classe en UML et montre la relation qu'il existe entre une méta-classe et les classes créées sur son modèle. Il s'agit de la même relation d'***instanciation*** qu'il existe entre une classe et ses objets (cette relation existe bien évidemment aussi entre une méta-fonction et une fonction). A partir d'une méta-classe, comme pour une méta-fonction, il est très simple d'instancier des classes pour

manipuler différents types d'objets. Il faut cependant bien faire la différence entre l'instanciation d'un objet et l'instanciation d'une classe. La première consiste en l'allocation physique d'une zone mémoire pour l'évolution d'un élément dynamique du programme, alors que la seconde consiste en la création d'un nouveau type de donnée qui est statique dans le programme (nous excluons de cette discussion les langages qui permettent une manipulation dynamique des classes) et permettra par la suite d'instancier des objets. L'exemple suivant crée des objets de classes instanciées à partir du patron `Liste<T>`.

```
Liste<entier> * liste1 = new Liste<entier>();
Liste<Dessin> * liste2 = new Liste<Dessin>();
Liste<Noeud> *  liste3;
```

L'expression `Liste<Noeud>` par exemple est seulement un moyen de représenter un type de donnée, elle ne correspond en aucun cas à l'exécution d'un quelconque code (hormis le fait que la variable `liste3` est créée), mais on parle pourtant d'instanciation, puisqu'elle donne de vrais types aux paramètres du patron. Il y a tout de même une phase de création de la classe au moment de la compilation. Lorsque le type `Liste<Noeud>` est rencontré pour la première fois, le compilateur va instancier la classe, en dupliquant simplement le modèle de la méta-classe et en remplaçant le paramètre `T` par le type `Dessin`.

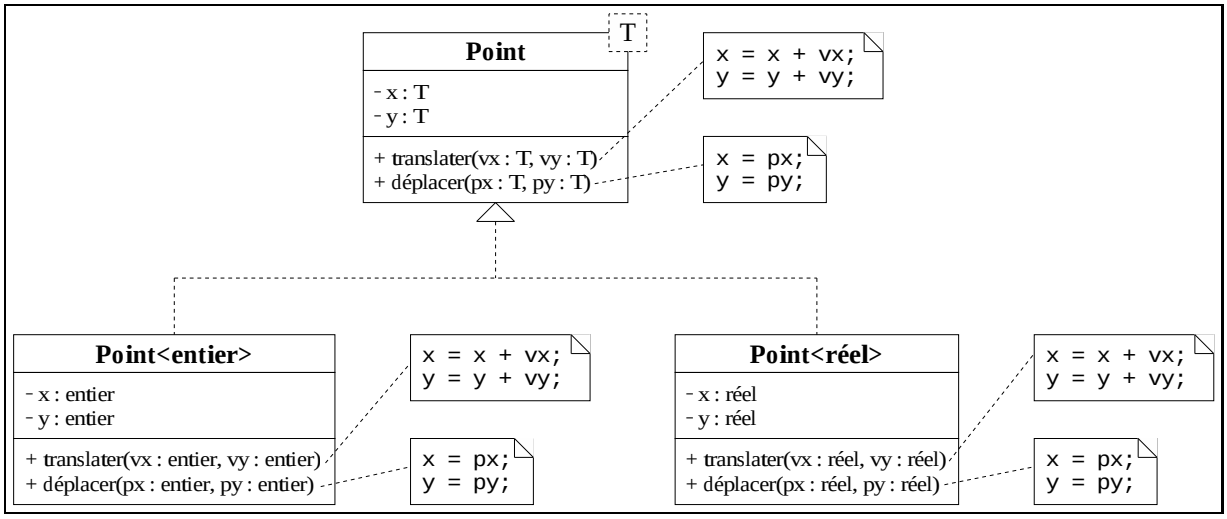


Figure 7.17: Un exemple de duplication de code lors de l'instanciation d'un patron.

Le mécanisme d'instanciation d'une méta-classe (ou d'une méta-fonction) revient donc exactement à ce que le programmeur duplique une classe (ou une fonction) qu'il utilise pour un type donné et qu'il remplace manuellement ce type par un autre. Il n'y a donc aucun coût lié à l'utilisation d'un patron en terme de temps d'exécution, les mêmes optimisations que pour une recopie manuelle peuvent être effectuées. Au niveau de la taille du code, chaque instanciation d'un patron correspond à une duplication du code (cf. figure 7.17). Mais il faut savoir qu'actuellement des techniques d'instanciation permettent de factoriser le code dupliqué et réduisent significativement la taille du code généré (cf. [ORio02]). Heureusement, à quelques exceptions près (e.g. les systèmes embarqués), la taille du code d'un logiciel n'est plus critique à l'heure actuelle.

Il faut savoir également que, comme un patron est un modèle de composant, son implémentation ne peut pas être cachée, elle doit être connue du compilateur pour l'instanciation. Comme pour les méthodes *inline*, un patron se retrouve entièrement dans l'interface. Une modification de l'implémentation d'un patron entraîne nécessairement une recompilation de toutes ses instances. Ce qui peut conduire à des temps de compilation assez longs si les instances sont complexes et nombreuses (e.g. notre bibliothèque).

7.4.3. Polymorphisme statique, notion de concept

L'intérêt principal du patron est que l'on développe un composant sans se soucier de la classe réelle de certaines données qu'il manipule, on suppose simplement qu'elles répondent à certains messages (i.e. que leur interface possède certaines propriétés) nécessaires pour la conception du composant. Ensuite, lorsque le réutilisateur veut exploiter le patron, il lui suffit de l'instancier en précisant les types des paramètres. Le composant est alors prêt à l'emploi, à condition que les paramètres possèdent les éléments d'interface requis. Cela fournit donc une forme de ***polymorphisme statique***, puisqu'au moment où le patron est conçu, on fait appel à des méthodes sur des objets dont on ne connaît absolument pas le type, et c'est au moment de l'instanciation de la classe, i.e. à la compilation, que l'implémentation de la méthode pourra être déterminée.

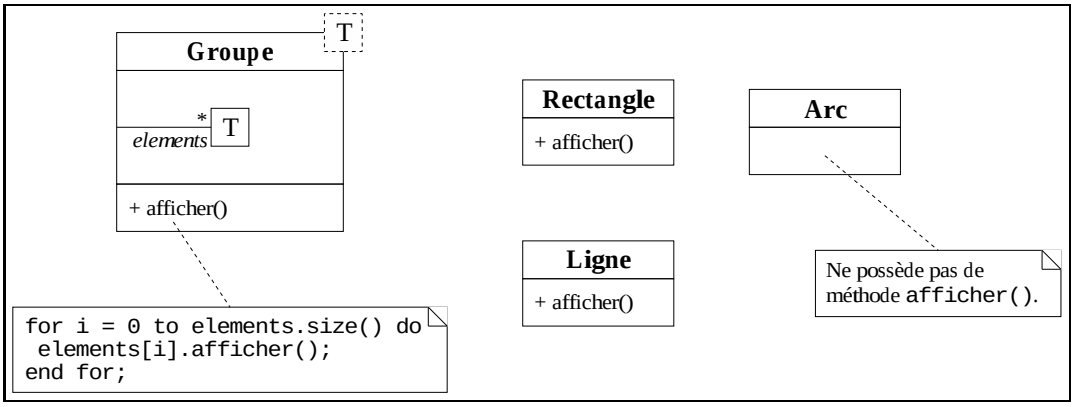


Figure 7.18: Un exemple de polymorphisme statique avec un patron de classe.

Pour préciser tout cela, considérons l'exemple de la figure 7.18. Une méta-classe `Groupe<T>` représente un groupement d'objets et possède une méthode `afficher()` qui appelle la méthode du même nom pour tous les objets du groupe. L'instanciation de la méta-classe avec la classe `Ligne` (ou la classe `Rectangle`) s'effectue sans difficulté. Le compilateur, au moment où il écrit la méthode `afficher()` de la classe `Groupe<Ligne>`, connaît le type des éléments du groupe et peut donc identifier l'implémentation de la méthode `afficher()` de `Ligne`. En revanche, s'il tente d'écrire la classe `Groupe<Arc>`, il ne va pas trouver de méthode `afficher()` pour la classe `Arc` et va donc échouer dans l'instanciation.

Il est alors possible de spécifier pour un patron les *concepts* (cf. [Aust99]) qu'une classe (ou un type de manière générale) doit modéliser pour être un candidat à l'instanciation de l'un de ses paramètres. Un ***concept*** est une fonctionnalité, i.e. un ensemble de propriétés dans l'interface, qu'une classe doit fournir. Un concept se traduit en programmation orientée objet par une ***interface*** qui est l'équivalent d'une classe possédant uniquement des signatures de méthodes dans sa partie publique. L'interface permet ainsi d'abstraire un service qu'une classe peut fournir. La relation d'héritage est naturellement possible entre interfaces.

Le formalisme UML et le langage Java permettent une distinction entre une classe et une interface (cf. figure 7.19a). En revanche, dans le langage C++ une interface est simplement représentée par une ***classe abstraite*** qui est une classe possédant uniquement des signatures de méthodes, sans aucun corps (les plus avertis comprendront que l'on assimile ici la classe abstraite au terme exact de *classe abstraite pure*). Une classe non abstraite est dite ***concrète***. Etant dépourvue de toute implémentation, il est impossible d'instancier une classe abstraite. Le fait qu'une classe fournisse une certaine interface est formalisé par une relation d'***implémentation***. Par exemple, la figure 7.19a montre que la classe `Ligne` implémente l'interface `Affichable`. En C++, cela se traduit par le fait que la classe `Ligne` hérite de la classe abstraite `Affichable` (cf. figure 7.19b).

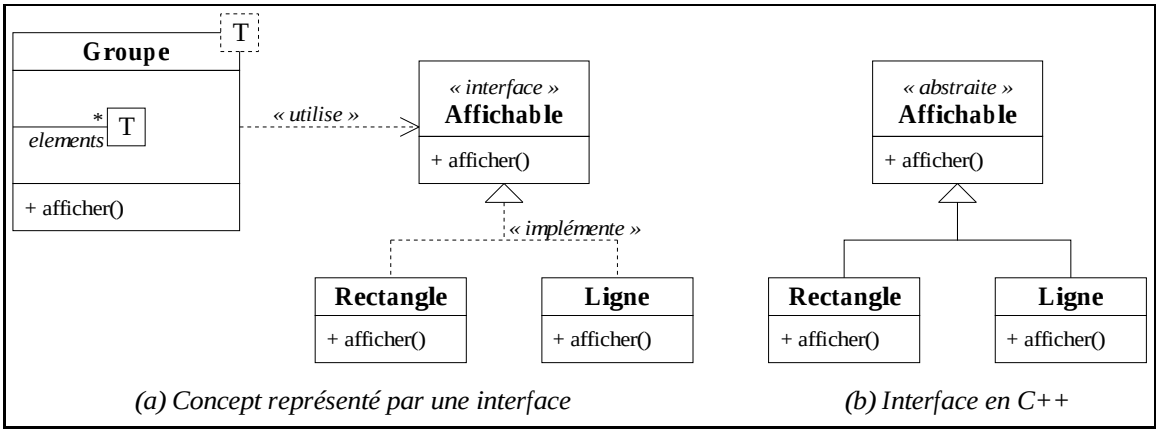


Figure 7.19: Un exemple de concept.

Il est donc possible de formaliser cette notion de concept en UML par la notion d'interface. Cependant, en C++ (Java n'implémentant pas les patrons), il n'est pas possible de spécifier directement qu'un patron n'accepte que certains concepts. Le moyen habituel pour prévenir le réutilisateur est de fournir une documentation détaillée du patron et des concepts qu'il supporte, e.g. le cadriciel STL (*Standard Template Library* [SGIWb]). Mais si le réutilisateur tente une instanciation impossible, il est confronté à un message d'erreur dans un code qu'il n'a pas écrit (le message apparaît lorsque le compilateur tente d'écrire un envoi de message à une méthode qui n'existe pas, cf. [Siek00]). Un défaut majeur des patrons, du moins en C++, est qu'en cas de mauvaise utilisation, le message d'erreur retourné par le compilateur n'est compréhensible que par le créateur du patron. Nous parlons au dernier chapitre de techniques pour atténuer ce désagrément.

7.4.4. Réutilisabilité

A première vue, le patron n'est pas un concept fondamental pour la réutilisabilité. En effet, son unique intérêt est qu'il permet de paramétrer un composant sur un ou plusieurs types de donnée, ce qui offre une bonne abstraction des données manipulées par le composant. Mais l'héritage permet lui aussi ce genre d'abstraction. Si l'on reprend l'exemple de la figure 7.18, il est possible de créer une classe unique `Groupe` qui agrège des objets de la classe abstraite `Affichable` spécialisée par les classes `Ligne` et `Rectangle` (cf. figure 7.20). La même abstraction des objets agrégés est alors fournie. Mais quel est donc le réel avantage du patron ?

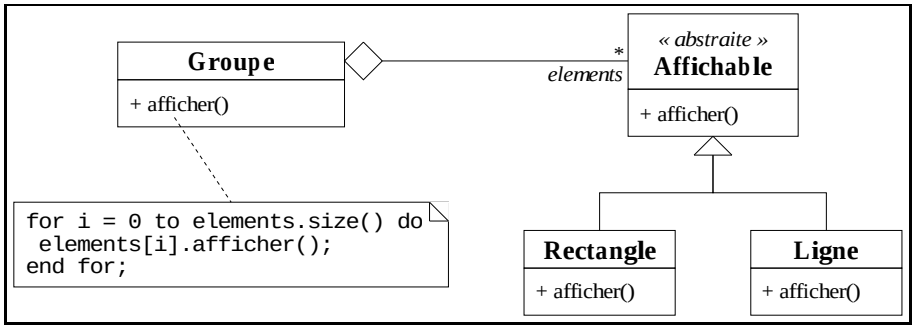


Figure 7.20: Un autre exemple d'héritage.

Principalement, il évite d'employer la virtualité, ce qui permet un gain significatif sur le temps d'exécution (surtout si des méthodes deviennent *inline*). En revanche, les types des données qu'il manipule étant fixés à l'instanciation, le patron est beaucoup moins flexible. Dans notre exemple, il ne permet de stocker que des

éléments d'un même type. Certains diront que cela constitue un avantage (en effet, si l'on veut s'assurer que l'on a un groupe d'objets de la même classe), d'autres diront que c'est un défaut (en effet, si l'on veut traiter en une même commande un ensemble d'objets hétérogène). Choisir entre l'héritage et le patron dépend donc fortement de l'utilisation que l'on souhaite faire des composants. Dans un système dynamique, on choisira plutôt l'héritage alors que dans un système statique, on préférera le patron qui offre une efficacité bien meilleure.

Une différence notable au niveau conceptuel réside dans le fait que les objets manipulés dans l'approche par héritage ont un lien de parenté, ils héritent tous de la même super-classe `Affichable` alors que dans l'approche par patron, aucun lien de parenté n'est présent, les objets doivent juste implémenter (souvent implicitement) le concept `Affichable`. Cela permet pour l'approche par patron de simplifier énormément la hiérarchie de l'héritage et d'éviter des dépendances a priori inutiles entre les composants.

Mais cette différence conceptuelle a également un impact important sur la réutilisabilité. En effet, un patron est totalement indépendant des types avec lesquels il est instancié. Cela signifie que si l'on récupère une classe dont on n'est pas l'auteur, mais qui implémente le concept nécessaire au patron, alors il sera possible d'instancier le patron avec cette classe. Supposons la même chose avec l'approche par héritage. Il est impossible de profiter de la classe `Groupe` pour une classe dont on n'est pas l'auteur, puisqu'elle n'hérite pas de `Affichable`. Deux solutions sont alors possibles: modifier la classe pour qu'elle hérite de `Affichable`, mais il faut alors mesurer la portée du changement; ou utiliser la délégation pour agréger la classe étrangère dans une nouvelle classe héritant de `Affichable`.

Une autre différence conceptuelle du patron est qu'il manipule indifféremment les classes et les types primitifs (e.g. entiers, réels). Ses paramètres sont simplement des types qui doivent répondre à certains messages. L'avantage du patron, si l'on reprend l'exemple de la figure 7.16, est que la classe `Liste` peut stocker aussi bien des types primitifs que des classes (e.g. `Dessin`...), alors que la même classe avec l'approche par héritage ne pourrait stocker que des objets d'une même classe de base, et malheureusement la plupart des langages (e.g. Java et C++) ne considère pas les types primitifs comme des classes.

Après réflexion, le patron s'avère être une manière toute particulière de réutiliser, avec une approche *boîte noire* et une abstraction sans distinction des types primitifs et des classes, il permet un découplage fort entre les composants. Contrairement à la composition, l'utilisation en *boîte noire* s'effectue sans aucun héritage, ce qui est un facteur indéniable d'efficacité. En revanche, le patron ne fournit qu'une structure statique et le polymorphisme dynamique fourni par l'héritage ou la délégation reste une nécessité pour beaucoup de systèmes.

7.5. Conclusion

La notion d'objet est un concept simple lié à notre perception de la plupart des systèmes réels. Cependant, tout au long de ce chapitre, nous avons découvert que concevoir une représentation orientée objet dans l'objectif d'une implémentation informatique s'avère être une tâche délicate. Les notions mises en oeuvre possèdent des nuances et des impacts sur l'implémentation dont on voudrait bien faire abstraction, mais ce faisant, cela pourrait conduire à un logiciel désastreux en termes de qualité et de réutilisabilité. Aux vues des problèmes introduits par les différents concepts de l'orientation objet, on s'aperçoit que pour des applications dites *scientifiques* où le temps d'exécution est critique, il est tout à fait impossible de faire abstraction des problématiques "bas niveau" que la programmation orientée objet induit. Ainsi, avec une bonne connaissance des mécanismes présentés dans ce chapitre, il est possible d'éviter les pièges classiques d'une modélisation orientée objet trop

conceptuelle et d'aboutir à des composants tout à fait réutilisables et efficaces.

De l'orientation objet, nous pouvons retenir trois concepts majeurs permettant la réutilisabilité: l'héritage, la composition et le patron. Dans leurs apports à la réutilisabilité, il faut faire une distinction entre l'abstraction des composants qui se traduit par le polymorphisme, et l'extension d'un composant qui permet de lui ajouter de nouvelles fonctionnalités. L'héritage intervient au niveau de l'abstraction des composants, grâce à un polymorphisme coûteux, et au niveau de l'extensibilité, dont la mise en oeuvre est discutable à cause des risques latents de rupture d'encapsulation. La composition intervient au niveau de l'extension, par l'assemblage de composants. Enfin, le patron intervient au niveau de l'abstraction des composants, plus forte que celle de l'héritage, grâce à un polymorphisme sans aucun coût. L'abstraction fournie par l'héritage est dynamique alors que celle fournie par le patron est statique. L'extension fournie par l'héritage est statique alors que celle fournie par l'agrégation est dynamique. D'ailleurs, une combinaison judicieuse de l'héritage et de la composition produit le concept de délégation qui permet une abstraction dynamique et une extension dynamique.

Ainsi, en fonction du dynamisme et de l'efficacité que l'on souhaite pour un logiciel, on cherchera la meilleure combinaison des concepts d'héritage, de composition et de patron. Dans le cadre du développement d'une bibliothèque de composants réutilisables pour la recherche opérationnelle, il semble logique d'éviter le polymorphisme dynamique et de concentrer la conception plutôt sur le polymorphisme statique. Il est bien évident que cela n'exclut en aucun cas l'héritage de la conception. En évitant ce polymorphisme dynamique très coûteux, une approche dite ***programmation générique*** est apparue (cf. [Muss89]). Elle consiste à favoriser le concept de patron à la place de l'héritage dans la conception de composants *génériques*. Cette notion de généricité n'est pas limitée à la seule utilisation de patrons, mais à l'idée qu'un composant doit être le plus général possible sans que cela n'ait un impact négatif significatif sur son efficacité. Plusieurs travaux dont nous aurons l'occasion de discuter au chapitre suivant ont abouti dans ce sens.

Nous pouvons dès maintenant exclure le langage Java de nos possibilités, puisqu'il n'offre pas pour l'instant la notion de patron. C++ sera donc notre langage pour l'implémentation. Pour conclure, nous citons deux travaux élaborés avec ce langage et qui semblent tout à fait répondre aux critères de qualité, d'efficacité et de réutilisabilité. Ils offrent un espoir dans notre tentative de développer des composants réutilisables pour la recherche opérationnelle, à laquelle nous tentons d'apporter des éléments de réflexion au chapitre suivant.

Le premier travail est le désormais célèbre cadriciel STL (*Standard Template Library* [SGIWb]) qui offre des structures de données et des algorithmes associés tout à fait efficaces et réutilisables. Le second travail est présenté dans [Hane99], il propose une classe représentant une abstraction des tableaux multidimensionnels. Les algorithmes manipulent cette classe sans connaître la représentation interne du tableau. Ils sont cependant très efficaces avec des performances similaires à celles d'une conception dédiée. Cet exemple est fort intéressant pour le calcul scientifique, puisqu'il règle le dilemme d'utiliser le langage Fortran avec sa représentation en colonne des tableaux, ou le langage C/C++ avec sa représentation en ligne.

CHAPITRE 8
UNE EXPÉRIENCE DE RÉUTILISABILITÉ POUR LES PROBLÈMES DE GRAPHES

Dans le début de cette partie nous avons tenté de montrer l'intérêt de développer des composants réutilisables pour la recherche opérationnelle et d'expliquer les possibilités, avec leurs avantages et leurs défauts, offertes par la programmation orientée objet choisie pour notre conception. De ces constats et de notre propre expérience du développement d'une bibliothèque réutilisable dans le cadre de notre étude des problèmes de synchronisation hypermédia, nous présentons dans ce dernier chapitre quelques réflexions concernant différentes solutions de conception.

