Institut
Supérieur d'
Informatique de
Modélisation et de leurs
Applications

Complexe des Cézeaux
BP 1235-63173 AUBIÈRE CEDEX

Rapport de projet de 2^{e}année

Modelib

MODÉLISATION DE PROGRAMMES LINÉAIRES

Présenté par:
Quentin Lequy et Romain Gaucher

Responsable ISIMA:
Christophe Duhamel

Projet de 100 heures
Année 2004|2005

Institut
Supérieur d'
Informatique de
Modélisation et de leurs
Applications

Complexe des Cézeaux
BP 1235-63173 AUBIÈRE CEDEX

Rapport de projet de 2^{e}année

Modelib

MODÉLISATION DE PROGRAMMES LINÉAIRES

Présenté par:
Quentin Lequy et Romain Gaucher

Responsable ISIMA:
Christophe Duhamel

Projet de 100 heures
Année 2004|2005

Résumé

Lorsque la complexité des programmes linéaires augmente, il devient nécessaire d'utiliser un modeleur pour le générer. Actuellement, il existe de nombreux modeleurs commerciaux qui sont réellement efficaces. Mais ces modeleurs ont un coût très important.

Nous avions à créer un modeleur relativement léger mais non moins robuste. Notre projet était donc de créer une bibliothèque utilisable en *C++* et grâce à un langage de script, le Python. Le fait d'utiliser un langage de script orienté objet permet de créer des modèles de programme linéaires rapidement et en ayant une abstraction totale du système sur lequel il devra être résolu.

Le projet a été réalisé avec le compilateur *C++* GNU 3.3.x et l'interpréteur Python 2.4 tout en utilisant la bibliothèque standard du *C++* (STL) ainsi que Boost.Python. Notre modeleur permet la résolution des programmes linéaires grâce à l'appel de solveurs tels que CPLEX ou GLPK.

Mots clés : Programme linéaire, Modeleur, *C++* , CPLEX, Boost.Python

Abstract

Building a linear program requests a modeller when his complexity grows up. By the way, lots of different modelers already work fine. While they are designed for a professional using, they are high-priced and each modeler has his own language.

We had to make a lighter one without loosing robustness. The project was to build some library made in *C++* and callable by *C++* programs and Python scripts. This interpreted object-oriented language allows us to make program quickly and easily.

We used the GNU compiler G++ and the interpreter Python 2.4 combined with the Standard Template Library and Boost.Python. Our modeler can also solve some linear programs with a system call of the solvers CPLEX and GLPK.

Keywords : Linear program, Modeler, *C++* , CPLEX, Boost.Python

Remerciements

Nous tenons à remercier chaleureusement notre chef de projet Christophe Duhamel d'avoir accepter notre projet et d'avoir été là à chaque fois que nous passions sans prévenir dans son bureau pour discuter du projet, de nos idées et de notre progression.

Nous souhaitons aussi remercier Loïc Yon car, sans vraiment le savoir, il nous a débloqué sur un problème relatif à l'appel de CPLEX.

Enfin, nous remercions Jonathan Brandmeyer, un utilisateur expérimenté de Boost.Python et participant à sa maintenance, avec qui nous avons échangés quelques courriers électroniques et qui nous a permis de créer une simple librairie et non pas un package pour l'interpréteur Python.

Introduction

Notre deuxième année d'études à l'ISIMA incluait la réalisation un projet de cent heures réparties entre novembre 2004 et mars 2005, qui dans notre cas portait sur le thème de l'optimisation linéaire.

L'optimisation est l'un des domaines liant informatique et mathématiques, et plus particulièrement l'optimisation linéaire. L'optimisation linéaire consiste à résoudre des programmes linéaires, qui sont eux-mêmes des modèles de problèmes tirés de la réalité. Résoudre un programme linéaire grâce à l'outil informatique passe par sa retranscription informatique en vue de sa résolution par un solveur. Si l'on exclut la formalisation mathématique, cette tâche est la plus longue dans la recherche de la solution au problème correspondant. En outre, elle doit être réitérée à chaque changement dans la formalisation. Pour réduire la longueur de cette tâche, des modeleurs de programme linéaire ont été développés. .

Les entreprises ayant développé des outils relatifs à la programmation linéaire propose généralement des modeleurs dans leurs gammes de produits. Les trois principaux inconvénients de ces modeleurs sont leur disponibilité, leur coût, et l'unicité de leur solveur associé. Le but de notre projet est de créer un modeleur de programme linéaire plus léger qu'un modeleur professionnel mais qui en offre les mêmes fonctionnalités de base.

Après avoir introduit l'étude, nous allons étudier l'existant pour ensuite présenter la conception de notre solution. Enfin nous présenterons nos résultats et nous discuterons de la réalisation du projet pour finir sur nos conclusions.

Glossaire

Langage de script Un langage de script est un langage de programmation, qui n'est pas compilé, mais interprété. Les scripts sont des ensembles d'instructions stockés dans des fichiers textes. On connaît surtout les scripts shells qui permettent de lancer et coordonner d'autres programmes.

Bibliothèque dynamique Ce fichier contient des fonctions qui pourront être appelées pendant l'exécution d'un programme, sans que celles-ci soient incluses directement dans le fichier exécutable.

Bibliothèque statique Contrairement à la bibliothèque dynamique, la bibliothèque statique est utilisée à l'édition des liens. Les fonctions contenues dans cette bibliothèque seront ajoutées dans l'exécutable.

Modeleur Un *modeleur* est un outil informatique qui permet de générer et d'utiliser un modèle mathématique et plus précisément, dans notre cas, un programme linéaire.

Programme linéaire En mathématiques, les problèmes de programmation linéaire (PL) sont des problèmes d'optimisation où la fonction objectif est une fonction monotone croissante de chaque variable considérée et les contraintes sont toutes linéaires. La programmation linéaire désigne également la manière de résoudre les problèmes de PL.

Solveur Un solveur est un logiciel qui résout des programmes linéaires.

Relaxation lagrangienne La relaxation lagrangienne est un outil mathématique qui permet de trouver des bornes supérieures d'un problème de maximisation.

Table des matières

Première partie

Introduction à l'étude

Chapitre 1

Sujet de l'étude

Il nous a été proposé de réaliser un modeleur de programmes linéaires. Pour palier à certains problèmes tels que le coût d'un modeleur et la variété des solveurs existants. Notre projet propose une alternative aux modeleurs commerciaux existants.

Pour le réaliser, nous avons dû tout d'abord poser les objectifs de notre modeleur tout en proposant une alternative raisonnable vis-à-vis des différentes offres commerciales.

1.1 Essence du projet

Ce projet a pour motivation première de permettre la facilité d'accès l'outil « programme linéaire ».

1.1.1 Le problème de la modélisation

L'utilité des programmes linéaires est bien réelle puisqu'ils permettent de résoudre des problèmes tels que l'ordonnancement de vols, le dimensionnement des réseaux de télécommunication, ou encore la gestion de production. Ces problèmes aux milliers de variables ne peuvent se traiter à la main et donc, le passage par un traitement informatique est nécessaire.

Pour un ingénieur, la programmation linéaire passe par plusieurs phases :
- La formalisation mathématique du problème
- La recherche des méthodes utilisables pour résoudre ce problème
- La partie développement avec l'utilisation des bibliothèques existantes

La tâche critique étant naturellement la formalisation du problème. En effet, une mauvaise formalisation peut amener à résoudre un programme linéaire complexe inutilement. La perte de temps dans la résolution d'un problème mal formalisé est conséquente et donc induit un déficit pour l'entreprise.

L'utilisation d'un modeleur se pose alors naturellement : la partie développement ne doit pas être un frein pour pouvoir tester la modélisation (l'utilisation d'un modeleur facilitant considérablement la partie développement).

1.1.2 Demande implicite de l'école

Pour notre projet, nous voulions à la fois concilier une partie développement importante[1] et un sujet concerant notre filière. Nous avons eu vent du désir de l'ISIMA d'acquérir un modeleur de programme linéaire. Nous avons alors proposé à notre futur chef de projet d'en développer un. Celui-ci a accepté et nous avons défini ensemble un cahier des charges.

1.1.3 Le problème de licences

Un des problèmes intrinsèques aux solveurs commerciaux est leurs coûts. En effet, un solveur comme CPLEX[2][ILO95] est hors de prix pour un particulier et fonctionne aux jetons[3].

1.2 Objectifs du projet

Le but initial du projet est de fournir un modeleur de programme linéaire en *C++* . Actuellement, le projet est disponible sous deux formes : une bibliothèque utilisable en *C++* et un langage de script exploitant cette bibliothèque.

1.2.1 La bibliothèque

Elle contient un ensemble de classes, de méthodes, de conteneurs et d'algorithmes qui permettent de manipuler des programmes linéaires au sein d'un code source *C++* .

Cette bibliothèque a été pensée pour faciliter son propre usage grâce, notamment, à une syntaxe intuitive proche de celle des modeleurs existants.

De plus, la bibliothèque fournit une interface très simple aux solveurs et un système d'entrée/sortie pour l'utilisateur.

Malgré la simplicité d'utilisation, il reste une phase relativement longue qui est la compilation du fichier source : elle intervient nécessairement après toute modification du fichier source.

C'est pour cela que nous avons décidé de développer un langage de script.

1.2.2 Le langage de script

Le langage de script permet d'éviter la phase de compilation au prix d'une perte de temps lors de l'execution. En effet, un langage de script est interprété et donc, on peut modifier en temps réel des parties de code.

Nous avons donc dû chercher un langage de script performant et même tenter d'en créer un...

1.3 Connaissance du problème

Pour réaliser notre projet, nous avions déjà les connaissances nécessaire en programmation linéaire qui sont au programme de première et deuxième année. Nous avons commencé renseigné sur les diverses offres commerciales et sur leur fonctionnement général et leurs capacités.

[1] A posteriori, notre projet compte plus de dix milles lignes de codes
[2] Solveur créé par la société ILOG.
[3] Chaque session de CPLEX utilise un jeton et l'ISIMA dispose d'un certain quota de jetons par serveur.

