HAI606I - Projet de Programmation 2
L3 - Licence Informatique
Faculté des Sciences de Montpellier

Cédric Cahuzac
Enzo Goulesque
Lorenzo Puccio
Margaux Renoir
Tom Simula

Encadrants : David Delahaye, Hinde Bouziane, Julie Cailler, Simon Robillard

13 mai 2022

Table des matières

Introduction générale **3**

1 Gestion du groupe **4**

2 Simplexe **6**
2.1 Description du simplexe 7
2.2 Exemple 11
2.3 Retour à la Base 12

3 Séparation & évaluation **14**
3.1 Implémentation de l'algorithme de séparation & évaluation 14
3.2 La concurrence dans l'algorithme de séparation & évaluation 15
3.3 Exemple 15

4 Optimisations **18**
4.1 Simplexe général : incrémentalité et programmation dynamique 18
4.2 Exemple d'utilisation de la programmation dynamique 19
4.3 Séparation et évaluation : contraintes supplémentaires 20
4.4 Exemple de Coupure de Gomory 23

5 Interface avec Goéland **24**
5.1 Le prouveur Goéland 25
5.2 Présentation des règles du prouveur Goéland 26
5.3 Implémentation dans Goéland 27
5.4 Règles arithmétiques : quotient et reste 28
5.5 Jonction Goéland - procédures de décision 30
5.6 Exemple de Jonction 31
5.7 Les tests 32
5.8 Exemple complet d'un problème arithmétique traité par Goéland 33

Bilan et conclusion **35**

Références **37**

6 Annexes **38**
Gestion du groupe 38
Optimisations 39
Interface avec Goéland 40

Table des figures

1.1 Diagramme de Gantt 5

2.1 Schémas représentant l'enveloppe des solutions d'un système linéaire avant et après ajout d'une coupure de Gomory 6

4.1 Schéma représentant la recherche d'un minimum global. [5] 21
4.2 Schéma représentant l'enveloppe convexe restreignant l'espace des solutions 22

5.1 Règles de la méthode des tableaux 25
5.2 Exemple de la résolution d'une expression avec la méthode des tableaux 26
5.3 Règles du prouveur Goéland 26
5.4 Constructeur : MakerID() 27
5.5 Structure : Id 28
5.6 Méthode : GetName() 28
5.7 Exemples d'exécution des différentes règles de quotient et de reste du prouveur . . . 29
5.8 Calcul de $-119 \bmod 13$ en Go, puis en Python 30
5.9 Exemple de tests unitaires avec le cas du uminus entier 32
5.10 Exemple de test unitaire avec le cas de la somme rationnelle négative 33
5.11 Exemple de preuve utilisant les règles du prouveur Goéland 34

6.1 Canal de discussion écrite todo_list sur Discord 38
6.2 Graphique de la quantité de "commits" sur une des branches du projet 38
6.3 Cas de l'opération "round" 40

Introduction générale

Ce rapport décrit le déroulement de notre projet de programmation dans le cadre de l'unité d'enseignement HAI606I, ainsi que son fonctionnement et ses spécificités. Ce projet a pour objectif d'utiliser les connaissances et aptitudes que nous avons acquises depuis le début de notre licence en Informatique ainsi que d'en acquérir de nouvelles par l'intermédiaire d'un nouveau langage de programmation (Go), ainsi que d'un nouveau paradigme de programmation (la programmation linéaire), et ce de façon collaborative.

Une de nos encadrantes, la doctorante Julie Cailler, développe un prouveur automatique en logique du premier ordre nommé Goéland dans le cadre de sa thèse. Il utilise une méthode de recherche de preuve connue sous le nom de méthode des tableaux. Son apport vient en grande partie de son utilisation du parallélisme pour rechercher des preuves efficacement. Bien que les règles de preuve axées sur la logique du premier ordre aient été implémentées par Julie Cailler dans Goéland, elle souhaitait également implémenter des règles de preuve axées sur l'arithmétique. L'arithmétique de Peano n'est que semi-décidable et donc inadaptée à son utilisation dans un prouveur automatique. C'est pourquoi l'arithmétique de Presburger est ici utilisée ; elle est moins puissante que l'arithmétique de Peano mais a l'avantage d'être décidable et donc mieux adaptée à son utilisation dans un prouveur automatique.

L'objectif principal du projet, réalisé au sein de l'équipe MaREL du Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier, est d'intégrer des procédures de décision pour l'arithmétique linéaire de Presburger à Goéland. Il est question ici de répondre à la problématique suivante : comment résoudre des problèmes arithmétiques par le truchement du prouveur Goéland ?

Ce rapport détaille le déroulement du projet en quatre parties différentes.

La gestion du groupe est décrite dans le chapitre 1. Il témoigne de la façon dont nous nous sommes organisés à cinq pour fournir un travail le plus complet possible. Les outils dont nous nous sommes servis pour rendre notre collaboration efficace au sein des différentes parties du projet y sont également introduits.

La partie concernant la description de l'algorithme du simplexe général et de nos choix d'implémentation se trouve dans le chapitre 2. C'est la procédure charnière du projet, car l'ensemble des mécanismes de résolution de problèmes linéaires passer par cet algorithme.

Une autre procédure développée dans ce projet au chapitre 3 est l'algorithme de séparation et d'évaluation (*Branch & Bound algorithm* en anglais). Cet algorithme s'associe à celui du simplexe en offrant la possibilité de trouver une solution entière sur les variables ainsi souhaitées plutôt qu'une solution où toutes les variables sont rationnelles. Dans ce chapitre est détaillée notre contribution au parallélisme par l'intermédiaire des goroutines.

Dans la section 4 référant aux optimisations, une attention particulière est portée aux techniques qui ont été mises en oeuvre pour accélérer la recherche de preuve. Ces optimisations complémentent les deux procédures citées plus haut.

Enfin, une partie est dédiée à l'intégration du module arithmétique au prouveur automatique Goéland dans le chapitre 5. Cette intégration est accompagnée de celle de plusieurs règles de calcul arithmétique. L'enjeu principal du projet et de l'implémentation des algorithmes du simplexe et de l'algorithme de séparation & évaluation repose ici, afin de pouvoir résoudre des problèmes de logique du premier ordre de nature arithmétique.

1 Gestion du groupe

Le début du projet pour notre groupe était en décembre 2021, au moment de la composition de notre groupe. Nous nous étions réunis en prenant compte de la diversité de nos parcours (Cursus Master Ingénierie Informatique, IUT Informatique, Licence Maths-Info, Polytech Montpellier, PACES) et des avantages offerts par leur nature. Selon la formation, chacun des membres avait des appétences particulières pour certaines constituantes du projet, certains préférant la partie théorique, d'autres la conception ou le développement de code. C'est la volonté d'apprendre des autres qui devait permettre au groupe de s'équilibrer.

La répartition du travail était au centre des priorités car nous avions une séquence de programmes à développer et non différents programmes indépendants, ce qui rendait indispensable une bonne communication. Ce diagnostic nous a permis de comprendre qu'il ne fallait pas enfermer chacun des membres du groupe dans une position définie au risque de bloquer le projet. Nous avons donc mis en place un système de missions hebdomadaires qui devait permettre à chaque membre de reprendre de façon fluide le travail des autres sans pour autant avoir laissé en suspens ses propres tâches.

Voici un exemple pour illustrer. La première semaine, trois types de missions étaient proposées par le chef d'équipe : certaines étaient théoriques, d'autres étaient des fonctions à coder ou encore des préparations pour la suite du projet. Chacun avait une charge de travail devant occuper environ trois heures de travail. Ces missions étaient toutes atomiques et devaient être réalisées dans la semaine. Si elles n'étaient pas finies, le membre du groupe qui en avait la responsabilité devait préciser les raisons derrière leur inachèvement. Ce rapport écrit permettait au chef d'équipe d'estimer si un autre membre possédait déjà la ou les connaissances manquantes via une précédente mission ou s'il fallait rajouter cette recherche dans la liste des missions à proposer. La semaine suivante, tout le monde changeait de mission, qu'elle soit achevée ou non. Ainsi, aucun membre du groupe ne restait bloqué sur des connaissances manquantes ou dans l'attente d'une aide externe. Il arrivait cependant très souvent qu'en cas d'échec la mission suivante soit la recherche des éléments qui faisaient défaut lors de la mission antérieure. Chaque semaine, ce changement a apporté une nouvelle fraîcheur psychologique au groupe, qui a permis de débloquer des situations qu'un acharnement aurait pu masquer.

Cette façon de travailler nous a aussi permis de prendre en compte les perspectives d'apprentissage de chacun. En rendant le travail moins inquiétant quelle que soit la charge à fournir en fin d'étape, en donnant de la confiance aux membres qui se sentaient en difficulté quelques semaines ou au contraire en donnant des défis très courts permettant à certains de propulser le projet vers une autre partie sans pour autant avoir la charge mentale de supporter l'intégralité du projet. Grâce aux réunions hebdomadaires avec nos encadrants, la conduite du projet était à jour et nous a permis de planifier nos objectifs à long et court terme.

Pour communiquer facilement, nous avons utilisé la plateforme de discussion Discord (6.1). La discussion privée offerte par Discord associée à la plateforme GitLab nous a permis un travail plus horizontal. Cependant ce n'était pas suffisant, c'est pourquoi nous sommes rapidement passés à un serveur Discord permettant l'utilisation de canaux spécifiques à chaque sous partie du projet. Discord permet aussi d'inclure du code d'un langage à spécifier dans un message avec une coloration syntaxique rendant la compréhension du travail de chacun plus simple. Suite à chaque réunion, un compte-rendu des nouvelles missions était publié dans le canal de discussion écrite todo_list.

Afin de pouvoir travailler efficacement, comme cité plus haut, nous avons utilisé GitLab (6.2). C'est à partir d'ici que notre répertoire de projet a été créé. L'unité d'enseignement HAI501I - Ateliers de Génie Logiciel nous a apporté les connaissances nécessaires pour utiliser et gérer correctement ce logiciel, notamment en développant chacun sur nos propres branches dans le répertoire du projet. En effet, l'organisation sur Discord était ainsi faite : chaque développeur s'occupant de ses fonctionnalités travaillait en autonomie ou en binôme. Cette disposition permettait, en plus de ne pas perturber le travail des autres en modifiant le code commun, de laisser s'exprimer pleinement chaque membre du groupe. Nous pensons en rétrospective que sans une telle organisation, certains auraient pu ressentir de la pression quant à l'accomplissement de leur travail. A la fin d'une semaine de travail, les différentes fonctionnalités étaient ajoutées au projet, à condition d'avoir fourni un code fonctionnel.

Nous avons utilisé la plateforme en ligne Overleaf pour la rédaction de ce rapport. Overleaf nous a permis d'éditer un fichier LaTeX en collaboration. LaTeX est un langage de balisage spécialisé dans le traitement de texte. Plusieurs logiciels peuvent prendre en charge le langage LaTeX, dont Overleaf. La complexité du langage est équilibrée par la diversité du contenu que comprend celui-ci : une grande variété d'extensions permet d'enrichir un document d'une manière plus flexible encore que ce qu'offrent les logiciels de traitement de texte classique.

Le langage de programmation utilisé pour le projet est le Go, un langage open source de programmation concurrente partiellement inspiré du C et du Pascal. Comme mentionné dans l'introduction, l'aspect concurrent du Go est très important pour le projet, notamment pour l'algorithme de séparation & évaluation. En effet, la concurrence permet d'exécuter plusieurs itérations de cet algorithme efficacement par l'utilisation de goroutines (threads spécifiques à Go).

Ci-dessous, le diagramme de Gantt schématisant les périodes de travail de notre groupe. Plusieurs points sont à souligner. D'abord, l'aspect séquentiel du travail : les procédures sont implémentées dans un ordre précis et s'appuient les unes sur les autres. Ensuite, certaines phases du projet commencent un peu avant la fin de la précédente. En effet, nous avons profité de notre effectif pour préparer les phases avant leur démarrage pendant que les autres membres du groupe terminaient la phase en cours. Enfin, chaque phase s'est étalée sur un mois, permettant à tous les membres du groupe de situer l'avancée du travail et d'estimer le plus tôt possible les éventuels retards sur le projet ou l'avance permettant de développer une amélioration supplémentaire.