L'enjeux étant important, quelques tentatives ont naturellement déjà vu le jour, avec plus ou moins de succès. Une liste relativement complète peut être trouvée dans [Maco97]. Dans notre discussion, nous ne retenons qu'un échantillon de bibliothèques C++ dont la réutilisabilité est avérée, puisqu'elles sont employées dans différents projets: LEDA (*Library of Efficient Data types and Algorithms* [ASWb], débutée en 1988), BGL (*Boost Graph Library* [SiekWb], débutée en 1998) et GTL (*Graph Template Library* [ForsWb], débutée en 1999). Chacune est le fruit de choix de conception différents liés aux objectifs auxquels elle a été assignée.

Il faut alors préciser que notre bibliothèque, la *B++ Library* [BachWb], débutée en 1999, a pour but de satisfaire à la fois des néophytes en matière de recherche opérationnelle (i.e. une réutilisation *boîte noire*), et des personnes averties (i.e. une réutilisation *boîte blanche*). Dans la première situation, il s'agit de permettre à une personne capable de modéliser un problème sous la forme d'un graphe d'implémenter sa résolution aisément, sans connaissances particulières de la recherche opérationnelle. Dans la seconde situation, notre volonté est de permettre un prototypage rapide d'une méthode de résolution. Nous espérons également, à travers un développement homogène, favoriser la comparaison de l'efficacité pratique de différents algorithmes.

La première partie de ce chapitre présente diverses manières de concevoir des composants ***génériques***, c'est-à-dire les plus indépendants et les plus extensibles possibles. Nous nous intéressons en particulier aux structures de données et aux algorithmes. Dans un premier temps, nous discutons de la structure d'un graphe et des possibilités de la rendre paramétrable sur les données portées par les noeuds et les arcs. Nous présentons ensuite différentes manières d'abstraire une structure de données dans un algorithme. Enfin, nous expliquons comment permettre l'extensibilité d'un algorithme. Nous terminons cette discussion sur un problème qui semble plus secondaire, mais qui est en fait très courant: la gestion de données additionnelles sur un graphe nécessaires à la plupart des algorithmes pour leur exécution. Pour tous ces problèmes de conception, nous ne proposons en aucun cas de solution absolue, mais tentons d'expliquer au mieux les forces et les faiblesses de chacune des principales possibilités.

Dans une seconde partie, nous présentons succinctement notre bibliothèque. Il s'agit de la conclusion de notre discussion sur la conception de composants réutilisables pour la recherche opérationnelle. Nous exposons ainsi l'état d'avancement de notre bibliothèque en proposant un aperçu rapide des principales fonctionnalités liées à la réutilisabilité que nous n'avons pas traitées jusqu'à présent. Ces points concernent très certainement des aspects plus pratiques de la réutilisabilité, notamment la portabilité, qu'il semble néanmoins important de souligner pour les personnes désireuses de se lancer dans une telle entreprise. Cette présentation permet de situer notre bibliothèque dans l'objectif à plus long terme de concevoir un environnement de développement pour les problèmes de graphes, dont nous proposons en fin de discussion une rapide ébauche.

8.1. Généricité des structures de données

Penchons-nous tout d'abord sur les différentes façons de modéliser la structure d'un graphe. La figure 8.1 montre de manière très générale comment représenter en orientation objet cette structure. Nous considérons ici que le graphe est composé (au sens objet) d'arcs et de noeuds, il est donc chargé de les gérer entièrement. Nous ne nous intéressons pas à la représentation même du graphe, c'est-à-dire si le graphe est représenté par une matrice d'adjacence, une matrice d'incidence, une liste d'adjacence (cf. chapitre 2)... Notre propos est justement de discuter d'une manière qui permet d'abstraire de tels détails pour les algorithmes. Ces derniers doivent être applicables indépendamment de la structure véritable du graphe, et ce avec le maximum d'efficacité.

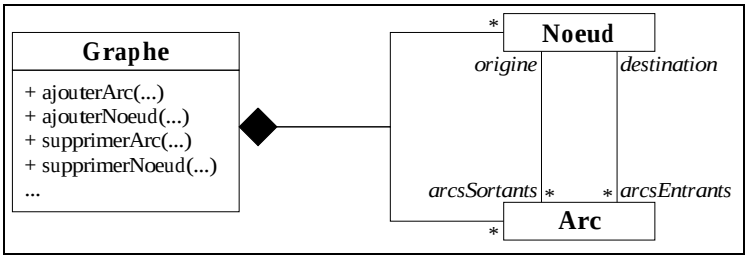


Figure 8.1: Représentation générique d'un graphe.

Dans un premier temps, nous nous intéressons à l'extensibilité de la structure. Il est en effet indispensable que l'on puisse créer un graphe et le manipuler avec n'importe quelles données sur ses arcs et ses noeuds. Par exemple, si l'on traite un problème de plus court chemin, le graphe portera des informations géographiques (distances, coordonnées...), alors que pour un problème de flot il portera plutôt des informations de flux (capacités, demandes...). L'utilisateur ne doit coder que ces données additionnelles, et ne doit en aucun cas avoir à toucher à la structure même du graphe. Dans cette optique, nous proposons deux manières de représenter la structure d'un graphe: une approche par héritage et une approche par patrons. Une fois que nous aurons jugé de la meilleure modélisation pour nos besoins, nous rappelons une technique très connue pour masquer la structure même du graphe (matrice, liste...), qui est la première étape dans l'indépendance des algorithmes par rapport aux structures de données qu'ils manipulent.

8.1.1. Modélisation d'un graphe par héritage

La première manière de représenter un graphe extensible consiste à reprendre la structure générique présentée à la figure 8.1, en ajoutant aux arcs et aux noeuds un attribut supplémentaire qui représente les données, outre les informations structurelles du graphe, qu'ils peuvent porter (cf. figure 8.2). Pour les arcs, cet attribut serait de la classe `DonnéeArc` et pour les noeuds de la classe `DonnéeNoeud`. L'unique rôle de ces deux classes est d'être étendues, c'est-à-dire que l'utilisateur puisse définir ses propres données en héritant de ces deux classes. La manipulation est donc très simple, l'utilisateur a seulement à définir une classe qui hérite de `DonnéeArc` ou `DonnéeNoeud`, possédant les attributs représentant les données qu'il souhaite attribuer aux arcs ou aux noeuds du graphe.

Dans notre exemple, nous avons ajouté sur les arcs des données de flot et sur les noeuds un potentiel. On pourrait penser que le défaut majeur de cette structure est l'inefficacité liée à l'héritage. Cependant le mécanisme de virtualité n'est pas utilisé ici, tout du moins pour l'accès aux données ajoutées (en effet, il est possible par exemple d'avoir une méthode virtuelle qui se charge d'afficher les données contenues dans un objet `DonnéeArc`). Le défaut de cette approche se trouve néanmoins dans l'accès aux données de flot d'un arc.

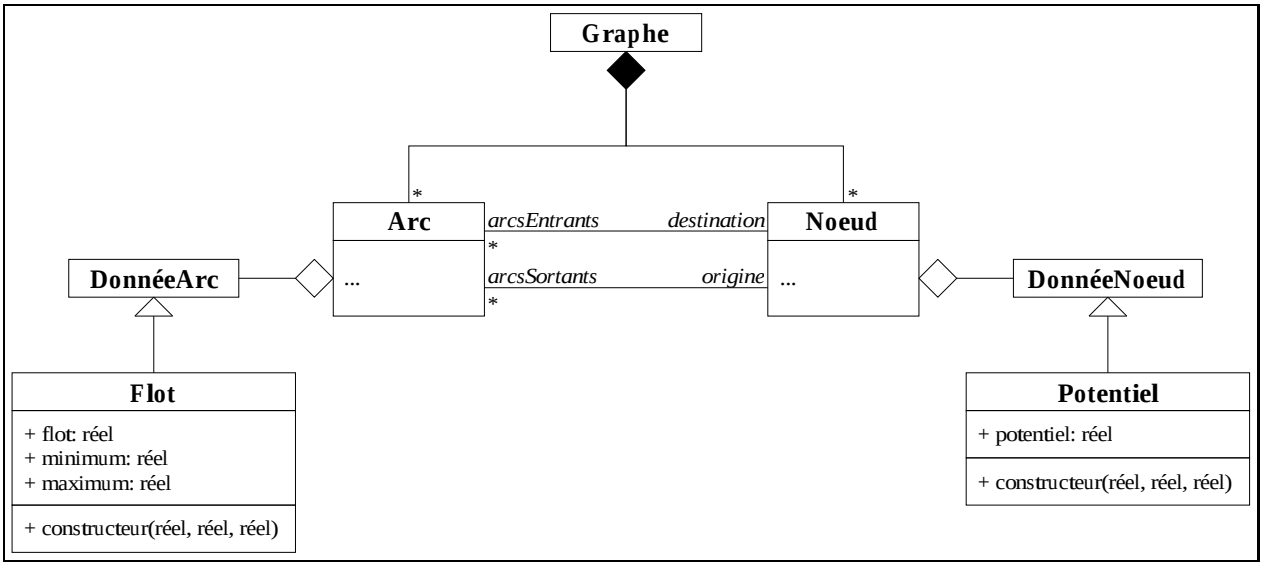


Figure 8.2: Représentation d'un graphe avec une approche par héritage.

En effet, l'arc est de type `Arc`, son attribut `donnée` est donc de type `DonnéeArc`. Mais cette classe ne possède pas, par exemple, l'attribut `flot`. Il faut alors transformer `donnée` en un objet de type `Flot`. Autant la conversion d'une sous-classe vers l'une de ses super-classes (appelée souvent ***upcast***) est instantanée (puisque l'objet appartient forcément à la classe dans laquelle on veut le transformer), autant la conversion d'une super-classe vers l'une de ses sous-classes (appelée souvent ***downcast***) demande vérification. Les langages permettent naturellement le *downcast*, mais sa vérification ne peut être effectuée qu'à l'exécution. Cela signifie donc une perte de temps à chaque accès aux données, même si le programmeur est sûr de la conversion. L'exemple suivant tente d'illustrer la problématique exposée ici.

```
Graphe *    g = new Graphe(...);
Flot *      f = new Flot(...);
DonnéeArc * d = new DonnéeArc(...);

g.ajouterNoeud(1);
g.ajouterNoeud(2);
...
g.ajouterArc(1,2,f);
g.ajouterArc(2,5,d);
...
(Flot)(g.getArc(1,2)).flot() = 8;
(Flot)(g.getArc(2,5)).flot() = 3;
```

Un graphe `g` est créé avec notamment l'arc $(1;2)$ qui porte des données de flot et l'arc $(2;5)$ qui ne porte aucune donnée additionnelle. Les deux dernières lignes de l'exemple tentent d'affecter un réel au flot des deux arcs cités précédemment, en effectuant une conversion (représentée par `(Flot)`). Pour l'arc $(1;2)$ l'action réussit, alors que pour $(2;5)$ elle échoue naturellement. Et cette erreur ne peut être vue qu'à l'exécution du programme.

Mais la virtualité, et surtout l'inefficacité qu'elle peut engendrer, ne sont pas pour autant exclues de cette modélisation. Avec cette approche, un algorithme est dépendant des types des données portées par le graphe. En effet, dans l'exemple précédent, le code manipule explicitement des objets de la classe `Flot`. Si l'on souhaite utiliser ce code avec une classe différente, en voulant par exemple ajouter une tension sur l'arc, on est

obligé d'étendre la classe `Flot` par héritage. Tant que l'on ne tente pas de surcharger une méthode de cette classe, aucun coût supplémentaire n'est à noter. En revanche, si le flot qui était un simple attribut devient une valeur calculée, alors il faut que la méthode qui retourne le flot dans la classe `Flot` soit virtuelle pour qu'une sous-classe puisse la surcharger. L'accès au flot entraîne alors un appel au mécanisme de virtualité dont nous connaissons les problèmes.

En résumé, la modélisation du graphe avec une approche par héritage entraîne une perte de temps due au *down-cast*, certes plus faible que la virtualité. Cependant, en observant de plus près les algorithmes qui manipulent cette structure, on s'aperçoit que les méthodes d'accès aux données des noeuds et des arcs doivent être le plus souvent virtuelles afin de permettre une plus grande réutilisabilité des algorithmes. Tous les défauts que nous avons énumérés ici portent sur l'efficacité de la modélisation. Il faut tout de même noter que l'effort à fournir ici pour décrire les données portées par le graphe est minimal et que cette représentation permet, contrairement à celle qui suit, de stocker des données de types variés sur les arcs et les noeuds du graphe, ce qui offre une certaine souplesse qui peut être appréciable, non pas pour la recherche opérationnelle, mais pour la simulation par exemple (cf. [Bach98]).

8.1.2. Modélisation d'un graphe par patrons

Nous cherchons donc une modélisation d'un graphe qui permette d'ajouter des données sur les arcs et les noeuds tout en garantissant une indépendance des algorithmes par rapport au type de ces données. Nous proposons une représentation sous forme de patrons où les paramètres sont les classes des données portées par les arcs et les noeuds, notées respectivement `TA` et `TN` sur la figure 8.3. Comme dans la modélisation par héritage, les arcs et les noeuds possèdent un attribut `donnée` qui représente les données qu'ils portent.

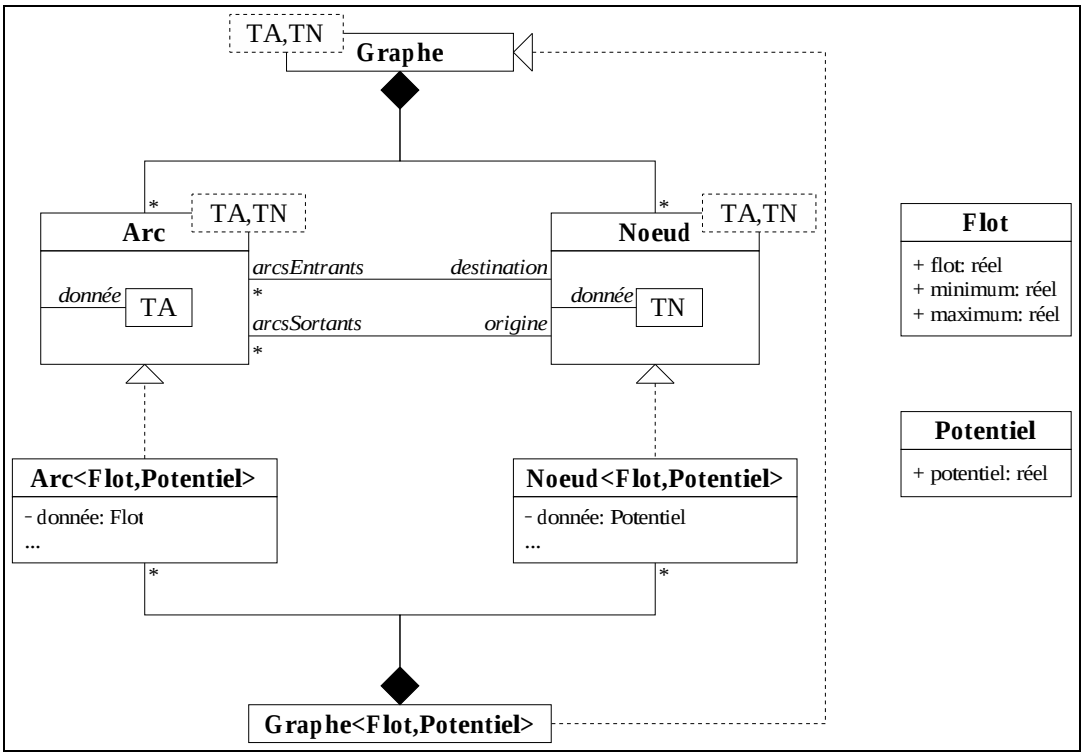


Figure 8.3: Représentation d'un graphe avec une approche par patrons.

Pour créer un graphe avec les données qu'il souhaite, un utilisateur doit simplement créer les classes représentant ces données, dans notre exemple `Flot` et `Potentiel`, et ensuite instancier le graphe avec ces deux classes. L'avantage de cette structure est celui de tout patron, aucun coût n'est ajouté par rapport à l'opération manuelle qui consisterait à répliquer et à modifier la structure de graphe pour chaque type de données.

En ce qui concerne les algorithmes, il est tout à fait possible comme nous le verrons par la suite d'écrire un patron d'algorithme paramétré sur les types des données portées par le graphe, le patron supposant implicitement que les types implémentent certains concepts. Ces derniers seront vérifiés au moment de la compilation. Nous avons ainsi l'indépendance minimale que nous recherchions depuis le début, i.e. un algorithme conçu pour certains types de données sur les arcs et les noeuds peut fonctionner pour d'autres types. Nous allons nous intéresser dans la dernière partie de cette section à renforcer l'indépendance de l'algorithme par rapport aux structures de données qu'il manipule, en particulier celle du graphe.

Mais avant cela, revenons rapidement sur les bibliothèques que nous avons présentées en début de chapitre. LEDA, comme BGL et notre bibliothèque, ont opté pour une modélisation par patrons avec certes des différences, mais avec la même idée principale d'abstraction. En revanche, GTL propose une représentation proche de celle par héritage, mais ne préconise même pas l'héritage pour attribuer des données aux arcs et aux noeuds, et propose plutôt de construire des listes à part de la structure de graphe, supposant que les noeuds et les arcs sont ordonnés (ce qui empêche tout modification structurelle du graphe). Les raisons qui font que GTL est réutilisable correspondent donc à un type d'utilisation différent de celui qui nous intéresse ici.

8.1.3. Abstraction par les itérateurs

Dans beaucoup de traitements, les structures de données qui sont manipulées sont des collections d'objets (liste, tableau, pile, file d'attente, arbre...). En particulier le graphe possède deux collections, l'une d'arcs et l'autre de noeuds. Nous discutons ici d'une manière d'abstraire la structure même de ces collections, afin que les traitements sur celles-ci soient indépendants de la structure de données effectivement manipulée. Les approches présentées ici peuvent être étendues à l'abstraction de tout type de structure de données.

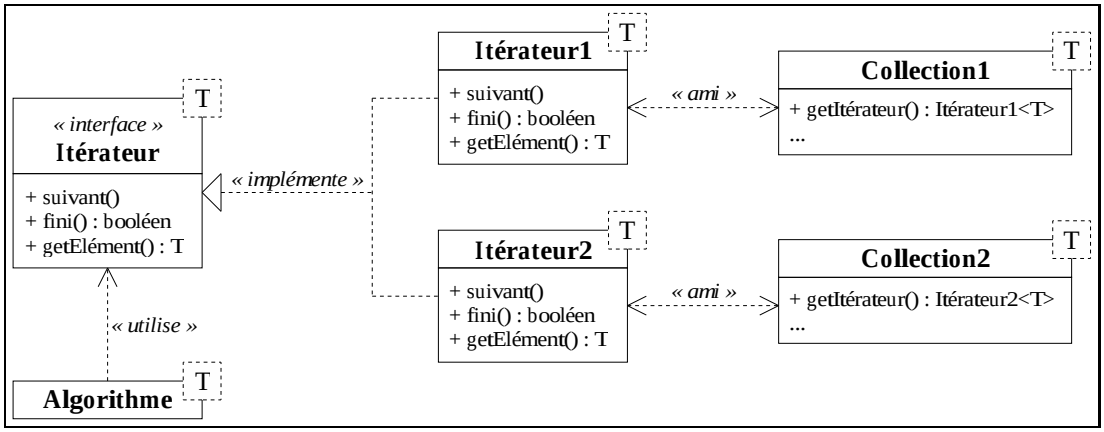


Figure 8.4: Un exemple d'itérateur.

Dans notre discussion, nous considérerons l'exemple d'une procédure qui parcours tous les éléments d'une collection et leur applique un traitement, et nous nous intéresserons à rendre son code unique pour tout type de collection. Le principe consiste à intercaler, pour chaque collection, une classe entre la collection et les algorithmes qui la manipulent. Cette classe possède la même interface quelque soit la collection. En l'utilisant

de préférence aux méthodes propres à la collection, les algorithmes deviennent plus indépendants de la structure qu'ils manipulent et les collections plus facilement interchangeables. L'interface qui convient à notre exemple est un ***itérateur***, il s'agit d'un patron de conception proposé dans [Gamm95] et exploité intensivement dans la bibliothèque STL.

L'interface d'un itérateur propose la fonctionnalité de déplacement dans une collection triée (implicitement ou explicitement). La figure 8.4 montre le principe d'un itérateur qui offre les méthodes pour se déplacer dans un sens arbitraire d'un élément à un autre d'une collection. Dans notre exemple, nous disposons de deux collections qui peuvent créer un ou plusieurs itérateurs (grâce à leur méthode `getItérateur()`) sur leur propre structure.

Il est à noter qu'une collection doit accéder à des attributs normalement cachés pour initialiser un itérateur et inversement un itérateur doit accéder à la structure cachée de sa collection. Les deux classes doivent donc être mutuellement ***amies***, cela signifie que chacune autorise explicitement l'autre à accéder à ses données cachées (tous les langages n'implémentent pas cette fonctionnalité, en C++ il existe le mot-clé `friend` qu'une classe peut utiliser pour en autoriser une autre à accéder à ses données cachées, en Java cette fonctionnalité n'est pas directement implémentée, mais par le mécanisme de *classe interne*, il est possible que certaines classes voient des choses normalement cachées pour d'autres). Nous présentons maintenant différentes manières d'utiliser les itérateurs qui traduisent différents niveaux d'abstraction d'une collection pour un algorithme.

8.1.3.1. Abstraction par les itérateurs

Le premier niveau d'abstraction consiste, même si l'on connaît précisément une structure, par exemple un objet de la classe `Collection1`, à utiliser ses itérateurs pour le manipuler, en évitant les méthodes propres à la structure. Comme le montre l'exemple suivant, il est ainsi possible, même dans un code a priori non réutilisable, de modifier la structure de données sans modifier aucune autre ligne de code.

```
Collection1<T> c = ...;
Itérateur1<T>  i = g.getItérateur();

while not i.fini() do
 ...i.getElément()...
 ...
 i.suivant();
end while;
```

Si l'on remplace les classes `Collection1` et `Iterateur1` par les classes `Collection2` et `Itérateur2`, aucune autre ligne n'a besoin d'être modifiée pour que le code fonctionne encore. Cette première abstraction d'une structure de données est intéressante mais insuffisante puisqu'elle nécessite une intervention manuelle, néanmoins il est toujours conseillé d'employer cette approche qui ne coûte absolument rien en efficacité, ni même en temps de développement et qui, à défaut d'une meilleure approche, apporte une certaine maintenabilité au code.