1.3.1 Les points de départ

Pour construire notre modeleur, nous avions à disposition :
- Une documentation sur la libraire statique de CPLEX de ILOG
- Un livre de référence sur le langage AMPL [4]
- Une documentation sur le format de fichier LP[5]

Nous n'avions aucune structure interne de référence car le code source des modeleurs n'est pas libre d'accès en général.

1.3.2 La variété de solveurs disponibles

Un modeleur a besoin d'un solveur pour résoudre les programmes linéaires qu'il manipule. Les solveurs disponibles sont :

CPLEX :

Cplex est le solveur le plus célèbre et certainement le plus utilisé en entreprise. Il est produit par une société française : ILOG.
C'est le solveur utilisé à l'ISIMA.

GLPK :

Glpk est un clône gratuit de Cplex dont le code source libre d'accès. Bien que moins puissant que Cplex, il permet de résoudre des programmes linéaires de tailles raisonnables assez rapidement.

Les autres :

Parmi les offres commerciales des solveurs, il existe de nombreux solveurs entre autres Coin et Xpress. Mais nous n'avons pas eu accès à ces solveurs, donc ils ne sont pas gérés par notre modeleur.

[4] Langage de script spécialisé dans la programmation linéaire
[5] Format de stockage de programme linéaire

Deuxième partie

Méthodes matérielles et Conception

Chapitre 2

Analyse de l'existant

2.1 Aperçu de la programmation linéaire

2.1.1 Définition d'un programme linéaire

Un programme linéaire est un objet mathématique qui se compose :
- de variables au sens mathématique du terme, c'est-à-dire des inconnues dont on ignore la valeur, la résolution du programme linéaire étant l'attribution de valeurs correctes à ces inconnues
- de contraintes sous forme d'équations linéaires, c'est-à-dire des égalités ou inégalités dont un terme puisse se mettre sous la forme d'un produit scalaire entre un vecteur d'inconnues et un vecteur de constantes et dont l'autre terme soit constant.
- d'une fonction objectif souvent linéaire mais qui peut prendre d'autres formes. Ici on se limitera aux fonctions objectifs linéaires.

On peut donc mettre un programme sous la forme normalisée[1] suivante :

Min	z	=	$c^T x$
s.c.	Ax	<=	b
	x	>=	0

2.1.2 Les fichiers LP

Le format de fichier LP est un format qui privilégie l'écriture d'un programme linéaire en ligne (comme la forme normalisée).
Ce type de fichier de décompose en plusieurs champs :
- **Maximize/Minimize :** déclaration de la fonction objectif
- **Subject to :** déclaration des contraintes
- **Bounds :** les bornes des variables si elles sont bornées
- **Generals-Binaries-integers :** les types des variables

[1] Ou forme standard

Un fichier LP se présente sous la forme d'un fichier éditable avec un éditeur de texte et ayant les champs définis suivant les problèmes à traiter.

Exemple de fichiers LP :

```
Maximize                                  \ commentaire
  Obj : 9*x1 + 7*x2 + 3*x3 - x4           \ fonction objectif
Subject to                                \ les contraintes
  C1 : x1 - 3*x2 + x5 > 12
  C2 : x3 + 9*x4 + x5 < 42
Bounds                                    \ bornes de variables
  -3 <= x1 <= 45
   x2 <= 12
Generals
  x3
Binaries                                  \ variables booleennes
  x4 , x5
End
```

2.2 Les spécifications

Les objectifs du projet étaient de créer une bibliothèque utilisable en *C++* pour pouvoir créer, manipuler et traiter les programmes linéaires et leur résultats.

2.2.1 Les objectifs principaux

Au départ, le sujet étant vaste, nous avons discuté des objectifs principaux avec notre chef de projet. Il fallait des objectifs à la fois simples, réalisables mais aussi suffisamment généraux pour permettre la modélisation de cas réels.
Le cahier des charges se composait au départ de :

- la gestion d'un modèle de programme linéaire robuste et supportant un grand nombre de variables
- les outils basiques de manipulations du modèle en *C++* comme la gestion des variables et des contraintes
- la lecture et l'écriture dans des fichiers au format LP
- un appel à différents solveurs simplifié grâce à une API[2]

Mais heureusement le cahier des charges a volontairement été facilement extensible. En effet, une fois les objectifs principaux atteints, nous avons pu définir d'autres objectifs.

2.2.2 Les objectifs secondaires

Dès lors, nous nous sommes interrogés sur la façon dont serait utilisé notre modeleur. Cette réflexion a abouti à la forme actuelle de bibliothèque et à la décision de faire un langage de script. Le cahier des charges s'est donc vu étendre à :

[2]Traduction : Interface de Programmation

- La mise sous la forme d'une bibliothèque statique qui permet de manipuler le modeleur par un ensemble d'objets et de fonctions directement dans le code source, et non pas par le biais d'un exécutable
- La création d'un langage de script pour faciliter l'utilisation du modeleur
- Des outils permettant la relaxation lagrangienne
- La possibilité de travailler à la fois sous la forme *duale* et *primale*.
- La possibilité d'exploiter directement des fichiers HTML, LaTeXet PostScript
- L'affichage sous forme fractionnaire

Le projet avait donc un cahier des charges ouvert qui s'est étoffé en fonction de notre progression. Cependant celle-ci a été ralentie par des spécifications techniques.

2.2.3 Les contraintes techniques

Pour être réellement exploitable, notre modeleur devait satisfaire les contraintes suivantes :

- être rapide
- occuper un faible espace en mémoire
- avoir un code d'aspect simple
- avoir un code maintenable

2.3 Les solutions existantes

Nous avons donc vu que, pour palier aux principaux problèmes techniques liés à la complexité de transcription informatique des programmes linéaires, il existe des modeleurs.
Ces modeleurs permettent de développer rapidement des programmes linéaires car :

- ils minimisent les connaissances techniques et pratiques à avoir.
- ils optimisent automatiquement les expressions et les opérations ainsi, les solveurs sont moins sollicités.
- ils permettent d'implémenter rapidement plusieurs méthodes pour les comparer.
- ils présentent une grande faciliter de relecture.

Nous retrouvons deux types de modeleurs commerciaux, ceux qui proposent un environnement complet (OPL Studio et MOSEL) et ceux qui se basent sur AMPL principalement. Nous verrons donc ces deux cas.

2.3.1 OPL Studio et MOSEL

Ce type modeleur est proposé avec la bibliothèque Cplex pour OPL Studio et Xpress pour MOSEL. Ils se composent tout deux d'une IDE[3] ainsi qu'un traceur qui aide lors de la résolution.
Ces suites disposent de nombreux outils, comme l'exportation des résultats sous tableaux Excel etc.
Un effort très important a été produit sur l'interface et la sortie des résultats ce qui fait que les résultats sont directement utilisables par des personnes non spécialistes en recherche opérationnelle.

[3]Traduction : Environnement de Développement Intégré

2.3.2 AMPL

AMPL[RF93] est un langage puissant et facilement compréhensible pour la modélisation de problèmes de linéaires ou non. Ce langage a été développé dans les laboratoires de BELL et permet de manipuler divers solveurs.
Il faut savoir tout de même que les solutions disponibles se basant sur AMPL disponible sur le site d'AMPL[4] se composent d'un interpréteur AMPL [D.H92] et d'un solveur (généralement CPLEX).

[4]http ://www.ampl.com

Chapitre 3

Conception de la bibliothèque

La bibliothèque a été conçue autour d'une entité qui est la représentation informatique du programme linéaire. Toutes les autres entités de la bibliothèque sont rattachées à cette entité ainsi que tous les flux d'entrées et de sorties.

3.1 Fonctionnement global du modeleur

Le modeleur est donc l'ensemble formé par l'entité *Model* représentant le modèle et par tous les outils qu'ils lui sont associés. En fait l'utilisateur va plus utiliser les outils que l'entité *Model* lui-même.

3.1.1 L'entité *Model*

C'est au sein de cette entité que vont se trouver toutes les données du programme linéaire. Comme le projet est codé en *C++* , cette entité est représentée par une classe. Cette classe est la clef de voûte de la bibliothèque puisque dans toutes les autres classes, représentant les autres entités, on aura une agrégation de la classe *Model*. La classe *Model* possède des méthodes qui vont permettre de gérer les principales manipulations sur le programme linéaire comme :

- L'ajout de contraintes et la définition de la fonction objectif
- Résolution et affichage des résultats sous différents formats
- Acquisition d'informations sur le programme linéaire(dimensions, états des variables,...)
- La manipulation des multiplicateurs intervenant dans la relaxation lagrangienne

La classe *Model* va également contenir toutes les valeurs après résolution du programme linéaire afin de pouvoir exploiter les résultats soit comme solution définitive au problème, soit comme une aide à la mise au point et à l'amélioration du programme.

3.1.2 Les variables, les expressions et les contraintes

Pour faciliter la gestion des variables et des contraintes lors de la programmation du modeleur, d'autres entités ont été créées. Il s'agit de classes représentant les variables, les expressions et les contraintes.

Les variables

Les variables sont primordiales dans l'écriture d'un programme linéaire. Elles sont les composantes du vecteur « inconnues ». Elles possèdent généralement un nom, un type et un intervalle de définition. Elles sont soit réelles, soit entières, soit booléennes. Par défaut, une variable

est réelle et positive. L'utilisateur dispose de trois classes représentant les trois types de variables. Pour déclarer une variable dans le programme linéaire, il suffit d'instancier[1] une de ces classes et la variable sera automatiquement ajoutée au vecteur « inconnues ».

Les expressions linéaires

Les expressions linéaires sont simplement une somme de variables coefficientées. Grâce à une surcharge des opérateurs d'addition, de soustraction et de multiplication avec un scalaire, il est possible à l'utilisateur de manipuler les expressions aussi naturellement qu'il le fait en mathématiques.

Pour conserver leur linéarité, il n'a le droit qu'aux opérations d'addition et de multiplication par un coefficient. Le code en devient limpide et très naturel. Il est aussi possible de créer des expressions à partir certains autres outils. La fonction objectif va être définie par une expression que l'utilisateur va designer à la classe *Model*.

