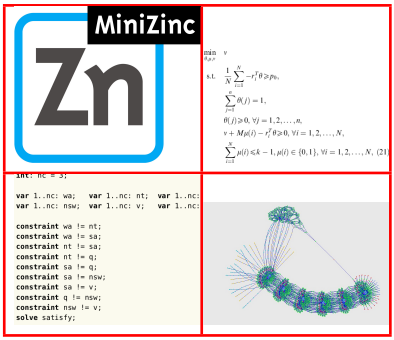
INTRODUCTION À LA RECHERCHE OPÉRATIONNELLE ET AIDE À LA DÉCISION

DUT Informatique – M4202C
Promotion 2018–2020

Julien Vion – julien.vion@uphf.fr
Thierry Bay – thierry.bay@uphf.fr



INSTITUT UNIVERSITAIRE DE TECHNOLOGIE VALENCIENNES • CAMBRAI • MAUBEUGE
Université Polytechnique HAUTS-DE-FRANCE

Introduction

Ce cours se veut être un *manuel de référence* que vous devrez avoir **systématiquement et obligatoirement** sur vous lors de toutes les séances d'apprentissage et d'évaluation. Vous irez y piocher les informations dont vous aurez besoin pour résoudre les problèmes auxquels vous serez confrontés. Contrairement à un cours habituel, *ne vous attendez pas* à une présentation systématique du contenu de ce livret lors des séances de cours. L'ensemble des notions expliquées ici devront être progressivement mises en œuvre lors de la résolution de chaque problème.

Ce livret n'est pas exhaustif. N'hésitez pas à consulter les documents listés dans la section bibliographique si vous avez besoin de plus d'informations.

Programme

Ce cours correspond au module *Introduction à la recherche opérationnelle et aide à la décision* (M4202C) du Programme Pédagogique National du DUT Informatique [3]. L'objectif est de vous former aux compétences suivantes :

Objectif du module :

- Connaitre l'existence d'outils de base pour aider la décision : programmation linéaire, etc. ;
- Comprendre le fonctionnement et les limitations de ces méthodes.

Compétences visées :

- Modéliser une situation complexe à l'aide d'un graphe ou de variables corrélées ;
- Prendre une décision raisonnée en optimisant un ou plusieurs critères.

Contenus :

- Programmation linéaire ;
- optimisation discrète ;
- méthodes arborescentes.

Modalités de mise en œuvre :

Pour la mise en œuvre pratique de ce module, nous nous appuierons sur le langage *MiniZinc*, qui inclut un solveur de problèmes de satisfaction et optimisation de contraintes simple.

Organisation des séances

La formation est organisée sous forme de *problèmes* sur lesquels vous travaillerez lors des séances de TD et TP. L'ensemble des sujets des problèmes que vous aurez à résoudre au cours de la formation se trouve en annexe de ce document. Généralement, la résolution d'un problème suivra le schéma suivant :

1. Séance de TD (1 heure 30) : étude du sujet *en groupe de 3 ou 4*. Préparation d'une solution.
2. Séance de TP (3 heures) : réalisation *individuelle* de la solution élaborée lors de la séance de groupe.
3. Séance de TD (1 heure 30) :
    - présentation du travail réalisé par un groupe,
    - correction du problème,
    - exercices d'application complémentaires.
4. Séance de cours (1 heure 30) :
    - restructuration et réponse aux questions ;
    - éventuellement une évaluation *individuelle*.

Soit 7 heures 30 en séance par problème. Entre chaque séance : travail autonome (non encadré), individuel et/ou en groupe, pour compléter les préparations et travaux non terminés (les évaluations sont faites en considérant que les problèmes sont entièrement résolus) et acquérir les connaissances nécessaires à la résolution des problèmes (normalement d'après ce livret).

L'annexe A de ce livret donne des indications sur les méthodes de travail. Consultez-la avant le début du premier problème.

Table des matières

Chapitre 1

Optimisation combinatoire et recherche opérationnelle : comment, pourquoi ?