8.1.3.2. Abstraction par paramétrage des itérateurs

Le second niveau d'abstraction consiste à paramétrer un algorithme sur les classes des itérateurs qu'il manipule. En reprenant l'exemple précédent, il est possible de proposer un algorithme paramétré sur le type de l'itérateur utilisé, le paramètre `I` dans l'exemple qui suit. L'algorithme n'aura aucune connaissance de la structure qu'il manipule et n'utilisera que les itérateurs qui lui sont fournis en argument.

```
function algo<I>(I * i)
 while not i.fini() do
  ...i.getElément()...
  ...
  i.suivant();
 end while;
end function;
```

L'algorithme nécessite alors simplement un paramètre qui implémente le concept d'itérateur pour fonctionner. L'inconvénient de cette approche est que l'utilisateur de l'algorithme connaît des détails du fonctionnement interne de la fonction (dans notre cas, que l'algorithme parcours un à un les éléments), ce qui peut rompre le concept d'encapsulation. En outre, si l'algorithme a besoin de beaucoup d'itérateurs, l'utilisateur doit les fournir, sans forcément en comprendre les raisons. Cette approche a été celle choisie par la STL, mais il faut reconnaître que les inconvénients que nous venons de citer ne s'y appliquent pas, puisque le transfert d'itérateurs aux algorithmes est justifié.

8.1.3.3. Abstraction par paramétrage de la structure de données

Aux vues des inconvénients des approches précédentes, un troisième niveau d'abstraction a été proposé notamment dans [Lee99], qui a suivi les premiers travaux de [Kuhl96] (utilisant plutôt le second niveau d'abstraction), et a débuté la conception de la bibliothèque BGL.

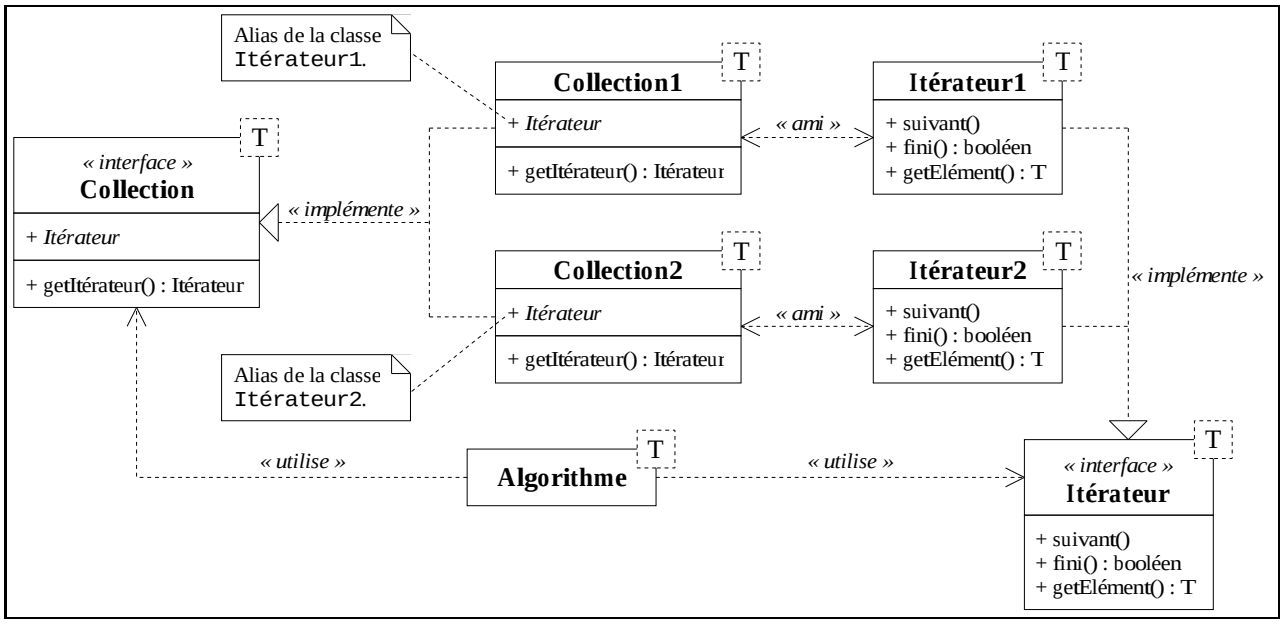


Figure 8.5: Un exemple d'abstraction d'une collection.

Au lieu de paramétrer l'algorithme sur les itérateurs, pourquoi ne pas le paramétrer sur la structure de données même qui doit être manipulée. La structure doit alors implémenter un concept qui permet la création d'itérateurs pour la parcourir. La figure 8.5 montre l'interface que doit posséder une collection. Mais il faut tout de même que l'algorithme connaisse le type de l'itérateur qu'il manipule. Comme on ne souhaite pas fournir explicitement d'itérateur en paramètre à l'algorithme, il ne peut y avoir que la collection qui possède ce renseignement. Pour cela, il est possible de définir des types internes à une classe qui sont accessibles comme toute autre propriété de la classe.

En reprenant notre exemple, le type interne `Itérateur` (représenté en italique sur la figure) est défini dans l'interface `Collection`, il s'agit en fait d'un alias sur la véritable classe de l'itérateur de la collection. Dans la classe `Collection1`, il s'agit donc du type `Itérateur1`. L'exemple suivant illustre cette manière d'abstraire la structure de données. Cette approche fournit une abstraction complète, les collections sont simplement obligées d'implémenter un certain concept relatif au mécanisme des itérateurs.

```
function algo<C>(C * c)
 c.Itérateur i = c.getItérateur();

 while not i.fini() do
  ...i.getElément()...
  ...
  i.suivant();
 end while;
end function;
```

Le seul véritable défaut de cette approche (également présent dans le deuxième niveau d'abstraction) se situe au niveau pratique. Il est en effet très difficile de déboguer un patron d'algorithme élaboré avec ce type de paramètre. La raison en est très simple. Lorsque le patron est analysé à la compilation, avant toute instanciation, le type `C`, si l'on revient à l'exemple précédent, est inconnu. Il est donc impossible de vérifier que le type `C.Iterateur` (ou le type `I` pour l'exemple du deuxième niveau d'abstraction) existe et surtout implémente le concept d'itérateur. Il faut donc attendre une instanciation du patron pour démarrer une véritable vérification du code. Ceci est très problématique lorsque l'on possède de nombreux modules à compiler et que le patron se trouve tôt dans la compilation alors que son instanciation s'effectue très tard. La phase de débogage, par la longueur des tentatives de compilation, devient rapidement presque impossible. En revanche, en tant qu'utilisateur, cette dernière approche présente la meilleure alternative, puisqu'elle offre le plus d'indépendance entre un algorithme et les structures de données qu'il manipule.

8.1.3.4. Un compromis d'abstraction

Comme précisé précédemment, la bibliothèque BGL propose cette dernière approche qui satisfait pleinement les utilisateurs, mais rend la tâche de la conception de la bibliothèque difficile. LEDA ne semble pas poursuivre tout à fait le même objectif de rendre les structures de données indépendantes des algorithmes. Sa manière de concevoir les composants se rapproche de notre choix, puisque nous proposons une abstraction intermédiaire qui tente de satisfaire à la fois les concepteurs (qui sont aussi réutilisateurs) et les utilisateurs finaux. Considérons l'exemple suivant qui illustre notre approche.

```
function algo<TA,TN>(Graphe<TA,TN> * g)
 g.ItérateurArc i = g.getItérateurArc();

 while not i.fini() do
  ...i.getElément()...
  ...
  i.suivant();
 end while;
end function;
```

Il est possible de paramétrer les algorithmes sur les types des données portées par les noeuds et les arcs, et dans l'algorithme d'appliquer le tout premier niveau d'abstraction, c'est-à-dire manipuler directement la classe `Graphe` pour récupérer, à la manière du troisième niveau d'abstraction, les classes des itérateurs. L'algorithme

n'est alors pas vraiment indépendant de la structure de graphe, puisqu'il manipulera toujours la classe `Graphe`. Cependant, toutes les manipulations se font par les itérateurs. Ainsi, un utilisateur qui désire changer de structure de graphe peut le faire sans avoir à modifier autre chose. Mais cela a un impact sur toute la bibliothèque, puisqu'il ne peut exister qu'une seule classe `Graphe`.

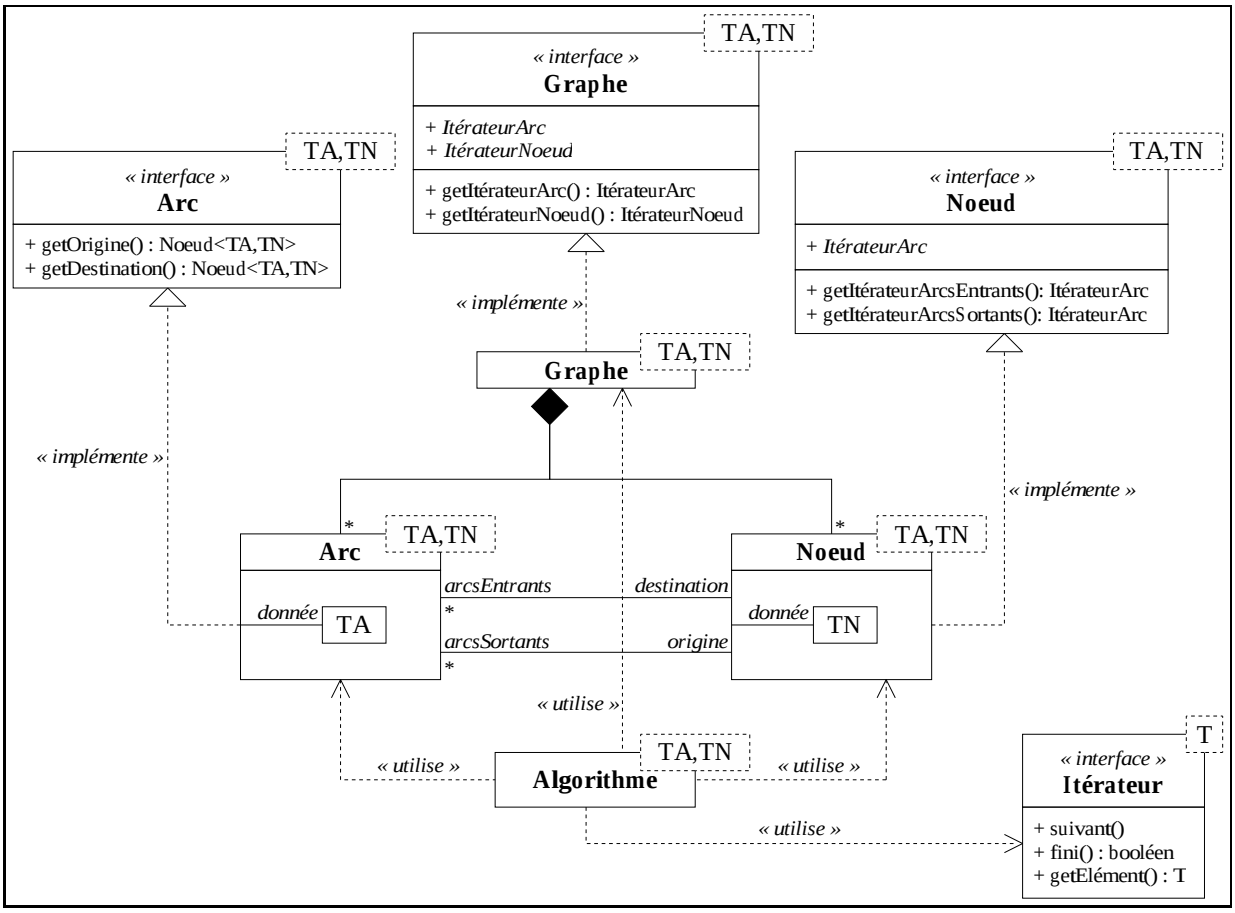


Figure 8.6: L'abstraction d'un graphe.

Notre bibliothèque est ainsi nettement moins flexible, mais lorsque nous nous penchons sur la manière de réutiliser les algorithmes, on s'aperçoit qu'il est très délicat de manipuler plusieurs structures de graphe. Imaginons que l'on soit en train d'élaborer une nouvelle méthode, basée sur des algorithmes existants. Si l'un des algorithmes fonctionne efficacement avec une structure de graphe et un autre avec une structure totalement différente, quelle structure choisir pour notre méthode ? Celle de l'un des deux algorithmes existants ou une structure qui sera un compromis pour les deux algorithmes ? L'utilisateur se trouve alors confronté à un dilemme pour lequel il ne possède pas tous les détails; et d'ailleurs dans un souci d'encapsulation, il ne doit pas. Il connaît tout au plus quelle structure est la mieux adaptée à un algorithme.

Ainsi, pour l'utilisation que nous envisageons de notre bibliothèque, un développement rapide de prototypes et une vitesse raisonnable des algorithmes pour un produit fini, nous avons choisi de fixer la structure de données du graphe. Nous détaillons légèrement en annexe notre choix. La figure 8.6 résume finalement le modèle que nous avons retenu. Certes il n'est pas le meilleur mais sans une évolution notable des compilateurs C++ sur la vérification des patrons, il nous semble peu envisageable de maintenir une bibliothèque comme la nôtre avec une abstraction totale de la structure d'un graphe.

8.2. Généricité des algorithmes

Nous abordons ici la généricité même des algorithmes. Les manières de rendre un algorithme indépendant des structures de données qu'il manipule ont été discutées à la section précédente. Nous présentons maintenant une façon de le concevoir plus indépendant des sous-algorithmes qu'il emploie. Ensuite, nous discutons de différentes approches pour le rendre paramétrable, afin qu'il soit extensible ultérieurement. Pour des raisons de clarté, les figures de cette section ne présentent pas les algorithmes sous forme de patrons comme il l'a été décidé à la section précédente, nous considérons implicitement qu'ils sont paramétrés sur les données portées par les noeuds et les arcs.

8.2.1. Abstraction des algorithmes

Il existe un patron de conception très connu, appelé ***stratégie*** (cf. [Gamm95]), qui consiste à modéliser un algorithme sous la forme d'un objet possédant une méthode, `run()` par exemple, appelée pour exécuter l'algorithme. Il est alors possible de proposer une classe abstraite pour représenter une catégorie d'algorithmes (e.g. les algorithmes pour résoudre le problème de la tension de coût minimale, cf. chapitre 4).

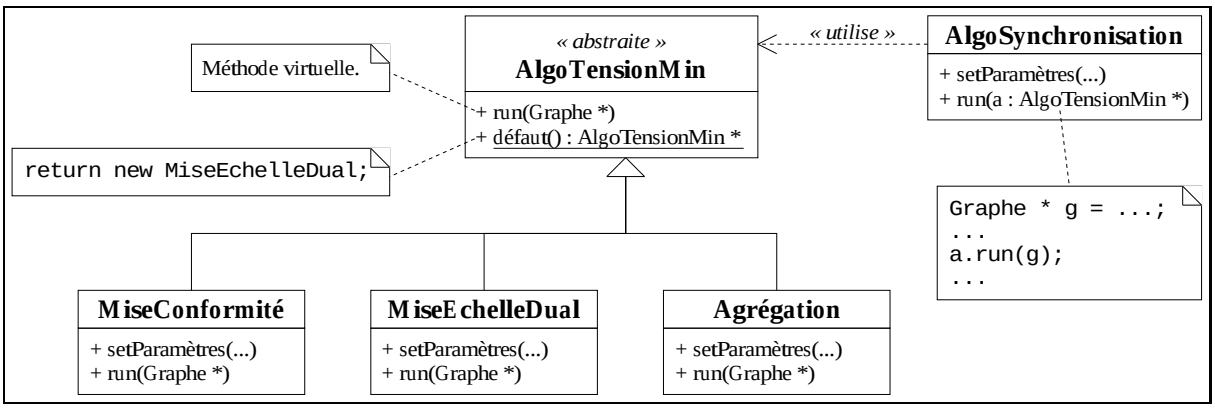


Figure 8.7: Un exemple d'abstraction d'une catégorie d'algorithmes.

Comme le montre la figure 8.7, cette classe abstraite fournit une interface commune à tous les algorithmes, ce qui les rend totalement interchangeables lorsqu'ils sont ensuite utilisés dans un autre algorithme, e.g. la classe `AlgoSynchronisation` sur la figure. La virtualité impliquée ici est négligeable dans la plupart des cas, puisque les méthodes appelées ainsi sont des algorithmes complets, i.e. avec un nombre d'opérations nettement supérieur à celui induit par la virtualité. En outre, l'interface étant commune à tous, elle ne peut pas être utilisée pour fournir des paramètres spécifiques à un algorithme. Au patron stratégie, il faut donc ajouter pour chacun une méthode propre, e.g. `setParamètres()`, dont le rôle est d'initialiser les paramètres de l'algorithme.

```
AlgoTensionMin      * a = new MiseConformité;
AlgoSynchronization * s = new AlgoSynchronisation;

s.setParamètres(...);
a.setParamètres(...);
s.run(a);
```

L'exemple précédent montre qu'il est ainsi possible de décider de l'algorithme de tension de coût minimal à employer pendant l'exécution du programme, de le créer et de le paramétrer avant de le fournir à l'algorithme

`AlgoSynchronisation`. Il faut noter que le rôle des méthodes `setParamètres()` peut être joué par les constructeurs des algorithmes. Ainsi, au moment de leur création ces derniers sont systématiquement paramétrés.

Il semble également important, lorsque l'on dispose de plusieurs méthodes pour résoudre un même problème, d'en fournir une par défaut, afin d'éviter que l'utilisateur ne se perde dans des détails inutiles sur les performances des algorithmes. C'est le rôle de la méthode de classe `défaut()` de `AlgoTensionMin`, qui fournit a priori l'algorithme reconnu le plus performant. Mais il est tout à fait possible d'envisager une approche plus évoluée où `défaut()` reçoit le graphe à traiter en paramètre, et à partir d'une analyse (e.g. la mesure de la densité du graphe) propose l'algorithme le mieux adapté. Cette méthode de classe est également importante pour l'évolution des programmes des utilisateurs. En l'utilisant un composant bénéficie automatiquement des progrès apportés par l'intégration d'une nouvelle méthode plus performante dans la bibliothèque.

8.2.2. Extension des algorithmes

Nous proposons ici trois manières de rendre un algorithme extensible, l'idée étant que certaines parties de l'algorithme sont déléguées dans des méthodes qui peuvent être remplacées par le réutilisateur. Dans la suite de notre discussion, même si les figures ne le montrent pas toujours, nous considérons que les algorithmes suivent le patron stratégie, avec les évolutions proposées à la section précédente, à savoir une méthode pour paramétrer chaque algorithme et une méthode de classe qui fournit l'algorithme le mieux adapté.

8.2.2.1. Approche par méthode virtuelle

La première approche, le patron de conception *méthode paramètre* (*template method* [Gamm95]), consiste tout simplement, de la méthode `run()` d'un algorithme, à déporter une partie du code dans des méthodes virtuelles, les ***méthodes paramètres***, appartenant toujours à l'algorithme. Ainsi, par héritage, ces méthodes peuvent être modifiées, et sans altérer le code de la méthode `run()` peuvent changer son comportement. La figure 8.8 illustre ce mécanisme.

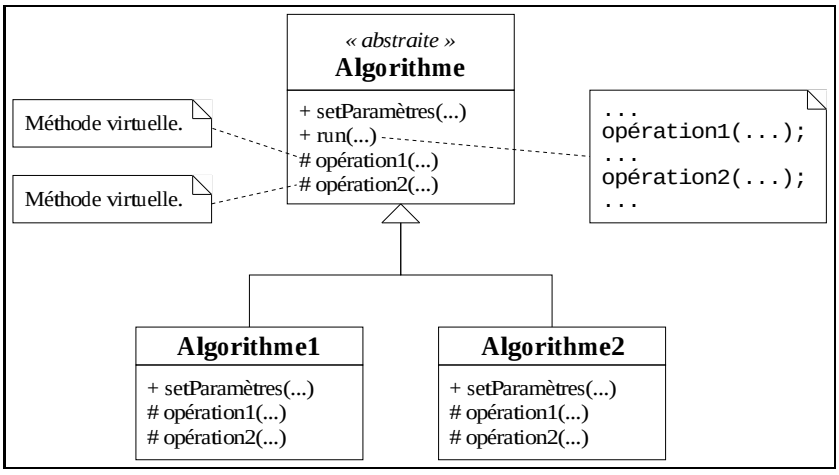


Figure 8.8: Un exemple d'extension d'un algorithme par méthodes virtuelles.

Le défaut majeur de cette approche est évidemment l'emploi du mécanisme de virtualité qui peut, si les méthodes paramètres sont souvent appelées, entraîner une perte de performance conséquente. Le second défaut de cette technique est la rigidité de l'extension de l'algorithme. Il est en effet impossible en temps réel de proposer un paramétrage différent de ceux prévus par les sous-classes de l'algorithme.

8.2.2.2. Approche par visiteur abstrait

La seconde approche repose sur le concept de ***visiteur***, également un patron de conception proposé dans [Gamm95]. Il consiste à modéliser les méthodes paramètres comme un objet. Plus précisément, le visiteur possède des méthodes qui correspondent aux méthodes paramètres. Pour qu'un objet algorithme fonctionne (i.e. sa méthode `run()` soit opérationnelle) il faut qu'il agrège un objet visiteur qui lui fournit les parties manquantes de son code. Cet objet peut être fourni par exemple à la construction de l'algorithme. La figure 8.9 présente la classe `Algorithme` qui manipule dans sa méthode `run()` un objet `visiteur` implémentant l'interface de la classe `Visiteur`, cette dernière fournit les méthodes paramètres `opération1()` et `opération2()` nécessaires à l'algorithme.