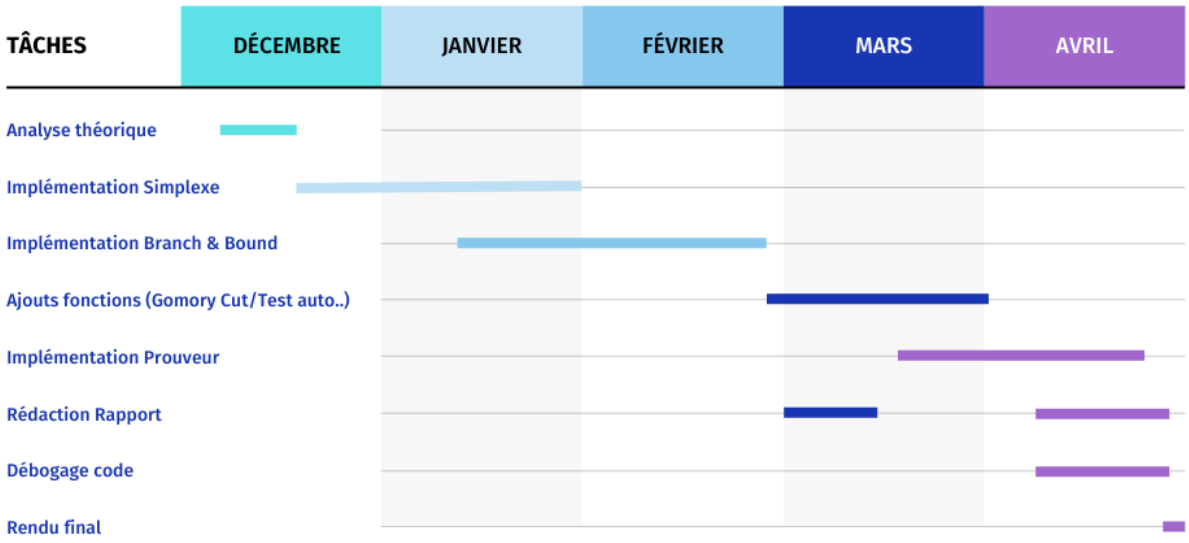

FIGURE 1.1 – Diagramme de Gantt

2 Simplexe

L'un des objectifs de ce TER est de prouver l'existence ou la non existence d'une solution rationnelle pour un système linéaire donné. Un système linéaire est un système composé d'équations ou d'inéquations du premier degré. La stratégie utilisée pour répondre au problème est l'algorithme du simplexe général, car il permet justement de déterminer, à condition qu'elle existe, une solution rationnelle au système traité. C'est donc cet algorithme qui va être présenté dans cette partie. Néanmoins en préambule, une explication rapide de l'utilisation du simplexe peut déjà être faite : Un système linéaire peut avoir une infinité de solutions. Ces solutions forment un espace délimitant le périmètre d'action du simplexe. Cet espace prend la forme géométrique d'une enveloppe convexe, c'est pourquoi des propriétés géométriques sont utilisées pour obtenir des solutions optimisées (entières, maximales, minimales, etc.). Ces propriétés qui seront décrites dans leur partie respective ont en commun l'objectif de restreindre l'espace des solutions afin de ne conserver que les solutions répondant à certains critères.

Ci-dessous (2.1) deux schémas représentant par des points les solutions entières d'un système linéaire quelconque par des droites les restrictions de l'espace de solutions par des contraintes. Le schéma de droite est une version optimisée du schéma de gauche, en effet, toutes les solutions entières présentes dans l'enveloppe convexe du schéma de gauche, sont représentées dans le schéma de droite. Seules des solutions rationnelles ont été écartées de l'enveloppe.

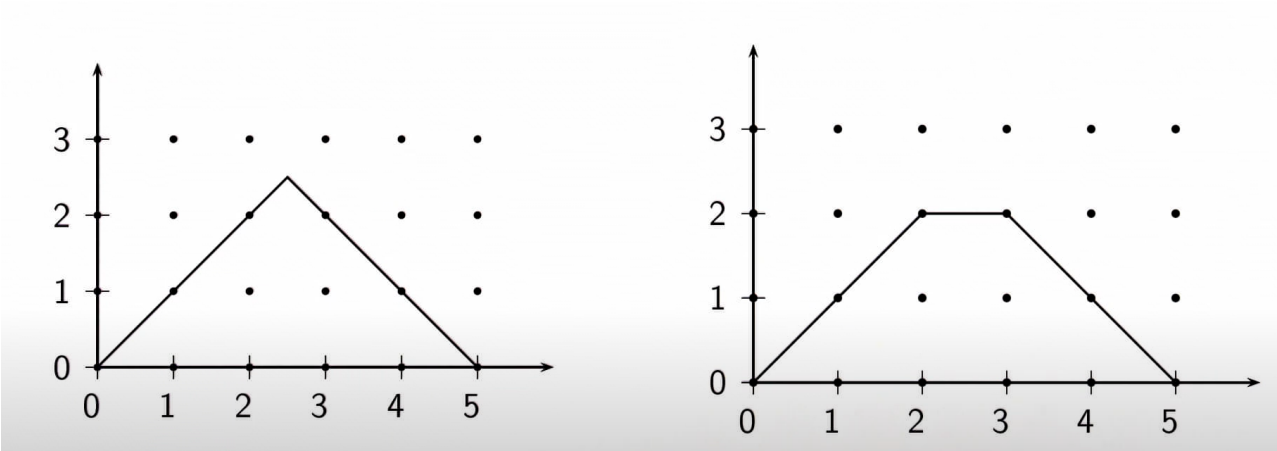

FIGURE 2.1 – Schémas représentant l'enveloppe des solutions d'un système linéaire avant et après ajout d'une coupure de Gomory

Avant d'expliquer le fonctionnement détaillé du simplexe général, une précision doit être apportée, la variante du simplexe que nous utilisons n'a pas de fonction objectif. C'est l'existence d'une solution qui est recherchée, et non son optimalité (maximale ou minimale). Cette variante est mise en avant car la procédure simplexe avec fonction de maximisation (respectivement minimisation) a été popularisée par d'autres disciplines comme l'économie, et reste la plus usitée.

2.1 Description du simplexe

Comme dit en introduction, le simplexe peut prendre en entrée des inéquations et des équations sans pré-traitement. Ces éléments sont appelés contraintes. Cependant, afin de simplifier le traitement des contraintes, nous avons fait le choix de normaliser les éléments du système linéaire en suivant le schéma suivant :

- Si l'inéquation est de la forme $x + y \leq b$, elle est réécrite en multipliant par -1 le membre gauche et le membre droit. Est ainsi obtenue l'inéquation : $-x - y \geq -b$.
- Si la contrainte est une équation de la forme $x + y = b$, elle est décomposée en deux inéquations : $x + y \geq b$ et $x + y \leq b$. Comme pour le point précédent, l'inéquation $x + y \leq b$ est réécrite en $-x - y \geq -b$.
- Si l'inéquation est de la forme $x + y \geq b$, elle ne subit aucune modification.

b étant un nombre rationnel quelconque et x,y les variables initiales du système.

Cette normalisation permet de ne traiter qu'un seul cas et non trois. Elle permet aussi d'éviter des vérifications qui seront explicitées par la suite sur les bornes des variables et le pivot de Gauss-Jordan.
Une fois le système normalisé, il est traité afin de construire la matrice des coefficients, un tableau de contraintes et un tableau contenant le nom des variables initiales du système que nous appellerons respectivement : `A, TabConst` et `TabVar`. Ces éléments nécessaires au simplexe peuvent être vus comme ses attributs, c'est pourquoi nous avons pris la décision de développer une structure nommée `info_systeme`. Il existe d'autres attributs dérivés comme `l'ordre de Bland` que nous détaillerons plus tard dans le rapport, un dictionnaire des affectations et un tableau des positions de chaque variable d'écart dans la matrice des coefficients. Pour le confort des utilisateurs et des testeurs, ces données peuvent être traitées automatiquement depuis un tableau de chaînes de caractères représentant des inéquations normalisées.

Posons le système linéaire suivant :

$$\begin{cases} x + y \geq 2 \\ 2x - y \geq 0 \\ -x + 2y \geq 1 \end{cases}$$

Supposons qu'il soit nécessaire de rendre le membre droit de l'inéquation nul pour obtenir une équation linéaire dont le membre droit serait 0. Une solution est de poser autant de variables que d'inéquations en appliquant comme contrainte sur ces variables le membre droit de l'inéquation d'origine. Avec le système précédent, est obtenu :

$$\begin{cases} -e_0 + x + y = 0 \\ -e_1 + 2x - y = 0 \\ -e_2 - x + 2y = 0 \end{cases} \quad \text{avec} \quad \begin{cases} e_0 \geq 2 \\ e_1 \geq 0 \\ e_2 \geq 1 \end{cases}$$

Ces nouvelles variables sont appelées variables d'écart.

Reprenons le système précédent. Si les variables dites d'écart sont déplacées vers le membre droit, alors il est possible d'exprimer le système initial par rapport à ces variables. Ces variables sont dites basiques (ou appartiennent à la base). Les variables initiales à cette étape sont donc toutes en dehors de la base. La représentation la plus simple du système ainsi obtenu est matricielle :

$$\begin{cases} x + y = e_0 \\ 2x - y = e_1 \\ -x + 2y = e_2 \end{cases} \quad \text{avec} \quad \begin{cases} e_0 \geq 2 \\ e_1 \geq 0 \\ e_2 \geq 1 \end{cases}$$

$$\text{devient} \begin{pmatrix} & x & y \\ e_0 & 1 & 1 \\ e_1 & 2 & -1 \\ e_2 & -1 & 2 \end{pmatrix}$$

$$\text{avec} \quad \begin{cases} e_0 \geq 2 \\ e_1 \geq 0 \\ e_2 \geq 1 \end{cases}$$

La matrice A ainsi acquise a pour dimensions le produit cartésien du nombre de variables d'écart et du nombre de variables initiales. Cela se traduit par :

$$|\text{TabConst}| \times |\text{TabVar}|$$

A ce stade, nous pouvons constater deux phénomènes sur notre matrice qui sont en fait des invariants de A :

- Comme la matrice est équivalente au système d'équations, l'affectation de la variable en ligne reste égale à la somme des affectations multipliées par les coefficients de la matrice.
- Les variables hors de la base respectent leurs bornes.

Comme défini plus haut, les variables d'écart ont une borne explicite. Pour le système suivant :

$$\begin{cases} e_0 + x + y = 0 \\ e_1 + 2x - y = 0 \\ e_2 - x + 2y = 0 \end{cases}$$

Les bornes suivantes sont observées sur les variables d'écart :

$$\begin{cases} e_0 \geq 2 \\ e_1 \geq 0 \\ e_2 \geq 1 \end{cases}$$

Ces bornes sont les bornes inférieures des variables d'écart. Ces variables ont cependant une autre borne implicite. En effet, elles sont aussi bornées par ∞. De manière analogue les variables initiales sont bornées par -∞ et ∞.

Si les variables ont des bornes, elles ont aussi une affectation α. C'est cette affectation qui sera retournée par l'algorithme du simplexe. En effet, le but du simplexe est de déterminer une solution rationnelle. Cette solution est un dictionnaire dont la clé est la variable et la valeur est l'affectation associée à la variable. Dans notre implémentation, nous avons utilisé un dictionnaire avec pour clé une chaîne de caractères du nom de la variable et comme valeur, son affectation. Ce dictionnaire est un attribut dérivé du simplexe car l'ensemble des affectations de chacune des variables (initiales comme d'écart) est initialisée à 0. Elle peut donc, à l'état initial, être déduite

du système d'inéquations.

Maintenant que toutes les variables ont été décrites exhaustivement, une façon d'ordonner les variables est d'utiliser l'ordre lexicographique en commençant par les variables initiales et en terminant par les variables d'écart. Dans notre implémentation, cette énumération est représentée par un tableau de chaînes de caractères où chaque chaîne porte le nom d'une variable. Cet ordre de Bland sera nécessaire lors de la détermination du pivot du simplexe. En effet, si le pivot est choisi aléatoirement ou par ordre des colonnes de la matrice, le simplexe peut boucler. C'est pourquoi le tableau de Bland est un attribut dérivé du simplexe. Il doit être déduit avant même le premier appel du simplexe. Pour le système précédent un ordre de Bland possible serait :

$$[\text{x, y, } e_0, e_1, e_2]$$

Tous les éléments du simplexe ont été présentés. Ils sont initialisés. Un appel au simplexe peut donc être fait pour déterminer ou non une solution au système.
La première étape consiste à vérifier les contraintes des variables de la base (ce sont les variables caractérisant les lignes de la matrice). La vérification valide le fait que l'affectation de la variable traitée soit dans ses bornes. Dans notre implémentation cela se traduit par le fait que l'affectation d'une variable d'écart est supérieure à sa borne inférieure (sa contrainte). Les variables initiales étant bornées par les infinis, elles respectent forcément leur contrainte.

Si toutes les variables de la base respectent leur contrainte alors le système a une solution : son dictionnaire d'affectation. Il est donc possible que le simplexe renvoie une solution immédiate. Prenons le système suivant :

$$\{x \geq 0$$

Ce système avec une unique variable sera traduit par la matrice suivante :

$$\begin{pmatrix} & x \\ e0 & 1 \end{pmatrix}$$

avec pour unique variable d'écart e_0 ayant la contrainte suivante :

$$\{e_0 \geq 0$$

L'affectation initiale de chaque variable est 0. e_0 va donc respecter sa contrainte, et la solution renvoyée sera $x = 0$. Cette solution est correcte car $0 \geq 0$.

Si une variable de la base ne respecte pas l'une de ses contraintes alors l'algorithme du simplexe propose deux suites possibles :

- si c'est possible, déterminer un pivot
- sinon arrêter la procédure. Il n'y a dans ce cas aucune solution possible pour ce système.