Les contraintes

Les contraintes sont le cœur d'un programme linéaire : elles sont donc un pilier du modeleur. Une contrainte peut être de plusieurs types : inférieure, supérieure, égale. Il y a deux classes représentant les contraintes :

- La classe permettant de construire les contraintes
- La classe permettant de manipuler les contraintes déjà présentes dans le programme linéaire, pour les relaxer par exemple

La classe qui permet de construire les contraintes est basée sur les expressions et sur le type des contraintes. Elle est construite de telle façon qu'il est possible de chaîner les contraintes, ce qui est notamment utile pour définir des contraintes de bornes sur les variables.

3.1.3 Les structures auxiliaires de données et d'algorithmes

Parmi les outils disponibles dans la bibliothèque, on trouve des structures de données aidant à la génération et à la gestion les variables, et, des algorithmes aidant à la construction des contraintes.

Les structures de données

Les structures de données ont pour but de rassembler des ensembles de variables (ou des ensembles de coefficient numériques) ayant le même rôle au sein du programme linéaire. Elles permettent à l'utilisateur de générer automatiquement les noms de variables grâce à un système d'indice, puis de leur appliquer un traitement par le biais d'algorithme sur ces structures.
Les structures implémentées sont :

- les vecteurs de coefficients numériques qui facilitent les conversions et les produits scalaires
- les ensembles et les parties, qui permettent de créer des ensembles au sens mathématique du terme et de gérer les opérations ensemblistes sur les parties de ces ensembles
- les vecteurs de variables, c'est-à-dire un tableau unidimensionnel de variables ayant le même nom à leur indice dans le tableau près
- les familles de variables, qui ont le même rôle que les vecteurs de variables, mais qui sont basés sur des ensembles de n-uplets caractérisant les variables

[1] Rappel : instancier une classe revient à la déclarer au sein du code source

Chaque structure est adaptée à certains types de programmes et a un usage particulier. Pour manipuler ces structures, l'utilisateur utilise des algorithmes qui ont généralement pour but de générer des contraintes à partir de ces structures.

Les structures d'algorithmes

Pour aider l'utilisateur à coder les algorithmes sur les structures de données, la bibliothèque inclut des structures prédéfinies d'algorithmes. Ce sont en fait des classes *C++* utilisant la généricité comme paramètres des algorithmes. Ces algorithmes sont essentiellement des boucles itératives et des boucles conditionnelles, avec des petits raffinements.

3.2 Structure de l'entité *Model*

L'entité *Model* doit pouvoir représenter entièrement un programme linéaire. Elle doit donc avoir accès à la fonction objectif, aux variables et aux contraintes.

3.2.1 La matrice creuse

Lorsque l'on est amené à modéliser un programme linéaire, on rencontre nécessairement le problème du stockage d'une matrice en mémoire. Nous avons décidé de stocker les contraintes sous forme matricielle pour suivre la forme standard.

Principe de la matrice creuse :

La majorité des valeurs de la matrice étant nulles, on dit que cette matrice est « creuse ». Une idée de stockage est de ne pas stocker les valeurs nulles, puisqu'on sait qu'elles sont nulles. Il suffit donc de savoir quels sont les indices des valeurs non nulles sur chaque ligne (ou chaque colonne) pour limiter la taille du stockage. En fait, nous utilisons un procédé un peu différent. Nous stockons les valeurs non nulles en les indexant par leurs coordonnées dans la matrice, tout en conservant les dimensions de la matrice. Cela revient au même mais permet d'économiser encore un peu plus de la place lors du stockage.

Application avec un conteneur associatif

On utilise un conteneur associatif (le type map de la STL) avec pour clé, la paire formée par les indices de la valeur stockée. Il faut tout de même souligner que ce gain de stockage a un coût en temps non négligeable. En effet toute recherche sur ce map est utilisé avec l'algorithme find. Si N est le nombre de valeurs non nulles de la matrice, cet algorithme s'exécute avec une complexité en $O(log(N))$ contre une complexité en $O(1)$ si on utilisait un tableau ou un vector. On peut enfin souligner que cette structure de « matrice creuse »est générique.

3.2.2 Le système interne de variables et de contraintes

Une attention toute particulière a été portée sur la façon de stocker les variables en mémoire afin d'éviter la duplication de données et d'éviter des problèmes de domaines d'existence. Le problème de stockage des contraintes et des variables est similaire (on peut le deviner si on considère les contraintes comme des variables duales) et se résume en trois points liés, à savoir le problème de duplication de données, le domaine d'existence et le type de conteneur.

Le problème de duplication de données

Comme nous l'avons vu précédemment, créer une variable revient à instancier une des classes représentant les variables. Si les données de la variables se trouve dans la classe, à chaque fois que l'on fait une copie de variable, par exemple lors de manipulation d'expression, on créé automatiquement une nouvelle variable.

Ce système va devenir très complexe s'il faut assurer l'unicité des variables. En effet, l'entité *Model* devra connaître toutes les instances correspondant à la même variable pour pouvoir leur repercuter chaque modification faite sur une variable à toutes les instances la représentant. Il est donc préférable de stocker les données des variables dans l'entité *Model* et d'y accéder avec un identifiant, une référence ou un pointeur. Ceci exige que les instances connaissent le *Model* auquel elles sont rattachées.

Le problème d'existence

L'autre problème de l'instanciation d'une classe pour représenter une variable est le domaine de validité de la variable. Lorsque une instance est détruite, on peut soit considérer que la variable l'est aussi, soit considérer que la variable existe encore dans le programme linéaire.

Or considérer que la variable est détruite empêche l'utilisateur de créer des variables dans des environnements locaux comme des fonctions ou des boucles. De même un problème se pose concernant les instances temporaires servant par exemple au manipulation d'expressions. Il ne vaut mieux pas qu'une nouvelle variable soit déclarée à chaque copie de variable. Pour résoudre ce problème, on a utilisé la surcharge des constructeurs en *C++* pour différentier les créations des copies.

Le type de conteneur

Il reste encore trois problèmes à résoudre : faire accéder la variable aux données, savoir comment représenter une variable et choisir quel type de conteneur de données utiliser. Mais ces problèmes ne sont que en fait que des choix de programmation. Nous avons opté pour référencer les variables par un identifiant entier : celui-ci n'a ni les contraintes d'initialisation et d'utilisation des références, ni les problèmes de réallocation des pointeurs et pourtant occupe la même place en mémoire.

L'inconvénient des identifiants est la recherche de l'identifiant de la variable parmi la totalité. Nous avons choisi de stocker les données des variables dans un conteneur associatif qui répond bien au problème de la recherche d'identifiant. Ce problème de conteneur est problème récurrent en informatique et aucune solution existante n'est vraiment meilleure qu'une autre. La solution dépend de l'utilisation du conteneur et dans notre cas le conteneur associatif nous a semblé la meilleure option.

3.2.3 Les interactions avec le système

Naturellement, après avoir modéliser un problème et après l'avoir implémenter dans le modeleur, il est indispensable d'avoir la solution obtenue pour le problème.
Pour ce faire, nous appelons divers solveurs que sont :

1. CPLEX
2. GLPK (GLPSOL)

Pour appeler ces solveurs, nous utilisons simplement le système. En fait, nous envoyons au système une ligne de commande spécifiant le fichier LP d'entrée et le fichier de sortie (dans lequel les résultats du solveurs seront stockés).

Afin de ressortir les résultats intéressants pour l'utilisateurs, il est nécessaire d'interpréter la sortie du solveur.

3.3 Interactions avec l'utilisateur

Pour que notre bibliothèque soit vraiment attractive, il faut que l'on puisse générer facilement des documents. C'est pour cela que nous avons décidé de gérer une somme intéressante de formats de sortie différents ainsi qu'un format d'entrée standard afin de pouvoir sauvegarder des problèmes (sous la forme d'un fichier LP).

3.3.1 Flux de fichiers LP

D'après le type même des fichier LP, le chargement d'un fichier LP revient « simplement »à lire chaque ligne et à la traiter.
Les principaux problèmes relatifs aux fichiers LP sont :

1. L'écriture sur plusieurs lignes de la fonction objectif ainsi que des contraintes
2. Le nombre d'écritures différentes comprises par le format LP
3. La rapidité de chargement (limitée par le temps des allocations)
4. Le stockage des différentes expressions afin de faciliter le passage du fichier LP chargé au *Model*

Tout d'abord, afin d'accélérer considérablement la transcription en mémoire du fichier LP chargé au *Model*, nous avons décidé d'utiliser des structures très proches. La grosse différence entre les deux étant le typage : lors du chargement du fichier LP, tous les éléments seront stockés sous forme de string[2] et n'ont pas de types spécifiques à leur entité propre.

L'algorithme de base, utilisé pour charger le fichier LP est le suivant :

[2]Conteneur de la STL qui permet de manipuler les chaînes de caractères

```
Procédure ChargerLP(nomFichier)
    motCourant : chaîne
    ligneCourante : chaîne
    Tant que (¬ fin de fichier) faire
        ligneCourante ← ChargerLigneCourante(nomFichier)
        motCourant ← PremierMot(ligneCourante)
        Selon que
            motCourant = "Minimize" : ChargerObjectif(ligneCourante)
            motCourant = "Subject to" : ChargerContrainte(ligneCourante)
            [Et de même pour tous les autres mots clés]
        Fin Selon que
    Fait
Fin
```

Algorithme 1: Chargement de fichiers LP

Afin d'accélérer cet algorithme, il faut éviter les tests sur les chaînes de caractères dans la partie « Selon que ». Pour cela, nous avons utilisé un conteneur associatif qui a une valeur de chaîne de caractère retourne un identifiant qui nous indique quelle champs est ciblé. Cette façon de procéder a un avantage supplémentaire : elle permet d'associer à plusieurs mots clés la même valeur, c'est indispensable pour le format de fichier LP car chaque nom de décline en quelques abréviations[3].

Enfin, pour gérer le fait que la fonctions objectif ainsi que les contraintes peuvent s'écrire plusieurs lignes, nous utilisons un drapeau qui nous indique dans quel champ on est et qui change de valeur lorsqu'un nouveau champ est commencé.