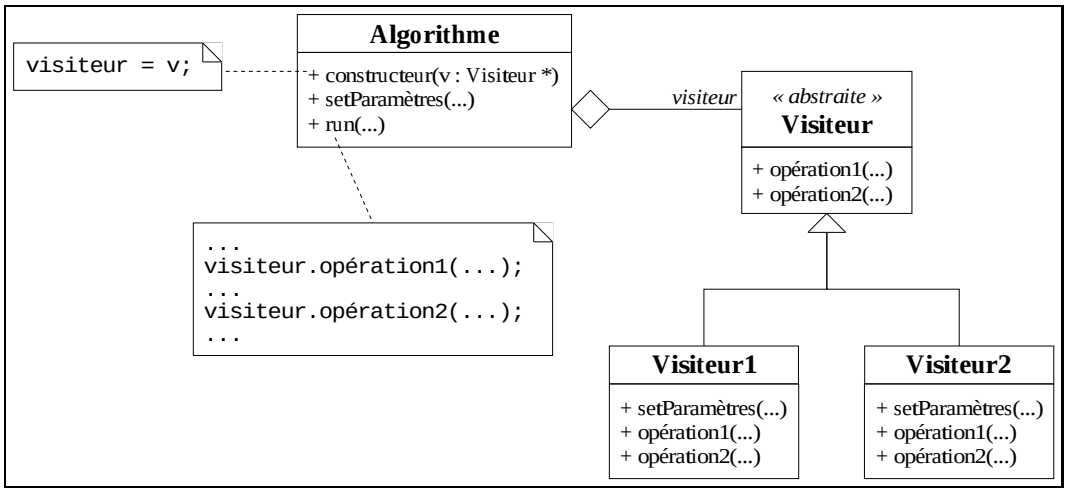


Figure 8.9: Un exemple d'extension d'un algorithme par visiteur abstrait.

L'exemple suivant montre la flexibilité apportée par la technique dans l'extension des algorithmes. Il est en effet possible de définir à l'exécution quel sera le visiteur d'un algorithme. Il faut noter que le visiteur peut avoir ses propres paramètres et qu'ils doivent être initialisés avant l'appel à l'algorithme principal.

```
Visiteur1  * v = new Visiteur1;
Algorithme * a = new Algorithme(v);

a.setParamètres(...);
v.setParamètres(...);
a.run(...);
```

Cependant, le défaut majeur, que l'on retrouvait déjà dans la première approche, réside dans l'utilisation du mécanisme de virtualité pour appeler les méthodes paramètres (e.g. les méthodes `opération1()` et `opération2()` du visiteur).

8.2.2.3. Approche par concept de visiteur

Pour éviter finalement le mécanisme de virtualité, il suffit de fournir le visiteur en paramètre et non pas en argument à l'algorithme, autrement dit ce dernier doit être un patron paramétré sur le type du visiteur qu'il manipule. Celui-ci doit alors simplement implémenter le concept de visiteur nécessaire à l'algorithme. La figure 8.10 illustre cette approche. L'exemple suivant montre que l'utilisation du visiteur est très similaire à l'approche précédente, la seule réelle différence à ce niveau étant que l'utilisateur n'a pas à créer explicitement de visiteur.

```
Algorithme<Visiteur1> * a = new Algorithme<Visiteur1>;

a.getVisiteur().setParamètres(...);
a.setParamètres(...);
a.run(...);
```

Le mécanisme de virtualité étant écarté, la technique présentée ici ne présente aucune perte d'efficacité. En revanche elle ne permet aucune flexibilité dans la construction des algorithmes, la relation algorithme-visiteur est fixée à la compilation. Cette flexibilité est pourtant importante si l'on considère qu'une méthode (comme nous l'avons expliqué dans la section sur le patron stratégie) est capable d'analyser la structure d'un graphe, de déterminer et de fournir l'algorithme le mieux adapté grâce à un paramétrage dynamique de cet algorithme avec des visiteurs.

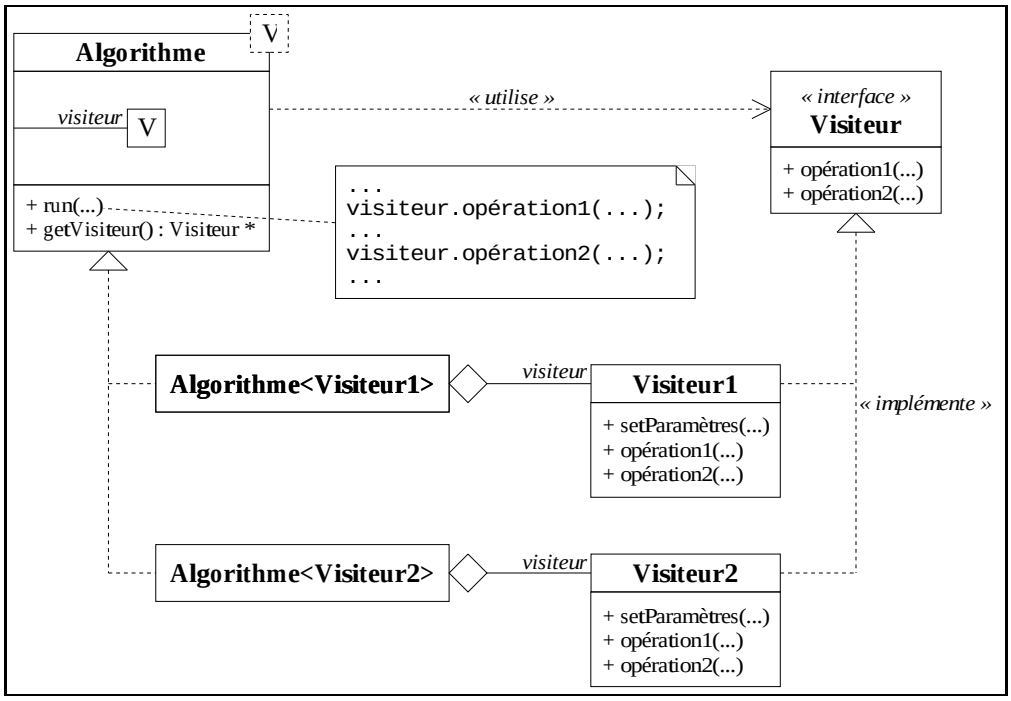


Figure 8.10: Un exemple d'extension d'un algorithme par concept de visiteur.

8.2.2.4. Conclusion

Pour permettre une bonne généricité des algorithmes, il semble donc important d'appliquer le patron stratégie avec les modifications que nous avons proposées à la section 8.1, tout en le combinant avec l'approche par concept de visiteur pour permettre une extension efficace des algorithmes. En ce qui concerne les différentes bibliothèques proposées pour la recherche opérationnelle, il faut savoir que seule BGL propose réellement une extension des algorithmes, les autres fournissant plutôt des algorithmes dans un but unique.

L'approche employée dans cette bibliothèque est très proche, mais plus générique, que celle que nous avons retenue. Nous avons expliqué les raisons qui nous ont fait choisir une voie moins intéressante pour les réutilisateurs, mais plus réaliste pour les concepteurs. Il faut également noter que la bibliothèque STL emploie la notion de *foncteur* très similaire à la notion de concept de visiteur. D'autres approches sont proposées, notamment dans [Dure01] qui s'applique à proposer des versions génériques (i.e. avec des patrons de composant) d'un grand nombre de patrons de conception proposés dans [Gamm95].

8.2.3. Gestion de données additionnelles

Il est très courant pour un algorithme d'avoir besoin d'affecter des données supplémentaires aux éléments des structures de données qu'il manipule. Par exemple un algorithme de résolution d'un problème de flot peut gérer un potentiel sur les noeuds du graphe, mais l'utilisateur ne doit pas connaître ce détail et se limiter aux données de flot. Nous discutons ici brièvement plusieurs manières de gérer ces données additionnelles.

La première approche consiste à stocker les données dans une structure à part du graphe. Un vecteur, par exemple, permet de conserver ces données dans le même ordre que les noeuds dans le graphe. L'accès aux données est immédiat, mais interdit toute modification du graphe qui perturberait l'ordre des noeuds. Il est alors possible d'utiliser un conteneur associatif pour stocker les données, par exemple un arbre binaire de recherche où l'identifiant d'un noeud joue le rôle d'une clé pour rechercher les données associées au noeud dans l'arbre. Cette approche autorise une modification structurelle du graphe, mais l'accès aux données est beaucoup plus lent ($O(\log n)$ opérations pour n noeuds dans le graphe).

Nous avons donc recherché une approche qui allie un accès efficace des données additionnelles à la flexibilité du graphe. Elle consiste à prévoir dans les classes des noeuds et des arcs un attribut qui référence pour chacun une zone mémoire que l'on appellera *zone de travail*. Cette zone n'est absolument pas gérée (ni même allouée) par la classe du graphe, mais par les algorithmes qui vont manipuler le graphe. Ainsi, lorsqu'un algorithme a besoin d'ajouter des données à un noeud, il lui suffit d'allouer ces données et de faire pointer l'attribut de la zone de travail dessus. A partir du noeud, l'accès aux données additionnelles est ainsi immédiat. Le code suivant illustre cette approche.

```
Graphe<Flot,Potentiel> * g = new Graphe<Flot,Rien>;
...
g.noeud(3).zoneTravail() = new Potentiel;
...
(Potentiel *)(g.noeud(3).zoneTravail()).potentiel = 2.5;
```

Cet exemple montre comment ajouter une donnée de la classe `Potentiel` (cf. section 8.1) à un noeud, mais soulève un problème important qui est qu'un *downcast* est nécessaire pour récupérer la donnée avec le bon type. Nous avons discuté à la section 8.1 des défauts de cette opération. La zone de travail doit néanmoins référencer n'importe quel type de données. Les langages de programmation ont différentes manières de représenter une référence sur un objet quelconque. Java propose la classe `Object` dont toute classe hérite directement ou indirectement. C++ propose le type `void *` qui représente une référence d'un type quelconque.

Un problème se pose également si plusieurs algorithmes ont besoin de la zone de travail, par exemple un algorithme de tension utilise la zone de travail mais fait appel à un algorithme de plus court chemin qui lui aussi a besoin de cette zone. Une approche consisterait, au début de chaque algorithme qui gère des données additionnelles, à sauvegarder les références de la zone de travail de chaque noeud par exemple avant de les remplacer par ses propres références. Il suffit alors de les restituer à la fin de la méthode, afin que l'algorithme appelant retrouve la zone de travail dans l'état où il l'a laissée.

Cela ne coûte qu'un nombre linéaire d'opérations en fonction du nombre de noeuds et reste négligeable pour la plupart des algorithmes. Cependant, il est possible d'obtenir une meilleure efficacité, en proposant simplement, au lieu d'une référence, une pile de références sur des zones de travail. Chaque algorithme qui gère des données additionnelles ajoute sa référence sur la pile, et celui qui est effectivement en cours d'exécution (on exclue ici une utilisation *multithread*) manipulera toujours la zone de travail au sommet de la pile.

8.3. Conception d'une bibliothèque réutilisable

Dans cette dernière section, nous proposons une brève présentation de l'avancement de notre bibliothèque, discutant de points certes pratiques, mais qui nous semblent très importants pour la réutilisabilité de la bibliothèque. Nous verrons qu'ils favorisent incontestablement la réutilisation de la bibliothèque par de tierces personnes, aussi bien pour un prototypage rapide que pour le développement d'un produit fini. Ces points concernent l'organisation même de la bibliothèque, mais également des aspects plus techniques comme la portabilité, dont nous expliquons qu'elle ne se limite pas simplement au portage sur un système d'exploitation. Cette présentation nous permettra de conclure sur les enjeux futurs de la bibliothèque, et de discuter de certaines corrections et évolutions à lui apporter.

8.3.1. Organisation de la bibliothèque

8.3.1.1. Code source ou code compilé ?

Notre manière de développer la bibliothèque repose sur une approche itérative. En effet, la première fois qu'un algorithme est créé, il est conçu principalement pour notre besoin immédiat, en pensant tout de même aux extensions les plus évidentes que l'on permet grâce à un paramétrage de l'algorithme (cf. les visiteurs). Mais il est très difficile (et trop coûteux en temps) d'envisager tous les usages d'un algorithme. C'est pourquoi nous procédons par raffinements successifs, ce qui explique qu'il nous semble indispensable de fournir le code source de la bibliothèque plutôt qu'une simple version compilée, afin qu'un réutilisateur puisse intervenir dans ce processus. Une autre raison pour fournir le code source est qu'un patron de composant se trouve entièrement dans l'interface, et que le seul moyen en C++ de l'instancier, pour les utilisateurs, est de posséder le code source associé.

Cependant, cette approche pose de nombreux problèmes. [Meye97] considère que fournir le code source est une solution de facilité. Il justifie cela par le fait qu'il n'est pas nécessaire alors de considérer le portage de la bibliothèque sur de nombreux compilateurs différents, cette tâche incombant aux concepteurs de ces compilateurs en garantissant un standard. Mais en pratique, cela revient à laisser l'utilisateur se débrouiller avec le code source pour le compiler. Cette phase est extrêmement délicate si le concepteur n'a pas pris la peine de la préparer pour l'utilisateur. En outre, ce dernier ne réutilisera pas la bibliothèque s'il n'arrive pas à la compiler rapidement. Nous pensons donc qu'il est important d'assister l'utilisateur dans la phase de compilation et qu'il est par conséquent plus difficile de produire un code source compilable sur n'importe quel système plutôt que de le compiler uniquement sur quelques plateformes et d'en distribuer simplement une version compilée.

8.3.1.2. Vérification de la bibliothèque

Il est impensable de fournir une bibliothèque réutilisable sans effectuer des tests certifiant d'une certaine fiabilité de ses composants. Il est peu réaliste de garantir une vérification absolue de la bibliothèque (e.g. [Balc98c]), mais une certaine rigueur et des séries de tests systématiques permettent de s'assurer du bon fonctionnement des composants lors de leur conception, et également que leur comportement est toujours fiable après une modification dans la bibliothèque. Pour cela nous avons opté pour trois techniques.

La première consiste à insérer des *assertions* dans le code source, c'est-à-dire des expressions qui doivent toujours être vraies au moment où elles sont testées dans le code. Si ce n'est pas le cas, une erreur est levée et

l'utilisateur est informé du problème. Ces assertions sont naturellement activées uniquement lors de la phase de débogage de la bibliothèque, afin de ne pas altérer les performances des composants dans leur fonctionnement normal.

Une seconde technique consiste à fournir pour chaque module une procédure de test de ses composants générant une trace textuelle. A la première exécution, cette trace est vérifiée entièrement et est conservée pour servir de référence lors des futures exécutions de la procédure de test. Cela permet ainsi de s'assurer qu'une modification de la bibliothèque n'a pas entraîné d'erreur apparente. Cette technique permet également de vérifier que la bibliothèque est portable (par la comparaison sur plusieurs plateformes de la trace) et garantit aux utilisateurs le bon fonctionnement de la compilation qu'ils obtiennent de la bibliothèque.

La dernière technique, proche de la précédente, consiste à fournir pour certains modules une procédure de test automatique où les jeux d'essais sont générés aléatoirement et vérifiés automatiquement. Par exemple, deux algorithmes totalement différents pour un même problème peuvent confirmer mutuellement leurs résultats. L'intérêt de combiner ce type de test avec le précédent est évident: l'un est déterministe et garantit un certain bon fonctionnement par rapport aux prévisions du concepteur, alors que l'autre est aléatoire et peut donc soulever une anomalie à chaque exécution. La combinaison de ces deux techniques rejoint les conclusions fortes intéressantes apportées dans [John02] concernant l'analyse par expérimentation d'algorithmes.

8.3.1.3. Documentation

La documentation d'une bibliothèque joue un rôle indéniable dans sa réutilisabilité. Comme nous en avons parlé à la section 6.1, la documentation fait désormais partie intégrante du code source. Il existe cependant différentes manières de présenter la documentation d'une bibliothèque, en fonction de l'organisation des composants que l'on choisit de mettre en avant.

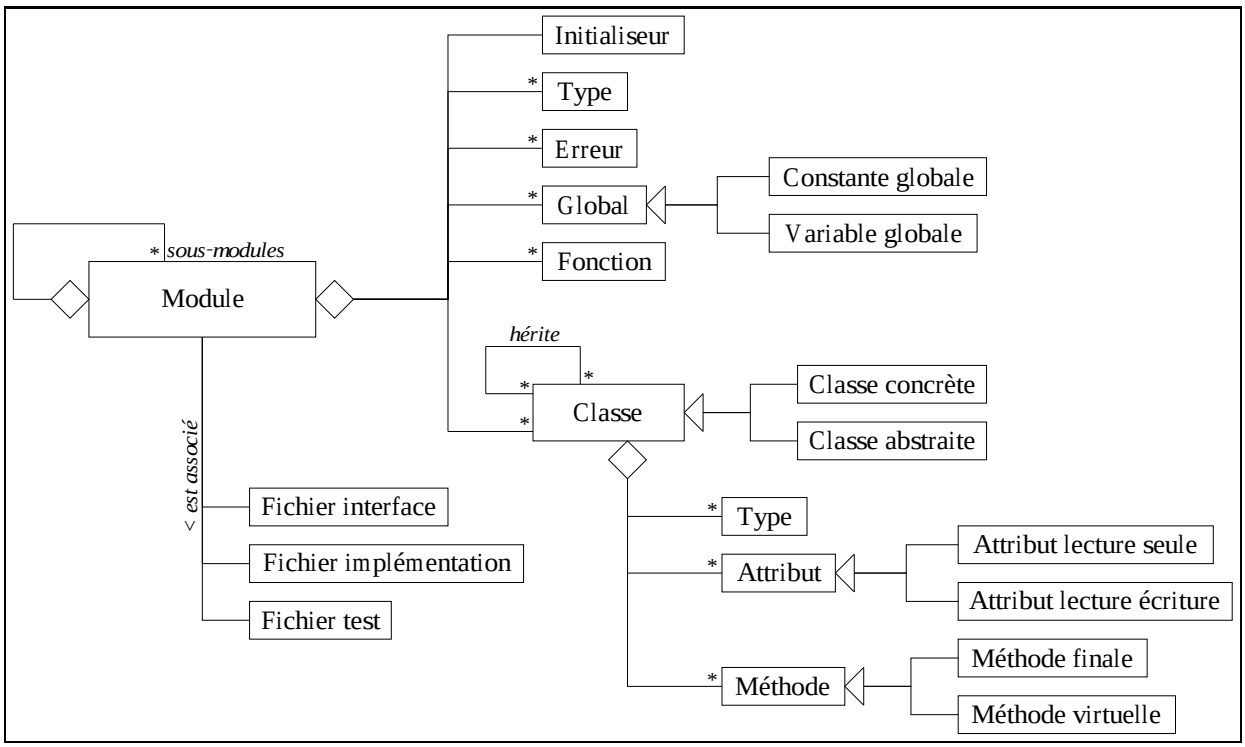


Figure 8.11: Organisation de la bibliothèque.

Cela dépend du langage et des concepts employés, mais aussi des outils utilisés pour générer la documentation qui ont tendance à favoriser une certaine vision. Pour notre bibliothèque, nous avons choisi une représentation qui reflète notre organisation en modules des composants. Nous proposons ici un schéma UML simplifié qui résume dans les grandes lignes cette organisation.

La figure 8.11 indique qu'un module est composé de trois fichiers sources: l'interface, l'implémentation et la procédure de test. Un module peut être décomposé en sous-modules. Chaque module contient des objets globaux (variables ou constants), des fonctions, des types, des erreurs (objets représentant les informations des erreurs que le module est susceptible de produire) et des classes. Chaque module contient également un *initialiseur*, objet chargé de gérer la création (avant la première utilisation du module) et la destruction (après la dernière utilisation du module) de ses objets globaux (nous revenons à la section suivante sur son utilité). Notre représentation ne met pas en avant la hiérarchie d'héritage comme la plupart des générateurs de documentation. La raison en est très simple: nous employons principalement les patrons et la composition à la place de l'héritage, contrairement à beaucoup de développements orientés objet.

8.3.2. Portabilité de la bibliothèque

Comme il l'a été rapidement expliqué à la section 6.1, nous pensons que la portabilité d'une bibliothèque ne se limite pas seulement à sa capacité à être réutilisée sur différents systèmes d'exploitation. Il est en effet très courant dans le milieu industriel de vouloir faire interagir dans un environnement logiciel des composants provenants de plateformes (compilateurs, langages, systèmes d'exploitation...) différentes (cf. [Stro95]). Afin de favoriser la réutilisabilité de notre bibliothèque, il est indispensable qu'elle puisse être intégrée dans de tels environnements. Nous avons donc tenté d'obtenir une indépendance du système d'exploitation, du compilateur et du système d'affichage, ce qui permet l'intégration de la bibliothèque dans une majorité d'environnements C++ existants. La compilation d'une version dynamique de la bibliothèque est également assurée, autorisant ainsi un chargement en temps réel de ses modules à l'exécution d'un programme, ce qui offre une possibilité d'interopérabilité avec d'autres langages que C++. Nous proposons dans ce sens un mécanisme permettant une forte interopérabilité avec Java, langage que nous avons longuement hésité à utiliser à la place de C++. Enfin le fonctionnement de la bibliothèque est garantit dans un environnement *multithread*. Nous discutons ici très brièvement de tous ces points, afin de justifier leur intérêt et d'éclairer le lecteur sur les approches employées et leurs éventuelles difficultés. Des détails pratiques sont disponibles dans [BachWb].

8.3.2.1. Indépendance de l'environnement

Le langage C++ n'est pas un langage totalement indépendant du système (système d'exploitation + compilateur) avec lequel il interagit. Un standard existe mais il reste malheureusement quelques ambiguïtés. La meilleure manière de rendre notre bibliothèque indépendante du système est de proposer une interface unique permettant à la bibliothèque de communiquer avec n'importe quel système. Le portage de la bibliothèque sur un nouveau système consistera donc simplement à implémenter l'interface. Pour simplifier cette étape, il est possible d'isoler dans quelques variables le peu de différences qu'il existe entre les implémentations de l'interface pour différents systèmes. Dans notre bibliothèque, un fichier source, dit *de dépendance*, très court, rassemble ces variables. Une partie de la figure 8.12 illustre le principe.