Dans le cas où une contrainte n'a pas été respectée, la méthode employée par le simplexe général est l'élimination de Gauss-Jordan. Ce pivot peut être effectué car les variables dans la base forment une sous matrice identité `I`. La matrice `A` augmentée de `I` permet l'application du pivot de Gauss pour éliminer de la base la variable qui ne respecte pas sa contrainte et ainsi l'obliger à rentrer dans ses bornes. Le simplexe peut donc vérifier qu'il existe un pivot compatible dans `l'ordre de Bland`. Les critères de compatibilité sont simplifiés par la normalisation du système.

En effet, si le coefficient est strictement positif, alors il n'y aucune autre vérification à faire ; le pivot est compatible.
Si le coefficient est strictement négatif, alors il faut que la variable hors de la base ait son affectation strictement dans ses bornes. Pour le dire autrement, si la variable hors base est une variable initiale alors le pivot est compatible car une variable initiale est bornée par les infinis. Sinon il faut comparer l'affectation de la variable d'écart hors base avec sa borne inférieure.
Si le coefficient est nul, le pivot est incompatible.
Un pivot compatible a été déterminé, il faut maintenant incrémenter (respectivement décrémenter) l'affectation de la variable hors base afin de modifier l'affectation de la variable de la base pour qu'elle respecte sa contrainte. C'est l'incrémenteur (respectivement le décrémenteur) θ qui occupe ce rôle. θ est défini ainsi :

$$\theta = \frac{b(x_i) - \alpha(x_i)}{a(x_j)}$$

$b(x_i)$ étant la borne inférieure de la variable de la base, $\alpha(x_i)$ étant l'affectation de la variable de la base, et $a(x_j)$ le coefficient de la variable hors de la base pour la variable dans la base.

L'affectation de la variable hors base peut donc être mise à jour par la simple addition de son affectation et de θ. Une fois cette affectation modifiée, la variable située dans la base peut à son tour être mise à jour par la somme des affectations hors base multipliées par leur coefficient.
Les deux variables peuvent enfin échanger leur position. L'opération du pivot est terminée.

Après le pivot, une variable hors base et une variable basique ont échangé leur position, cependant les variables hors base ne répondent plus au premier invariant de la matrice. En effet, la somme des affectations des variables colonnes multipliées par leur coefficient respectif n'est plus égale à l'affectation des variables de la base. Il faut donc mettre à jour la matrice des coefficients pour rétablir cet invariant.
A la ligne du pivot, les coefficients sont à jour. La nouvelle variable de la base peut être définie par les variables en dehors de la base. C'est cette propriété qui est utilisée pour obtenir la nouvelle matrice des coefficients.
Pour illustrer cette propriété, voici une matrice issue d'un pivot en ligne 1 et colonne 1 :

$$\begin{pmatrix} 1 & 1 \\ 4 & 1 \end{pmatrix}$$

$$\text{Soit le système associé : } \begin{cases} x = e_0 + y \\ e_1 = 4x + y \end{cases}$$

Les coefficients de la ligne 2 de la matrice sont les coefficients correspondant à l'ancienne matrice. Pour la mettre à jour, il faut exprimer x en fonction de e_0 et y.

$$\text{Le système devient donc le suivant : } \begin{cases} x = e_0 + y \\ e_1 = 4(e_0 + y) + y \end{cases}$$

La nouvelle matrice est donc :

$$\begin{pmatrix} 1 & 1 \\ 4 & 5 \end{pmatrix}$$

Les affectations qu'il reste à actualiser sont celles des variables basiques qui ne viennent pas de pivoter. En effet, les variables en ligne et colonne pivot ont déjà leur affectation à jour grâce à θ. Les autres variables hors base conservent leur affectation. En reprenant l'exemple précédent, il faut appliquer le calcul suivant : $\alpha(e_1)= 4 \times \alpha(e_0)+5 \times \alpha(y)$

2.2 Exemple

$$\text{Considérons le système suivant : } \begin{cases} x+y \geq 2 \\ 2x-y \geq 0 \\ -x+2y \geq 1 \end{cases}$$

$$\text{Ce système est équivalent à : } \begin{cases} -e_0+x+y=0 \\ -e_1+2x-y=0 \\ -e_2-x+2y=0 \end{cases} \quad \text{avec} \begin{cases} e_0 \geq 2 \\ e_1 \geq 0 \\ e_2 \geq 1 \end{cases}$$

Initialisons le système : les affectations de chaque variable sont à 0. L'ordre de Bland est le suivant : [x , y , e_0 , e_1 , e_2], la matrice associée au système est :

$$\begin{pmatrix} & x & y \\ e_0 & 1 & 1 \\ e_1 & 2 & -1 \\ e_2 & -1 & 2 \end{pmatrix}$$

$\alpha(e_0) = 0 < 2$ donc e_0 ne respecte pas sa contrainte. Dans l'ordre de Bland, il faut chercher un pivot compatible. Le coefficient à la ligne e_0, colonne x est positif. Ce pivot est donc compatible. S'ensuit le calcul de l'incrémenteur $\theta = \frac{Be_0 - \alpha(e_0)}{a}$, où Be_0 est la borne inférieure de e_0, $\alpha(e_0)$ est l'affectation de e_0, et `a` est le coefficient pivot.
Le résultat est $\theta = 2$. Donc $\alpha(x) = 0 + 2 = 2, \quad \alpha(e_0) = \alpha(x) + \alpha(y) = \alpha(x) = 2$
$e_0 = x + y$ donc $x = e_0 - y$, la première ligne de la matrice est donc mise à jour. $\begin{pmatrix} & e_0 & y \\ x & 1 & -1 \end{pmatrix}$

$e_1 = 2x - y = 2 \times (e_0 - y) - y = 2e_0 - 3y$

$e_2 = -x + 2y = -e_0 + y + 2y = -e_0 + 3y$

Les coefficients ayant été mis à jour, la matrice peut être réécrite : $\begin{pmatrix} & e_0 & y \\ x & 1 & -1 \\ e_1 & 2 & -3 \\ e_2 & -1 & 3 \end{pmatrix}$

S'ensuit un nouveau calcul de l'ensemble des affectations des variables de la base.

$\alpha(e_1) = 2 \times \alpha(e_0) + \alpha(y) = 2 \times 2 + 0 = 4$

$\alpha(e_2) = -\alpha(e_0) - 2 \times \alpha(y) = -2 - 0 = -2$

e_2 ne respecte pas sa contrainte, il faut donc chercher un pivot compatible dans l'ordre de Bland. Le coefficient entre e_2 et y est négatif. Il faut donc vérifier que l'affectation de y est strictement dans ses bornes. y est bornée par les infinis, donc son affectation $\alpha(y) = 0$ remplit la condition.

$\theta = \frac{Be_2 - \alpha(e_2)}{a} = \frac{1-(-2)}{3} = 1,$

$\alpha(y) = 0 + 1 = 1,$

$\alpha(e_2) = -\alpha(e_0) + 3\alpha(y) = -2 + 3 \times 1 = 1$

$e_2 = -e_0 + 3y$, donc $y = \frac{1}{3}e_0 + \frac{1}{3}e_2$, la première ligne de la matrice est donc mise à jour :

$$\begin{pmatrix} & e_0 & e_2 \\ y & 1/3 & 1/3 \end{pmatrix}$$

$x = e_0 - y = e_0 - (\frac{1}{3}e_0 + \frac{1}{3}e_2) = \frac{2}{3}e_0 - \frac{1}{3}e_2$

$e_1 = 2e_0 - 3y = 2e_0 - e_0 - e_2 = e_0 - e_2$

Les coefficients ayant été mis à jour, la matrice peut être réécrite : $\begin{pmatrix} & e_0 & e_2 \\ x & 2/3 & -1/3 \\ e_1 & 1 & -1 \\ y & 1/3 & 1/3 \end{pmatrix}$

S'ensuit un nouveau calcul de l'ensemble des affectations des variables de la base.

$\alpha(x) = \frac{2}{3}\alpha(e_0) + \frac{-1}{3}\alpha(e_2) == 1$

$\alpha(e_1) = \alpha(e_0) - \alpha(y) = 2 - 1 = 1$

Toutes les contraintes sont maintenant respectées, il existe donc une solution rationnelle au problème : $\{\alpha(x) = 1, \alpha(y) = 2\}$

2.3 Retour à la Base

Une question s'est posée dès le début du projet. Nous voulions connaître le nombre maximum de pivots qu'il pouvait y avoir lors d'un appel au simplexe. Pour poser la question autrement, nous avions besoin de savoir si une variable d'écart initialement situé dans la base, pouvait sortir de la base puis y revenir. Pour répondre à cette problématique, nous avons supposé que ce n'était pas possible et nous avons trouvé un contre-exemple :

$$\begin{cases} x \geq 0 \\ x \geq 1 \\ x \geq 2 \end{cases}$$

Ce système devient :

$$\begin{cases} -e_0 + x = 0 \\ -e_1 + x = 0 \\ -e_2 + x = 0 \end{cases} \quad \text{avec} \begin{cases} e_0 \geq 0 \\ e_1 \geq 1 \\ e_2 \geq 2 \end{cases}$$

Soit la matrice associée au système :

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

Les affectations de toutes les variables valent 0. Donc e_1 ne respecte pas sa contrainte.
Le coefficient est positif, c'est donc le pivot. $\theta = 1$, $\alpha(x) = 1$ $\alpha(e_1) = 1$ Le résultat est donc la même matrice, dont toutes les affectations sont à 1. x est dans la base, e_1 est hors de la base.

L'affectation de e_2 ne lui permet pas de respecter sa contrainte. De manière analogue à l'opération précédente $\theta = 1$, donc $\alpha(e_1) = 2$ et $\alpha(e_2) = 2$. Il est à remarquer que la variable d'écart e_1 qui avait quitté la base est revenue dans la base.

Sauf en cas de citation explicite, toutes les informations concernant ce chapitre sont citées dans le document suivant : [4, p. 113-119]

3 Séparation & évaluation

L'algorithme de séparation et évaluation est un algorithme générique de résolution de problèmes d'optimisation : il faut à chaque fois chercher à trouver la solution optimale à un problème donné, à savoir celle qui possède un coût minimal et qui respecte les contraintes d'intégrité imposées par le problème (sous forme de système d'inéquations).

L'algorithme de séparation et évaluation est capable de résoudre efficacement des problèmes NP-complets grâce à son fonctionnement : depuis une situation initiale, une contrainte est ajoutée d'une part, puis la contrainte opposée d'autre part ; les deux sont traitées en parallèle. Le nombre de résultats augmente exponentiellement avec le nombre de contraintes grâce à l'étape de séparation ; cependant, un résultat et les sous-résultats qui en découlent ne sont pas pris en compte si la contrainte n'est pas respectée ou si la valeur obtenue à l'évaluation est moins proche de la borne que la solution précédente. Cela permet « d'élaguer »les différentes branches et réduire le nombre de cas à tester. L'algorithme de séparation et évaluation est facile à représenter sous forme d'un arbre binaire, d'où l'utilisation du verbe « élaguer » ; il est préférable de le visualiser ainsi en théorie. L'algorithme de séparation et évaluation renvoie la mention *Insatisfiable* s'il ne trouve pas de solution et une solution valide sinon.

Cet algorithme est également utilisé en recherche opérationnelle pour résoudre des problèmes NP-complets tels que le problème du sac à dos (mettre le plus de poids et de volume possible dans un sac à dos de volume donné) ou encore le problème du voyageur de commerce (recherche d'un cycle hamiltonien de valuation totale minimum).

Dans le contexte de notre projet, l'algorithme de séparation & évaluation est utilisé pour trouver des solutions entières pour les variables initiales choisies. A savoir que le prouveur Goéland peut choisir quelles variables doivent avoir une valeur entière quand il fait appel à l'algorithme, c'est pour cela que l'algorithme de séparation & évaluation va seulement gérer les résultats rationnels choisis par le prouveur.

3.1 Implémentation de l'algorithme de séparation & évaluation

Tout d'abord l'algorithme de séparation & évaluation est un algorithme récursif. Ceci implique que son implémentation contient un cas d'arrêt. Tels que, l'algorithme de séparation & évaluation vérifie la solution renvoyée par le simplexe. Si la contrainte sur les valeurs entières donnée par le prouveur est respectée alors la résolution du problème est terminée et l'algorithme de séparation & évaluation peut retourner la solution. Dans le cas contraire, l'algorithme de séparation & évaluation affine la recherche de la solution s'il le peut.

En effet, dans ce cas de figure, l'algorithme se charge d'ajouter de nouvelles contraintes permettant d'éviter la précédente solution rationnelle. L'algorithme se retrouve donc a travaillé avec deux ensembles de contraintes différents. Il se sépare donc en deux pour travailler sur deux plans différents, d'où son nom : "Algorithme de séparation et évaluation" De plus, il met à jour toutes les autres informations nécessaires comme la matrice de coefficient ou encore un tableau enregistrant la position de chaque variable dans la matrice.