De nombreux problèmes, industriels ou quotidiens, nécessitent de réaliser des décisions difficiles ou d'optimiser : réaliser un emploi du temps, ordonnancement des tâches dans un atelier (*scheduling*), trouver le plus court chemin d'un point à un autre (*planning*), stocker des objets dans des containers (*bin packing*), allocation de fréquences pour les réseaux mobiles, routage dans les réseaux, dans les aéroports, réaliser des intelligences artificielles, décrypter un message codé, etc…

En général, tous ces problèmes peuvent être décomposés en variables, souvent des nombres entiers. Un problème de *décision* consiste à trouver une solution, c'est-à-dire à affecter une valeur à chaque variable. Un problème d'*optimisation* consiste à trouver *la meilleure* solution, suivant un critère donné, par exemple la solution la moins couteuse, la moins risquée, prenant le moins de temps, etc., voire une combinaison de critères.

La *recherche opérationnelle* (et dans une certaine mesure l'*intelligence artificielle*) est le domaine scientifique qui s'intéresse à ce genre de problème. Les problèmes d'optimisation sont étudiés par les mathématiciens dès le XVIII^{e} siècle, mais c'est au cours de la seconde guerre mondiale qu'elle trouve de premières applications concrètes dans la planification des opérations militaires (d'où son nom). À partir des années 1950, la discipline commence à s'imposer dans les milieux industriels et académiques, mais c'est surtout dans les années 1990 que la performance des ordinateurs permet de résoudre des problèmes industriels de taille raisonnable. Aujourd'hui, la recherche opérationnelle et l'intelligence artificielle sont des disciplines académiques très complètes, faisant intervenir des notions de mathématiques appliquées, d'informatique, d'économie, d'ingéniérie, de statistiques, etc.

1.1 Quelques exemples

1.1.1 Appariements de colocataires

On souhaite apparier quatre personnes A, B, C et D pour une colocation (dans deux appartements de deux places chacun). Chacune des quatre personnes a des affinités, pas toujours réciproques, avec chacune des trois autres. Ces affinités seront représentées par une note allant de 0 (aucune affinité) à 10 (très forte affinité). L'objectif est de trouver un appariement qui va maximiser la somme des affinités.

Voici les affinités sous forme de matrice :

	A	B	C	D
A		0	6	10
B	3		7	2
C	3	9		4
D	7	8	4	

Une solution à ce problème serait d'apparier A avec B et C avec D. La somme des affinités est alors de 0 (affinité de A avec B) $+ 3$ (affinité de B avec A) $+ 4 + 4 = 11$. Est-ce la meilleure solution ?

On peut aussi fixer une affinité minimum, et trouver un appariement qui va satisfaire ce minimum. On peut alors chercher à trouver le plus grand minimum possible. La solution précédente a pour minimum l'affinité de A avec B, soit 0. Peut-on trouver une solution avec un meilleur minimum ?

Ce genre de problème est appelé un problème *combinatoire* parce que l'on est tenté, pour le résoudre, de tester toutes les combinaisons d'appariements jusqu'à trouver la meilleure solution. Et le nombre de combinaisons augmente très vite ! Dans les années 1960, une université américaine souhaitait résoudre ce problème pour plus de 21 000 étudiants. Avec 22 personnes, il y a déjà plusieurs milliards de combinaisons à évaluer. Une telle progression, dite *exponentielle*, dépasse de loin les capacités de calcul des ordinateurs les plus puissants : les temps de calculs se compteraient en millénaires… Cependant, il n'est peut-être pas nécessaire de tester toutes les solutions pour répondre au problème…

1.1.2 Coloration de carte

Un problème très classique en optimisation combinatoire est la coloration de cartes.

Le principe consiste, comme dans l'exemple ci-dessus, à affecter une couleur à chaque région d'une carte, de sorte que deux régions adjacentes aient deux couleurs différentes. Il faut également trouver le nombre minimal de couleurs nécessaire pour optimiser la solution.

Pour résoudre ce problème, on représente la carte par un graphe, dont la particularité est de pouvoir être tracé sans croiser les arcs : on parle de graphe *planaire*.

FIG. 1.1 : Carte de France à colorier : combien de couleurs au minimum ?

Dans ce cas, on a démontré en 1976 qu'on pouvait toujours colorer une carte avec quatre couleurs seulement, mais la démonstration a résisté longtemps à la communauté des mathématiciens, depuis sa formulation au milieu du XIX^{e} siècle jusqu'à sa preuve. Pour l'anecdote, c'est la première preuve qui avait nécessité l'emploi d'un algorithme programmé, dans la mesure où la démonstration requiérait une décomposition en 1 478 cas particuliers (on a trouvé de meilleures preuves depuis).