Pour tester cette approche, nous avons compilé notre bibliothèque sur différents systèmes, chacun avec un fichier de dépendance spécifique. Le réglage des variables de ce fichier est pour l'instant effectué manuellement, mais nous envisageons de proposer un outil permettant la génération automatique du fichier de dépendance

grâce à des tests qu'il effecturait sur le système d'exploitation et le compilateur. Un tel outil existe déjà dans le monde UNIX/Linux avec la commande `configure` exécutée avant toute compilation d'un logiciel. Il est en effet très délicat, pour un utilisateur n'ayant pas conçu la bibliothèque, de régler le fichier de dépendance. En revanche, pour nous les concepteurs, cette opération est assez immédiate. Cette approche offre donc tout de même, dans son état actuel, un portage efficace (rapide et sûr) de la bibliothèque.

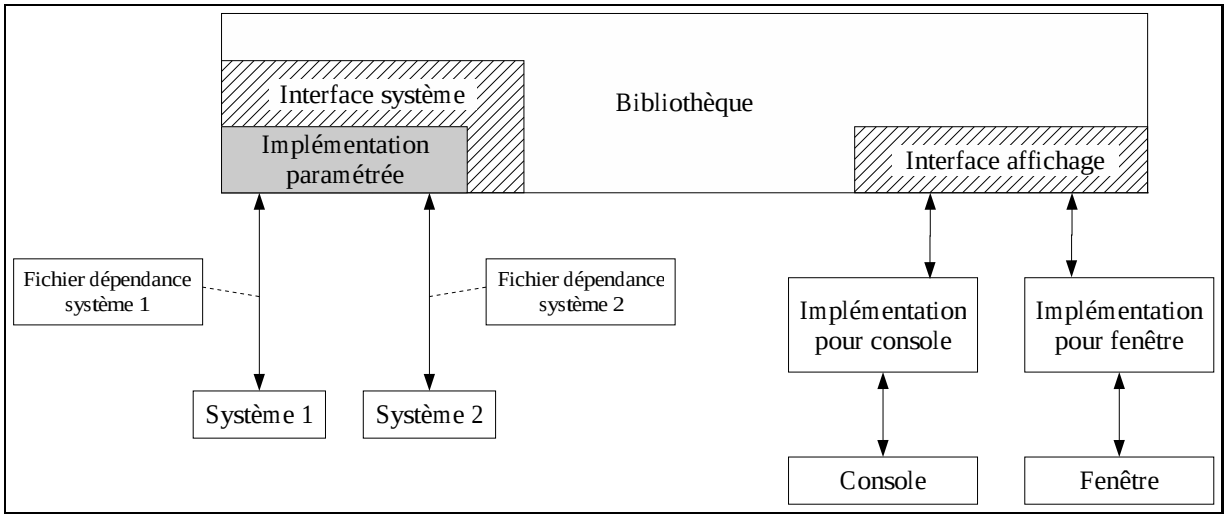


Figure 8.12: Indépendance de la bibliothèque face à son environnement.

Pour que la bibliothèque puisse coexister dans une application déjà existante, il faut également qu'elle soit indépendante du système d'affichage (texte, graphique ou autre). Un exemple simple est l'utilisation de la bibliothèque dans une interface graphique permettant de manipuler un graphe. Imaginons que l'on désire appeler un algorithme de la bibliothèque sur ce graphe. Il faut alors que la bibliothèque soit capable de dialoguer avec l'interface graphique, simplement pour informer l'utilisateur d'un mauvais fonctionnement ou de la progression d'un processus. Pour cela, il faut prévoir là aussi une interface pour que la bibliothèque communique avec tout système d'affichage (cf. figure 8.12). Ainsi, un utilisateur qui désire un affichage différent de celui fourni par défaut dans la bibliothèque devra implémenter l'interface, par exemple pour que les messages de la bibliothèque s'affichent dans une fenêtre plutôt que dans une console texte.

8.3.2.2. Bibliothèque dynamique

Une bibliothèque est un rassemblement de composants logiciels. Sous sa forme compilée, une bibliothèque peut être statique ou dynamique. Si elle est ***statique***, cela signifie qu'au moment de la compilation d'un programme (à l'édition de liens exactement), tous les composants nécessaires de la bibliothèque sont recopiés dans un fichier unique avec le reste du programme. Si la bibliothèque est ***dynamique*** (appelée souvent DLL, *Dynamic Link Library*), cela signifie qu'au moment de la compilation d'un programme, les composants nécessaires de la bibliothèque sont simplement référencés dans le fichier du programme (i.e. le nom du fichier et l'emplacement du composant dans celui-ci est noté), permettant ainsi en temps réel au programme d'aller chercher, quand il en a besoin, les composants dans les différents fichiers de la bibliothèque.

L'intérêt premier de cette forme dynamique est d'éviter une redondance de code, puisqu'un composant de la bibliothèque n'apparaît qu'une seule fois dans la mémoire de stockage, indépendamment du nombre de programmes qui l'utilisent. Mais l'intérêt secondaire qui nous intéresse ici est que la forme dynamique d'une bibliothèque renforce son interopérabilité avec un environnement C++ ou tout autre langage. En effet, il est

possible en temps réel de consulter une bibliothèque dynamique et d'en sélectionner un composant. Ce protocole est en général standard pour un système d'exploitation donné, ce qui permet à un programme créé avec n'importe quel compilateur (C++ ou autre) d'accéder à certains composants de la bibliothèque (nous présentons par la suite un tel mécanisme avec Java).

Le fait que la bibliothèque soit dynamique ne pose aucun problème de conception. Excepté les quelques bugs liés à une sous-utilisation du mécanisme à l'heure actuelle par certains compilateurs, cela n'introduit au pire que quelques modifications dans le code source. Une catégorie de compilateurs (e.g. GNU GCC pour Linux) permet une compilation dynamique sans aucune modification du code source, c'est le compilateur qui se charge de tout. Cependant la plupart des compilateurs (e.g. sous Windows) nécessitent de rajouter dans le code source une instruction pour chaque composant logiciel que l'on désire exporter, i.e. être visible des programmes qui utilisent la bibliothèque. Il faut donc prévoir ce mécanisme dès l'écriture des premières lignes d'une bibliothèque.

L'initialiseur (vu à la section précédente) joue ici un rôle très important, puisqu'il permet de manipuler un module comme une entité vraiment indépendante. En effet, il gère la préparation d'un module avant son utilisation et gère les dépendances d'utilisation qu'un module peut avoir avec d'autres. Par exemple, si un module A utilise un module B, l'initialiseur de A appelle automatiquement l'initialiseur de B. Ainsi, sous sa forme dynamique, un module de la bibliothèque peut être préparé par son initialiseur en temps réel et ses composants utilisés par un tiers sans aucun problème. Il faut savoir que cette notion d'initialiseur n'existe pas en C++, sans ce mécanisme l'ordre d'initialisation des objets globaux n'est pas contrôlé, ce qui peut causer d'énormes problèmes (e.g. une allocation de mémoire alors que le mécanisme global de gestion des allocations n'est pas initialisé). Dans un langage comme Java, cette notion d'initialiseur est nativement implémentée, il s'agit du bloc `static {...}` qu'une classe exécute à sa première utilisation.

8.3.2.3. Interopérabilité avec Java

Nous avons longuement hésité pour le développement de la bibliothèque entre C++ et Java, et nous nous sommes rendus compte qu'il n'était pas réaliste d'employer Java principalement pour les structures de données et les algorithmes. Cependant, Java reste un langage incontournable et il serait dommage que notre bibliothèque ne puisse pas être réutilisée par les utilisateurs de ce langage. L'intérêt de Java est principalement sa portabilité qui est absolue dans le sens où les composants compilés avec ce langage sont exécutables sur tout environnement qui possède une *machine virtuelle*, composant logiciel simulant un ordinateur. A l'heure actuelle, tout ordinateur dignement équipé possède cette machine virtuelle Java (JVM, *Java Virtual Machine*).

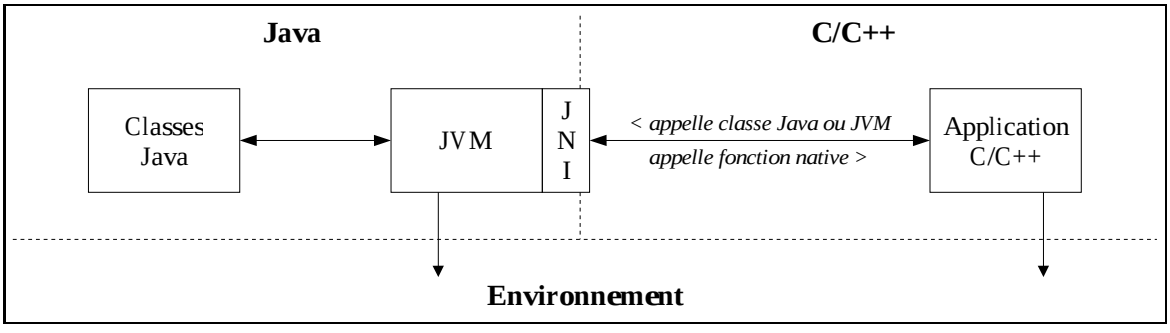


Figure 8.13: Protocole Java Native Interface.

Heureusement, il existe une interface simple et très complète qui permet aux composants Java (compilés dans le langage de la machine virtuelle) de communiquer avec des composants dits *natifs*, i.e. compilés dans le

langage de la machine physique (et stockés dans une bibliothèque dynamique). Ce protocole d'échange, appelé JNI (*Java Native Interface* [Lian99]), permet à Java d'appeler une fonction C/C++ et à C/C++ de manipuler des objets, des classes, bref tout composant du monde Java (cf. figure 8.13).

Ce mécanisme JNI offre peu de possibilités du côté Java, simplement l'appel de fonctions C/C++. Si l'on désire par exemple exécuter un algorithme C++ sur un graphe Java, il faut concevoir une fonction C++, appelable de Java, qui traduit le graphe Java en un graphe C++, exécute l'algorithme C++ et traduit finalement le résultat C++ en un résultat Java. Bien que très simple et très complet, JNI est difficilement utilisable directement. La manipulation simple d'un composant Java nécessite une dixaine de lignes de code C/C++ et la détection d'éventuelles erreurs, ce qui freine considérablement l'utilisation de ce mécanisme.

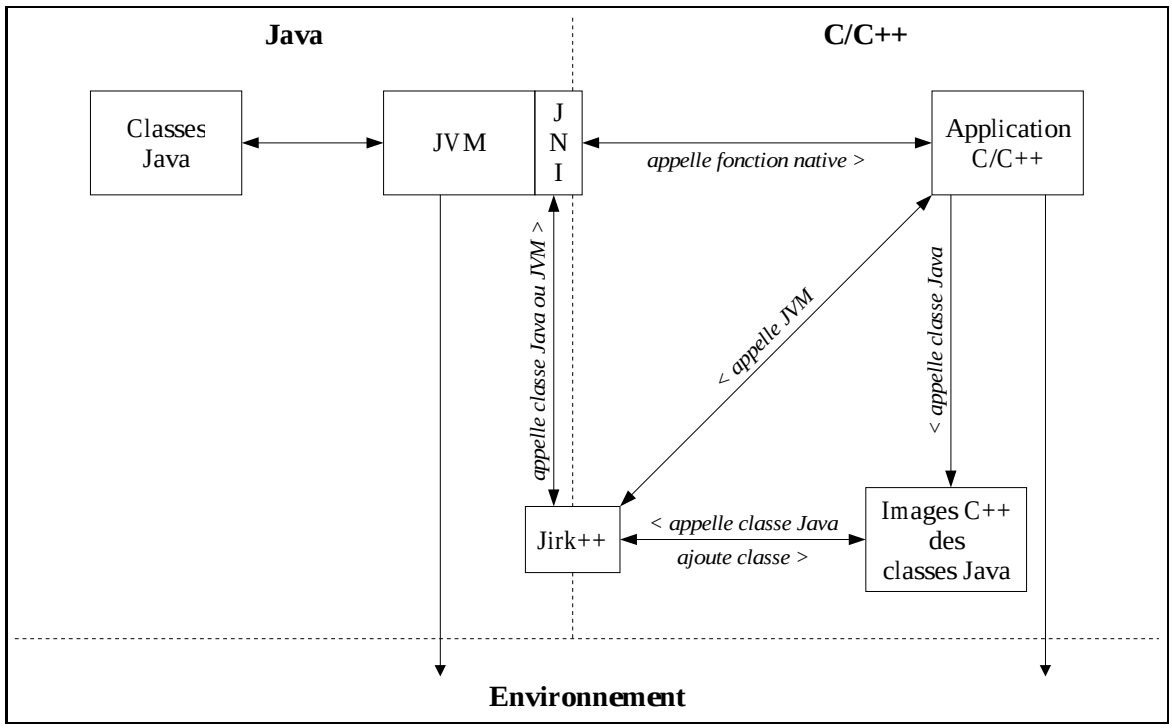


Figure 8.14: Mécanisme Jirk++.

Afin de permettre une réelle interaction entre notre bibliothèque et Java, nous avons proposé une modélisation C++ qui permet de manipuler directement et très simplement un composant Java en C++. Chaque classe Java possède une représentation C++, et c'est cette classe qui est appelée en C++, elle délègue ensuite toute exécution à son équivalent Java (cf. figure 8.14). De plus amples détails sur ce mécanisme, appelé *Jirk++*, sont disponibles dans [BachWb]. Cette interopérabilité suscite quelques intérêts puisque la bibliothèque GTL propose une version, *GTL Java*, qui utilise JNI pour permettre de manipuler ses composants en Java. Il existe également des outils commerciaux plus généralistes, équivalents à Jirk++, qui facilitent le développement de logiciels à fort couplage C++/Java (e.g. JunC++ion [CodeWb], JNI Assistant [KeyTWb]).

8.3.2.4. *Multithreading*

La portabilité passe également par la possibilité d'utiliser la bibliothèque dans un environnement où plusieurs tâches, et donc des composants de la bibliothèque, peuvent être utilisées en parallèle. Nous nous intéressons ici aux environnements *multithread* où un programme peut être séparé en plusieurs exécutions parallèles, les

threads, partageant les ressources (variables globales, fonctions, classes...) du programme. Cette fonctionnalité nous a semblé importante pour différentes raisons. Tout d'abord, Java exploite pleinement le *multithreading*, il faut donc s'attendre de la part des réutilisateurs Java à utiliser un composant de la bibliothèque en parallèle. Ensuite, en recherche opérationnelle la rapidité est un facteur majeur, il est donc souvent intéressant de pouvoir paralléliser des calculs sur des machines multiprocesseurs. La bibliothèque doit donc être portable et réutilisable dans un tel environnement. Ce portage n'entraîne pas de problèmes compliqués (cf. [WagnWb]), mais une certaine discipline de conception: protéger les ressources globales dynamiques (i.e. dont l'état peut changer au cours du programme) contre une manipulation simultanée par des *threads* différents. A l'heure actuelle, bien que la bibliothèque soit conçue pour le *multithreading*, tous les compilateurs ne l'implémentent pas pleinement.

8.3.3. Gestion des erreurs

Jusqu'à présent, nous avons totalement éludé les problèmes liés à la gestion des erreurs. Il est en effet possible qu'un utilisateur manipule mal un composant lors de l'exécution d'un programme. Lorsqu'une telle erreur se produit, il est possible de la gérer au niveau du composant, mais la plupart du temps, il est souhaitable d'informer l'utilisateur du mauvais fonctionnement. Cela pause un réel problème de portabilité: comment informer l'utilisateur ? Faut-il transmettre l'erreur au système d'affichage, faut-il émettre un son... Cela dépend naturellement de l'environnement logiciel avec lequel la bibliothèque interagit.

L'idéal serait que la gestion de l'erreur incombe à l'environnement logiciel même. Pour cela le mécanisme d'exception existe. Lorsqu'une erreur survient, soit la fonction (ou la méthode) où elle apparaît la gère, soit l'erreur est *jetée*. On l'appelle alors une ***exception***, et cela signifie que la fonction courante est arrêtée et que l'erreur est transmise au composant appelant. A son tour, ce dernier va décider de traiter l'erreur ou de la jeter à nouveau au composant qui l'a appelée. L'exception peut ainsi remonter jusqu'au premier composant appelant qui lui est alors obligé au final de la gérer.

Ce mécanisme est très intéressant d'un point de vue réutilisabilité, puisqu'il renforce le découplage entre un composant et ceux qui l'utilisent. La bibliothèque peut tout à fait détecter des erreurs, les gérer quand cela est possible et sinon les jeter. Le logiciel utilisateur n'aura qu'à se charger au final de la gestion des erreurs restantes et d'informer l'utilisateur s'il le souhaite. Ce mécanisme d'exception n'est pas toujours très utilisé car son fonctionnement est assez difficile à implémenter, tous les compilateurs jusqu'à récemment ne le géraient pas parfaitement. Ce mécanisme entraîne également un léger ralentissement du programme (cf. [ORio02]) puisque des informations doivent être accumulées en temps réel afin de permettre une remontée sans problème dans l'empilement des appels de fonction.

En revanche, l'utilisation des exceptions est très simple et favorise indéniablement la réutilisabilité. Elle offre une meilleure indépendance des composants et contribue à leur fiabilité: le compilateur se charge d'ajouter le code nécessaire pour stopper une fonction, opération qu'il effectue plus sûrement que tout programmeur; le concepteur est également déchargé d'une partie de sa tâche, il doit détecter les erreurs, mais il n'est pas obligé de toutes les gérer (il aura donc tendance à en détecter plus).

8.3.4. Implémentation de la notion de concept

Nous avons vu au chapitre 7 un défaut très important en pratique des patrons. Un mauvais paramétrage d'un patron entraîne des erreurs dans son implémentation et non pas dans celle du réutilisateur. Ce défaut est lié au

langage utilisé, en l'occurrence C++, et il serait intéressant de mettre en place un mécanisme qui permette à l'utilisateur de mieux comprendre son erreur. Dans cet objectif, les travaux [McNa00] et [Siek00] proposent une manière d'introduire explicitement la notion de concept en C++ et de permettre alors de vérifier avant l'instanciation d'un patron que ses paramètres implémentent bien le ou les concepts requis. Les messages d'erreur apparaissent ainsi plus explicites et lient la tentative d'instanciation d'un patron de l'utilisateur aux concepts requis par ce même patron. Nous étudions actuellement la possibilité d'implémenter un tel mécanisme dans notre bibliothèque.

8.3.5. Vers un environnement de développement

Nous avons présenté dans cette dernière section l'état d'avancement de notre bibliothèque et les modifications éventuelles assez immédiates que nous pourrions lui apporter. Nous expliquons maintenant très sommairement le rôle que nous souhaitons donner à cette bibliothèque dans une ambition à long terme de fournir un environnement de développement pour les problèmes de graphes, comme il en existe déjà dans de nombreux domaines, notamment la conception de logiciels et la simulation. Très brièvement nous présentons les différents outils que nous imaginons dans cet environnement résumé par la figure 8.15.

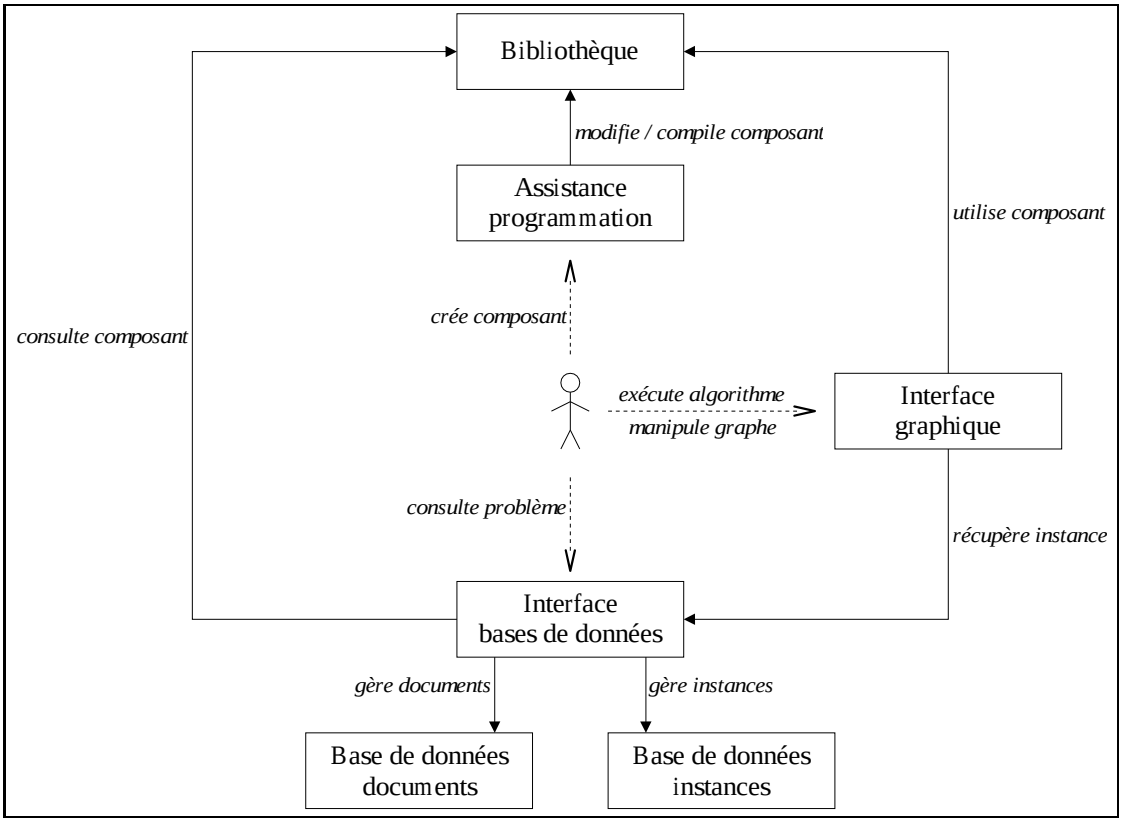


Figure 8.15: Environnement de développement pour les problèmes de graphes.