Par exemple pour : $\alpha(x) = 3.5$

Si x doit être entier alors l'algorithme de séparation & évaluation détecte le non-respect de la contrainte entière sur x et va donc créer deux nouvelles contraintes :

$$x \geq 4 \quad et \quad x \leq 3$$

Les deux contraintes doivent chacune être implémentées dans deux systèmes différents. De ce fait, l'algorithme fait deux copies du système initial pour ajouter les deux contraintes dans des système séparés. Cependant, comme vu précédemment avec l'algorithme du simplexe, les contraintes sont appliquées seulement sur les variables d'écart. De ce fait, l'algorithme de séparation & évaluation crée une nouvelle ligne dans le système et doit mettre à jour la matrice des coefficients en y ajoutant une ligne de coefficients nuls et un coefficient de 1 sur la colonne de la variable soumise a la contrainte (x dans notre exemple). Toutefois, notre implémentation du simplexe gère seulement les contraintes de type supérieur ou égal ; donc pour la contrainte inférieur ou égal il est nécessaire de mettre un coefficient de -1 au lieu de 1 pour inverser l'inéquation. Une fois cela fait, l'algorithme met à jour les variables des deux nouveaux systèmes telles que le tableau gérant l'ordre de Bland, celui contrôlant la position des variables et le dictionnaire renseignant les affectations de solution (leur alpha) de chaque variable. Il termine par rappeler le simplexe avec deux nouveaux systèmes puis renvoie sa solution à l'algorithme de séparation & évaluation pour mettre en place la récursivité jusqu'à l'obtention d'une solution confirmée par l'algorithme de séparation & évaluation.

3.2 La concurrence dans l'algorithme de séparation & évaluation

Le langage Go est principalement connu pour la mise en place de sa concurrence efficace grâce à ses goroutines. C'est pour cette raison qu'il a été choisi comme langage de programmation pour notre projet, car il y a plusieurs possibilités d'optimisation par la concurrence, dont une au moment de la séparation en deux systèmes dans l'algorithme de séparation & évaluation. En effet, notre programme lance deux goroutines avec les deux systèmes copiés et font l'ajout de contrainte ainsi que la mise à jour des variables. Cela permet de traiter chacune des possibilités développées par l'algorithme simultanément. Cependant, il est nécessaire de pouvoir communiquer avec chacune des goroutines pour pouvoir faire remonter une solution et arrêter toutes les autres goroutines en cours. Nous avons donc mis en place des channels permettant la communication uniquement dans le sens de l'enfant vers le parent. De cette manière, dès qu'une goroutine trouve une solution, il la remonte au parent. Une fois qu'elle reçoit une solution il ferme son channel ce qui entraînera la fermeture de toutes les autres goroutines.

3.3 Exemple

Considérons le système suivant :

$$\begin{cases} 20t - x + y - 18z \geq 8 \\ -5x + y \geq 5 \\ -7t + 3x + 5y + z \geq 33 \end{cases} \quad \text{soit avec les contraintes :} \quad \begin{cases} e_0 \geq 8 \\ e_1 \geq 5 \\ e_2 \geq 33 \end{cases}$$

La matrice associée au système est : $\begin{pmatrix} & t & x & y & z \\ e_0 & 20 & -1 & 1 & -18 \\ e_1 & 0 & -5 & 1 & 0 \\ e_2 & -7 & 3 & 5 & 1 \end{pmatrix}$

Le premier appel à l'algorithme simplexe renvoie la matrice : $\begin{pmatrix} & t & x & y & z \\ e_0 & \frac{1}{21} & \frac{-2}{147} & \frac{-1}{147} & \frac{127}{147} \\ e_1 & \frac{1}{84} & \frac{-107}{588} & \frac{5}{147} & \frac{53}{294} \\ e_2 & \frac{5}{84} & \frac{53}{588} & \frac{25}{147} & \frac{265}{294} \end{pmatrix}$

Ainsi que la solution : $(t = \frac{13}{147},\ x = \frac{181}{588},\ y = \frac{3845}{588},\ z = 0)$

Maintenant vient l'importance de l'algorithme de séparation et évaluation. Dans la solution retournée t est un décimal, il faut donc séparer puis rajouter les contraintes $t \geq 4$ et $x \leq 3$ ainsi que mettre à jour la matrice. Ce qui donne l'embranchement et les matrices suivants :

$$e_0 \geq 8,\ e_1 \geq 5,\ e_2 \geq 33$$

$$e_0 \geq 8,\ e_1 \geq 5,\ e_2 \geq 33,\ e_3 \geq 1 \qquad e_0 \geq 8,\ e_1 \geq 5,\ e_2 \geq 33,\ e_3 \leq 0$$

$$\begin{pmatrix} & t & x & y & z \\ e_0 & 20 & -1 & 1 & -18 \\ e_1 & 0 & -5 & 1 & 0 \\ e_2 & -7 & 3 & 5 & 1 \\ e_3 & 1 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} & t & x & y & z \\ e_0 & 20 & -1 & 1 & -18 \\ e_1 & 0 & -5 & 1 & 0 \\ e_2 & -7 & 3 & 5 & 1 \\ e_3 & -1 & 0 & 0 & 0 \end{pmatrix}$$

Les nouvelles contraintes sont représentées dans la dernière ligne de la matrice mise à jour. Comme dit précédemment, la nouvelle ligne prend un coefficient 1 ou -1 (pour inverser l'inéquation) dans la colonne t car c'est la variable ayant un résultat décimal. Et les autres coefficients de la ligne à 0 car seule une contrainte sur t est rajoutée. Ainsi sont répétées les exécutions de l'algorithme du simplexe, suivi de celle de l'algorithme de séparation & évaluation jusqu'à trouver une solution satisfiable. Ceci donne cet arbre de résultat :

$$(t = \tfrac{13}{147},\ x = \tfrac{181}{588},\ y = \tfrac{3845}{588},\ z = 0)$$

$$(t = 1,\ x = \tfrac{253}{508},\ y = \tfrac{3805}{508},\ z = \tfrac{134}{127}) \qquad (t = 0,\ x = \tfrac{147}{508},\ y = \tfrac{3275}{508},\ z = \tfrac{-13}{127})$$

$$(t = 1,\ x = 8,\ y = 45,\ z = -209)$$

Comme visible, l'arbre de solution s'arrête car l'algorithme à retourner un résultat satisfiable et donc arrête toute autre recherche de solutions grâce à la fermeture du channel parent.

Sauf en cas de citation explicite, toutes les informations concernant ce chapitre sont citées dans le document suivant : [4, p. 120-122]

4 Optimisations

4.1 Simplexe général : incrémentalité et programmation dynamique

Le simplexe général, comme dit plus haut, démarre par une initialisation de ses paramètres qui seront actualisés tout au long de la procédure en fonction des pivots effectués. S'il est nécessaire d'ajouter une inéquation ne contenant pas de nouvelles variables au système pour affiner une solution ou évaluer l'impact d'une nouvelle contrainte (l'ordre des pivots ne changeant pas), l'ensemble des paramètres seront recalculés en repassant par l'intégralité de la procédure, ce qui entraîne des ralentissements indésirables pour la recherche d'une solution. La méthode choisie pour permettre d'éviter la répétition de la procédure est celle de la programmation dynamique.

La programmation dynamique est une méthode algorithmique basée sur le principe d'optimalité de Bellman, à savoir la recherche d'une solution optimale d'un problème par la recherche de solutions optimales à des sous-problèmes du problème principal. L'idée derrière le choix de la programmation dynamique dans ce projet est la suivante : l'algorithme du simplexe procède de manière ordonnée au choix de la contrainte non respectée à pousser hors base (en suivant l'ordre des lignes de la matrice). L'ordre de Bland, lui, impose un ordre dans le choix du pivot. En est déduit qu'ajouter une contrainte n'a pas d'influence sur les résultats des étapes précédentes. Cette idée nous a été inspirée par le critère de monotonie (ou réactivité positive) du théorème de May 6. Ce critère dans les maths du choix social décrit la capacité d'une nouvelle proposition à influencer toutes les autres propositions.
Il est donc tout à fait possible de stocker les informations clé qui seront perdues dans un tableau et de reproduire tous ces calculs sur la nouvelle contrainte, pour l'intégrer comme si elle avait toujours fait partie du système.
La question que nous nous sommes posés pour l'implémentation est : quelles sont les données qui seront perdues par le déroulement normal du simplexe et qui sont pourtant nécessaires à la mise à jour des autres contraintes ?

Deux données distinctes sont utilisées pour effectuer le pivot : le numéro de la colonne de celui-ci et l'ensemble des coefficients de la ligne du pivot. Or, la taille d'une ligne de la matrice est invariante car définie par le nombre de variables initiales du système. Il est donc possible de n'utiliser qu'un seul tableau qui sera parcouru de la manière suivante : en première case, la colonne du pivot ; dans les cases suivantes, les coefficients ordonnés selon leur ancienne position. La taille de ce tableau indique le nombre de pivots ayant eu lieu précédemment car il correspond à la concaténation de tous les sous-tableaux représentant les informations perdues après chaque pivot.
Enfin, la programmation dynamique nous a permis de comprendre qu'il était possible d'implémenter une optimisation supplémentaire. Dans la partie simplexe, nous avons vu qu'après chaque mise à jour de la matrice des coefficients, il fallait mettre à jour les affectations de chacune des variables. La raison était qu'après chaque pivot, les nouvelles affectations pouvaient en tirer une solution. Ici ce n'est pas le cas. Reprenons les deux idées qui nous ont permis d'aboutir à la conclusion qu'il n'était pas nécessaire de systématiquement mettre à jour les affectations :

- le tableau contenant les informations permettant de mettre en place l'aspect incrémental du simplexe général est la concaténation des informations volatiles de chaque pivot

- nous tenons compte du critère de non-influence des nouvelles contraintes sur les étapes précédentes du simplexe

Les nouvelles contraintes ne peuvent ni changer l'ordre des pivots qui ont eu lieu sur les contraintes précédentes, ni supprimer de contraintes. Il est donc acquis que seul le dernier pivot peut faire émerger une éventuelle solution. C'est donc après le dernier pivot qu'il faut calculer les nouvelles affectations.

4.2 Exemple d'utilisation de la programmation dynamique

Pour illustrer, prenons l'exemple suivant à l'état initial :

$$\begin{cases} x + y = -e_0 \\ 2x - y = -e_1 \\ -x + 2y = -e_2 \end{cases} \quad \text{avec} \quad \begin{cases} e_0 \geq 2 \\ e_1 \geq -2 \\ e_2 \geq 1 \end{cases}$$

$$\text{devient} \begin{pmatrix} 1 & 1 \\ 2 & -1 \\ -1 & 2 \end{pmatrix}$$

e_0 ne respecte pas sa contrainte et le coefficient entre e_0 et x est strictement positif.
Est donc ajouté dans notre tableau destiné au comportement incrémental le numéro de la colonne du pivot : 0.
Après l'étape du pivot est obtenu la nouvelle matrice des coefficients :

$$\begin{pmatrix} 1 & -1 \\ 2 & -3 \\ -1 & 3 \end{pmatrix}$$

La ligne du pivot étant la ligne numéro 0, sont ajoutés à notre tableau de programmation dynamique les coefficients de cette ligne. Ce tableau est donc rempli ainsi : [0, 1, -1]
L'affectation $\alpha(e_2) = -2$, donc cette variable ne respecte pas sa contrainte. Un nouveau pivot va être déterminé en suivant l'ordre de Bland.
Le coefficient entre e_2 et y est strictement positif, il s'agit donc du pivot. Le numéro de la colonne est ajouté dans le tableau des informations volatiles.
La nouvelle matrice est :

$$\begin{pmatrix} 1/3 & -2/3 \\ 2/3 & -7/3 \\ 1/3 & 1/3 \end{pmatrix}$$

Les coefficients de la ligne numéro 2 sont ajouté toujours dans le même tableau :

$$[\ 0,\ 1,\ \text{-}1,\ 1,\ 1/3,\ 1/3\ ]$$

Toutes les variables respectent leurs contraintes. Nous souhaitons maintenant ajouter la contrainte suivante :

$$x + 2y = -e_3 \qquad \text{avec} \quad e_3 \geq 1$$

Plutôt que de recommencer toutes les étapes précédentes, nous posons la matrice à jour en rajoutant la nouvelle contrainte à l'état initial :

$$\begin{pmatrix} 1/3 & -2/3 \\ 2/3 & -7/3 \\ 1/3 & 1/3 \\ 1 & 2 \end{pmatrix}$$

Pour mettre à jour la nouvelle contrainte, il faut parcourir le tableau qui y est dédié. En première case, le pivot est à la case 0. Dans la colonne pivot, le coefficient de la nouvelle contrainte est multiplié par le coefficient de l'ancienne matrice en position 0. Pour les autres colonnes, le coefficient en colonne pivot de la nouvelle contrainte est multiplié par le coefficient de la colonne cible, le tout additionné au coefficient de la colonne cible de la nouvelle contrainte.