1.1.3 Problème du voyageur de commerce

Un voyageur de commerce doit visiter n villes en passant par chaque ville exactement une fois. Il commence par une ville quelconque et finit sa tournée à la ville de départ. Sachant que les distances entre les villes sont connues, quel chemin faut-il choisir afin de minimiser la distance parcourue ? La notion de distance peut-être remplacée par d'autres notions comme le temps qu'il met ou l'argent qu'il dépense. En termes mathématiques, l'objectif est de trouver un cycle hamiltonien (c'est-à-dire passant par chaque sommet) de coût minimal.

Ce problème est un représentant de la classe des problèmes NP-complets. L'existence d'un algorithme de complexité polynomiale reste inconnue. Pour 15 villes, il existe 43 milliards de possibilités. Temps de résolution : 12 h ! Les algorithmes pour résoudre ce type de problèmes peuvent être répartis en deux classes :

1. Les algorithmes déterministes qui trouvent la solution optimale.

2. Les algorithmes d'approximation qui fournissent une solution *presque* optimale.

1.2 Problème de satisfaisabilité

Étant donné un ensemble de règles logiques (c'est-à-dire une grosse formule booléenne pouvant comporter les opérateurs NON, ET et OU), ce problème consiste à décider s'il existe une assignation de valeurs aux variables qui renvoie la valeur VRAI. Il est toujours possible de réécrire ces problèmes sous forme d'une conjonction de disjonctions (appelées clauses). Cette réécriture permet de considérer chaque clause comme une contrainte au problème. Souvent, une telle assignation n'existe pas. Dans ce cas, il est naturel de chercher une assignation satisfaisant un nombre maximal de contraintes.

Un exemple de problème de satisfaisabilité est connu sous le nom de *problème SAT*, dont l'objectif est de vérifier si une formule propositionnelle est satisfaisable ou non. Cette modélisation a baucoup d'applications au niveau de la CAO, des bases de données, ou de la vision par ordinateurs.

1.3 Problème du plus court chemin

Vous avez déjà rencontré ce type de problèmes en théorie des graphes. La recherche d'un plus court chemin dans un graphe valué peut se faire entre deux sommets, d'un sommet à tous les autres, ou entre tous les couples de sommets. L'objectif est évidemment de minimiser le coût. Ce type de problèmes se rencontre en *Optimisation dans les réseaux*.

De nombreux algorithmes existent en fonction des caractéristiques du graphe :

- L'algorithme de *Dijkstra* s'utilise dans un graphe où tous les coûts des arcs sont positifs. Sa complexité peut être en $O(m + n \cdot \log(n))$ (avec n le nombre de sommets et m le nombre d'arcs).
- L'algorithme de *Bellman-Ford* fonctionne avec des arcs à coûts négatifs. Sa complexité peut être en $O(nm)$.
- L'algorithme de *Floyd-Warshall* sert à déterminer le plus court chemin entre toute paire de sommets. Sa complexité peut être en $O(n^3)$.
- Une bonne solution au problème du plus court chemin est fourni par l'algorithme de *Ford-Dikstra* dans lequel un marquage des sommets est effectué. La complexité est en $O(n^2)$.

Notez que ce problème n'est pas NP-difficile !

1.4 Explosion combinatoire et complexité algorithmique

Résoudre un problème de décision ou d'optimisation nécessite dans un premier temps de *modéliser* le problème, c'est-à-dire à le formaliser, généralement sous la forme d'un problème mathématique. Il faut également déterminer un algorithme qui

permettra de le résoudre, si possible de manière efficace (c'est-à-dire, sans tester toutes les combinaisons possibles). L'efficacité d'un algorithme s'évalue généralement par sa *complexité*, c'est-à-dire l'ordre de grandeur du nombre de calculs qu'il va falloir réaliser pour terminer l'algorithme.

Il n'est pas toujours possible de trouver un algorithme « efficace » à un problème, c'est-à-dire qui trouvera la solution en temps raisonnable. Il existe même quelques problèmes théoriques de référence pour lesquels on n'a jamais trouvé, en 50 ans de recherche, un algorithme exact qui ne soit pas de complexité exponentielle. On n'a jamais pu montrer non plus que c'était impossible. Ces problèmes sont dits *«NP-difficiles»*. La coloration de graphe évoquée dans l'exemple de la section 1.1.2 en est un. Trouver un tel algorithme, ou prouver qu'il n'en existe pas, est un des défis majeurs de l'informatique théorique. En attendant, on va généralement chercher à calculer une solution approchée de ces problèmes.