Une particularité de la gestion de la mémoire de cette partie (qui est regroupée dans *FichierLP.cpp*) est qu'elle est déléguée grâce à l'utilisation des smart pointers [4] de la STL, les *auto_ptr*.

Une fois ceci réalisé, la routine critique est la reconnaissance des expression, si elles sont valides ou non. Pour cela, nous avons quelques fonctions qui valident le fait qu'un élément soit un nombre ou une variable. Ensuite, il faut différencier les monômes ayant un PLUS ou un MOINS, c'est le rôle du *Tokenizer*, les tokens[5] étant les caractères PLUS et MOINS.

3.3.2 Les flux de sortie HTML, LaTeX, PostScript

Pour que notre bibliothèque soit vraiment attractive, il faut que l'on puisse générer facilement des documents. C'est pour cela que nous avons décidé de gérer une somme intéressante de formats de sortie différents :

- HTML
- LaTeX
- PostScript

En combinant cela avec la sauvegarde de la matrice creuse en image et la sortie à chaque instant du fichier LP et des informations relatives au problème, il est possible de créer un

[3] Par exemple, "Min" pour "Minimize", "st" pour "Subject to", etc.

[4] Ce sont des pointeurs qui sont stockés dans des objets et qui se desallouent à leur destruction

[5] Caractères redondant qui relient des monômes en expression

document complet et directement utilisable. (A noter que pour le format LaTeX, il faut tout de même compiler le fichier .tex !)

Voici les différents avantages des formats gérés :

HTML C'est un format de fichier très facile à gérer, de plus, avec une feuille de style CSS, on peut rendre un document simple, beau. Mais dans tous les cas, le format HTML est extrêmement portable.

LaTeX C'est le format de fichier très connu qui permet de faire des documents scientifiques en facilitant la mise en forme d'équations tout en ayant un bon rendu. Ce format, une fois compilé, permet de générer des fichiers PDF, PostScript, HTML etc. grâce à divers outils disponibles sous licence libre.

PostScript Ce format a été choisi car il fournit un document directement imprimable. Il faut tout de même noter que le support de ce type de fichier est très limité, on ne peut pas ajouter de formules mathématiques autrement que tapées directement à la main.

Sortie HTML et LaTeX :

Le principal intérêt des sorties HTML et LaTeXest la capacité de générer des documents directement utilisables, tels que des comptes-rendus, pages HTML et autres documents portables comme les fichier PDF.

Nous avons apporté à l'utilisation des ces formats de sortie un soin important, en effet, nous voulons que la création de documents grâce à nos flux de sorties soit aisée et puissante : il doit être possible de générer un rapport depuis les fichiers de sortie.

C'est pour cela que les méthodes principales des classes HTMLFile et LATEXFile sont des surcharges d'opérateurs et notamment la surcharge de flux.
Les éléments que l'on peut envoyer par flux dans les fichiers sont :

- Le modèle sous forme de fichier LP
- Des chaînes de caractères sous forme de strings
- Les types classiques du *C++* (float, int, char)

De plus, pour que l'utilisateur habitué aux flux en *C++* ne soit pas dérouté, il est possible de chaîner les flux. Techniquement, cela revient à retourner simplement, par les opérateurs, des références vers l'objet courant.

Sortie PostScript :

Le principal intérêt du format PostScript est que l'on peut avoir une abstraction totale du système pour la sortie. C'est-à-dire que sur tout système, le fichier sera imprimable sans dépendre du système : le format PostScript étant supporté par toutes les imprimantes depuis une vingtaine d'années.

Bien entendu, l'utilisation interne des fichiers PostScript est quasiment identique à celle des fichiers HMLT et LaTeX.

3.3.3 Le contrôle de la part de l'utilisateur

Un aspect crucial de notre modeleur est le fait qu'à chaque instant, lors de l'utilisation, il est possible d'obtenir toutes les informations contenues dans le emphModel.

Pratiquement, cela revient à pouvoir faire des tests afin d'améliorer le modèle implémenté pour maximiser les performances en terme de résultats et optimiser le modèle en général (suivant des cas particuliers).

3.4 Interfaçage Python

Arrivés au moment où nous avions une bibliothèque utilisable (comme l'utilisation de CPLEX en mode CALLABLE), nous nous sommes intéressés à la façon dont notre bibliothèque serait utilisée. A ce moment là, elle n'était utilisable qu'en *C++* , c'est-à-dire que tout la phase de compilation et édition des liens était inévitable pour la moindre petite modification sur le code : cette phase est longue surtout et fait perdre beaucoup de temps si on doit résoudre un petit système.

Nous avons donc tout d'abord décidé de créer un langage de script qui serait orienté programmation linéaire basé sur le langage AMPL.
Or, la conception d'un interpréteur AMPL n'aurait pas été réalisable en si peu de temps. Dès lors, nous avons décidé de se baser sur un langage de script moderne existant : le Python.

Le langage Python :

Python[6] est un langage portable, dynamique, extensible, gratuit, qui permet (sans l'imposer) une approche modulaire et orientée objet de la programmation.

3.4.1 Nos motivations pour le choix du Python

Le choix du Python n'est pas anodin. En effet, nous avions des conditions nécessaires à respecter. Notre modeleur utilisant très largement la notion d'héritage et la surcharge d'opérateurs, il nous fallait un langage orienté objet qui permet la surcharge d'opérateurs (pour facilité l'écriture).

Une fois le langage déterminé, il ne nous restait plus qu'à faire le portage de la bibliothèque en Python.

3.4.2 Analyse des différents types de portages

Pour porter notre modeleur en python, nous aurions principalement pu utiliser trois manières différentes :

- Utiliser la bibliothèque fournie avec Python (python-2.xx-dev) et réécrire toutes les fonctions à la main.
- Utiliser SWIG
- Utiliser Boost.Python

Tout d'abord, il n'était pas question de tout réécrire à la main en gérant tout ce que Python nécessitait, cela aurait pris beaucoup trop de temps. En effet, il aurait fallu dupliquer notre code en y ajoutant une compatibilité de format avec l'interpréteur Python, c'est-à-dire redéfinir tous les types utilisés pour une sortie quelconque.

Notre choix s'est donc porté vers SWIG qui est un outil de développement qui permet d'interfacer un programme écrit en C/*C++* avec de nombreux langages de plus haut niveau (Perl, Java, etc.).
Mais comme il est destiné à interfacer de nombreux langages, il n'est pas spécifique au *C++* . Il y a donc naturellement des manques, notamment la conversion entre tous les types de la STL doivent être explicites et la surcharge d'opérateurs n'est pas toujours bien supportée.

[6]Langage de script, interpréteurs disponibles à l'URL : http ://www.python.org

3.4.3 Utilisation du framework Boost.Python

Boost.Python[7] est un framework qui permet d'interfacer *C++* et Python. Le code pour générer l'interface est assez simple d'utilisation, mais quelques limitations dues à Python se sont révélées : il est impossible de retourner des pointeurs (hors fonctionnement interne).

Son utilisation étant relativement simple, la première difficulté fût l'installation de Boost (compilation des bibliothèques) pour qu'il prenne en compte des version de Python supérieures à la version 2.2. En effet, il y a un problème d'Unicode incompatible entre les version inférieures à 2.2 et les nouvelles versions.

Cette phase passée, nous avons donc porté la totalité du code de notre modeleur en déclarant pour chaque classe, une fonction d'exportation qui contenait le code python généré par Boost.Python.

Voici un exemple de création d'un interfaçage pour Python à partir d'un code source *C++* en utilisant Boost.

[7] Boost.Python est disponible sur le site de Boost : http ://www.boost.org

Fichier "Voiture.cpp":

```
class Voiture {
  float niveauEssence;
public:
  Voiture(const float& _e = 0.0) : niveauEssence(_e) {};
 ~Voiture() {};

  void SetEssence(const float& _e)
  {
    niveauEssence = _e;
  }

  float GetEssence() const {return niveauEssence;}

  Voiture& operator+=(float _e)
  {
    niveauEssence += _e;
  }

};

// Code de génération du code Python associé
BOOST_PYTHON_MODULE(Voiture)
{
  class_<Voiture>("Voiture",init<optional<float> >)
      .def("GetEssence", &Voiture::GetEssence)
      .def("SetEssence", &Voiture::SetEssence)
      .def(self += float())
  ;
}
```

Une fois compilé, passé en bibliothèque dynamique[8], nous obtenons un fichier *Voiture.so*.

```
>>> import Voiture
>>> v = Voiture.Voiture(42) # package.constructeur
>>> v += 9
>>> print "v est une voiture avec %f d'essence" % v.GetEssence()
```

Cet exemple permet de créer un objet Voiture dont le niveau d'essence est de 42. L'affichage du niveau peut donc se faire en utilisant la méthode *Voiture : :GetEssence*.
On voit de plus la facilité avec laquelle la surcharge d'opérateurs est faite. En fait, le principal problème est dû au polymorphisme faible : Boost.Python ne reconnaît pas les signatures différentes. La solution apportée à ce problème est de renommer les méthodes lors de la création de la bibliothèque dynamique Python. Ainsi aucune méthode n'a le même nom et le problème ne se pose plus.

[8]Voir glossaire

Troisième partie

Résultats et discussion

Chapitre 4

Réalisation et résultats

Une fois le produit terminé, nous avons procédé à une série de tests pour valider notre travail.

4.1 Le produit réalisé

Nous avions comme objectif de créer un modeleur. Au final, nous avons une collection d'outils permettant la modélisation de programme linéaire.

4.1.1 La bibliothèque C++ réalisée

Notre travail a abouti à la création d'une bibliothèque, constituée d'un ensemble de fichiers d'entête et d'une bibliothèque statique. Cette bibliothèque est configurable et adaptable au système d'exploitation sous condition de la recompiler.

Ceci est possible grâce à un script réalisé en Python qui génére selon des options un fichier de configuration et un fichier de construction de la bibliothèque adéquats. Comme nous n'avons utilisé que des bibliothèques disponibles par défaut du *C++* , la compatibilité UNIX/Windows est ainsi totalement assurée.

Nous avons donc rempli l'objectif principal et les contraintes de compatibilité inter système que nous nous étions fixées. Nous n'avons pas développé de GUI[1] car sa conception n'était pas triviale et que le portage sous Python a diminué considérablement ce qu'il aurait pu apporter.