La nouvelle matrice est :

$$\begin{pmatrix} 1/3 & -2/3 \\ 2/3 & -7/3 \\ 1/3 & 1/3 \\ (1 \times 1) & (1 \times -1 + 2) \end{pmatrix}$$

La nouvelle contrainte a donc été mise à jour d'un pivot. Il reste le dernier pivot à rétablir et la matrice sera initialisée non plus à l'état initial, mais à l'état final.

$$\begin{pmatrix} 1/3 & -2/3 \\ 2/3 & -7/3 \\ 1/3 & 1/3 \\ (1 \times 1/3 + 1) & (1 \times 1/3) \end{pmatrix} = \begin{pmatrix} 1/3 & -2/3 \\ 2/3 & -7/3 \\ 1/3 & 1/3 \\ 4/3 & 1/3 \end{pmatrix}$$

$\alpha(e_3) = 2 \times \ 4/3 + 2 \times 1/3 = 10/3$

$10/3 \geq 1$ donc le simplexe va retourner une solution. Si la contrainte sur e_3 n'avait pas été respectée, un nouveau pivot aurait été déterminé, le tableau auxiliaire de l'aspect incrémental de la procédure aurait été mis à jour avec les nouvelles informations et si nous venions à ajouter une nouvelle contrainte au système, elle aurait subi 3 calculs successifs sur ses coefficients pour pouvoir être intégrée au système.

Sauf en cas de citation explicite, toutes les informations concernant les deux sections précédentes sont citées dans le document suivant : [4, p. 120]

4.3 Séparation et évaluation : contraintes supplémentaires

L'algorithme de séparation et d'évaluation peut être imagé comme un tamis égrainant les solutions rationnelles et réelles pour ne conserver que les solutions entières. C'est une procédure agissant sur les affectations des variables. Malheureusement en l'état le problème de décision ILP (optimisation linéaire en nombre entier) n'est pas décidable par la procédure. En effet, le système

$1 \leq 3x-3y \leq 2$ n'a pas de solution entière. Cependant, la recherche d'une telle solution provoquera une boucle infinie de la procédure de décision. Il est donc nécessaire de corriger ce cas particulier en ajoutant des contraintes qui vont forcer l'arrêt de la procédure. Pour expliquer ces contraintes, utilisons le système défini précédemment : $1 \leq 3x-3y \leq 2$. Ce système n'a aucune solution entière. Cependant, il existe une infinité de solution réelles qui seront déterminées par l'algorithme de séparation & évaluation. Pour empêcher le programme de boucler, nous ajoutons des contraintes sur les variables initiales dont nous souhaitons obtenir une solution entière. L'idée pour les déterminer est proche de l'algorithme de Metropolis. Pour rappel, l'algorithme de Metropolis permet de calculer un minimum global en ne cherchant que des minima locaux. Ci-dessous, 4.1, un schéma montrant le procédé de l'algorithme de Metropolis. A partir d'une position choisie aléatoirement sur la courbe, la procédure cherche un minimum local tout en permettant de remonter un peu sur la courbe à condition de trouver un autre minimum local plus important. L'échantillon de minimums locaux ainsi obtenu permet de trouver avec une forte probabilité le minimum global quand une recherche directe aurait pû bloquer sur un minimum local non global.

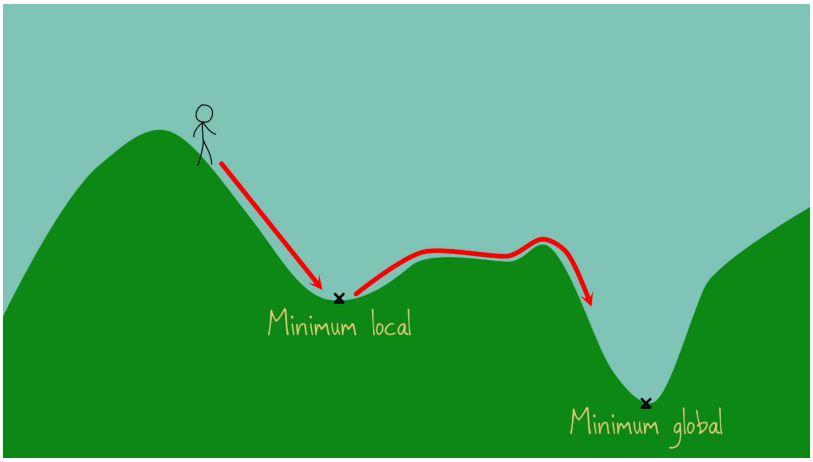

FIGURE 4.1 – Schéma représentant la recherche d'un minimum global. [5]

Dans notre procédure de décision, il n'y a pas d'aléatoire, mais l'intuition reste la même. Les contraintes de terminaisons sont des bornes suffisamment grandes pour empêcher le programme de s'enfermer dans un espace plus restreint que celui de l'enveloppe convexe.

$$VI = ((N+M) \times N \times \theta)^N$$

VI est une variable initiale sur laquelle l'algorithme de séparation & évaluation demande une solution entière ; N est le nombre de contraintes ; M est le nombre de variables initiales ; θ est l'élément de valeur maximale parmi les coefficients et bornes non-infinies.

L'ajout de ces bornes à très grande valeur va forcer la procédure de décision à rechercher des solutions aux frontières de l'enveloppe convexe. Si aucune solution n'est trouvée aux limites de l'enveloppe, alors il n'y a aucune solution au système. L'ajout de ces contraintes empêche donc le système de boucler. La contrepartie est l'ajout à chaque passe dans la procédure de séparation et d'évaluation d'autant de nouvelles contraintes que de variables initiales.

Il existe aussi des configurations particulières permettant de restreindre davantage l'espace des solutions sans pour autant perdre des solutions entières. Ces configurations se traduisent par l'ajout de nouvelles contraintes qui vont grandement augmenter la puissance de l'algorithme.

En effet, l'algorithme de séparation et d'évaluation ne suit pas toujours le meilleur chemin. C'est pourquoi nous avons implémenté les coupures de Gomory (ou *Gomory cuts* en anglais). Ce sont des contraintes qui permettent de restreindre l'espace des solutions sans perdre pour autant de solutions entières. Dans la littérature, lorsque cette optimisation est implémentée, l'algorithme de séparation et évaluation (*Branch & Bound algorithm*) est renommé en algorithme de séparation et coupure (*Branch & Cut algorithm*). Ci-dessous 4.2, un schéma montrant l'enveloppe des solutions en dissociant la composante de l'enveloppe retenant toutes les solutions entières, de celle qui n'en contient aucune. La coupe dans l'enveloppe est la coupure de Gomory ; elle ne coupe qu'une partie de la composante n'ayant pas de solutions entières.

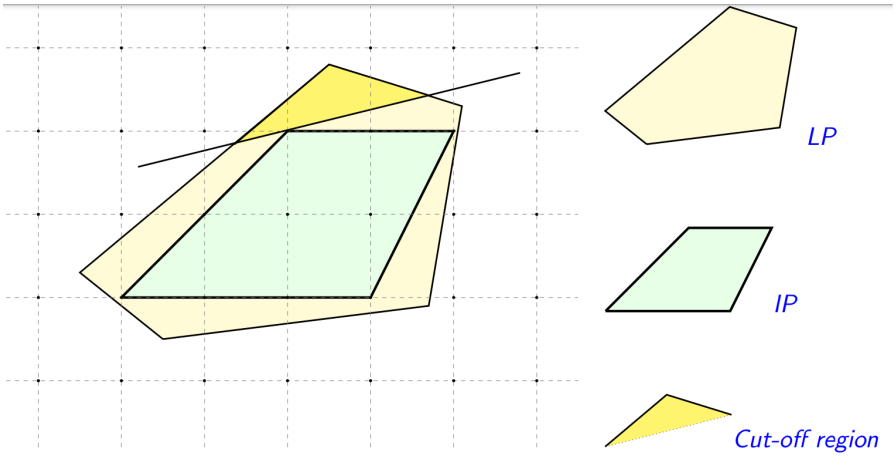

FIGURE 4.2 – Schéma représentant l'enveloppe convexe restreignant l'espace des solutions : en bleu clair (IP) les solutions entières, en beige (LP) les solutions rationnelles, en jaune (*Cut-off region*) une région de solutions rationnelles coupée de l'enveloppe des solutions. [3, p. 31]

Une coupure de Gomory requiert 2 conditions :

- Toutes les variables hors base ont leur affectation strictement égale à l'une de leurs bornes.
- Une variable initiale de la base possède une affectation non entière.

Dans notre implémentation, cela signifie qu'il n'y a que des variables d'écart hors de la base et que leur affectation vaut exactement leur borne inférieure. En effet, les variables initiales sont bornés par les infinis et les bornes supérieures valent $+\infty$. Leur affectation ne peut donc pas prendre ces valeurs.
La coupure de Gomory est construite par un jeu sémantique en 2 parties : Si les coefficients sont positifs, il faut soustraire aux variables d'une contrainte leur affectation. En est tirée l'inéquation suivante :

$$VI - \alpha(VI) \geq \sum_{j \in NB} (a_j \times (e_j - \alpha(e_j)))$$

Il reste ensuite à soustraire la partie décimale du membre gauche de l'inéquation :

$$VI - (\alpha(VI) - \lfloor \alpha(VI) \rfloor) \geq \lfloor \alpha(VI) \rfloor + \sum_{j \in NB} (a_j \times (e_j - \alpha(e_j)))$$

2 événements sont à observer :

- $\alpha(e_j)$ vaut strictement la borne inférieure de e_j, donc $e_j - \alpha(e_j) \geq 0$ et $a_j > 0$, donc la somme est forcément supérieure ou égale à 0.

- $\lfloor \alpha(VI) \rfloor$ est une partie décimale non nulle, donc elle est forcément strictement supérieure à 0.

Donc l'inéquation : $VI - (\alpha(VI) - \lfloor \alpha(VI) \rfloor) \geq \lfloor \alpha(VI) \rfloor + \sum_{j \in NB} (a_j \times (e_j - \alpha(e_j))) > 0$ est obtenue.
Cependant nous cherchons une contrainte entière, donc nous pouvons réécrire l'inéquation obtenue précédemment :

$$VI - (\alpha(VI) - \lfloor \alpha(VI) \rfloor) \geq \lfloor \alpha(VI) \rfloor + \sum_{j \in NB} (a_j \times (e_j - \alpha(e_j))) \geq 1$$

En faisant passer la partie décimale de VI dans le membre le plus à droite, puis en divisant les deux membres droits de l'inéquation par la différence entre 1 et $\lfloor \alpha(VI) \rfloor$ de la partie décimale, la première partie de la coupure de Gomory est obtenue :

$$\sum_{j \in NB} (a_j \times (e_j - \alpha(e_j))) \geq 1 - \lfloor \alpha(VI) \rfloor$$
$$\frac{1}{1 - \lfloor \alpha(VI) \rfloor} \times \sum_{j \in NB} (a_j \times (e_j - \alpha(e_j))) \geq 1$$

Pour les coefficients négatifs, de manière analogue est obtenue l'inéquation suivante :

$$VI - (\alpha(VI) - \lfloor \alpha(VI) \rfloor) \geq \lfloor \alpha(VI) \rfloor + \sum_{j \in NB} (a_j \times (e_j - \alpha(e_j))) \leq 0$$

En faisant passer la partie décimale de VI dans le membre le plus à droite, puis en divisant les deux membres droits de l'inéquation par la négation de la partie décimale, la deuxième partie de la coupure de Gomory est obtenue :

$$\sum_{j \in NB} (a_j \times (e_j - \alpha(e_j))) \leq -\lfloor \alpha(VI) \rfloor$$
$$\frac{-1}{\lfloor \alpha(VI) \rfloor} \times \sum_{j \in NB} (a_j \times (e_j - \alpha(e_j))) \geq 1$$

Enfin, il ne reste plus qu'à ajouter ces 2 inéquations pour obtenir une coupure de Gomory :

$$\frac{1}{1 - \lfloor \alpha(VI) \rfloor} \times \sum_{j \in NB} (a_j \times (e_j - \alpha(e_j))) - \frac{1}{\lfloor \alpha(VI) \rfloor} \times \sum_{j \in NB} (a_j \times (e_j - \alpha(e_j))) \geq 1$$

4.4 Exemple de Coupure de Gomory

Soit le système suivant :

$$\begin{cases} x \geq 1/4 \\ -x \geq -1/4 \end{cases}$$

Ce système devient, par ajout des variables d'écart :

$$\begin{cases} -e_0 + x = 0 \\ -e_1 - x = 0 \end{cases} \qquad avec \begin{cases} e_0 \geq 1/4 \\ e_1 \geq -1/4 \end{cases}$$

A la fin de la procédure simplexe sont obtenues les équations suivantes :

$$\begin{cases} x = e_0 \\ e_1 = -e_0 \end{cases}$$

Avec les affectations suivantes : $\alpha(x) = 1/5, \alpha(e_0) = 1/4$, les conditions d'une coupure de Gomory sont donc remplies. Par conséquent, la contrainte suivante est ajoutée au système :

$$e_2 = \frac{1}{1 - \frac{1}{5}} \times e_0 \geq 1 - \frac{1}{1 - \frac{1}{5}} \times \frac{1}{4}$$

Sauf en cas de citation explicite, toutes les informations concernant les deux sections précédentes sont citées dans le document suivant : [4, p. 122-125]

5 Interface avec Goéland

L'objectif était d'intégrer une procédure de décision pour l'arithmétique linéaire de Presburger à Goéland et de trouver comment résoudre des problèmes arithmétiques par le truchement du prouveur Goéland. Pour commencer, nous avions déjà une interface utilisateur. Cette interface permettait à l'utilisateur de rentrer une chaîne de caractères normalisée et notre parseur en faisait l'analyse. Pour construire notre parseur, il a fallut mettre en place des règles de normalisation. Chaque élément de la chaîne de caractères rentrée par l'utilisateur devait être séparée par un espace. L'équation devait toujours être un « $\geq$ » et chaque variable ne devait apparaître qu'une seule fois.