Face à un problème, on commence normalement par chercher s'il n'est pas équivalent à un autre problème dont la complexité et/ou un bon algorithme sont déjà connus… L'exemple de la section 1.1.1 n'est pas NP-difficile : il existe un algorithme issu de la théorie des graphes pouvant le résoudre avec un nombre d'opérations de l'ordre de n^3 [2].

Il existe également des outils, inspirés de l'intelligence artificielle, qui, sans toujours rivaliser avec une étude approfondie d'un problème donné, sont généralement assez efficaces, notamment quand le problème est NP-difficile. Nous allons découvrir quelques uns de ces outils dans la suite du cours.

1.5 Le cours

L'objectif de ce cours est de fournir les bases de la recherche opérationnelle à un public de DUT Informatique. La taxonomie de l'optimisation (ou classification d'entités) ne sera évidemment pas visitée dans son intégralité. Nous nous limiterons à l'optimisation discrète (en nombres entiers et combinatoire) et à l'optimisation continue pour la programmation linéaire (cf. figure 1.2). En parallèle, la programmation par contraintes, faisant intervenir une formalisation spécifique du problème, sera présentée.

Dans le domaine de ***l'optimisation discrète*** sont classés les problèmes pour lesquels certaines variables du modèle appartiennent à un ensemble discret. De nombreux problèmes bien connus se retrouvent dans cette catégorie : problèmes d'affectation, d'emploi du temps, d'ordonnancement… Parmi les branches existantes de ce domaine, deux sont principalement étudiées par la suite :

- la programmation en nombres entiers, pour laquelle l'ensemble discret est un ensemble d'entiers ;
- l'optimisation combinatoire, pour lequel l'ensemble discret est un ensemble d'objets. Cette partie est uniquement introduite à titre informatif dans ce document, mais sera décrite plus en détails dans le reste du module.

Dans le domaine de ***l'optimisation continue***, et plus spécifiquement en programmation linéaire, la fonction-objectif et les contraintes sont tous les deux linéaires

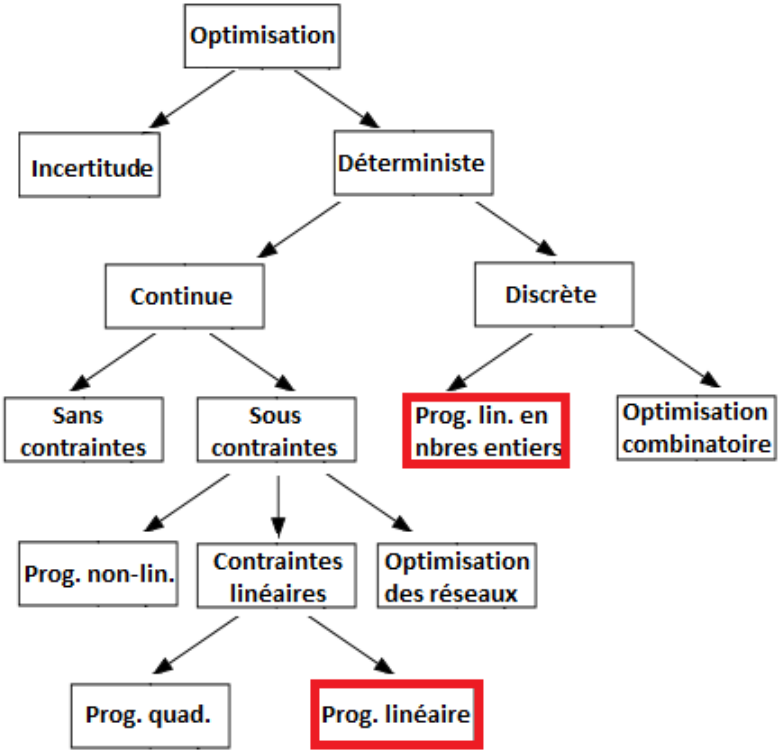


FIG. 1.2 : Taxonomie.

(ou affines).

Pour chacune de ces deux familles, les concepts de base sont présentés et un exemple-type illustre nos propos.

Chapitre 2

Programmation par contraintes

La programmation par contraintes puise ses racines dans les travaux des chercheurs en IA des années 1970. Au milieu des années 1990, l'intégration de techniques de recherche opérationnelle a permis de développer de nouveaux outils performants : extrêmement expressifs, simples d'utilisation grâce à l'intégration de technologies issues de l'IA, et relativement efficaces.