4.1.2 Le portage sous Python

Un intérêt très important de notre modeleur est le fait qu'il est interfacé avec un langage de script déjà existant et reconnu dans le monde de la programmation scientifique.

Une fois la bibliothèque utilisable en Python réalisée, nous avons été surpris de la rapidité de ce langage. En effet nous avions quelques appréhensions quant à l'utilisation d'un langage de script, un langage de script étant beaucoup plus lent qu'un langage compilé tel que *C++* .

4.1.3 La documentation fournie

Dans l'optique de facilité l'utilisation de notre modeleur, nous avons mis à disposition de l'utilisateur un manuel d'utilisateur, un manuel de référence ainsi que des exemples de codes.

[1] Traduction : Interface Graphique pour l'Utilisateur

Le manuel utilisateur est un document HTML rédigé sous la forme d'une FAQ[2]. Ce sont en fait des questions types auxquelles nous avons répondu en détaillant à chaque fois les points qui pourraient être obscurs.

Le manuel de référence répertorie l'ensemble des classes, les fonctions et les constantes de la bibliothèque en explicitant le rôle de chacune. Cette référence est générée automatiquement depuis notre code source grâce au logiciel Doxygen, elle est destinée principalement à favoriser la maintenance de la bibliothèque.

Des exemples de codes sont mis à disposition pour donner des exemples complets de l'utilisation de notre modeleur à l'utilisateur. Parmi ceux-ci, on retrouve le problème du « cutting-stock »et un problème d'ordonnancement de flotte aérienne.

4.2 Les tests et la validation

4.2.1 Tests de rapidité

Nous avons testé la rapidité de notre modeleur suivant trois critères.

Le chargement de fichier LP

Nous avons chargé un fichier LP qui traite du problème d'affectation sous contraintes de ressources. Ce fichier représente un problème d'une dimension de cinq cent milles variables pour quelques milliers de contraintes. Le modeleur actuel est capable de charger le fichier LP et de le convertir en *Model* en une cinquantaine de secondes, qui est une durée raisonnable : CPLEX charge le même fichier LP en vingt-six secondes et notre phase de chargement du fichier LP ne met que trois secondes en plus.[3] La conversion sous notre *Model* prend le même temps que le chargement en partie à cause de la reconnaissance des variables, mais aussi parce que le modeleur simplifie automatiquement les expressions des contraintes et de la fonction objectif.

L'écriture d'un fichier LP

L'écriture dans un fichier LP est une tâche critique car la résolution requiert une écriture dans un fichier LP. La sortie du fichier LP met beaucoup moins de temps que le chargement, temps négligeable devant la résolution.

La génération de fichier LP

Nous avons testé la rapidité de notre modeleur quant à la génération d'un fichier LP à partir d'un code source. Malheureusement, le temps de génération dépend de trop de paramètres pour pouvoir être vraiment évalué en terme de performances. Il dépend essentiellement des structures de données utilisées et de l'habileté de l'utilisateur à utiliser nos outils. Nous avons effectué principalement des tests sur le problème du flot maximum. Pour générer un tel problème avec deux millions de variables et quatre-vingt mille contraintes, le modeleur a besoin d' environ sept minutes réparties à peu près équitablement entre la phase de création du programme linéaire en mémoire et la phase d'écriture du fichier LP.

[2]Frequently Asked Questions
[3]Tester sur un serveur dans des conditions similaires

4.2.2 Potable vis-à-vis de la concurrence

Les modeleurs commerciaux intègrent généralement beaucoup plus d'outils que notre modeleur, mais nous avons développé les outils de bases qui se retrouvent dans tous les modeleurs. Les outils que nous avons a développés et que l'on retrouve dans les autres modeleurs sont :

- les conteneurs(vecteurs, nombres, variables)
- une encapsulation du Model
- la gestion automatique des expressions
- les simplifications
- un langage de script
- divers sorties, en particulier HTML.
- des structures d'algorithmes

Les outils intéressant que nous n'avons pas développés sont entre autres :

- la gestion des graphes
- la gestion d'un dual en parallèle
- un langage propre au modeleur
- un débogueur

4.3 Le bilan du projet

Nous avons rempli nos objectifs principaux mais notre projet ne correspond pas entièrement au cahier des charges au niveau des objectifs secondaires.

4.3.1 Les objectifs non résolus

Ces problèmes sont principalement la gestion du dual et des bibliothèques CPLEX.

Gestion du Dual : Le problème est de travailler à la fois sur la forme duale en ayant les répercussions sur la forme primale. Il est dû à la structure que nous utilisons qui nous aurait obligé à faire des duplications de données ou des échanges entre des *Models* .

Utilisation du modèle (avec solveur) sous Linux : Comme nous avons la bibliothèque CPLEX uniquement disponible sous station Sun, nous ne pouvons intégrer la bibliothèque CPLEX à notre modeleur. C'est ce qui nous a poussé à faire des sauvegardes des fichiers LP puis des appels système pour les résoudre : sans cela, le modeleur serait plus rapide.

4.3.2 Les améliorations possibles

Une importante perte de temps évitable se situe lors des simplifications des expressions. En effet à chaque simplification d'une expression, il y a création et destruction d'un conteneur associatif, opération qui devient coûteuse lorsque le nombre de variables augmente.

La gestion des flux pourrait être améliorée car elle reste encore basique. De nombreuses options pourraient être ajoutées aux fonctions déjà existantes et de nouvelles méthodes pourraient être créées.

La robustesse du modeleur pourrait encore être améliorée grâce à une meilleure gestion des fichiers d'échanges et des appels systèmes mais aussi grâce à une gestion mieux contrôlée de la généricité dans les structures de données et dans les algorithmes.

Chapitre 5

Discussion

5.1 Moyens utilisés pour conduire le projet

5.1.1 La STL, les compilateurs

La STL : Nous avons notamment utilisé la structure de *map* de la STL. C'est un conteneur associatif qui a une structure d'arbre binaire équilibré. Nous avons aussi massivement utilisé les structures classiques telles que les flux, les chaînes de caractères, les listes chaînées et les vecteurs.

Les compilateurs : La gamme de compilateurs utilisée est celle des G++ de 3.3.x à 3.4, autant sous Windows que sous Linux. Les options de compilations utilisées nous ont forcé à avoir un code source qui respecte le standard du *C++* et donc, compilable par n'importe quel autre compilateur *C++* .

5.1.2 Les rendez-vous fréquents avec le chef de projet

Lors du développement du noyau du modeleur, nous avons rencontré notre chef de projet au moins une fois par semaine. Nous échangions aussi fréquemment des courriers électroniques en y joignant l'état en cours du projet. Les rendez-vous nous ont permis de comprendre les besoins, de modifier le cahier des charges suivant notre progression et surtout d'avoir la critique constructive de notre chef de projet.

5.1.3 Boost.Python

Pour aborder sereinement Boost.Python, il faut en comprendre la philosophie. Boost.Python est principalement utilisée pour faire du portage d'applications professionnelles, en se rendant sur les mailings list, nous nous sommes aperçu qu'il y avait beaucoup de personnes de grandes entreprises telles que HP Inc. ou le CERN qui y posaient les mêmes questions que nous.

Nous nous sommes donc basés sur la documentation existante, disponible sur le site Internet de Boost.Python ainsi que sur la liste de diffusion destinée aux utilisateurs de ce framework en y posant quelques questions.

Nous avons d'ailleurs eu la chance d'échanger quelques courriers avec un des mainteneurs de Boost.Python qui nous a débloqué alors que nous n'avions pas réussi à résoudre un problème tout seuls.

5.2 Méthodologie et déroulement du projet

5.2.1 Les prévisions et le travail en parallèle

Après l'établissement du cahier des charges, nous avons décidé de passer tout d'abord du temps sur la conception théorique de la structure du modeleur afin de s'accorder sur la vision du projet.

Puis nous avons divisé le travail en deux parties relativement indépendantes que nous avons codées séparément pour accélérer le développement. Grâce à des structures homogènes, nous avons pu relier les deux parties de façon la plus efficace. Une fois ces parties reliées, nous avons effectué des tests sur ce qui allait devenir le noyau du modeleur.

Ensuite est venu le problème du langage de script et celui de la création des outils associés au modeleur.

5.2.2 La logistique employée

Nous avons travaillé sur deux ordinateurs séparés tout en communiquant constamment l'un avec l'autre et en se partageant le travail effectué. Cela a permis d'être continuellement au courant de la progression de l'autre et de pouvoir corriger, critiquer et apporter de nouvelles idées.

Le problème de cette méthode de travail est l'uniformisation des versions du projet une fois les parties indépendantes reliées. Toutefois, ce problème ne nous a pas trop ralenti et nous n'avons connu que peu de pertes de code.

Nous aurions pu résoudre ce problème en installant un logiciel comme un serveur *CVS* qui permet de mettre à jour intelligemment les codes sources.

5.2.3 La méthode de validation du travail

Lors du développement, nous avons essayé au maximum de vérifier notre code en le testant quasiment fonction par fonction tout en s'assurant l'intégrité du travail global.

Pour réaliser certains tests, nous nous sommes inspirés de la méthode de développement eXtreme Programming, et notamment des tests unitaires. Ce genre de tests a été effectué sur des routines critiques, notamment pour la détection des nombres de tous types ainsi que pour la séparation de monôme en coefficient multiplié par une variable.

Exemple de test unitaire

Un test unitaire est un test que l'on fait alors que l'on connaît déjà le résultat.

Voici un exemple de test unitaire sur une fonction très simple : le produit de deux entiers.

```
int mul(int a, int b) {return (a*b);}

bool TEST_UNITAIRE(int test, int valeur) {return test == valeur;}

cout << TEST_UNITAIRE(mul(1,2),2)     << endl; cout <<
TEST_UNITAIRE(mul(5,5),25)    << endl; cout <<
TEST_UNITAIRE(mul(9,-5),-45)  << endl; cout <<
TEST_UNITAIRE(mul(0,1<<15),0) << endl;
```

On s'arrange ainsi pour tester tous les cas possibles. Ce genre de tests permet d'être sûr que la fonction retournera toujours un bon résultat (en fait, cela dépend des tests réalisés par le programmeur).