Exemple :

- $-2x + 2y \geq 9$ ne correspond pas car il manque les espaces entre les éléments
- $5\ y\ +\ 2\ z\ \leq\ 44$ ne correspond pas car le signe n'est pas bon
- $7\ x\ +\ 3\ z\ -\ 2\ x\ \leq\ 3$ ne correspond pas car l'opération « 7 x - 2 x »doit être remplacée par « 5 x »
- $6\ x\ +\ 8\ z\ \leq\ 44\ -\ 3$ ne correspond pas car l'opération « 44 - 3 »doit être remplacée par « 41 »
- $5\ z\ +\ x\ \geq\ 3$ correspond bien

Comme nous avions la volonté, au-delà du projet, de permettre la démocratisation de ces outils de décision, nous avons permis à l'utilisateur de construire ses propres procédures basées sur le simplexe par différents moyens :

- Via un parseur de chaînes de caractères
- Via un système d'entrée-sortie pour que l'utilisateur puisse entrer les valeurs dans le tableau de coefficients ainsi que les contraintes sur chaque ligne de celui-ci, tout en étant guidé
- Via un jeu de couleurs dans le terminal qui décrit chaque étape du simplexe.

Nous voulions répondre au manque que nous avons constaté lorsque nous avons fait ce projet. En effet, nous n'avions pas de tests et donc rien pour comparer nos résultats. Notre volonté était de proposer un outil formateur et interactif pour répondre à ce manque.

Tout ceci n'est pas la problématique de notre TER, mais c'en était assez proche pour que puissions nous en préoccuper. Nous devions trouver comment résoudre des problèmes arithmétiques par l'intermédiaire du prouveur Goéland.

Afin de comprendre comment les deux systèmes (arithmétique et prouveur) communiquent, les règles de Goéland sont présentées dans la section 5.1. Ensuite, les nouvelles règles qu'apportent le module arithmétique sont exposées en section 5.2. Enfin, l'explication de l'implémentation de ces règles dans le prouveur est décrite dans la section 5.3.

5.1 Le prouveur Goéland

Le prouveur Goéland est un prouveur automatique de théorème en logique du premier ordre, développé par Julie Cailler, une de nos encadrantes pour le projet. Reprenant une grande partie de la méthode des tableaux, il s'en différencie cependant par l'utilisation de programmation concurrente (notamment pour l'algorithme de séparation & évaluation).

Ci-dessous, les règles issues de la méthode des tableaux :

$$\frac{\bot}{\odot}\,\odot_{\bot} \qquad \frac{\neg\top}{\odot}\,\odot_{\neg\top} \qquad \frac{P \qquad \neg P}{\odot}\,\odot$$

$$\frac{\neg\neg P}{P}\,\alpha_{\neg\neg} \qquad \frac{P \Leftrightarrow Q}{\neg P, \neg Q \mid P, Q}\,\beta_{\Leftrightarrow} \qquad \frac{\neg(P \Leftrightarrow Q)}{\neg P, Q \mid P, \neg Q}\,\beta_{\neg\Leftrightarrow}$$

$$\frac{P \wedge Q}{P, Q}\,\alpha_{\wedge} \qquad \frac{\neg(P \vee Q)}{\neg P, \neg Q}\,\alpha_{\neg\vee} \qquad \frac{\neg(P \Rightarrow Q)}{P, \neg Q}\,\alpha_{\neg\Rightarrow}$$

$$\frac{P \vee Q}{P \mid Q}\,\beta_{\vee} \qquad \frac{\neg(P \wedge Q)}{\neg P \mid \neg Q}\,\beta_{\neg\wedge} \qquad \frac{P \Rightarrow Q}{\neg P \mid Q}\,\beta_{\Rightarrow}$$

$$\frac{\exists x.P(x)}{P(\epsilon(x).P(x))}\,\delta_{\exists} \qquad \frac{\neg\forall x.P(x)}{\neg P(\epsilon(x).\neg P(x))}\,\delta_{\neg\forall} \qquad \frac{\forall x.P(x)}{P(X)}\,\gamma_{\forall M}$$

$$\frac{\neg\exists x.P(x)}{\neg P(X)}\,\gamma_{\neg\exists M} \qquad \frac{\forall x.P(x)}{P(t)}\,\gamma_{\forall inst} \qquad \frac{\neg\exists x.P(x)}{\neg P(t)}\,\gamma_{\neg\exists inst}$$

FIGURE 5.1 – Règles de la méthode des tableaux

Les règles des tableaux présentées ci-dessus se divisent en trois catégories distinctes :

— les règles de clôture (dont la règle de coupure) permettant de clôturer la branche d'un arbre de preuve,

— les règles analytiques (α, β, δ) inspirées des règles de calcul des séquents en logique du premier ordre

— les γ-règles

Dans la méthode des tableaux, s'il est nécessaire de traiter un problème composé d'hypothèses et d'une conséquence, il faut chercher à démontrer l'insatisfiabilité du problème en soumettant la conjonction des hypothèses et de la négation de la conséquence aux règles des tableaux.

Le système de clôture d'arbre de preuve est basé sur le conflit dans la négation d'une expression logique.

Voici un exemple pour illustrer :

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg((\forall x.\neg(x<-1)\vee x<-1)\wedge\exists y.y>0)}{\neg(\forall x.\neg(x<-1)\vee x<-1)}\,\beta_{\neg\wedge}}{\neg(a<-1)\wedge a<-1}\,\delta_{\neg\forall}}{\neg(a<-1),a<-1}\,\alpha_{\neg\vee}}{\odot}\,\odot \qquad \dfrac{\dfrac{\neg\exists y.y>0}{\neg(Y>0)}\,\gamma_{\neg\exists}}{\sigma=Y\mapsto 1}\,\odot_{\sigma}$$

FIGURE 5.2 – Exemple de la résolution d'une expression avec la méthode des tableaux

D'abord est appliquée une règle β qui branche sur le connecteur racine. La branche gauche est skolémisée en introduisant le symbole a. Ensuite, nous le cassons la formule avec une règle α (qui ne crée pas de branche), ce qui donne une contradiction entre les deux prédicats, clôturant cette branche. Pour la branche droite, une métavariable est instanciée puis nous trouvons une substitution telle que $y > 0$ pour clore la branche.

5.2 Présentation des règles du prouveur Goéland

Ci-dessous, les règles liant Goéland au module arithmétique.

Règles de Constante

$$\dfrac{k \bowtie k'}{\odot}\ \text{Const} \qquad \dfrac{k = k'}{\odot}\ \text{Const}$$

Règles de Normalisation

$$\dfrac{e = e'}{e \leq e', e' \leq e}\ \text{Eq} \qquad \dfrac{e \neq e'}{e < e' \mid e' > e}\ \text{Neq} \qquad \dfrac{\neg e \bowtie e'}{e \overline{\bowtie} e'}\ \text{Neg}$$

$$\dfrac{e < f}{e \leq f - 1}\ \text{Int-Lt} \qquad \dfrac{e > f}{e \geq f + 1}\ \text{Int-Gt}$$

FIGURE 5.3 – Règles du prouveur Goéland

Il y a deux catégories de règles :

- Les règles de constante, qui permettent de clôturer une branche. Elles sont appelées si k et k' sont des constantes ($k, k' \in \mathbb{Z}$ ou $k, k' \in \mathbb{Q}$) et l'opérateur est un des opérateurs de comparaison arithmétique suivants : $\{=, >, <, \leq, \geq, \neq\}$.
  Par exemple $1 > 2$ est une contradiction et va ainsi permettre à la branche de se clôturer.
- Les règles de normalisation, qui créent des branches. Elles sont appelées si e ou f sont des expressions entières ($e, f \in \mathbb{N}$) et l'opérateur est un des opérateurs de comparaison

arithmétique suivants : $\{=, \neq, >, <\}$.
Le but est d'arriver à un conflit afin de clôturer ces branches avec les règles de constante. Ces règles vont ensuite faire appel au simplexe et à l'algorithme de séparation & évaluation pour trouver des solutions en envoyant la négation de leurs formules logiques. Si l'algorithme du simplexe ou de séparation & évaluation renvoie une solution à cette négation, c'est qu'il a bien trouvé un conflit.

5.3 Implémentation dans Goéland

Pour permettre l'implémentation dans Goéland, nous avons utilisé la programmation orientée objet. En effet, il n'y a pas de classes en Go comme il y en a dans d'autres langages orientés objet. Cette contrainte du langage pourrait laisser penser que Go n'est pas un langage objet. Cependant, Go permet bel et bien de programmer des objets.

Premièrement, nous allons parler des constructeurs, qui sont des fonctions qui retournent une structure de données. Comme en C, les structures de données sont un regroupement d'attributs typés. Et nous pouvons, via des interfaces, créer des fonctions ne pouvant être appelées que par des types d'objets spécifiques.
En exemple, il y a « MakerID » qui est un constructeur qui renvoie un ID. Un ID est une structure qui contient plusieurs attributs.

```
/* ID maker */
func MakerId(s string) Id {
        lock_id.Lock()
        i, ok := idTable[s]
        lock_id.Unlock()
        if ok {
                return MakeId(i, s)
        } else {
                return MakerNewId(s)
        }
}
```

FIGURE 5.4 – Constructeur : MakerID()

Les attributs servent à définir la structure. Par exemple, pour Id, ces attributs sont *index* et *name*. L'attribut *index* sert à classer les ID : c'est une numérotation des ID pour ne pas avoir à les reconstruire quand ils existent déjà et *name* à les nommer.

Ce constructeur a deux *return*, soit MakerId qui récupère un ID qui existe déjà, soit MakerNew qui crée un nouvel ID. Le constructeur est « malin » car il ne recrée pas ce qu'il connaît déjà.

```
/* id (for predicate) */
type Id struct {
        index int
        name  string
}
```

FIGURE 5.5 – Structure : Id

Pour finir, en Go, toutes les méthodes sont des fonctions d'interface. (Id) GetName() est une fonction d'interface qui ne peut être appelé que par un ID.

```
func (i Id) GetName() string {
        return i.name
}
```

FIGURE 5.6 – Méthode : GetName()

Maintenant qu'ont été expliqués les outils de base de la programmation orientée objet en Go, les prédicats peuvent être présentés. Les prédicats sont les objets que nous traitons car ils présentent l'avantage d'avoir pour sémantique une valeur de vérité. Un prédicat pour Goéland est une structure de données qui contient trois paramètres :

- args ([]Term), une liste de termes utilisés comme arguments du prédicat. Les termes sont des métavariables ou des constantes.
- typeHint (typing.TypeScheme). Goéland a ses propres types, tRat et tInt correspondant à des termes respectivement rationnels et entiers.
- id (ID)

Pour faire le lien entre notre travail et le prouveur, Goéland envoie des prédicats arithmétiques au module arithmétique. Ces prédicats sont ensuite traités via un parseur, puis sémantiquement interprétés.
Selon le prédicat, la sémantique est vérifiée, il reste à regarder si la solution recherchée est entière ou rationnelle et renvoyer un résultat.

Le prouveur récupère un problème disponible dans la bibliothèque libre de TPTP [6], une bibliothèque de problèmes destinés à la preuve de théorème automatisée. Chacun de ces problèmes contient une description de celui-ci (ignorée par le prouveur), puis la formule logique à évaluer. Parmi les problèmes disponibles dans cette bibliothèque, les problèmes arithmétiques sont ceux qui nous intéressent dans le cadre de ce projet.

5.4 Règles arithmétiques : quotient et reste

Parmi les règles en arithmétique implémentées dans le prouveur Goéland, les règles "quotient" et "remainder" (reste) se démarquent des autres règles arithmétiques du prouveur par leur séparation en sept règles distinctes : "quotient", "quotient_e", "quotient_t", "quotient_f", "remainder_e", "remainder_t", "remainder_f". L'effet de chacune des règles est défini ainsi :

- quotient : applicable seulement sur deux rationnels ; calcule la division entre deux rationnels.
- quotient_e : calcule la division euclidienne entre deux entiers ou deux rationnels. [1, p. 132-134]
- quotient_t : calcule la division réelle entre deux entiers ou deux rationnels ; le quotient est ensuite tronqué à sa partie entière. [1, p. 131-132]
- quotient_f : calcule la division réelle entre deux entiers ou deux rationnels ; le quotient est ensuite recalculé à sa partie entière inférieure. [1, p. 131, 133]
- remainder_e : calcule le reste de la division euclidienne entre deux entiers ou deux rationnels.
- remainder_t : calcule le reste de la division réelle tronquée entre deux entiers ou deux rationnels.
- remainder_t : calcule le reste de la division réelle sous-évaluée entière entre deux entiers ou deux rationnels.