Pour ce cours, nous allons nous familiariser avec le langage de modélisation en programmation par contraintes *MiniZinc* [4]. Une série d'outils basés sur MiniZinc, notamment un solveur de contraintes, un environnement de développement (IDE) et une documentation sont librement téléchargeables sur la page du projet. Ils sont compatibles avec les systèmes Windows, Mac OS et Linux.

2.1 Problème de satisfaction ou optimisation de contraintes

Un problème de satisfaction de contraintes est composé de *variables* et de *contraintes*. Les variables peuvent prendre n'importe quelle valeur prises dans un *domaine* à définir. Généralement, le domaine est un intervalle de nombres entiers : 1..10 représente les valeurs $\{1, 2, 3, \ldots, 10\}$. Dans l'exemple de la coloration de cartes de la section 1.1.2, chaque région sera représentée par une variable. Elle pourra prendre une valeur entre 1 et 4 suivant la couleur dans laquelle la région sera colorée.

Attention, si *MiniZinc* (comme d'autres langages de modélisation) est considéré comme un langage de programmation, il est fondamentalement différent des langages impératifs comme le Java ou le C. On ne manipule pas d'états mémoire, il n'y a pas de structure de contrôle, les instructions ne sont pas exécutées dans un ordre défini : les variables ne sont pas des emplacements dans la mémoire de l'ordinateur

qu'il est possible de modifier, mais des « inconnues », au sens mathématique, que le solveur devra affecter d'une manière ou d'une autre sans intervention humaine. *MiniZinc* fait partie de la famille des langages « déclaratifs » comme le *SQL* ou même le *HTML* : on définit le problème et la solution que l'on recherche, mais pas les algorithmes ou méthodes de résolution.

Les contraintes sont simplement des prédicats (également appelés « expressions booléennes »), tels qu'on les retrouve dans la plupart des langages de programmation. Ce sont des formules ou des fonctions qui prennent en argument un ensemble de variables. Elles renvoient *vrai* si la contrainte est validée par une affectation de ces variables. Par exemple, pour des variables X, Y, et Z, on pourrait avoir : $X < Y, X \neq Y, X = Y + Z, X = |Y|$, etc. Les fameux systèmes de n équations à n inconnues étudiés au collège et au lycée peuvent être considérés comme des problèmes de satisfaction de contraintes.

Ici, la seule limite sera l'expressivité du langage et les fonctionnalités du solveur. MiniZinc supporte une centaine de types de contraintes. Pour la coloration de graphes, les contraintes vont imposer que deux nœuds voisins soient colorés différemment ($X \neq Y$).

Voici un programme MiniZinc qui modélise la coloration de la carte de France en quatre couleurs (représentées par les entiers 1 à 4) :

```
var 1..4: nord_picardie; var 1..4: normandie;
var 1..4: ile_de_france; var 1..4: est;
var 1..4: bretagne;      var 1..4: pays_de_la_loire;
var 1..4: centre;        var 1..4: bourgogne_franchecomte;
var 1..4: sud_ouest;     var 1..4: rhonealpes_auvergne;
var 1..4: sud;           var 1..4: paca;

constraint nord_picardie != normandie;
constraint nord_picardie != ile_de_france;
constraint nord_picardie != est;
constraint normandie != bretagne;
constraint normandie != pays_de_la_loire;
constraint normandie != centre;
constraint normandie != ile_de_france;
constraint ile_de_france != centre;
% ...

solve satisfy;
```

L'instruction **var** 1..4 : nord_picardie ; déclare une variable nord_picardie, qui peut prendre une valeur entière comprise entre 1 et 4. Plus loin, la ligne **constraint** nord_picardie != normandie ; impose que la variable nord_picardie prenne une valeur différente de la variable normandie. Enfin, **solve satisfy** ; indique que le solveur devra trouver une solution quelconque, satisfaisant toutes les contraintes.