5.3 Problèmes rencontrés

5.3.1 La procédure d'appel des solveurs

Notre premier souhait concernant l'appel des solveurs était de le faire grâce aux librairies statiques mis à disposition par ILOG aux détenteurs de licences CPLEX. Le problème est venu du fait que ces librairies n'étaient disponibles que sous SunOS et donc inutilisable sur les autres plateformes. Nous prévoyions de convertir les fichiers dits objets de la plateforme Sun en fichier objets pour d'autres plateformes mais nous avons dû très vite renoncer devant l'ampleur de la tâche. En effet, cette conversion pourrait très bien constituer un projet de cent heures de deuxième année. Nous avons donc préféré nous rabattre sur un autre solution.
Un problème est aussi survenus lors de l'appel de GLPK sur des gros fichiers : GLPK n'accepte pas les fichiers LP dont les lignes mesurent plus de 255 caractères. Ceci fait perdre encore un peu plus de temps lors de l'appel de ce solveur.

5.3.2 L'utilisation de Boost.Python

Boost.Python est un framework très bien fait et simple d'utilisation. Quelques problèmes sont survenus au niveau de l'encodage de caractères qui avait changé lors du passage de Python 2.2 à Python 2.3, mais les gros problèmes de Boost.Python a été la surcharge des opérateurs. En effet, comme les langages Python et *C++* ne fonctionnent pas de la même façon même s'ils sont tous deux orientés objets, la façon d'écrire les expressions sous en *C++* ne s'est pas retrouvé en Python. De plus, la surcharge des opérateurs existe en Python mais n'est pas identique à celle des opérateurs de *C++* . Ce qui fait que les expressions linéaires et les contraintes se sont retrouvées très mal gérées sous Python.

5.3.3 La forme duale

L'extraction de la forme duale d'un programme linéaire est déjà un problème en soi. En effet, si le programme linéaire n'est pas normalisé, ce qui représente la majorité des cas, les algorithmes sont vraiment très complexes. Nous avons préféré passer par une phase de normalisation avant d'extraire la forme duale d'un programme linéaire. Mais notre objectif était de travailler en même temps sous la forme duale et primale.

La recherche de solution n'a été que peu fructueuse, en sachant que notre structure de matrice creuse pour stocker les contraintes rendait la tâche encore plus difficile. De toutes les façons, l'intégration de cet outil à notre modeleur nous aurait obligé soit à recoder toutes les fonctions mais pour la forme duale, soit à changer la structure interne de notre modeleur et donc à modifier toutes les fonctions, soit à dupliquer les données et créer un système de communication entre les différentes formes tout en assurant l'unicité du programme linéaire, opération lourde tant lors de sa mise en place que lors de son exécution

Conclusion

Les objectifs principaux ont été remplis et la plupart des objectifs secondaires aussi. Nous avons été plus loin que nous le prévoyions et donc le résultat est satisfaisant, malgré les problèmes concernant les objectifs secondaires non remplis et les améliorations encore possibles. Le portage sous Python ne s'est pas déroulé aussi bien que prévu mais l'expérience aura été intéressante. La conception du modeleur, et en particulier des structures de données et des structures d'algorithmes, a été enrichissante d'un point de vue maîtrise du langage *C++* et du langage Python.

Néanmoins, nous pouvons déjà nous réjouir car notre modeleur sert actuellement à un binôme qui fait un projet sur l'étude des flots sur des graphes complets. Notre modeleur leur permet de générer facilement des solutions à leurs problèmes.

Le modeleur est actuellement exploitable mais reste largement incomplet vis-à-vis des modeleurs professionnels, ce qui est tout à fait acceptable au vue de la durée du projet. Néanmoins, ce modeleur pourra être exploité par l'ISIMA, notamment tant au niveau de l'enseignement que de la recherche. Il pourra aussi être amélioré pour devenir vraiment attractif si la partie langage de script abouti. Outre les améliorations concernant les flux et les structures de données, le modeleur gagnerait peut-être à avoir un GUI, réalisé par exemple sous Qt[1] pour conserver la portabilité.

[1] Bibliothèques de composants graphiques multiplateforme développée par Trolltech.

Annexes

Exemple : étude du flot max

Ce problème correspond à un problème fréquemment rencontré en télécommunication : le problème de routage.

Description du problème

On considère un graphe orienté $G = (S, A)$. Les arcs ne diposent pas de capacité initiale mais vont devoir être choisie parmi un ensemble d'équipement $E = \{(c_e, u_e)\}$ où $u_e \geq 0$ désigne une capacité en communication et $c_e \geq 0$ leur coût d'acquisition.

Le problème est la mise en place d'une structure de communication permettant à un ensemble K d'entités de communiquer. L'ensemble des commodités K est constitué de triplets (o_k, d_k, q_k) représentant respectivement le sommet d'origine des flots, le sommet de destination et la quantité de flot à faire transiter sur le réseau.

Ces commodités définissent un multiflot, à savoir un ensemble de flux élémentaires non miscibles et dont la somme sur un arc ne doit pas dépasser sa capacité.

Le but du problème est de définir une installation de coût minimum permettant de satisfaire toutes les commodités.

Modèle mathématique

Nous allons avoir besoin de deux types de variables pour chaque arc. Des variables continues $y_a^k \geq 0$ vont représenter les flots sur le graphe pour chaque commodité de K et des variables booléennes $x_a^e \in \{0\}$ qui vont représenter le choix de l'équipement $e \in E$ pour un arc $a \in A$.

Nous aurons trois types de contraintes :

1. des contraintes de conservation de flux.
2. des contraintes d'unicité d'affectation de l'équipement
3. des contraintes de capacité

Mise en équation

On notera par la suite pour que, $\forall i \in S, \Omega_i^- = \{(j, i) \in A\}$ est l'ensemble des arcs entrants de i et $\Omega_i^+ = \{(i, j) \in A\}$ l'ensemble des arcs sortants de i. Nous allons travailler sur un graphe non orienté pour faciliter la tâche du solveur.

Le programme s'écrit alors

$$
\begin{array}{ll}
\text{Min z} = \sum_{a\in A}\sum_{e\in E} c_e x_a^e & \\
\text{s.c.} & \\
\sum_{a\in\Omega^+_{o_k}} y_a^k = q_k & \forall k \in K \\
\sum_{a\in\Omega^-_{o_k}} y_a^k = 0 & \forall k \in K \\
\sum_{a\in\Omega^-_i} y_a^k - \sum_{a\in\Omega^+_{o_i}} y_a^k = 0 & \forall k \in K, \forall i \in S\backslash\{o_k, d_k\} \\
\sum_{a\in\Omega^-_{d_k}} y_a^k = q_k & \forall k \in K \\
\sum_{a\in\Omega^+_{d_k}} y_a^k = 0 & \forall k \in K \\
 & \\
\sum_{e\in E} x_a^e \leq 1 & \forall a \in A \\
x^e_{(i,j)} = x^e_{(j,i)} & \forall a = (i,j) \in E, \forall e \in E \\
\sum_{k\in K} y_a^k \leq \sum_{e\in E} u_e x_a^e & \forall a \in A \\
 & \\
y_a^k \geq 0 & \forall a \in E, \forall k \in K \\
x_a^e \in \{0,1\} & \forall a \in E, \forall e \in E
\end{array}
$$