La création d'une bibliothèque suppose une organisation, cela signifie que si l'on veut rajouter un composant, il ne suffit pas d'écrire son implémentation, il faut également l'insérer dans la bibliothèque avec des manipulations qui permettent une interaction avec les autres composants. En fonction du type de bibliothèque et des objectifs qui lui sont assignés, la phase d'intégration est plus ou moins évidente. Dans le cas de notre bibliothèque,

nous proposons des squelettes de fichiers qu'un réutilisateur peut compléter pour intégrer un composant dans la bibliothèque. Dans le but d'abstraire ces mécanismes internes totalement inutiles pour un réutilisateur, il serait intéressant de lui fournir une interface, un éditeur, où il n'y ait qu'à écrire le composant, et un mécanisme caché se chargerait d'intégrer automatiquement le composant dans la bibliothèque en garantissant sa portabilité. Cette interface, couplée à la documentation, pourrait servir à la recherche et à l'accès rapide à un composant.

Les graphes ont souvent une représentation graphique, il semble intéressant pour la vérification, le débogage et, d'un point de vue pédagogique, l'apprentissage d'algorithmes de posséder un outil permettant de visualiser un graphe et les données qu'il porte. Cette interface est également importante pour la création de graphes. Nous pensons que manipuler une représentation graphique des graphes facilite énormément la phase de développement et de vérification d'un algorithme. Comme cela a été le cas en simulation, le fait de posséder une représentation visuelle du problème permet à l'humain de détecter des incohérences qu'il ne pourrait pas détecter par des traces textuelles (cf. [Hill96], [Balc98c]). Nous travaillons actuellement au développement d'une interface graphique pour la représentation et l'édition de graphes en Java (cf. [Bach02b]). Cet outil pourra être interfacé avec tout type de composant logiciel Java, C ou C++, et permettra donc d'appliquer des algorithmes provenant de ces différents langages sur les graphes manipulés dans l'interface et d'en visualiser la progression (sous une forme textuelle ou directement sur le graphe manipulé). Cette interface à elle seule offrira déjà un progrès important dans la conception d'algorithmes pour les problèmes de graphes. Cet outil n'est en aucune façon dédié spécifiquement à notre bibliothèque, mais grâce au mécanisme d'interaction Jirk++ qu'elle propose avec Java, elle est favorisée pour utiliser cet outil.

A ces interfaces de visualisation et d'assistance, nous pensons également associer des bases de données permettant de gérer et de cataloguer les problèmes de graphes. Pour un problème donné, il est important de pouvoir consulter et ajouter des références bibliographiques, de tester et de créer des instances de ces problèmes (stockage d'instances avec leur solution exacte ou la meilleure solution connue, algorithmes pour générer de nouvelles instances...), de réutiliser et de concevoir des méthodes de résolution... L'environnement que nous envisageons a donc pour but d'assister l'utilisateur dans toute la phase d'implémentation de sa méthode, en passant par l'intégration de la méthode à la bibliothèque, son débogage, la génération d'instances et sa vérification. Nous espérons que cela permette à terme (i.e. une fois la bibliothèque suffisamment garnie) le prototypage rapide et sûr de nouvelles méthodes de résolution.

8.4. Conclusion

Les premiers chapitres ont consisté à justifier l'intérêt de la réutilisabilité dans la conception d'une bibliothèque de composants de recherche opérationnelle, et à démontrer que l'approche orientée objet n'est pas forcément un facteur pénalisant pour l'efficacité des composants logiciels, mais permet au contraire, en utilisant judicieusement les possibilités du paradigme objet, une généricité, une facilité d'utilisation et une portabilité des composants.

Dans le dernier chapitre, nous avons tenté d'illustrer ces propos en discutant de choix de conception rencontrés lors de l'élaboration de notre bibliothèque. Nous n'avons présenté que quelques points, insistant principalement sur les structures de données et les algorithmes qui sont les composants majeurs d'une application de recherche opérationnelle. Nous avons montré comment les rendre génériques, c'est-à-dire indépendants et extensibles, avec une perte d'efficacité tout à fait négligeable. Les approches que nous avons retenues ne sont pas toujours les meilleures d'un point de vue conceptuel (notamment notre choix d'une abstraction modérée

des structures de données contrairement à celle proposée par la bibliothèque BGL), mais il a fallu prendre en considération certaines difficultés techniques qui nous ont conduit à un compromis qui satisfait plus les concepteurs/réutilisateurs que les utilisateurs finaux. Nous précisons simplement que les approches que nous avons écartées pour ces raisons n'ont pas été abandonnées, et que la bibliothèque a été conçue de manière à permettre d'y revenir ultérieurement. Notamment, le fait que nos algorithmes ne soient pas indépendants totalement des structures de données qu'ils traitent peut être corrigé aisément, avec seulement la modification des signatures des composants. Les structures sont suffisamment isolées à l'aide d'itérateurs qu'il est simple de les abstraire totalement. Nous attendons maintenant dans ce cas une évolution des compilateurs C++ dans la gestion des patrons de composant.

Pour conclure sur ces aspects de conception, nous citerons simplement quelques exemples de notre bibliothèque où nous avons appliqué les concepts d'algorithmes génériques retenus ici. Ceci n'est qu'un bref aperçu du potentiel de cette approche.

- Le premier exemple concerne le tri topologique (cf. annexe) qui consiste à numéroter les noeuds d'un graphe dans l'ordre de parcours à partir de la source. Un visiteur pour cet algorithme consiste à fournir une méthode qui indique si un arc peut être utilisé ou non dans le parcours. Cela permet, en revenant à la discussion du chapitre 4 sur l'algorithme de mise à l'échelle du dual, d'appliquer un tri topologique sur le graphe résiduel sans avoir physiquement à créer ce graphe.
- Un second exemple assez évident dans notre étude des problèmes de tension est l'algorithme de Minty qui se base sur la coloration (4 couleurs au total) des arcs pour rechercher certains cycles ou cocycles (cf. chapitre 2). Le visiteur ici consiste à fournir une méthode qui indique la couleur d'un arc. Ainsi, en fournissant différents visiteurs mais un seul algorithme, il est possible de produire des algorithmes pour trouver des circuits simples, des cycles avec flot augmentant, des cocycles avec tension augmentante...
- Notre troisième exemple concerne l'un des algorithmes majeurs de ce mémoire, la mise à conformité. Plus précisément la procédure qui améliore la conformité d'un arc. Elle est employée avec différents types de coût: coût linéaire, coût linéaire par morceaux, coût convexe... L'idée ici est de paramétrer la fonction avec un visiteur qui fournit toutes les informations concernant la courbe de conformité d'un arc. Ainsi, la procédure est toujours la même, mais pour savoir si une augmentation par exemple est possible sur un arc, elle interroge le visiteur de conformité. Le corps de la fonction n'a été codé qu'une seule fois et utilisé dans des situations très différentes.

Nous avons finalement présenté notre bibliothèque qui est loin d'être aboutie, mais qui a atteint à notre avis un stade où sa réutilisabilité apporte des gains en temps de développement pour la conception rapide de prototypes de recherche. Nous espérons très rapidement prouver son utilité dans l'élaboration de produits finis, puisque des efforts de conception (facilité d'utilisation des composants, possibilité d'intégration dans divers environnements...) ont été fournis dans ce but, notamment avec l'avancée du développement d'une interface graphique en Java permettant de manipuler visuellement des graphes et de leur appliquer de manière très intuitive différents algorithmes qui pourraient éventuellement provenir de notre bibliothèque.

Afin de rester objectif, il faut tout de même souligner le manque de lisibilité immédiate de notre bibliothèque, elle nécessite en effet un certain effort d'apprentissage de la part de l'utilisateur. Cette difficulté est liée à une syntaxe chargée en C++ pour décrire des patrons de composant. Elle est dûe également au fait qu'un grand nombre de composants sont organisés dans la bibliothèque et que l'ajout d'un composant doit respecter certaines règles pour que tout se passe bien. Nous pensons que cette étape peut être améliorée par une interface (visuelle ou un nouveau langage) entre l'utilisateur et le code source C++.

En conclusion, nous espérons avoir convaincu, à travers nos discussions dans cette dernière partie, de l'intérêt de concevoir une bibliothèque de composants réutilisables pour la recherche opérationnelle, et plus important de la faisabilité de cette ambition, à condition d'y consacrer suffisamment de ressources humaines. Notre bibliothèque est une tentative dans ce projet dont la réussite dépend énormément des contributions de chacun, qui permettront un enrichissement progressif par l'apport de nouveaux composants logiciels. Enfin, il nous semble important à terme d'envisager un environnement complet d'assistance au développement d'outils pour les problèmes de graphes, dans le but de faciliter et d'accélérer le déploiement de nouvelles méthodes d'optimisation.

CONCLUSION

Plusieurs travaux proposaient une représentation sous forme de graphe des contraintes temporelles d'un document hypermédia synchronisé. Nous avons montré que cette modélisation permet de considérer les problèmes de synchronisation hypermédia comme des problèmes de tension dans un graphe et donc de profiter de la théorie des graphes pour résoudre certains de ces problèmes.

Dans un premier temps, nous nous sommes intéressés à la recherche d'une tension compatible dans un graphe, traduisant la recherche d'une planification possible pour un scénario. Les méthodes étudiées sont suffisamment efficaces pour envisager leur usage en temps réel, il est tout à fait concevable de les relancer au cours de la présentation du document hypermédia afin d'ajuster la planification après une anomalie. En outre, la méthode basée sur le plus court chemin est tout à fait adaptée à l'aspect incrémental de la création d'un document. La non réalisabilité d'un ensemble de contraintes se traduit par la détection d'un circuit de longueur négative, il est alors possible d'indiquer à l'utilisateur où se situe le problème: il suffit d'exhiber les objets qui se trouvent sur ce circuit.

Pour le problème de la tension optimale, qui reflète le problème de déterminer la meilleure planification d'un scénario, nous proposons plusieurs méthodes, chacune adaptée à des phases différentes de la création et de la présentation du document synchronisé. Avec des coûts linéaires sur les objets multimédia pour mesurer la qualité de la synchronisation, la méthode de la mise à l'échelle du dual est la plus efficace et peut être envisagée non seulement pour le formatage final du document, mais également pour les prévisualisations. La méthode de la mise à conformité, certes moins performante sur des graphes quelconques, semble plus efficace sur des graphes plus structurés, une étude sur des cas réels est nécessaire pour identifier la méthode la mieux adaptée aux instances de synchronisation hypermédia.

La méthode de la mise à conformité nous semble néanmoins très importante dans l'aspect temps réel, puisque son approche pour optimiser la planification est distribuée sur le graphe (comme le montre d'ailleurs la méthode de reconstruction). En effet, si la tension est modifiée localement, ce serait le cas après un retard par exemple dans la présentation du document, alors il suffit de rendre à nouveau conformes les quelques arcs qui ne le sont plus pour conserver une planification optimale. Pour chaque arc, cette méthode nécessite $O(m(A+B))$ opérations, ce qui est tout à fait acceptable pour un usage temps réel.

Nous avons également proposé d'adapter la mise à conformité à des coûts convexes quelconques pour mesurer la qualité d'une planification. L'intérêt de cette approche est de permettre aux personnes qui élaborent les outils de conception de documents hypermédia synchronisés d'expérimenter différents types de coûts, dans un premier temps sans se préoccuper de la manière d'optimiser la planification. Comme l'explique par exemple [Kim95], il peut être intéressant d'employer des coûts quadratiques plutôt que des coûts linéaires, afin d'obtenir une meilleure répartition des déformations entre les objets multimédia.

Nos travaux sur les problèmes de tension se terminent dans ce mémoire par l'étude d'une classe particulière de graphes, les graphes série-parallèles, qui semblent présenter une structure très proche de celle des graphes issus de synchronisation hypermédia. Nous avons proposé une méthode d'agrégation efficace, $O(m^3)$ opérations, pour résoudre ces cas idéaux. Les bonnes performances de cette méthode ont ensuite été exploitées pour élaborer la première ébauche d'une méthode de reconstruction de la tension optimale d'un graphe quelconque à partir d'une décomposition le représentant en composantes série-parallèles.

Des résultats numériques montrent le potentiel de cette approche, dont nous n'avons pas réussi pour l'instant à profiter pleinement. Néanmoins, nous avons proposé deux variantes qui s'avèrent efficaces pour des graphes presque série-parallèles avec une SP-perturbation d'au plus $2\ \%$. Les tests ont révélé que la méthode peut être performante jusqu'à $10\ \%$ de SP-perturbation, ce qui permettrait de résoudre très certainement encore plus efficacement les problèmes pratiques. Un point intéressant de cette étude sur les graphes presque série-parallèles est d'avoir permis de mettre en évidence l'intérêt des différentes méthodes en fonction de la structure du graphe: les graphes série-parallèles sont traités avec la méthode d'agrégation, les graphes presque série-parallèles avec la méthode de reconstruction ou la mise à conformité simple, et enfin les graphes quelconques avec la méthode de mise à l'échelle du dual.

Une partie de ce mémoire a été consacrée à des réflexions sur la manière de concevoir des composants réutilisables pour la recherche opérationnelle. Il nous a semblé important d'en discuter ici, puisque les problèmes de tension ont été traités dans l'objectif de répondre à des besoins dans le domaine de l'hypermédia. Il était inconcevable pour nous de fournir simplement comme réponse des algorithmes sous une forme trop abstraite. Il nous semblait important de produire des algorithmes implémentés tout à fait opérationnels pour qui voudrait les utiliser. Cela nous a amené à une réflexion plus large sur la manière de mener à la fois une recherche efficace, c'est-à-dire pouvoir expérimenter rapidement une nouvelle méthode, et en même temps fournir des implémentations réutilisables par une majorité de personnes, à savoir des initiés en recherche opérationnelle, mais également des novices ayant identifié leur problème et désirant une implémentation rapide. Une autre motivation importante était de pouvoir évaluer les performances pratiques de différents algorithmes, il est en effet très délicat de comparer des implémentations si elles n'ont pas été développées sur le même modèle. Le fait de fournir un environnement réutilisable où finalement tous les algorithmes utilisent les mêmes "briques" de base facilite forcément cette comparaison.

Nous espérons dans cette dernière partie avoir convaincu de l'intérêt de proposer un environnement réutilisable de développement pour les problèmes de graphes, les enjeux étant d'accélérer l'implémentation de nouveaux algorithmes et de permettre plus aisément de fournir des produits finis. Les problèmes et les besoins de la recherche opérationnelle sont différents de ceux d'autres domaines de l'informatique, il est donc hors de question de suivre les schémas classiques proposés par le génie logiciel pour conduire le développement d'un système logiciel vaste pour la recherche opérationnelle. Nous avons donc tenté dans notre présentation du paradigme objet de montrer en quoi cette approche pouvait être bénéfique pour la recherche opérationnelle, comme elle l'a été pour de nombreux autres domaines, à condition d'éviter certaines tentations durant la phase de développement: des possibilités au premier abord alléchantes en termes de réutilisabilité et d'extensibilité peuvent conduire à une inefficacité catastrophique. A partir de notre expérience et de nos réflexions, nous avons discuté plusieurs techniques de conception qui permettent de développer des composants génériques, c'est-à-dire indépendants des structures de données qu'ils manipulent et des sous-algorithmes qu'ils emploient, tout en étant fortement extensibles, et cela avec une perte d'efficacité minimale.

Nos diverses implémentations nous ont menés progressivement au développement d'une bibliothèque assez vaste comprenant des modules pour différents problèmes de recherche opérationnelle sur les graphes (concernant surtout la tension, mais aussi les flots, les plus court chemins...), l'enjeu étant qu'elle puisse être utilisée ou réutilisée par une majorité de personnes, ce qui signifie une bonne réutilisabilité, une portabilité éprouvée et une efficacité honorable. Nous envisageons d'intégrer cette bibliothèque dans un environnement de développement pour les problèmes de graphes qui assisterait les personnes dans leurs implémentations, dans un processus d'expérimentation de nouvelles méthodes ou dans l'élaboration d'un produit fini. Notre première étape est actuellement l'élaboration d'une interface graphique permettant non seulement de manipuler des graphes, mais également de construire plus aisément des algorithmes à partir des "briques" de la bibliothèque.

Notre ligne de conduite tout au long de l'étude a été de fournir pour les problèmes de tension des algorithmes et des résultats théoriques, des implémentations informatiques réutilisables (documentées, portables, efficaces, robustes...) et des jeux d'essais (génération aléatoire). Le but d'un environnement de développement serait de pouvoir mieux intégrer toutes ces phases et de faciliter ainsi le passage d'une idée à sa réalisation, ce qui ne peut que dynamiser la recherche d'algorithmes pour les problèmes de graphes.

ANNEXE
IMPLÉMENTATION

Cet annexe contient des détails sur les implémentations des méthodes présentées dans le mémoire que nous n'avons pas jugés importants de faire apparaître directement dans notre étude. L'objectif de cet annexe est de fournir un supplément d'informations qui peuvent aider dans l'interprétation des résultats, dans leur reproduction et dans l'intégration des divers algorithmes dans un logiciel de conception et/ou de présentation de documents hypermédia synchronisés.

Dans une première partie, nous présentons des algorithmes qui viennent compléter ceux détaillés dans les chapitres précédents. Ensuite, nous expliquons les méthodes que nous avons utilisées pour générer nos instances de problèmes de flot et de tension. Nous présentons également très brièvement les structures de données employées pour représenter les différents éléments manipulés dans nos algorithmes: graphe, cycle, cocycle... Nous terminons enfin par quelques paragraphes sur la manière dont ont été menés les tests: la machine, le système d'exploitation, le compilateur, le nombre d'instances...

9.1. Algorithmes

9.1.1. Arbre recouvrant

L'algorithme présenté ici construit un arbre recouvrant d'un graphe $G = (X; U)$. L'arbre est obtenu sous la forme d'une fonction $p : x \in X \longmapsto u \in U$ qui associe à un noeud x l'arc u qui le connecte au reste de l'arbre. Pour établir cette fonction, l'algorithme parcours le graphe, en partant d'un noeud quelconque et en employant les arcs indifféremment dans le sens direct ou indirect, et $p_x = u$ si u est le premier arc qui a permis de visiter x. Cette méthode étant un simple parcours de graphe, elle nécessite $O(m)$ opérations.

```
pour tout x ∈ X faire p_x ← ∅;
soit s un noeud de G;
S ← {s};

tant que S ≠ ∅ faire
 soit x un noeud de S;
 S ← S \ {x};

 pour tout u = (x; y) ∈ U faire
  si p_y = ∅ et y ≠ s alors p_y ← u; S ← S ∪ {y};
 fin pour;

 pour tout u = (y; x) ∈ U faire
  si p_y = ∅ et y ≠ s alors p_y ← u; S ← S ∪ {y};
 fin pour;
fin tant que;
```

9.1.2. Base de cycles

Cet algorithme permet de construire une base de cycles B pour un graphe $G = (X;U)$. Sa justification et son fonctionnement sont présentés à la section 2.2.1. La méthode repose sur la manipulation d'un arbre recouvrant $T = (X;U')$. En lui ajoutant un par un les arcs de $U \setminus U'$, cela introduit à chaque fois un cycle qui est alors inséré à la base B. Cette méthode nécessite $O(mn)$ opérations, il y a $m - n + 1$ arcs à ajouter et chaque fois une détection du cycle ainsi créé est nécessaire, ce qui s'effectue au pire en $n-1$ opérations (puisqu'il y a $n-1$ arcs dans l'arbre).

```
trouver un arbre recouvrant T = (X;U') défini par la fonction p;
B ← ∅;
S ← U \ U';

tant que S ≠ ∅ faire
 pour tout x ∈ X faire m_x ← faux; /* Marque pour trouver le cycle. */
 soit u = (x;y) un arc de S;
 S ← S \ {u};
 x' ← x;
 y' ← y;

 tant que p_x' ≠ ∅ faire
  soit z tel que p_x' = (x';z) ou p_x' = (z;x');
  m_z ← vrai;
  x' ← z;
 fin tant que;

 tant que non m_y' faire
  soit z tel que p_y' = (y';z) ou p_y' = (z;y');
  y' ← z;
 fin tant;

 soit le cycle γ formé des chaînes de y à y', de y' à x et de l'arc u = (x;y);
 B ← B ∪ {γ};
fin tant que;
```

9.1.3. Plus court chemin

Nous rappelons ici l'algorithme dont l'idée générale a été proposée par L.R. Ford en 1956 pour la recherche d'un plus court chemin entre un noeud s et tous les autres noeuds dans un graphe $G = (X;U)$. Les longueurs l_u des arcs u peuvent être négatives et des circuits peuvent exister dans le graphe. La procédure consiste à repérer un arc $(x;y)$ qui ne satisfait pas les conditions d'optimalité du plus court chemin (cf. section 3.2.4) et à modifier le potentiel de l'une de ses extrémités pour vérifier la condition, cette opération est répétée jusqu'à ne plus détecter aucun arc. Cette idée a été reprise par R.E. Bellman en 1958 (cf. [Bell58]) en proposant un ordre pour traiter les arcs qui réduit la complexité de la méthode à $O(mn)$ opérations. Elle est toujours la meilleure complexité fortement polynômiale à ce jour. Cette méthode permet également de détecter la présence d'un

circuit de longueur négative (dont la somme des longueurs des arcs est négative): lorsqu'un noeud possède un potentiel inférieur à $-nC$ (C étant la longueur maximale d'un arc). Dans l'algorithme que nous proposons, les plus courts chemins sont obtenus sous la forme d'une fonction $p : x \in X \longmapsto u \in U$ qui associe à un noeud x l'arc u par lequel le plus court chemin de s arrive.

```
pour tout x ∈ X faire
 π_x ← +∞; /* Le potentiel du noeud. */
 p_x ← ∅;
fin pour;

S ← {s};
S' ← ∅;
π_s ← 0;

tant que S ≠ ∅ ou S' ≠ ∅ faire
 si S' ≠ ∅ alors sortir un noeud x de S'; /* Gestion FIFO de l'ensemble. */
 sinon sortir un noeud x de S; /* Gestion FIFO de l'ensemble. */

 pour tout u = (x; y) faire
  si π_y > π_x + l_(x;y) alors
   si π_y = +∞ alors S ← S ∪ {y};
   sinon S' ← S' ∪ {y};

   π_y ← π_x + l_(x;y);
   si π_y < −Cn alors /* Circuit de longueur négative. */
   p_y ← u;
  fin si;
 fin pour;
fin tant que;
```

9.1.4. Tri topologique

Considérons un graphe $G = (X; U)$ sans cycle, avec un seul noeud source. Le tri topologique des noeuds d'un tel graphe consiste à les ordonner de sorte qu'un noeud est toujours après tous ses prédécesseurs. L'algorithme proposé ici est un simple parcours des noeuds du graphe, en partant de sa source s. Les noeuds sont numérotés dans l'ordre dans lequel ils sont parcourus, un noeud n'étant visité qu'après tous ses prédécesseurs, ce qui correspond bien à un ordre topologique. Cette méthode n'étant qu'un simple parcours, elle nécessite $O(m)$ opérations.

```
pour tout x ∈ X faire
 m_x ← d_x^-; /* Marque pour le parcours topologique. */
 Δ_x ← 0; /* Numéro dans l'ordre topologique. */
fin pour;

S ← {s};
k ← 0;
```

```
tant que S ≠ ∅ faire
 choisir x dans S;
 S ← S \ {x};

 pour tout u = (x; y) ∈ U faire
  m_y ← m_y − 1;

  si m_y = 0 alors
   S ← S ∪ {y};
   k ← k + 1;
   Δ_y ← k;
  fin pour;
 fin si;
fin tant que;
```

9.2. Génération aléatoire de problèmes

La génération de problèmes pour nos jeux d'essais a posé deux difficultés, la première concerne la structure même des graphes, il a fallu créer des graphes connexes (la non connexité ne pose aucune difficulté pour les problèmes que l'on traite, mais nécessite de détecter les composantes connexes et de lancer les méthodes de résolution sur chacune). Il peut être intéressant aussi de générer des graphes sans circuit. Nous avons également généré des graphes série-parallèles et presque série-parallèles. La seconde difficulté consiste à créer les données mêmes du problème, c'est-à-dire les capacités de flot ou de tension selon le problème.