Chacune de ces règles renvoie une erreur en cas de division par 0 (pour les entiers) ou 0/1 (pour les rationnels). Afin de distinguer les différentes règles de quotient et de reste présentées ci-dessus, voici quelques exemples :

$$quotient(\tfrac{6}{7}, \tfrac{2}{9}) = \tfrac{27}{7}$$
$$quotient(\tfrac{-6}{7}, \tfrac{2}{9}) = \tfrac{-27}{7}$$

$quotient_e(5,2) = 2$	$remainder_e(5,2) = 1$
$quotient_e(\tfrac{6}{7}, \tfrac{2}{9}) = 3$	$remainder_e(\tfrac{6}{7}, \tfrac{2}{9}) = \tfrac{6}{7}$
$quotient_e(-5,2) = -2$	$remainder_e(-5,2) = -1$
$quotient_e(\tfrac{-6}{7}, \tfrac{2}{9}) = -3$	$remainder_e(\tfrac{-6}{7}, \tfrac{2}{9}) = \tfrac{-6}{7}$
$quotient_t(5,2) = 2$	$remainder_t(5,2) = 1$
$quotient_t(\tfrac{6}{7}, \tfrac{2}{9}) = 3$	$remainder_t(\tfrac{6}{7}, \tfrac{2}{9}) = \tfrac{6}{7}$
$quotient_t(-5,2) = -2$	$remainder_t(-5,2) = -1$
$quotient_t(\tfrac{-6}{7}, \tfrac{2}{9}) = -3$	$remainder_t(\tfrac{-6}{7}, \tfrac{2}{9}) = \tfrac{-6}{7}$
$quotient_f(5,2) = 2$	$remainder_f(5,2) = 1$
$quotient_f(\tfrac{6}{7}, \tfrac{2}{9}) = 3$	$remainder_f(\tfrac{6}{7}, \tfrac{2}{9}) = \tfrac{6}{7}$
$quotient_f(-5,2) = -3$	$remainder_f(-5,2) = 1$
$quotient_f(\tfrac{-6}{7}, \tfrac{2}{9}) = -4$	$remainder_f(\tfrac{-6}{7}, \tfrac{2}{9}) = \tfrac{1}{7}$

FIGURE 5.7 – Exemples d'exécution des différentes règles de quotient et de reste du prouveur

Pendant le développement de ces règles, nous avons remarqué que l'opérateur modulo de Go se comportait différemment de son implémentation dans d'autres langages tels que Python. Par exemple, sur le calcul de $-119 \bmod 13$, nous avons un résultat différent selon le langage utilisé :

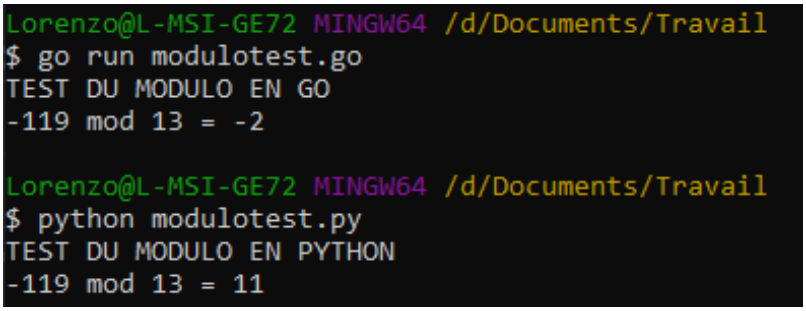

FIGURE 5.8 – Calcul de -119 mod 13 en Go, puis en Python

Cette différence pouvant entraîner de potentielles irrégularités dans le calcul, nous avons préféré implémenter le reste en calculant la formule suivante pour chacune des opérations de reste détaillées plus haut :

$$r = dd - ds \times q$$

avec r le reste de la division, dd son dividende, ds son diviseur et q son quotient. [1, p. 131]

5.5 Jonction Goéland - procédures de décision

Goéland fournit une liste de prédicats pouvant être interprété sémantiquement comme des équations ou des inéquations. Cependant nos procédures de décision ne permettent pas de lire directement ces prédicats. Il faut donc réaliser un pré-traitement. Pour ce faire, notre méthode d'analyse s'appuie sur 3 étapes dont les 2 premières sont simultanées.

La liste de prédicat est parcourue, pour chaque prédicat une analyse lexicale récursive permet de récupérer ce que nous appellerons des métavariables. Ces variables particulières traitées par Goéland seront les variables initiales de nos procédures de décision. Ce parcours permet de stocker dans un tableau la liste des variables initiales dont nous aurons besoin dans les procédures de décision. En même temps, un compteur de prédicat est incrémenté, si le prédicat est une inéquation, il augmente de 1, si le prédicat est une équation, il augmente de 2. Il est à noter que le nombre de variables initiales donne le nombre de colonnes de la matrice des coefficients, et le compteur offre le nombre de lignes.

Parallèlement, durant le parcours une analyse récursive sémantique est effectuée. Notre choix d'implémentation normalise les prédicats de manière à ce que les métavariables avec leurs coefficients soient dans le membre gauche du prédicat, quand les constantes sont dans le membre droit. Ce choix a une influence dans le signe des coefficients et des constantes après passage de l'analyseur. Sont traitées par l'analyseur l'ensemble des fonctions issus de la bibliothèque ARI de TPTP [6]. Parmi ces fonctions certaines s'analysent de manière analogue, c'est le cas des opérations additives (additions, soustraction) ou multiplicatives (multiplication, quotients). Les fonctions unaires comme les restes ne peuvent prendre comme paramètre que des constantes, c'est pourquoi l'analyse est proche pour ces fonctions malgré leur arité différente. Il semble pertinent de noter que pour l'analyse de la soustraction, nous avons fait le choix de jouer avec notre conception plutôt que de l'analyser brutalement. Une soustraction est considérée dans notre implémentation comme une addition dont les arguments ne sont pas dans le même membre du prédicat.
Pour illustrer cela : $x - y \geq 3$ est analysé comme $x \geq 3 + y$
Cette étape permet de récupérer l'ensemble des calculs entre les fonctions et les métavariables

pouvant être effectué avant utilisation de nos procédures de décision. Cependant il peut y avoir plusieurs prédicats qui n'ont pas les mêmes métavariables, et il peut rester des paires coefficient-métavariable ayant la même métavariable mais qui n'ont pas encore été additionné. D'où l'intérêt de la 3e étape. A la fin de l'étape 2 la liste de prédicat est réécrite en une liste de paires coefficient-métavariable avec une métavariable nulle pour les constantes.

Cette étape est la seule qui ne s'effectue pas en même temps que les autres. En fonction du tableau de métavariables et des listes de paires coefficients-métavariables citées plus haut, cette étape effectue un tri ordonné des pairs et additionne les pairs qui ont la même métavariable pour simplifier la lecture finale permettant de remplir la structure de données initiale du simplexe. Quand cette étape est finie, le simplexe peut être appelé avec cette structure de données ainsi qu'un tableau de booléens listant les métavariables souhaitées entières. Ainsi le simplexe pourra appeler l'algorithme de séparation & évaluation sur les variables initiales désignées de manière automatique.

5.6 Exemple de Jonction

Illustrons une jonction entre Goéland et la procédure simplexe avec un exemple. Soit un système contenant deux prédicats :

$$\begin{cases} \text{Sum(Difference(Uminus(4), Product(3,x)),x)} \geq \text{Uminus(9)} \\ \text{Product(4, y)} = 2 \end{cases}$$

En première étape la liste de prédicats va être analysée, par une fonction qui va chercher récursivement les types métavariables. Les fonctions (constantes inclues) sont donc négligées par cette étape. Dans le même temps, le type de prédicat est lui aussi analysé. Le premier prédicat est de type $\geq$ quand le second est de type =. Le compteur de contraintes s'incrémente donc de 1 puis de 2 pour atteindre la valeur de 3.
En parralèle les fonctions sont analysées d'un point de vue sémantique.
Sum(Difference(Uminus(4), Product(3,x)),x) $\geq$ Uminus(9)
devient Sum(Difference(-4, 3x), x) $\geq$ -9 (Une paire [-9, vide] est créée),
puis Sum(-4,x) $\geq$ -9 -3x (-3x étant dans le membre droit, une paire [3, x] est créée),
enfin est obtenue -4 x $\geq$ -9 -3x (-4 étant dans le membre gauche, la paire [4, vide] est créée, en même temps que la paire [1, x]).
De manière analogue, Product(4, y) = 2 devient 4y = 2 et les paires [4, y] et [2, vide] sont crées.
Nous obtenons donc 2 tableaux de paires, une par prédicat. Un traitement postérieur est effectué. Le prédicat de type = est dupliqué, l'un des fils voit tous ses coefficients et toutes ses constantes être multipliés par -1.
Ainsi nous obtenons le tableau de tableau de paires suivant :

[[[-9, vide],[3, x], [1, x], [4, vide]], [[4, y], [2, vide]],[[-4, y],[-2, vide]]]

C'est maintenant que s'effectue la troisième étape. Au sein d'un sous tableau toutes les paires de même variable s'additionne. Les variables manquantes sont aussi ajoutées avec pour coefficient : 0.

[[[4, x], [0, y], [-5, vide]], [[0, x], [4, y], [2, vide]], [[0, x], [-4, y], [-2, vide]]

Il faut maintenant remplir les attributs du simplexe comme expliqué dans la partie éponyme. Le système qui sera envoyé au simplexe est donc :

$$\begin{cases} 4x + 0y \geq -5 \\ 0x + 4y \geq 2 \\ 0x - 4y \geq -2 \end{cases}$$

L'étape 1 fournit, en plus de la liste de variables initiales, une liste de variables dont que nous souhaitons entières. Cette liste est transformée en tableau de booléens qui est aussi passé au simplexe pour permettre l'appel éventuel à la procédure de séparation & évaluation.

5.7 Les tests

Les tests ont été notre stratégie de vérification. Ils sont très importants car ils nous permettent de voir que notre implémentation traite l'ensemble des cas dont nous avons anticipé le comportement. Ils nous permettent aussi de voir si notre gestion d'erreur est au point.

Comment fonctionnent-ils ?
Ci-dessous le cas de l'analyse sémantique des fonctions arithmétiques : "uminus" qui transforme un nombre positif en nombre négatif et inversement.

```
func TestUminusInt() {
    fmt.Println(" -------- TEST Uminus Int -------- ")
    fmt.Println(" 4  devient -4 ")
    quatre := types.MakerConst(types.MakerId("4"),tInt)
    uminus := types.MakerFun(
                types.MakerId("uminus"),
                []types.Term{quatre},
                tInt)
    solution,_:=ari.EvaluateFun(uminus)
    fmt.Println("solution = ", solution)
}
```

FIGURE 5.9 – Exemple de tests unitaires avec le cas du uminus entier

Nous avons décidé que pour chaque fonction à tester, nous devions faire en sorte d'englober tous les cas possibles. Donc pour chaque fonction il y a des tests pour les cas :

- la fonction reçoit un entier
- la fonction reçoit un entier négatif
- la fonction reçoit un rationnel (exemple : 4/2)
- la fonction reçoit un rationnel négatif (exemple : -4/2)
- la fonction reçoit un rationnel qui n'est pas entier (exemple : 2/3)
- la fonction reçoit un rationnel négatif non entier (exemple : -2/3)

Pour faciliter la relecture et la vérification, nous affichons le résultat attendu, ici -4.

"MakerConst" nous sert à créer une constante ; son "MakerID" est sa valeur. Nous allons rentrer cette valeur (ici 4) dans la fonction à évaluer (ici uminus).
"MakerFun" sert à créer la fonction "uminus".
Enfin, nous appelons la fonction créée avec "EvaluateFun", qui renvoie "ma solution est une erreur" et nous affichons cette solution pour vérifier manuellement si elle correspond au résultat attendu.

Prenons l'exemple maintenant l'exemple de la fonction "sum" qui additionne 2 fonctions (une constante est une fonction d'arité 0). Dans cet exemple, ces constantes sont des rationnels et l'un d'eux est négatif.

```
func TestSumRat2() {
    fmt.Println(" -------- TEST Sum Rat 2 -------- ")
    fmt.Println(" 1/3 + 2.5 = 17/6?")
    un_sur_trois := types.MakerConst(types.MakerId("1/3"), tRat)
    deux_cinq := types.MakerConst(types.MakerId("5/2"), tRat)
    sum := types.MakeFun(
                        types.MakerId("sum"),
                        []types.Term{un_sur_trois, deux_cinq},
                        typing.GetTypeScheme(
                                    "sum",
                                     typing.MkTypeCross(tRat, tRat)))
    solution,_:=ari.EvaluateFun(sum)
    fmt.Println("solution = ", solution)
}
```

FIGURE 5.10 – Exemple de test unitaire avec le cas de la somme rationnelle négative

Ici, en plus des fonctions montrées dans la partie des fonctions unaires, nous devons rajouter "MkTypeCross" pour définir le produit cartésien. Dans cet exemple, le produit cartésien est $\mathbb{Q} \times \mathbb{Q}$.

Dû à la grande variété d'opérateurs arithmétiques utilisés par la bibliothèque TPTP [6], le nombre total de tests sur ces opérateurs s'avère être bien plus grand : le nombre de lignes de code consacrées exclusivement à ces tests dépasse 2500.