Une fois le programme écrit, il suffit de lancer le solveur pour obtenir une solution :

```
> minizinc france.mzn
```

```
nord_picardie = 1;
normandie = 3;
ile_de_france = 2;
est = 3;
bretagne = 1;
pays_de_la_loire = 2;
centre = 1;
bourgogne_franchecomte = 4;
sud_ouest = 3;
rhonealpes_auvergne = 2;
sud = 1;
paca = 3;
```

L'énoncé complet du problème consiste à trouver le nombre minimal de couleurs pour colorer la carte. Il y a un objectif à optimiser, on parle donc de *problème d'optimisation de contraintes*. Concrètement, on ajoute ce nombre de couleurs comme variable au problème, et on va demander au solveur de minimiser ce nombre. Comme on sait que c'est toujours possible avec 4 couleurs, et toujours impossible avec une seule couleur, on peut se fixer ce minimum et maximum. Voici la variante du problème :

```
var 2..4:nb_couleurs;

var 1..4:nord_picardie; var 1..4:normandie;
var 1..4:ile_de_france; var 1..4:est;
% ...

constraint nord_picardie <= nb_couleurs;
constraint normandie <= nb_couleurs;
constraint ile_de_france <= nb_couleurs;
constraint est <= nb_couleurs;
% ...

constraint nord_picardie != normandie;
constraint nord_picardie != ile_de_france;
constraint nord_picardie != est;
constraint normandie != bretagne;
% ...

solve minimize nb_couleurs;
```

constraint nord_picardie <= nb_couleurs;... est une série de contraintes qui permet de fixer facilement le nombre de couleurs maximal du problème. Enfin, la dernière ligne a été modifiée en **solve minimize** nb_couleurs;, pour que le solveur trouve maintenant une solution pour laquelle nb_couleurs soit le plus petit possible.

2.2 Le langage MiniZinc

2.2.1 Instructions et commentaires

Une instruction MiniZinc est une déclaration de paramètre, de variable, de contrainte ou d'objectif de résolution. Toute déclaration se termine par un point-virgule.

On peut commenter une ligne avec le caractère %.

2.2.2 Paramètres

On peut déclarer en MiniZinc des paramètres (ou constantes) avec la syntaxe : **par** type : nom [= valeur] ;

```
par int: nb_couleurs = 3;
```

Les types supportés par MiniZinc sont **bool**, **int**, **float** et **string**. Les tableaux et matrices sont également supportés (voir section suivante).

On peut également déclarer le paramètre et donner sa valeur plus loin dans le programme :

```
par int: parameter;

%...

parameter = 3;
```

Pour de nombreux problèmes, cela peut permettre de définir une modélisation générale du problème, et les cas particuliers dans une autre section du programme, voire dans un fichier séparé. On ne peut évidemment affecter un paramètre qu'une seule fois.

Les valeurs peuvent être des expressions (3 * 2 + 4...)

Les paramètres apparaissent dans les contraintes exactement comme s'il s'agissait de variables.

2.2.3 Variables

Les variables sont les « inconnues » du problème, que le solveur va essayer d'affecter. Chaque variable doit être définie avec son domaine, c'est-à-dire l'ensemble des valeurs possibles pour la variable. Les variables peuvent être booléenes, entières ou flottantes.

On déclare une variable avec la syntaxe : **var** domaine : nom [= expression] ;

```
var int: ex1;
var 1..6: ex2;
var {1, 2, 5, 9}: ex3;
```

Le domaine peut être un type, un intervalle ou une liste de valeurs. Les domains infinis (notamment **int**) sont mal supportés par les solveurs (MiniZinc considère qu'il s'agit du domaine $-10\,000\,000..10\,000\,000$, qui n'est pas vraiment infini), évitez-les si possible.

2.2.4 Contraintes

Les contraintes définissent le problème. Toute contrainte est une expression booléenne, qui peut être définie à l'aide des opérateurs et fonctions suivantes :

Opérateurs logiques (si x et y sont des booléens) :

Conjonction (« et » logique) : x /\ y

Disjonction (« ou » logique) : x \/ y

Implication (si - alors) : x -> y

Équivalence (si et seulement si) : x <-> y

Négation : not(x)

« ou » exclusif : x xor y

Opérateurs de comparaison : On retrouve les opérateurs classiques dont la signification est évidente : !=, <, <=, =, >, >=

Opérateurs arithmétiques : On retrouve les opérateurs classiques dont la signification est évidente : *, +, -, /, mais aussi les fonctions mathématiques classiques abs(x), max(x, y), min(x, y), mod(x ,y), pow(x, y), div(x, y) (ce dernier réalise une division entière).

Fonctions sur les réels : Quand on travaille sur les types réels, on a accès à des fonctions mathématiques spécifiques : fonctions trigonométriques, logarithmes, racines, etc.