9.2.1. Graphes connexes

Pour générer des graphes connexes avec n noeuds et m arcs, nous avons choisi de créer aléatoirement tout d'abord un arbre connectant les n noeuds du graphe. Ensuite, $m - n + 1$ arcs sont ajoutés au hasard pour compléter le graphe. Lorsqu'un arc est inséré dans la deuxième phase, il faut éventuellement vérifier qu'il n'introduit pas de circuit, si c'est le cas il faut alors générer au hasard un autre arc. Pour créer l'arbre, la procédure consiste à ajouter l'un après l'autre les noeuds dans le graphe, en les connectant à chaque fois par un arc avec un noeud déjà présent dans le graphe choisi au hasard.

9.2.2. Graphes série-parallèles ou presque

La génération de graphes série-parallèles est simple. Elle repose sur la définition même de cette classe de graphes (cf. section 5.1.1). Le graphe se réduit au départ à un seul arc, ensuite à chaque itération, un arc du graphe est sélectionné au hasard pour être dédoublé, soit par une opération série, soit par une opération parallèle. Mais chaque opération série ajoute un nouveau noeud et un nouvel arc, alors que chaque opération parallèle ajoute seulement un arc. Cela signifie qu'il faut s'assurer qu'il y a exactement $n - 2$ opérations séries d'ajoutées dans le graphe et $m - n + 1$ opérations parallèles. Il y a plusieurs possibilités pour contrôler ces quotas. Nous avons choisi d'appliquer au hasard, mais uniformément, soit une opération série, soit une

opération parallèle quand cela est possible. Une opération série n'est ajoutée que si le nombre de noeuds limite n'a pas été atteint, et une opération parallèle n'est appliquée que si le nombre de noeuds restants à ajouter est inférieur strictement au nombre d'arcs restants.

Générer un graphe presque série-parallèle est très simple. A partir d'un SP-graphe, il suffit d'ajouter au hasard autant d'arcs que nécessaire pour compléter le graphe. Nous verrons pour la génération des problèmes de tension qu'il peut être possible de choisir d'inverser le sens d'un arc au moment où il est inséré, pour se rapprocher des problèmes de synchronisation hypermédia qui ne manipulent que des tensions positives.

9.2.3. Problèmes de flot

Nous proposons ici une méthode pour générer des problèmes dits de flot compatible qui peuvent être étendus très simplement à des problèmes de flot maximal ou de flot de coût minimal.

Soit a et b deux fonctions qui associent une valeur (réelle ou entière) à chaque arc u du graphe $G = (X;U)$ telles que $a_u \leq b_u$. a_u et b_u sont les valeurs minimales et maximales du flot qui peut traverser l'arc u. Le ***problème du flot compatible*** est de trouver un flot φ sur G tel que $\forall\, u \in U,\ a_u \leq \varphi_u \leq b_u$.

On s'intéresse ici à générer un graphe avec des valeurs de flot minimales et maximales sur les arcs telles qu'il existe au moins un flot compatible. Pour cela, on s'appuie sur la preuve de la section 2.2.1.3 qui permet d'affirmer le corollaire suivant. Tout flot sur un graphe est déterminé par ses seules composantes sur les arcs qui ne sont pas sur un arbre recouvrant donné.

Autrement dit, si on veut générer aléatoirement un flot sur un graphe G, il suffit de trouver un arbre recouvrant de G et de tirer totalement au hasard des valeurs de flot pour les arcs qui ne font pas partie de l'arbre. Ensuite, il faut déterminer les valeurs de flot pour les arcs de l'arbre afin d'obtenir la conservation des flots. Pour cela nous proposons la méthode suivante.

On choisit un noeud x de l'arbre $T = (X;U')$ qui n'a pas de successeur. On cherche à déterminer le flot φ_u de l'arc u qui le connecte au reste de l'arbre. Pour cela on calcule la valeur suivante.

$$\lambda = \sum_{(x;y)\in U\setminus U'} \varphi_u - \sum_{(y;x)\in U\setminus U'} \varphi_u$$

Pour garantir la conservation des flots sur le noeud, il suffit de poser $\varphi_u = \lambda$. Ensuite on retire le noeud x et l'arc u de l'arbre T et on recommence avec un autre noeud. On constate à chaque itération que l'on fixe le flot d'un arc. On obtient finalement bien un flot. Il suffit ensuite de tirer au hasard un intervalle $[a_u;b_u]$ pour chaque arc u autour du flot généré et on obtient un graphe qui admet au moins un flot compatible.

9.2.4. Problèmes de tension

De manière similaire au flot, nous proposons une méthode pour générer des problèmes de tension compatible qui peuvent être étendus très simplement à des problèmes de tension maximale ou de tension de coût minimal.

Soit a et b deux fonctions qui associent une valeur (réelle ou entière) à chaque arc u du graphe $G = (X;U)$ telles que $a_u \leq b_u$. a_u et b_u sont les valeurs minimales et maximales de la tension de l'arc u. Nous rappelons que le problème de la tension compatible est de trouver une tension θ sur G telle que $\forall\, u \in U,\ a_u \leq \theta_u \leq b_u$.

On s'intéresse ici à générer un graphe avec des valeurs de tension minimales et maximales sur les arcs telles qu'il existe au moins une tension compatible. Pour cela, il suffit de tirer au hasard un potentiel pour chaque noeud du graphe. Ensuite, il faut calculer la tension des arcs du graphe à l'aide de la définition 2.5. On obtient bien ainsi une tension. Il suffit alors de tirer au hasard un intervalle $[a_u; b_u]$ pour chaque arc u autour de la tension générée et on obtient un graphe qui admet au moins une tension compatible.

Il faut noter que jusqu'à présent, aucune contrainte n'a été imposée au signe de la tension. Avec la synchronisation hypermédia, la tension de tout arc doit être positive. La méthode de génération présentée ici ne garantit pas pour les instances qu'il existe une tension compatible positive. Pour cette raison, la méthode de génération aléatoire est légèrement modifiée pour les problèmes de synchronisation hypermédia. Au moment du calcul de la tension d'un arc, si celle-ci s'avère négative, alors l'arc est inversé, i.e. la source devient la destination et vice-versa.

Pour les graphes presque série-parallèles, nous avons opté pour une approche légèrement différente. En effet, une fois le graphe série-parallèle généré, il est possible de le parcourir à partir de la source et d'attribuer un potentiel de plus en plus important à mesure de l'avancée dans le parcours (qui doit être bien sûr topologique). Dans une deuxième phase, des arcs perturbateurs sont ajoutés pour rendre le graphe presque série-parallèle. Pour cela, deux noeuds sont choisis au hasard et un arc est créé entre les deux, si la tension de cet arc est négative, il faut inverser ses extrémités.

9.3. Structures de données

Une seule structure de données a été utilisée pour représenter les graphes dans les différents algorithmes implémentés. La modélisation objet est résumée dans la section 8.1. L'approche choisie est une liste d'adjacence. Un graphe est donc composé d'une liste d'arcs et d'une liste de noeuds. Chaque noeud possède une liste de ses arcs entrants et une liste de ses arcs sortants. Pour la plupart des algorithmes, la structure même de la liste a peu d'importance, car les procédures correspondent principalement à des parcours de ces listes.

Opération	Complexité
Ajout d'un noeud	$O(\log n)$
Ajout d'un arc	$O(\log m)$
Suppression d'un noeud	$O(\log n)$
Suppression d'un arc	$O(\log m)$
Accès à un noeud (par sa clé)	$O(\log n)$
Accès à un arc (par sa clé)	$O(\log m)$
Accès à l'origine d'un arc	$O(1)$
Accès à la destination d'un arc	$O(1)$
Parcours des p arcs entrants d'un noeud	$O(p)$
Parcours des p arcs sortants d'un noeud	$O(p)$
Parcours des noeuds du graphe	$O(n)$
Parcours des arcs du graphe	$O(m)$

Tableau 9.1: Complexité des opérations sur la structure de graphe.

Cependant, pour la méthode d'agrégation notamment, il est nécessaire de retirer des noeuds et des arcs du graphe. A l'inverse, la méthode de reconstruction par exemple ajoute des noeuds et des arcs. Il est indispensable que ces opérations s'effectuent de manière efficace. Malheureusement, il est impossible de garantir une complexité $O(1)$ pour toutes ces opérations à la fois, nous avons donc choisi une structure qui propose la même complexité pour toutes les opérations. Les listes sont modélisées sous la forme d'arbres binaires de recherche (la classe `map` en C++), la clé étant l'identifiant des éléments stockés. Toutes les opérations dans une `map` sont

en $O(\log p)$ opérations (p étant le nombre d'éléments de la structure). Seul le parcours de tous les éléments nécessite $O(p)$ opérations. Le tableau 9.1 résume la complexité de chaque opération, il faut juste noter que l'accès aux éléments par leur clé est à éviter, il est possible d'obtenir un accès en $O(1)$ par leur référence. L'accès par clé est normalement très rare.

Très souvent, les algorithmes nécessitent de manipuler des ensembles de noeuds ou d'arcs. Nous avons choisi comme structure de données le vecteur (la classe `vector` en C++). Elle garantit un accès à un élément en $O(1)$ opérations, un parcours en $O(p)$ (p étant le nombre d'éléments), et un ajout et une suppression en fin de liste en $O(1)$ opérations. Quand il est nécessaire de gérer une politique $FIFO$ (*First In, First Out*), une file d'attente est utilisée (la classe `deque` en C++) qui garantit les mêmes complexités que le vecteur avec en plus un ajout et une suppression en tête de liste en $O(1)$ opérations.

Il faut noter que ces structures, `vector` et `deque`, reposent sur la structure de tableau, elles réallouent automatiquement de la mémoire si nécessaire, ce qui peut conduire dans certaines utilisations à une perte de performance d'environ $10\ \%$. Lorsque les ensembles manipulés sont triés, nous avons choisi d'utiliser la structure de `map`, qui permet un accès en $O(1)$ au premier élément, toutes les autres opérations étant en $O(\log p)$ (p étant le nombre d'éléments).

9.4. Résultats numériques

Voici quelques détails supplémentaires sur les expérimentations menées. Les programmes ont été écrits en C++ et compilés avec GNU Compiler Collection (GCC) 2.95.3, sous le système d'exploitation Unix AIX 4, avec une machine RISC-6000 à 160 MHz. L'option de compilation `-O3` a été utilisée pour optimiser la vitesse des programmes.

Les expérimentations ont été menées sur des graphes générés aléatoirement avec les méthodes présentées précédemment. Pour prélever les résultats numériques, 10 instances de chaque problème avec les mêmes caractéristiques ont été générées. Les résultats présentés dans le mémoire sont donc des moyennes des données ainsi obtenues, ce qui atténue l'influence d'une instance particulière sur les résultats.

BIBLIOGRAPHIE

[Ahuj93] Ravindra K. Ahuja, Thomas L. Magnanti et James B. Orlin. **Network Flows - Theory, Algorithms, and Applications**. Prentice Hall, 1993.

[Ahuj99a] Ravindra K. Ahuja, Dorit S. Hochbaum et James B. Orlin. **Solving the Convex Cost Integer Dual Network Flow Problem**. Dans *7th International IPCO Conference*, 1999.
`http://www.ise.ufl.edu/ahuja/PAPERS/Paper03.pdf`.

[Alle83] James F. Allen. **Maintaining Knowledge about Temporal Intervals**. Dans *Communications of the ACM*, volume 26-11, pages 832–843, 1983.

[Alle91] James F. Allen. **Time and Time Again: the Many Ways to Represent Time**. Dans *International Journal of Intelligent Systems*, volume 6-4, pages 341–355, 1991.
`http://citeseer.nj.nec.com/allen91time.html`.

[ASWb] Algorithmic Solutions Software GmbH. **Library of Efficient Data Types and Algorithms**. Site Web.
`http://www.algorithmic-solutions.com/as_html/products/leda/products_leda.html`.

[Aust99] Matthew H. Austern. **Generic Programming and the STL: Using and Extending the C++ Standard Template Library**. Addison-Wesley, 1999.

[Bach01b] Bruno Bachelet, Gaëlle Boisnard et Jérémy Braud. **Synchronisation dans les documents hypermédia**. Rapport Technique, ISIMA, Université Blaise Pascal, Clermont-Ferrand, France, 2001.

[Bach02b] Bruno Bachelet, Christophe Duhamel, Lucie Masson et Alice Villeger. **Conception d'un éditeur de graphe en Java**. Rapport Technique, ISIMA, Université Blaise Pascal, Clermont-Ferrand, France, 2002.

[Bach98] Bruno Bachelet. **Creation of Libraries of Reusable Model Components for the Visual Simulation Environment**. Mémoire de DEA, ISIMA, Université Blaise Pascal, Clermont-Ferrand, France, 1998.
`http://www.nawouak.net/?doc=ms_thesis`.

[BachWb] Bruno Bachelet. **B++ Library**. Site Web.
`http://www.nawouak.net/?doc=bpp_library`.

[Baio97] Mourad Baïou et Ali Ridha Mahjoub. **Steiner 2-Edge Connected Subgraph Polytopes on Series-Parallel Graphs**. Dans *SIAM Journal on Discrete Mathematics*, volume 10-3, pages 505–514, 1997.

[Balc98c] Osman Balci. **Verification, Validation and Accreditation**. Dans *1998 Winter Simulation Conference*, pages 41–48, 1998.
`http://manta.cs.vt.edu/balci/papers/WSC98%20VVA%20Tutorial.pdf`.

[Bell58] R.E. Bellman. **On a Routing Problem**. Dans *Quaterly of Applied Mathematics*, volume 16, pages 87–90, 1958.

[Berg62] C. Berge et A. Ghoula-Houri. **Programmes, jeux et réseaux de transport**. Dunod, 1962.

[Bern87] M.W. Bern, E.L. Lawler et A.L. Wong. **Linear-Time Computation of Optimal Subgraphs of Decomposable Graphs**. Dans *Journal of Algorithms*, volume 8-2, pages 216–235, 1987.

[Bert95] Cathy Berthouzoz, Petra Fabian et Günter Dotzel. **Oberon: extensibilité et persistance**. Dans *L'objet*, volume 1-2. SOL, Paris, 1995.
`http://www.modulaware.com/objetpap.htm`.

[Bodl96] Hans L. Bodlaender et Babette de Fluiter. **Parallel Algorithms for Series Parallel Graphs**. Dans *4th Annual European Symposium on Algorithms*, pages 277–289. Springer-Verlag, 1996.
`http://citeseer.nj.nec.com/bodlaender96parallel.html`.

[Booc87] Grady Booch. **Software Engineering with Ada, 2nd Edition**. Benjamin / Cummings, 1987.

[Booc91] Grady Booch. **Object-Oriented Design with Applications**. Benjamin/Cummings, 1991.

[BorlWb] Borland Software Corporation. **Borland C++ Builder**. Site Web.
`http://www.borland.com/cbuilder/`.

[Boud95] Thomas Meyer-Boudnik et Wolfgang Effelsberg. **MHEG Explained**. Dans *IEEE MultiMedia*, volume 2-1, pages 26–38. Spring, 1995.

[Brow96] Alan W. Brown. **Foundations for Component-Based Software Engineering**. Dans *Component-Based Software Engineering*, pages vii–x. IEEE Computer Society Press, 1996.
`http://www.computer.org/cspress/CATALOG/BP07718/preface.htm`.

[Buch93a] M. Cecelia Buchanan et Polle T. Zellweger. **Automatic Temporal Layout Mechanisms**. Dans *ACM Multimedia 93*, pages 341–350, 1993.
`http://citeseer.nj.nec.com/buchanan93automatic.html`.

[Buch93b] M. Cecelia Buchanan et Polle T. Zellweger. **Automatically Generating Consistent Schedules for Multimedia Documents**. Dans *Multimedia Systems*, pages 55–67. Springer-Verlag, 1993.

[Bult98] Dick C.A. Bulterman, Lynda Hardman, Jack Jansen, K. Sjoerd Mullender et Lloyd Rutledge. **GRiNS: a GRaphical INterface for Creating and Playing SMIL Documents**. Dans *Computer Networks and ISDN Systems*, volume 30, pages 519–529, 1998.

[Camp00] André Campos. **Une architecture logicielle pour le développement de simulations visuelles et interactives individu-centrées**. Thèse de Doctorat, Université Blaise Pascal, Clermont-Ferrand, France, 2000.
`http://andre.campos.free.fr/these/these.zip`.

[Chin99] Chin-Wen Ho, Sun-Yuan Hsieh et Gen-Huey Chen. **Parallel Decomposition of Generalized Series-Parallel Graphs**. Dans *Journal of Information Science and Engineering*, volume 15-3, pages 407–417, 1999.
`http://www.iis.sinica.edu.tw/JISE/1999/199905_06.pdf`.

[Clem96] Paul C. Clements. **From Subroutines to Subsystems: Component-Based Software Development**. Dans *Component-Based Software Engineering*, pages 3–6. IEEE Computer Society Press, 1996.
`http://www.sei.cmu.edu/publications/articles/cb-sw-dev.html`.

[CodeWb] Codemesh. **JunC++ion**. Site Web.
`http://www.codemesh.com/en/products.html`.

[Cohe98] Sholom Cohen et Linda M. Northrop. **Object-Oriented Technology and Domain Analysis**. Dans *5th International Conference on Software Reuse*, pages 86–93, 1998.

[Coul98] Bernard Coulange. **Software Reuse**. Springer-Verlag, 1998.

[Cour96b] J.P. Courtiat, M. Diaz, R.C. de Oliveira et P. Sénac. **Formal Models for the Description of Timed Behaviors of Multimedia and Hypermedia Distributed Systems**. Dans *Computer Communications*, volume 19, pages 1134–1150. Elsevier Science, 1996.
`http://www.laas.fr/~courtiat/PAPERS/95400.ps`.

[Datt99] Alak Kumar Datta et Ranjan Kumar Sen. **An Efficient Scheme to Solve Two Problems for Two-Terminal Series Parallel Graphs**. Dans *Information Processing Letters*, volume 71, pages 9–15. Elsevier Science, 1999.

[Drie96] Karel Driesen et Urs Hölzle. **The Direct Cost of Virtual Function Calls in C++**. Dans *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1996.
`http://www.cs.ucsb.edu/labs/oocsb/papers/oopsla96.shtml`.

[dSil96] Monica Ferreira da Silva et Claudia Maria Lima Werner. **Packaging Reusable Components Using Patterns and Hypermedia**. Dans *4th International Conference on Software Reuse*, pages 146–154, 1996.

[Duff65] R.J. Duffin. **Topology of Series-Parallel Networks**. Dans *Journal of Mathematical Analysis and Applications*, volume 10, pages 303–318, 1965.

[Dure01] Alexandre Duret-Lutz, Thierry Géraud et Akim Demaille. **Generic Design Patterns in C++**. Dans *6th USENIX Conference on Object-Oriented Technologies and Systems*, pages 189–202, 2001.
`http://www.lrde.epita.fr/product/download/coots01.html`.

[Edmo72] J. Edmonds et R.M. Karp. **Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems**. Dans *Journal of the ACM*, volume 19, pages 248–264, 1972.

[Epps92] David Eppstein. **Parallel Recognition of Series-Parallel Graphs**. Dans *Information and Computation*, volume 98-1, pages 41–55, 1992.
`http://citeseer.nj.nec.com/eppstein92parallel.html`.