Code *C++* correspondant

```
/*-------------------------------------------------------------------*/
/* Fichier d'exemple de modélisation de programme linéaire           */
/* à l'aide de la bibliothèque statique Modelib                      */
/* Auteurs: Quentin Lequy quentin.lequy@poste.isima.fr               */
/*          et Romain Gaucher                                        */
/*                                                                   */
/*  Description: Génération du fichier et résolution du fichier LP   */
/*  correspondant au problème du flot maximal sur un graphe.         */
/*  C'est un problème de routage multiflots avec commodités et       */
/*  et dont le but est de minimiser le cout de l'équipement          */
/*                                                                   */
/* compilation: > g++ flotmax.cpp -o flotmax modelib.a -Wall         */
/*-------------------------------------------------------------------*/
#include "../Modelib.h" //on inclut Modelib #include <fstream>
//on doit gérer des fichiers d'entrée #include <sstream>      //on
doit générer des noms de contraintes facilement

using namespace std;   //on se place dans l'espace de nommage
standart

int main(int argc, char *argv[]) {
  cout << "Creation des données" << endl;

  /* Création des arcs */
  Ensemble< Nuplet<2,unsigned> > arcs;
  //on crée un ensemble de couple
  ifstream file("topo.dat");
  //on ouvre le fichier topo.dat dans lequel se trouve la topologie du →
     ↪  graphe
  unsigned nbsommets = 0;  //on déclare le nombre de sommets
```

```
unsigned nbarcs = 0;      //on déclare le nombre d'arcs
file >> nbsommets;    //on lit le nombre de sommets dans le fichier
file >> nbarcs;       //on lit le nombre d'arcs dans le fichier
for(unsigned i=0;i<nbarcs;++i)
{
  file >> arcs; //on lit les arcs
}
//On duplique les arcs parce que l'on résout le problème non orienté
for(unsigned i=0;i<nbarcs;++i)  //Pour tous les arcs
{
    Nuplet<2,unsigned> tmp = arcs.Get(i);
    //On récupère le couple correspondant à l'arc en cours
    unsigned swap = tmp[0]; //on permute les variables
    tmp[0] = tmp[1];
    tmp[1] = swap;
    arcs.Add( tmp ); //on créé le nouvel arc
}
file.close(); //on ferme le fichier

/*Création des commodités*/
Ensemble< Nuplet<3,unsigned> > commo;
//on crée un ensemble de triplet représentant
//le point de départ, la destination et la quantité de flux
file.open("commo.dat");
unsigned nbcommo = 0;
file >> nbcommo;
for(unsigned i=0;i<nbcommo;++i)
{
  file >> commo;
}
file.close();

/*Création des equipements*/
Ensemble< Nuplet<2,unsigned> > equip;
//on crée un ensemble de couple (cout,capacité)
file.open("equip.dat");
unsigned nbequip = 0;
file >> nbequip;
for(unsigned i=0;i<nbequip;++i)
{
  file >> equip;
}
file.close();

cout << "Création du PL" << endl;

Model flot; //On déclare le modèle

flot.SetProblemName("Probleme du flot maximal sur un graphe");

/*Définition des variables*/
vector< Family<2>* > y;
//On créer un vecteur de familles de variables indexées sur des
   ↪ couples
for(unsigned i=0;i<nbcommo;++i)
{
  ostringstream nom;
  nom << "y_%_%_" << i; //on créé les variables y[arc,no de
     ↪ commodités]
  //qui représentent la circulation du flux sur le graphe pour
     ↪ chaque commodités
  y.push_back(new FloatFamily<2>(flot,0.0f,Infinity,arcs,nom.str()))
     ↪ ; //On initialise chaque famille
}
```

```
vector< Family<2>* > x;
//On créer un vecteur de familles de variables indexées sur des couples
for(unsigned i=0;i<nbequip;++i)
{
  ostringstream nom;
  nom << "x_%_%_" << i;//on créé les variables x[arc,no d'equipement ]
  //qui représentent si on choisi de l'équipement i pour chaque arc
  x.push_back(new BoolFamily<2>(flot,arcs,nom.str())); //On initialise chaque famille
}


/*Définition de la fonction objectif*/
Expr obj; //on déclare une expression qui sera la fonction objectif
for(unsigned i=0;i<nbequip;++i)
//Pour chaque equipement
{
  NumVarArray t = x[i]->GetFamilyPart( arcs.all() );
  //on récupère tous les variables de la famille x[i]
  obj+= (equip.Get(i))[1]*Sum(t);
  //on ajoute à la fonction objectif le cout de l'equipement
}

flot.Minimize(obj,"obj"); //On minimisera la fonction objectif



/*Définition des contraintes de conservation de flux*/
for(unsigned i=0;i<nbcommo;++i) //pour chaque
{
  unsigned o = (commo.Get(i))[0]; //on récupère l'origine de la commodité
  unsigned q = (commo.Get(i))[2]; //on récupère sa quantité de flux
  ostringstream nom;
  nom << "c1a_" << i;
  //On créé le nom de la contrainte
  NumVarArray t = y[i]->GetFamilyPart( arcs.partie(Omega<0,2, unsigned>(o)) );
  //On prend la partie de l'ensemble des variables représentant la quantité
  //de flux sur les arcs sortant de l'origine
  flot.Add( Sum(t) ==  q, nom.str() );
  //La somme des flux sortant de l'origine est égale à la quantité de flux de la commodité
  NumVarArray u = y[i]->GetFamilyPart( arcs.partie(Omega<1,2, unsigned>(o)) );
  //On prend la partie de l'ensemble des variables représentant la quantité
  //de flux sur les arcs entrant de l'origine
  flot.Add( Sum(u) ==  0, nom.str()+'2' );
  //La somme des flux entrant dans l'origine est nulle
}

for(unsigned i=0;i<nbcommo;++i) //Pour chaque commodité
{
  unsigned o = (commo.Get(i))[0]; //on récupère l'origine de la commodité
  unsigned d = (commo.Get(i))[1]; //on récupère la destination de la commodité
```

```
  for(unsigned j=0;j<nbsommets;++j) //pour chaque sommets
  {
      if(j!=o && j!=d) //différents de l'origine et de la →
         ↪ destination de la commodité
      {
          ostringstream nom;
          nom << "c1.b_" << i << "_" << j;
          //On crée le nom de la contrainte
          NumVarArray t = y[i]->GetFamilyPart( arcs.partie(Omega →
             ↪ <0,2,unsigned>(j)) );
          //On prend la partie de l'ensemble des variables →
             ↪ représentant la quantité
          //de flux sur les arcs sortant du sommet j
          NumVarArray u = y[i]->GetFamilyPart( arcs.partie(Omega →
             ↪ <1,2,unsigned>(j)) );
          //On prend la partie de l'ensemble des variables →
             ↪ représentant la quantité
          //de flux sur les arcs entrant dans le sommet j
          flot.Add( Sum(t) - Sum(u) == 0, nom.str() );
          //Le flux est conservé, ie la quantité de flux rentrante →
             ↪ est égale à la
          //quantité de flux sortante
      }
  }
}

for(unsigned i=0;i<nbcommo;++i) //Pour chaque commodité
{
  unsigned d = (commo.Get(i))[1]; //on récupère la destination de →
     ↪ la commodité
  unsigned q = (commo.Get(i))[2]; //on récupère sa quantité de flux
  ostringstream nom;
  nom << "c1c_" << i;
  //on créé le nom de la contrainte
  NumVarArray t = y[i]->GetFamilyPart( arcs.partie(Omega<1,2, →
     ↪ unsigned>(d)) );
  //On prend la partie de l'ensemble des variables représentant la →
     ↪ quantité
  //de flux sur les arcs entrant dans la destination
  flot.Add( Sum(t) == q, nom.str() );
  //La quantité de flux entrant dans l'origine est égale à la →
     ↪ quantité de flux
  NumVarArray u = y[i]->GetFamilyPart( arcs.partie(Omega<0,2, →
     ↪ unsigned>(d)) );
  //On prend la partie de l'ensemble des variables représentant la →
     ↪ quantité
  //de flux sur les arcs sortant de la destination
  flot.Add( Sum(u) == 0, nom.str()+'2' );
  //La quantité de flux entrant dans la destination est nulle
}

/*Déclaration des contraintes d'unicité de l'équipement*/
for(unsigned i=0;i<nbarcs*2;++i) //Pour chaque arc(on les a →
   ↪ dédoublés)
{
  Expr tmp; //on déclare une expression temporaire
  ostringstream nom;
  nom << "c2a_" << i;
  //on créé le nom de la contrainte
  for(unsigned j=0;j<nbequip;++j)
  {
      tmp += x[j]->Get(i);
```

```
        //on fait la somme des variables de choix d'équipement pour →
            ↪ cet arc
    }
    flot.Add( tmp <= 1, nom.str() );
    //cette somme ne doit pas être supérieure à 1, ce qui signifie qu' →
        ↪ il
    //n'y aura qu'un équipement par arc
  }

  for(unsigned i=0;i<nbequip;++i)//Pour chaque équipement
  {
    for(unsigned j = 0; j < nbarcs; ++j) //Pour chaque arc
    {
        ostringstream nom;
        nom << "c2b_" << i << "_" << j;
        flot.Add( x[i]->Get(j) - x[i]->Get(j+nbarcs) == 0, nom.str() ) →
            ↪ ;
        //il y a le même équipement dans les deux sens
    }
  }

  /*Déclarations de contrainte de capacité maximum sur un arc*/
  for(unsigned i =0; i<nbarcs*2 ;++i) //Pour chaque arc
  {
     //On somme la quantité de flux passant sur cet arc
     Expr tmp1;
     for(unsigned j =0; j < nbcommo; ++j)
     {
       tmp1+=y[j]->Get(i);
     }
     //On calcule la capacité de l'arc en fonction de l'équipement →
         ↪ choisi
     //sachant qu'il n'y a qu'une variable de choix d'équipement non →
         ↪ nulle par arc
     Expr tmp2;
     for(unsigned j=0; j < nbequip; ++j)
     {
       tmp2+=x[j]->Get(i)*(equip.Get(j))[0];
     }
     ostringstream nom;
     nom << "c3_" << i;
     flot.Add( tmp1 <= tmp2 , nom.str() );
     //La capacité de l'arc ne doit pas être dépassée
  }

  cout << "Resolution" << endl;

  flot.InitSolver(GLPK); //on choisit le solveur qui sera utilisé pour →
      ↪  résoudre le LP
  flot.Solve(); //On résout

  flot.SortieHTML("flotmax.html"); //On récupère les résultats sous →
      ↪ forme HTML

  //On désalloue les familles de variables
  for(unsigned i=0;i<nbequip;++i)
    delete x[i];

  for(unsigned i=0;i<nbcommo;++i)
    delete y[i];
}
```

Manuel d'utilisation

La déclaration du modèle et utilisation des variables

La bibliothèque Modelib se présente, dans le code, sous la forme d'une entité contenant le modèle s'appelant `Model`.

Il est donc nécessaire de déclarer une instance de la classe `Model` ainsi que les variables nécessaires.

Il existe deux types principaux de variables : les variables et les types numériques. Ceux-ci se divisent chacun en deux parties, les vecteurs ou les scalaires.

Les variables

Ce type de conteneur sera utilisé pour déclarer les variables dans les expressions qui seront passées par la suite dans votre modèle.

Il existe plusieurs types de variables :

Les variables Ce sont des conteneurs qui ne contiendront donc qu'une seule valeur variable. Ces variables peuvent être des `IntVar`, des `FloatVar` ou des `BoolVar` qui représentent respectivement des entiers, des réels et des booléens.

Les vecteurs de variables Ce type contient un certain nombre de variables scalaires, la taille étant passée au moment de la déclaration de ces vecteurs. Ces vecteurs variables sont des `IntVarArray`, des `FloatVarArray` ou bien des `BoolVarArray`.

Le principe de déclaration est le même pour tous les types de variables et les paramètres des constructeurs sont :

- Le Model auquel la variable sera rattachée
- La taille si c'est un vecteur de variables
- La borne inférieure des valeurs prises par la ou les variables
- La borne supérieure
- Le nom de la variable

Il est utile de spécifier à chaque fois que l'on déclare une variable un nom. En effet, grâce à ce nom, on pourra acceder rapidement à cette dernière.
La borne supérieure et la borne inférieure peuvent prendre une valeur particulière qui est Infinity.

Les types numériques

Tout comme les variables, ils se déclinent en deux catégories, les vecteurs et les scalaires.

Les types numériques Ces conteneurs contiennent un nombre définit d'un certain type : `Int`,`Float`,`Bool`.

Les vecteurs contenant ces types Ce sont des vecteur d'`Int`,`Float` ou `Bool`, donc, ce sont des `IntArray`, `FloatArray` ou `BoolArray`.