5.8 Exemple complet d'un problème arithmétique traité par Goéland

Prenons un exemple pour illustrer comment Goéland résoud des problèmes arithmétiques à partir de formule logique du premier ordre :

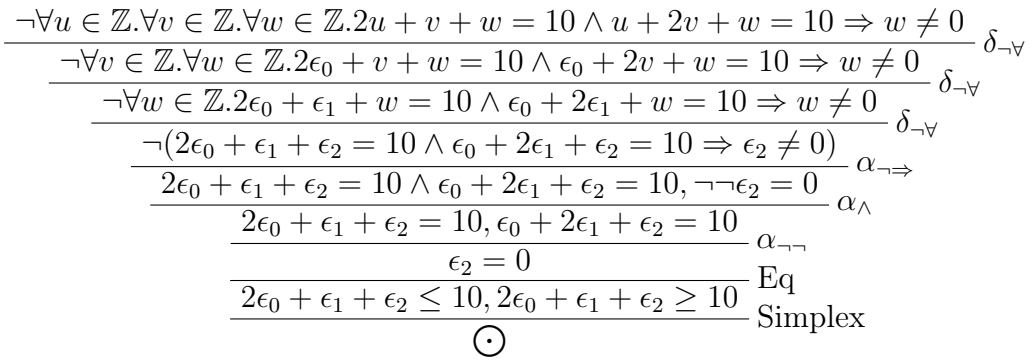

FIGURE 5.11 – Exemple de preuve utilisant les règles du prouveur Goéland

Pour commencer, la règle « $\delta_{\neg\forall}$ »est appliquée trois fois : elle enlève les quantificateurs pour skolémiser les variables liées. Puis la règle « $\alpha\neg \Rightarrow$ »est appliquée ce qui permet de « casser »l'implication. La règle $\alpha_\wedge$ »permet elle aussi de « casser »l'opérateur « $\wedge$ ». Maintenant avec la règle du « $\alpha_{\neg\neg}$ »qui enlève la double négation, nous trouvons que « $e_2 = 0$ ». La règle Eq permet de casser l'opérateur d'égalité, appliqué a la première formule « $2\epsilon_1 + \epsilon_2 + \epsilon_3 = 10$ » ; il la divise en deux inéquations. La deuxième formule n'est pas réécrite car elle reste inchangée. Reste enfin à appeler la règle « *Simplex* »qui fera la jointure (décrite dans la partie « Jonction Goéland - procédures de décision »30) entre le système obtenu précédemment et les procédures de décision, permettant de clôturer l'arbre de recherche de preuve.

Sauf en cas de citation explicite, toutes les informations concernant ce chapitre sont citées dans le document suivant : [2]

Bilan et conclusion

Les algorithmes du simplexe général et de séparation & évaluation (devenu *Branch & Cut* par l'introduction des coupures de Gomory) ont été implémentés en programmation dynamique et parallèle dans le prouveur Goéland.

Dès le départ, ce TER nous aura marqué par sa complexité mêlant connaissances théoriques - que nous devions acquérir- et pratiques, ainsi que par une liberté souvent vertigineuse dans le choix des implémentations. La principale difficulté que nous avons rencontrée dans l'implémentation de l'algorithme du simplexe général était d'ordre théorique, notamment sur la détermination d'un pivot qu'il fallait associer à notre normalisation des systèmes d'équations.

L'algorithme de séparation & évaluation nous a imposé un travail supplémentaire, autant théorique que pratique : nous devions comprendre et implémenter cet algorithme par le truchement du paradigme multitâche. Si le parallélisme a contribué à la vitesse de prise de décision de Goéland, il a en revanche troublé notre lecture dans le débogage, alors que celle-ci est habituée au séquentiel.

L'algorithme du simplexe général a été complètement implémenté, selon le document de référence cité en début de chapitre. Parmi les fonctionnalités incluses sont comprises : introduction des variables d'écart, mise en tableau des coefficients et des variables, pivot entre variables de base et d'écart et traitement des problèmes incrémentaux.

C'est pourquoi nous avons dû changer nos habitudes de programmation. En effet, tout au long de ce TER, nous avons ressenti des carences de bonnes pratiques et avons bénéficié des précieux conseils de nos encadrants. Nous avons utilisé les couleurs de bash pour les affichages des nombreux éléments du simplexe et pour distinguer les enfants des goroutines. Nous avons utilisé des structures afin de rendre lisibles les paramètres de nos fonctions. Pour ajouter un nouveau système ou choisir un système codé en dur à tester, nous n'avions plus besoin de toucher au code ; un système d'entrée-sortie a été implémenté à cet effet. Le code a été factorisé en sous-fonctions pour pouvoir être réutilisé facilement et gagner en lisibilité. Enfin, nous avons normalisé nos indentations, nos noms de variables et nos commentaires. Toutes ces bonnes pratiques nous ont à la fois permis de gagner du temps dans nos productions, en même temps qu'elles ont apporté à chaque membre du groupe la capacité de s'approprier l'intégralité du code.

Ces pratiques de programmation ont fortement influencé la gestion du groupe. En effet, le projet était linéaire : les procédures de décision se suivaient séquentiellement, venaient ensuite les optimisations, puis l'intégration des procédures au prouveur. De plus, la vision sur le long terme dépendait beaucoup de la communication entre nos encadrants et nous, afin de ne pas avoir de surprise au moment de l'intégration. Il fallait donc une parfaite osmose entre les membres du groupe ainsi qu'une bonne compréhension globale, pour pouvoir avancer étape par étape sans se faire obstruction. La contrainte de faire le projet à cinq dans ce contexte est devenue une force, dès l'instant où nous avons pris conscience du fort lien entre la conception modulaire du code et la production rapide de petites briques indépendantes. Nous avons donc choisi de répartir le travail de façon à ce que chaque membre puisse opérer indépendamment sur des briques hebdomadaires.

Une perspective pour pousser le projet plus loin serait de tenir compte des éléments à valeur dans les réels. Dans le cadre de ce TER, nous nous sommes limités aux entiers relatifs et aux rationnels. Cependant, le type flottant est limité par construction pour traiter les réels. Il semblerait pertinent de s'intéresser à la construction d'un type symbolique avec ses opérations internes.
Une autre perspective serait de traiter le cas des formules disjonctives. En effet, un problème

d'arithmétique linéaire peut être formulé avec des systèmes dont les éléments sont connectés par des disjonctions. La difficulté sera de traiter côté Goéland ces ensembles de prédicats par le truchement de branches distinctes.

Références

[1] Raymond T. Boute. The euclidian definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems*, 14(2), April 1992.

[2] Guillaume Bury and David Delahaye. Integrating simplex with tableaux.

[3] Oktay Günlük. Cutting planes for mixed-integer programming : Theory and practice, April 2018.

[4] Daniel Kroening and Ofer Stritchman. *Decision Procedures : An Algorithmic Point of View*. Springer-Verlag Berlin Heidelberg, 2008.

[5] ScienceEtonnante. Comment déchiffrer (presque) n'importe quel message codé ? https://www.youtube.com/watch?v=z4tkHuWZbRA à 9 :26, April 2021.

[6] G. Sutcliffe. The tptp problem library and associated infrastructure. from cnf to th0, tptp v6.4.0. *Journal of Automated Reasoning*, 59(4) :483–502, 2017.

6 Annexes

Gestion du groupe

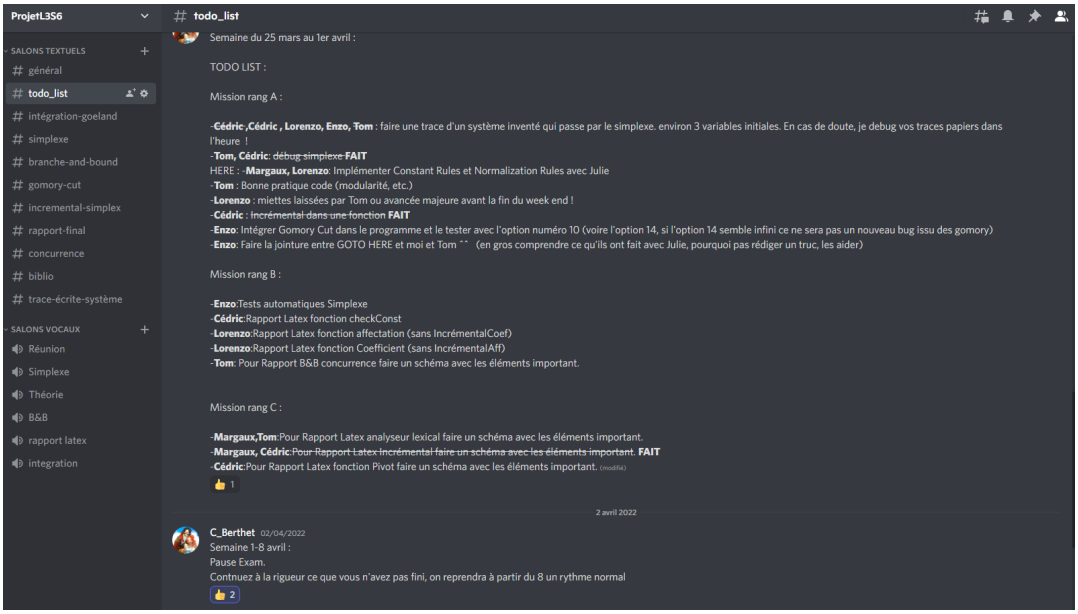

FIGURE 6.1 – Canal de discussion écrite todo_list sur Discord

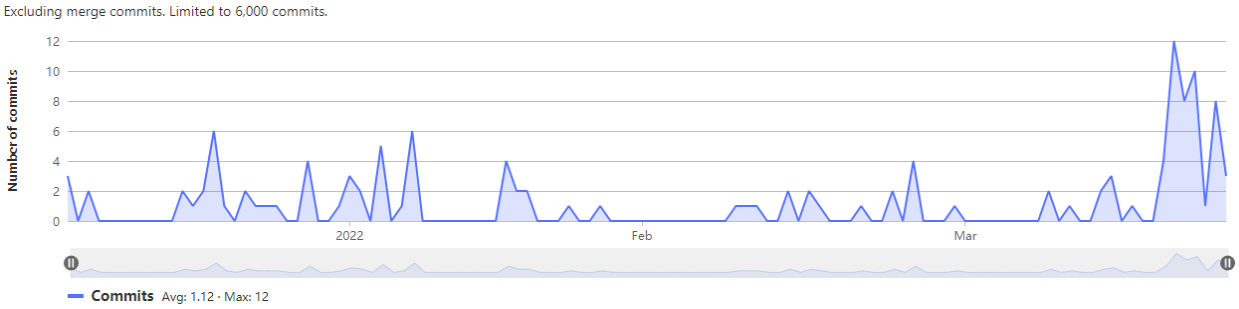

FIGURE 6.2 – Graphique de la quantité de "commits" sur une des branches du projet

Revenir à la gestion du groupe (page 4)

Optimisations

Théorème de May

"Le théorème de May dit que si le choix du groupe est limité à deux possibilités, la règle du vote à la majorité simple ou relative constitue une règle de choix social qui satisfait à un ensemble d'exigences raisonnables qui sont :

— critère de l'anonymat : si deux votants échangent leurs bulletins le résultat de l'élection sera le même (tous les votants ont le même poids)

— critère de la neutralité : si tous les votants échangent leurs votes alors le résultat de l'élection s'inverserait (chaque vote a le même poids pour tous les candidats)

— critère de monotonie (ou réactivité positive) : un candidat qui bénéficie d'un soutien accru des votants ne doit pas voir se dégrader sa position au regard de la décision collective

— choix décisif : pour deux options quelconques, une exactement doit être choisie."

Revenir aux optimisations (page 18)

```
case "round":
    fmt.Println(f.GetTypeHint())
    switch f.GetTypeHint() {
    case tInt:
        if res1, err := checkError1Arg(arg1); err != nil {
            return zero_rat, err
        } else {
            res_1_f64, _ := res1.Float64()
            diff := res_1_f64 - float64(int(res_1_f64))
            res := newRat()
            if diff >= 0.5 {
                res = res.SetFloat64(math.Ceil(float64(res_1_f64)))
            } else {
                res = res.SetFloat64(math.Floor(float64(res_1_f64)))
            }
            return res, nil
        }
    case tRat:
        if res1, err := checkError1Arg(arg1); err != nil {
            return zero_rat, err
        } else {
            res_1_f64, _ := res1.Float64()
            res_1_f64_floor := newRat().SetFloat64(math.Floor(res_1_f64))
            diff := newRat().Sub(res1, res_1_f64_floor)
            res := newRat()
            if diff.Cmp(big.NewRat(1,2)) == 1 || res1.Cmp(big.NewRat(1,2)) == 0 {
                res = res.SetFloat64(math.Ceil(float64(res_1_f64)))
            } else {
                res = res.SetFloat64(math.Floor(float64(res_1_f64)))
            }
            return res, nil
        }
    default:
        return zero_rat, errors.New("Error in evaluate : floor")
    }
```

FIGURE 6.3 – Cas de l'opération "round"

Revenir à l'interface avec Goéland (page 24)