[Farg98] Hélène Fargier, Muriel Jourdan, Nabil Layaïda et Thierry Vidal. **Using Temporal Constraints Networks to Manage Temporal Scenario of Multimedia Documents**. Dans *ECAI 98 Workshop on Spatial and Temporal Reasoning*, 1998.
`ftp://ftp.inrialpes.fr/pub/opera/publications/ECAI-TempConstraints98.ps.gz`.

[Fold93] Stephan Foldes et François Soumis. **PERT and Crashing Revisited: Mathematical Generalizations**. Dans *European Journal of Operational Research*, volume 64, pages 286–294. Elsevier Science, 1993.

[ForsWb] Michael Forster, Andreas Pick et Marcus Raitner. **The Graph Template Library**. Site Web. `http://www.infosun.fmi.uni-passau.de/GTL/`.

[Fulk61] D.R. Fulkerson. **An Out-of-Kilter Method for Minimal Cost Flow Problems**. Dans *SIAM Journal on Applied Mathematics*, volume 9, pages 18–27, 1961.

[Gamm95] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, 1995.

[Gold87] A.V. Goldberg et R.E. Tarjan. **Solving Minimum Cost Flow Problem by Successive Approximation**. Dans *19th ACM Symposium on the Theory of Computing*, pages 7–18, 1987.

[Gond79] Michel Gondran et Michel Minoux. **Graphes et Algorithmes**. Eyrolles, 1979.

[Hadj96] Malika Hadjiat. **Problèmes de tension sur un graphe - Algorithmes et complexité**. Thèse de Doctorat, Université de la Méditerranée, Marseille, France, 1996.

[Hane99] Scott Haney et James Crotinger. **How Templates Enable High-Performance Scientifi c Computing in C++**. Dans *IEEE Computing in Science and Engineering*, volume 10-4, 1999. `http://www.acl.lanl.gov/pooma/papers/GenericProgramming_CSE/dubois.html`.

[Herm98] I. Herman, N. Correia, D.A. Duce, D.J. Duke, G.J. Reynolds et J. Van Loo. **A Standard Model for Multimedia Synchronization: PREMO Synchronization Objects**. Dans *Multimedia Systems*, volume 6, pages 88–101. Springer-Verlag, 1998.

[Hill96] David R.C. Hill. **Object-Oriented Analysis and Simulation**. Addison-Wesley, 1996.

[Jaza95] Mehdi Jazayeri. **Component Programming - a Fresh Look at Software Components**. Dans *5th European Software Engineering Conference*, pages 457–478, 1995. `http://citeseer.nj.nec.com/72914.html`.

[John02] D. S. Johnson. **A Theoretician's Guide to the Experimental Analysis of Algorithms**. Dans *5th and 6th DIMACS Implementation Challenges*. American Mathematical Society, 2002. `http://www.research.att.com/~dsj/papers/experguide.pdf`.

[John88] Ralph E. Johnson et Brian Foote. **Designing Reusable Classes**. Dans *Journal of Object-Oriented Programming*, volume 1-2, 1988. `ftp://www.laputan.org/pub/foote/DRC.pdf`.

[John91] Ralph E. Johnson et Jonathan Zweig. **Delegation in C++**. Dans *Journal of Object-Oriented Programming*, volume 4-11, pages 22–35, 1991.

[Jour98] Muriel Jourdan, Nabil Layaïda et Cécile Roisin. **A Survey on Authoring Techniques for Temporal Scenarios of Multimedia Documents**. Dans *HandBook of Multimedia Computing*, 1998.

[Jour99] Muriel Jourdan, Cécile Roisin et Laurent Tardif. **Constraint Techniques for Authoring Multimedia Documents**. Dans *Constraints Journal*, volume special, pages 1–17. Kluwer Academic Publishers, 1999.

[JTC1Wb] International Standardization Working Group ISO/IEC JTC1/SC22/WG15. **POSIX**. Site Web. `http://std.dkuug.dk/JTC1/SC22/WG15/`.

[KeyTWb] Key Technology. **JNI Assistant**. Site Web. `http://www.keytech.com.au/jniassistant.html`.

[Kim95] Michelle Y. Kim et Junehwa Song. **Multimedia Documents with Elastic Time**. Dans *Multimedia '95*, pages 143–154, 1995.

[Knut81] D.E. Knuth et M.F. Plass. **Breaking Paragraphs into Lines**. Dans *Software - Practice and Experience*, volume 11, pages 1119–1184, 1981.

[Kuhl96] Dietmar Kühl. **Design Patterns for the Implementation of Graph Algorithms**. Mémoire de DEA, Technische Universität Berlin, Germany, 1996.
`ftp://ftp.informatik.uni-konstanz.de/pub/algo/personal/kuehl/da.ps.gz`.

[Laya02] Nabil Layaïda, Loay Sabry-Ismail et Cécile Roisin. **Dealing with Uncertain Durations in Synchronized Multimedia Presentations**. Dans *Multimedia Tools and Applications Journal*, volume 18, pages 213–231. Kluwer Academic Publishers, 2002.
`http://citeseer.nj.nec.com/441592.html`.

[Laya96b] N. Layaïda et L. Sabry-Ismail. **Maintaining Temporal Consistency of Multimedia Documents using Constraint Networks**. Dans *Multimedia Computing and Networking*, pages 124–135, 1996.
`http://citeseer.nj.nec.com/374052.html`.

[Laya97] Nabil Layaïda. **Madeus: système d'édition et de présentation de documents structurés multimédia**. Thèse de Doctorat, Université Joseph Fourier, Grenoble, France, 1997.
`http://opera.inrialpes.fr/people/Nabil.Layaida/these/1997-Layaida.Nabil.ps.gz`.

[Leac97] Ronald J. Leach. **Software Reuse**. McGraw-Hill, 1997.

[Lee99] Lie-Quan Lee. **The High Performance Generic Graph Component Library**. Mémoire de DEA, Department of Computer Science and Engineering, University of Notre Dame, Indiana, USA, 1999.
`http://www.lsc.nd.edu/downloads/research/ggcl/papers/thesis.pdf`.

[Lian99] Sheng Liang. **The Java Native Interface**. Addison-Wesley, 1999.

[Lipp92] Stanley B. Lippman. **L'essentiel du C++**. Addison-Wesley France, 1992.

[Lipp97] Stanley B. Lippman. **Le modèle objets du C++**. International Thomson Publishing France, 1997.

[Litt90] T. Little et A. Ghafoor. **Synchronization and Storage Models for Multimedia Objects**. Dans *IEEE Journal on Selected Areas in Communications*, volume 18-3, pages 413–427, 1990.

[Liu77] P. Liu et R. Geldmacher. **On the Deletion of Nonplanar Edges of a Graph**. Dans *10th Conference on Combinatorics, Graph Theory and Computing*, pages 727–738, 1977.

[Maco97] Maciej Macowicz. **Approche générique des traitements de graphes**. Thèse de Doctorat, INSA, Lyon, France, 1997.
`http://macowicz.home.cern.ch/macowicz/PhD/thesis.ps.gz`.

[Mang69] O. Mangasarian. **Nonlinear Programming**. Mac-Graw Hill, 1969.

[Matt96] Michael Mattsson. **Object-Oriented Frameworks - A Survey of Methodological Issues**. Rapport Technique, Department of Computer Science, Lund University, Sweden, 1996.
`http://www.ipd.hk-r.se/michaelm/papers/Mattsson.Lic.thesis.pdf`.

[McNa00] Brian McNamara et Yannis Smaragdakis. **Static Interfaces in C++**. Dans *First Workshop on C++ Template Programming*, 2000.
`http://www.oonumerics.org/tmpw00/mcnamara.pdf`.

[Medi02] M.T. Medina, C.C. Ribeiro et L.F.G. Soares. **Automatic Scheduling of Hypermedia Documents with Elastic Times**. Dans *Parallel Processing Letters*, 2002.
`http://www-di.inf.puc-rio.br/~celso/artigos/multi.ps`.

[Meye97] Bertrand Meyer. **Object-Oriented Software Construction, 2nd Edition**. Prentice Hall, 1997.

[MicrWb] Microsoft Corporation. **Microsoft Visual C++**. Site Web.
`http://msdn.microsoft.com/visualc/`.

[Mint61] G. Minty. **Solving Stady-State Non Linear Networks of Monotone Elements**. Dans *IRE Transactions on Circuit Theory*, pages 99–104, 1961.

[Mull97] Pierre-Alain Muller. **Modélisation objet avec UML**. Eyrolles, 1997.

[Muss89] David R. Musser et Alexander A. Stepanov. **Generic Programming**. Dans *Lecture Notes in Computer Science*, volume 358, pages 13–25. Springer-Verlag, 1989.
`http://citeseer.nj.nec.com/musser88generic.html`.

[Newc91] Steven R. Newcomb, Neill A. Kipp et Victoria T. Newcomb. **The "HyTime", Hypermedia / Time-Based Document Structuring Language**. Dans *Communications of the ACM*, volume 34-11, pages 67–83, 1991.

[Niem97] Patrick Niemeyer et Joshua Peck. **Exploring Java**. O'Reilly, 1997.

[OMGW1] Object Management Group. **Unified Modeling Language Specification**. Site Web.
`http://www.rational.com/uml/resources/documentation/index.jsp`.

[OMGW2] Object Management Group. **Corba**. Site Web.
`http://www.corba.org/`.

[OrcaWb] Orca Computer. **Visual Simulation Environment**. Site Web.
`http://www.orcacomputer.com/vse/VSESet.html`.

[ORio02] Martin J. O'Riordan. **Technical Report on C++ Performance**. Rapport Technique, International Standardization Working Group ISO/IEC JTC1/SC22/WG21, 2002.
http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2002/n1355.pdf.

[Pidd99] Michael Pidd, Noelia Oses et Roger J. Brooks. **Component-Based Simulation on the Web**. Dans *1999 Winter Simulation Conference*, pages 1438–1444, 1999.
http://www.informs-cs.org/wsc99papers/210.PDF.

[Pla71] Jean-Marie Pla. **An Out-of-Kilter Algorithm for Solving Minimum Cost Potential Problems**. Dans *Mathematical Programming*, volume 1, pages 275–290, 1971.

[PLuq96] M.J. Pérez-Luque et T.D.C. Little. **A Temporal Reference Framework for Multimedia Synchronization**. Dans *IEEE Journal on Selected Areas in Communications*, volume 14-1, pages 36–51, 1996.
http://hulk.bu.edu/sync95/papers/mjpl_paper.ps.

[Rock70] R.T. Rockefellar. **Convex Analysis**. Princeton University Press, 1970.

[Rock84] R.T. Rockafellar. **Network Flows and Monotropic Optimization**. Wiley-Interscience, 1984.

[Rous98] Franck Rousseau et Andrzej Duda. **Synchronized Multimedia for the WWW**. Dans *Computer Networks and ISDN Systems*, volume 30, pages 417–429, 1998.

[Rous99] Franck Rousseau. **Présentations multimédia synchronisées pour le WWW**. Thèse de Doctorat, Institut National Polytechnique de Grenoble, France, 1999.
ftp://ftp.imag.fr/pub/bibliotheque/theses/1999/Rousseau.Franck/these.dir.

[Samp00a] P.N.M. Sampaio et J.P. Courtiat. **A Formal Approach for the Presentation of Interactive Multimedia Documents**. Dans *ACM SIGMM Electronic*, 2000.

[Sant99] C.A.S. Santos, P.N.M. Sampaio et J.P. Courtiat. **Revisiting the Concept of Hypermedia Document Consistency**. Dans *ACM MM'99 Electronic*, 1999.
http://citeseer.nj.nec.com/santos99revisiting.html.

[Satz96] John W. Satzinger et Tore U. Orvik. **Object-Oriented Approach**. Boyd and Fraser, 1996.

[Schm96] Doug Schmidt. **The Road to Reuse: Design Patterns**. Dans *4th International Conference on Software Reuse*, pages 229–230, 1996.

[Scho95] Berry Schoenmakers. **A New Algorithm for the Recognition of Series Parallel Graphs**. Rapport Technique, No CS-59504, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1995.
http://www.cwi.nl/ftp/CWIreports/AA/CS-R9504.ps.Z.

[SGIWb] Silicon Graphics. **Standard Template Library Programmer**. Site Web.
http://www.sgi.com/tech/stl/.

[Siek00] Jeremy Siek et Andrew Lumsdaine. **Concept Checking: Binding Parametric Polymorphism in C++**. Dans *First Workshop on C++ Template Programming*, 2000.
http://www.oonumerics.org/tmpw00/siek.pdf.

[SiekWb] Jeremy Siek, Lie-Quan Lee et Andrew Lumsdaine. **The Boost Graph Library**. Site Web. `http://www.boost.org/libs/graph/doc/`.

[Slea83] D.D. Sleator et R.E. Tarjan. **A Data Structure for Dynamic Trees**. Dans *Journal of Computer and System Sciences*, volume 24, pages 362–391, 1983.

[Soar00] Luiz Fernando G. Soares, Rogério F. Rodrigues et Débora C. Muchaluat Saade. **Modeling, Authoring and Formatting Hypermedia Documents in the HyperProp System**. Dans *Multimedia Systems*, volume 8, pages 118–134. Springer-Verlag, 2000.

[Song99] Junehwa Song, G. Ramalingam, Raymond Miller et Byoung-Kee Yi. **Interactive Authoring of Multimedia Documents in a Constraint-Based Authoring System**. Dans *Multimedia Systems*, volume 7, pages 424–437. Springer-Verlag, 1999.

[Stei90] Ralf Steinmetz. **Synchronization Properties in Multimedia Systems**. Dans *IEEE Journal on Selected Areas of Communication*, volume 8-3, pages 401–412, 1990.

[Step95] Alexander Stepanov. **The Standard Template Library**. Rapport Technique, Hewlett Packard Laboratories, Palo Alto, California, USA., 1995. `http://citeseer.nj.nec.com/stepanov95standard.html`.

[Stro95] Bjarne Stroustrup. **Why C++ is not just an Object-Oriented Language**. Dans *ACM Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1995. `http://www.research.att.com/resources/articles/oopsla.pdf`.

[Stro96] Bjarne Stroustrup. **Language-Technical Aspects of Reuse**. Dans *4th International Conference on Software Reuse*, pages 11–19, 1996.

[Stro97] Bjarne Stroustrup. **The C++ Programming Language, 3rd Edition**. Addison-Wesley, 1997.

[SunW1] Sun Microsystems. **The Source for Java Technology**. Site Web. `http://java.sun.com/`.

[SunW2] Sun Microsystems. **Javadoc Tool**. Site Web. `http://java.sun.com/j2se/javadoc/`.

[Taka82] K. Takamizawa, T. Nishizeki et N. Saito. **Linear-Time Computability of Combinatorial Problems on Series-Parallel Graphs**. Dans *Journal of the ACM*, volume 29, pages 623–641, 1982.

[Vald82] Jacobo Valdes, Robert E. Tarjan et Eugène L. Lawler. **The Recognition of Series Parallel Digraphs**. Dans *SIAM Journal on Computing*, volume 11-2, pages 298–313, 1982.

[Vazi98] Michael Vazirgiannis, Yannis Theodoridis et Timos Sellis. **Spatio-Temporal Composition and Indexing for Large Multimedia Applications**. Dans *Multimedia Systems*, volume 6, pages 284–298. Springer-Verlag, 1998.

[vHeeWb] Dimitri van Heesch. **Doxygen**. Site Web. `http://www.doxygen.org/`.

[Vida97] Thierry Vidal et Hélène Fargier. **Contingent Durations in Temporal CSPs: from Consistency to Controllabilities**. Dans *4th International Workshop on Temporal Representation and Reasoning*, 1997.
http://www.enit.fr/~Thierry/Pubdoc/TVtime97.pdf.

[vRos93] Guido van Rossum, Jack Jansen, K. Sjoerd Mullender et Dick C.A. Bulterman. **CMIFed: a Presentation Environment for Portable Hypermedia Documents**. Dans *ACM Multimedia 93*, 1993.
http://citeseer.nj.nec.com/vanrossum93cmifed.html.

[W3CW1] W3C. **Synchronized Multimedia**. Site Web.
http://www.w3c.org/AudioVideo/.

[W3CW2] W3C. **Extensible Markup Language (XML)**. Site Web.
http://www.w3c.org/XML/.

[WagnWb] Tom Wagner et Don Towsley. **Getting Started with POSIX Threads**. Site Web, 1995.
http://dis.cs.umass.edu/~wagner/threads_html/tutorial.html.

[Wald98] Jim Waldo. **The Solution to the Reuse Problem**. Dans *5th International Conference on Software Reuse*, pages 369–370, 1998.

[Werr90] Dominique de Werra. **Eléments de programmation linéaire avec application aux graphes**. Presses Polytechniques Romandes, 1990.

[XOrgWb] X.Org. **X Window System**. Site Web.
http://www.x.org/.

INDEX

A
abstraction, 135
accréditation, 122
adapteur, 127
agrégation, 95
analyse de domaine, 130
analyse de système, 129
approche basée composant, 137
approche orientée objet, 135, 139
arborescence, 23
arbre, 23
arbre binaire, 23
arbre binaire de décomposition, 86
arbre recouvrant, 23
arc ε-conforme, 77
arc adjacent, 20
arc admissible, 79
arc agrégé, 95
arc entrant, 20
arc multiple, 20
arc saturé, 46
arc sortant, 20
argument, 153
attribut, 140

B
base de cocycles, 31
base de cycles, 30
base de vecteurs, 30
bibliothèque dynamique, 176
bibliothèque statique, 176
borne à droite, 78
borne à gauche, 78
branchement, 90
branchement fermé, 91

C
cadriciel, 131
capacité, 65
chaîne, 20
chaîne élémentaire, 21
chemin, 20
chemin de fermeture, 91
chemin élémentaire, 21
circuit, 21
circuit élémentaire, 21
classe abstraite, 155
classe amie, 164
classe concrète, 155
classe d'objet, 135, 140
classe de base, 145
cocircuit, 22
cocycle, 21
combinaison linéaire, 30
compatibilité, 123
composant générique, 159
composant logiciel, 121
composante connexe, 22
composante fortement connexe, 22
composante série-parallèle, 103
composante série-parallèle maximale, 103
concept, 155
conception, 130
conditions d'optimalité du PCC, 50
conformité, 57
conformité approchée, 70
conservation des flots, 29
constructeur, 141
cost-scaling, 76
courbe de conformité, 57
courbe de conformité à droite, 78
courbe de conformité à gauche, 78
coût, 36
cycle, 21
cycle élémentaire, 21

D
décomposition série-parallèle, 103
degré, 20
degré entrant, 20

degré sortant, 20
délégation, 152
dérivée approchée, 69
design pattern, 130
destructeur, 142
downcast, 161
durée idéale, 12

E
efficacité, 124
encapsulation, 135
ε-conformité, 70
ε-dérivée, 69
évacuation saturante, 80
exception, 179
extensibilité, 123
extension, 122

F
facilité d'utilisation, 125
fiabilité, 122
flot, 29
flot ε-optimal, 77
fonction de coût minimal t-centrée, 96
fonction de coût minimal, 95
framework, 131

G
graphe connexe, 22
graphe fortement connexe, 22
graphe orienté, 20
graphe partiel, 23
graphe presque série-parallèle, 103
graphe série-parallèle, 84
graphe temporel, 32
génie logiciel, 121
généralisation, 143

H
héritage, 142
héritage multiple, 151
hiérarchie, 136
homogénéité, 123

I
implémentation, 130, 134, 141, 155
instance, 140
instanciation, 140, 153
intelligibilité, 123
interface, 134, 141, 155
itérateur, 164

L
listes d'adjacence, 26

M
maintenabilité, 122
matrice d'adjacence noeud-noeud, 25
matrice d'incidence noeud-arc, 24
matrice unimodulaire, 25
message, 141
méta-classe, 153
méta-fonction, 152
méthode, 140
méthode finale, 144
méthode paramètre, 169
méthode virtuelle, 144
mise à conformité, 60
mise à l'échelle, 65
mise à l'échelle des coûts, 76
mise à l'échelle du dual, 80
modularité, 136
module, 134
multigraphe, 20

N
noeud adjacent, 20
noeud connecté, 22
noeud de branchement, 90
noeud de synchronisation, 90
noeud destination, 20
noeud fortement connecté, 22
noeud origine, 20
noeud puits, 20, 85
noeud source, 20, 85

O
objet, 135
opération parallèle, 84
opération série, 84

out-of-kilter, 60

P
paradigme objet, 135
parallélisation, 84
paramètre, 153
patron de classe, 153
patron de conception, 130
patron de fonction, 152
patron stratégie, 168
planification, 13
planification réalisable, 13
pointeur virtuel, 146
polymorphisme, 144
polymorphisme dynamique, 144
polymorphisme statique, 144, 155
portabilité, 123
potentiel, 31
problème de la tension compatible, 35, 43
problème de la tension de coût minimal, 36, 53
problème de la tension maximale, 36, 43
problème de plus court chemin, 50
problème du flot compatible, 193
programmation générique, 158
propriété, 140
propriété d'instance, 140
propriété de classe, 140
propriété privée, 140
propriété protégée, 140
propriété publique, 140
pseudo-flot, 77

R
racine, 23
réduction parallèle, 89
réduction série, 89
relation parallèle, 84
relation série, 84
relations d'Allen, 11
réutilisabilité, 125
réutilisation, 125
robustesse, 122

S
scaling, 65
sens direct, 20
sens indirect, 20
sérialisation, 84
sous-arbre, 24
sous-classe, 143
sous-graphe, 23
sous-programme, 133
SP-arbre, 86
SP-composante, 103
SP-décomposition, 103
SP-graphe, 84
SP-perturbation, 103
SP-réduction, 89
SP-relation, 84
SP-relation simple, 86
spécialisation, 143
structure, 140
super-classe, 143
surcharge de méthode, 144
synchronisation, 91

T
table virtuelle, 146
tension, 31
tension compatible, 35
tension idéale, 36
tension principale, 85
type abstrait de donnée, 134

U
upcast, 161
utilisation, 125

V
validité, 122
vecteur dépendant, 30
vecteur indépendant, 30
visiteur, 170