Contrairement aux variables, les types numériques n'ont pas à être rattachés au *Model* courant. Leur utilisation courante est de s'en servir comme coefficients de variables.
Pour la construction des type, il suffit donc d'indiquer :

- La taille si ce sont des vecteurs
- La valeur contenue (si c'est un vecteur, la valeur sera recopiée dans tous les éléments).

Exemple d'utilisation des types

Voici un exemple de code *C++* qui permet de déclarer des variables et des types numériques.

```
#include "Modelib.h" // Entête des fonctions et définition des
types. #include <iostream> using std::cout; ... int main() {
    Model model;

    // déclarations des bornes pour les différences variables
    float f_borneInf = -42.24, f_borneSup = 42.2442;
    int   i_borneInf = 0, i_borneSup = Infinity;
    // déclaration de la dimension -- qui est commune ici
    unsigned taille = 42;
    // déclarations des vecteurs
    FloatVarArray array_f(model,taille,f_borneInf,f_borneSup,"arrayf_" →
       ↪ );
    IntVarArray   array_i(model,taille,i_borneInf,i_borneSup,"arrayi_" →
       ↪ );
    // déclaration de variables
    FloatVar     f(model,f_borneInf,f_borneSup,"f");
    // déclaration d'un scalaire
    Bool         b(0); // Booléen initialisé à False=0
    // Vecteur de scalaires
    IntArray     i(taille); // Les éléments seront tous mis à 0

    ... traitement ...

    // affectation des variables (valables pour tous les types)
    FloatVarArray tmp = array_f;
    // quelques informations
    cout << tmp.GetName() << " de taille" << tmp.Size();
    // affiche la première composante
    cout << "son 1er elmt vaut " << tmp[0];
}
```

Et le même type exemple en Python :

```
from PL import * # chargement des éléments de notre bibliothèque
                 # dans l'environnement courant de Python
model = Model() array_f = FloatVarArray(model,10,0,42.42,'arrayf_')
array_i = IntVarArray(model,10,0,42.42,'arrayi_')

i = IntVar(model,10)

j = i # le typage automatique de Python fait que j est un IntVar
```

La définition d'une fonction objectif

La fonction objectif étant tout simplement une expression, il suffit d'appeler `Model::Maximize` ou `Model::Minimize` avec en paramètre l'expression pour que la fonction objectif soit créée.

```
Model m;
//Déclarations des variables
BoolVar x1(m,"x1"); BoolVar x2(m,"x2"); BoolVar x3(m,"x3"); BoolVar
x4(m,"x4");

//Création du problème linéaire
m.Maximize(4*x1 + 5*x2 + 6*x3 + 7*x4, "Nom");
// ou
// m.Minimize(*x1 + 5*x2 + 6*x3 + 7*x4);
```

La Manipulation des contraintes

Afin de facilité la lecture du code source des modèles utilisés pour résoudre des programmes linéaires, nous avons décidé d'adopter une écriture naturelle pour l'ajout des contraintes. En effet, il suffit de l'ajouter au *Model* comme cela :

```
Model model;
// déclaration des variables
FloatVar X(model,0,Infinity,"var_x"); FloatVar
Y(model,0,Infinity,"var_y");

model.Add(X+2*Y <= 10, "Nom_Contrainte");
```

Il est à noter que la fonction `Model::Add` retourne un type `Constraint`. Ce type, représentant une contrainte, est nécessaire si vous souhaitez utiliser les coefficients de lagrange associés aux contraintes.

Le chargement de fichier de type LP

Dans Modelib, nous avons décidé de n'utiliser que le format LP. Ce format a été choisi pour plusieurs raisons :

1. Sa représentation ressemble a la forme canonique d'un LP
2. Le format est très utilisé
3. CPLEX et GLPK (glpsol) l'utilisent
4. Il est naturel a écrire, et très facile a comprendre !

Pour charger un fichier LP, il suffit d'appeler la fonction `Model::LoadLP`. Elle s'utilise comme cela :

```
Model model;
// Chargement du fichier qui est dans le repertoire courant
bool chargement = model.LoadLP("./flot.lp"); cout << model; //
affichage du fichier LP
```

A noter qu'il y a une option mise à false par défaut. Cette option permet d'afficher des informations lors du chargement du fichier.

```
model.LoadLP("./flot.lp",true);
```

Vous avez vu que l'on peut afficher le fichier LP directement dans la sortie standard, il est donc possible d'enregistrer le model à chaque instant comme cela :

```
#include "Modelib.h" #include <fstream> using std::ofstream; ... int
main() {
    Model model;

    ... opérations sur le model ...

    ofstream out("sortie.lp");
    out << model;
    out.close();
}
```

Les expressions

La gestion des expressions est un point clé de notre modeleur. Grâce à celles-ci, les écritures de contraintes, de fonctions objectifs se révèlent très simples.

Les expressions s'utilisent sur les types de Modelib même s'il est possible de manipuler aussi des types du *C++* .

En fait, les expressions sont utiles dès que les contraintes ou la fonction objectif ne peuvent pas s'exprimer comme le produit scalaire d'un `NumVarArray` avec un `NumArray`

Imaginons un problème dont les variables sont :

```
Model model; IntVarArray  affect(model,N,0,Infinity,"a_"); IntArray
cout(N); IntArray     profit(N);

// Construction des variables liées au problème
Initialisation(cout,profit);
```

Nous avons dont un système classique, avec un vecteur *affect* qui représente nos variables et deux vecteurs *cout* et *profit* qui déterminent le système : en effet, le but de ce problème est de :

$$\min_{\forall i}(cout[i] - profit[i]) * affect[i] \tag{5.1}$$

Or, on ne peut pas se permettre d'écrire

```
model.Minimize((-cout + profit) * affect, "fct_obj");
```

car le modèle ne gère pas ces opérations (- et + entre deux entités Array).

Il faut donc écrire séparément :

```
IntArray CMP(N); for (unsigned i=0 ; i<N ; i++) {
    float v = cout[i].GetValue() - profit[i].GetValue();
    CMP[i] = static_cast<int>(v);
}

// On minimise CMP[i] * affect[i]
model.Minimize (
  affect * CMP
);
```

On peut remarquer qu'une expression est générée à partir d'un produit scalaire d'un vecteur de variables avec un vecteur de scalaires.

Nous avons donc vu comment générer une fonction Objectif assez simplement en un foncteur d'*Algorithm.h*. Cela cache en fait la construction d'une expression etc.

Voilà comment palier la construction d'une expression en la cachant via un algorithme. Construisons maintenant une contrainte qui sera la somme des éléments d'*affect* si quelques contraintes sont vérifiées.

Formellement :

$$C_i = \sum_{a \in affect, test1[a]=test2[a]} affect[a] = MAX\_AFFECT \tag{5.2}$$

```
BoolArray test1(N,0); // tout initialisé à Faux==0 BoolArray
test2(N,0);

... initialisation des valeurs dans les test1 ... ...  généralement,
fonction de coût, profit   ... ...     et d'autres paramètres du
système     ...

// On début un environnement de validité de expr
START_ENV_EXPR(expr) {
    // Pour tout élément de affect[a], en C++
    // on travaille sur les indices
    for (unsigned a=0;a<N;a++)
    {
      if (test1[a] == test2[a])     // Si le test est validé Alors
         expr += Expr(affect[a]);    // l'expression est modifiées
    }
    // On ajoute donc notre contrainte, une fois la construction finie
    model.Add(expr == MAX_AFFECT,"C_i");
} END_ENV_EXPR
```

C'est comme cela que l'on peut construire des contraintes plus fines que des simples produits scalaires ou sommes de vecteurs.

Si vous voulez un exemple complet sur l'utilisation des expressions, il y a un exemple (airline.cpp) fourni dans le package. Les contraintes sont toutes construites comme ci-dessus. (Il s'agit d'un ordonnanceur de flotte aérienne).

L'appel des solveurs

Pour que le solveur soit appelé correctement, il faut initialiser le solveur avant de lui demander les informations que vous désirez. Voici un exemple d'appel du solver GLPSOL, cet

exemple est compilable, il prend en paramètre un nom de fichier LP et retourne des informations relatives au problème donné.

```
//F solveLP.cpp
//C g++ solveLP.cpp -o solveLP /chemin/de/Modelib.a

#include "Modelib.h" #include <iostream> #include <string> using
namespace std;

int main(int argc, char *argv[]) {
    Model model;
    model.InitSolver(GLPK); // GLPSOL marche aussi ainsi que CPLEX si →
       ↪ vous voulez
                            // utiliser ce solveur.

    // On va charger le fichier LP passé en paramètre
    model.LoadLP(argv[1]);

    float resultat;
    resultat = model.Solve();
    cout << "Résultat = " << resultat << endl;

    // Récupération des sorties importantes
    string operation,informations;
    model.SolverInfos(operation,informations);

    // Minimisation ou Maximisation
    cout << "Le problème était de " << operation << endl;

    // INTEGER OPTIMAL etc.
    cout << "La solution trouvée est du type: " << informations << →
       ↪ endl;
}
```

Obtenir des informations sur la matrice creuse

Le modèle possède une méthode qui vous donne cette information.

```
Model model; ... chargement, traitement ... double occupation =
model.MatriceOccupation();
```

Vous pouvez aussi ressortir une image (au format PNG) de la matrice creuse. Pour cela, il faut que vous ayez une version de la librairie et un système qui supporte le png (avoir libpng12-dev etc.).

```
model.ExportSparseMatrixPNG();
```

Cette méthode permet de créer le fichier MatriceCreuse.png et rempli avec une couleur qui dépend de la valeur (rouge=négatif, bleu=positif).

Si vous n'avez pas d'image qui se crée, c'est que vous n'avez pas ajouté la définition de la constante

```
#define EXPORT_SPARSE_MATRIX_PNG
```

dans *Config.h*.

Index

Bibliographie

[D.H92] D.Holmes. *AMPL at the University of Michigan (Documentation)*, 1992.

[ILO95] ILOG. *CPLEX User's Manual*. CPLEX Optimization Incorporated, 1995.

[RF93] Brian W. Kernighan Robert Fourer, David M. Gay. *AMPL : A Modeling Language for Mathematical Programming*. Brooks/Cole Publishing Company, 1993.