2.2.5 Type de problème

Un programme MiniZinc se termine toujours par l'objectif à réaliser. Celui-ci peut être de trois types :

- **solve satisfy**;
- **solve minimize** *expression arithmétique*;
- **solve maximize** *expression arithmétique*;

Dans le premier cas, on cherche simplement une solution, dans les autres cas on cherche une solution qui optimise (en min ou en max) l'expression donnée.

2.2.6 Tableaux

Comme dans la plupart des langages de programmation, on peut définir des tableaux (et des matrices) de variables ou constantes. Ici, les tableaux permettent notamment de définir des modèles de problèmes dont la taille peut varier (comme le nombre de personnes dans l'exemple de la section 1.1.1).

La syntaxe de la déclaration est :
array [*intervalle1*, *intervalle2*, ...] **of** *variable/paramètre* : *nom*

Les intervalles correspondent aux indices de la matrice, chaque intervalle correspondant à une dimension de la matrice. Notez qu'on peut donc les démarrer à n'importe quelle valeur (le plus souvent 1). Dans l'exemple suivant, on définit un tableau *tab* de 10 variables (indicées de 1 à 10) dont le domaine est l'intervalle 1..20 :

```
array [1..10] of var 1..20: tab;
```

Ici, on déclare les paramètres, notamment la matrice, qui vont servir à définir le problème de l'exemple de la section 1.1.1 :

```
par int: nbpers;
array [1..nbpers, 1..nbpers] of par int: preferences;
```

Les personnes sont représentées par un tableau de variables. Le premier couple est représenté par les variables des indices 1 et 2, le deuxième couple par les variables des indices 3 et 4, etc. La personne A sera représentée par la valeur 1, la personne B par la valeur 2, etc.

```
array [1..nbpers] of var 1..nbpers: couples;
```

Le contenu des tableaux peut être défini, pour les tableaux à une dimension, par la syntaxe : [*expr1*, *expr2*, ...], et pour les tableaux/matrices à deux dimensions, par la syntaxe :

```
[|  l1c1, l1c2, ...,  |
    l2c1, l2c2, ...,  |
    ...               |]
```

Les données de la section 1.1.1 pourraient être définies comme suit :

```
nbpers = 4;
preferences = [|
  0, 0, 6, 10,|
  3, 0, 7,  2,|
  3, 9, 0,  4,|
  7, 8, 4,  0 |];
```

La solution « A avec C, B avec D » sera ainsi représentée par le tableau couples = [1, 3, 2, 4].

On accède aux éléments d'un tableau avec une syntaxe classique : a[i] correspond à l'élément d'indice i dans le tableau a. Pour une matrice, m[i, j] correspond à l'élément situé à la ligne d'indice i et la colonne d'indice j dans la matrice m.

2.2.7 Générateurs

Minizinc propose une syntaxe permettant de générer le contenu d'un tableau. Il s'agit d'une syntaxe que l'on retrouve souvent en programmation fonctionnelle et en mathématiques.

L'idée est d'appliquer une expression (i.e., une formule) sur un ensemble, généralement défini à partir d'intervalles, de produits cartésiens et de filtres. On obtient un tableau qui résulte de l'application de l'expression sur chaque élément de l'ensemble. La syntaxe est [*expression* | *ensemble*]

Par exemple, [2 * i | i in 1..3] correspond au tableau [2, 4, 6]. On la lit « pour tout i de l'intervalle 1..3, réaliser l'opération $2 \times i$ ».

On peut faire des choses plus compliquées, par exemple : [i * j | i, j in 1..3 where i < j] . « i, j in 1..3 » réalise le produit cartésien $1..3 \times 1..3 = \{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)\}$. Le where filtre ce produit cartésien pour ne garder que les éléments pour lesquels $i < j$. Enfin, on réalise pour chaque élément du tableau l'opération $i \times j$. On obtient le tableau [1 * 2, 1 * 3, 2 * 3] = [2, 3, 6].

On peut utiliser les générateurs pour générer des contraintes, à l'aide notamment du mot-clé « **forall** » :

```
constraint forall ( [ couples[i] != couples[j]
    | i, j in 1..nbpers where i < j ]);
```

Équivalent à couples[1]!= couples[2], couples[1]!= couples[3], couples[2]!= couples[3], etc., ce qui signifie que tous les éléments du tableau couples devront être différents.

Enfin, il existe une syntaxe alternative, un peu plus proche du langage mathématique pour le mot-clé **forall**, qui consiste à placer les ensembles servant à la génération avant l'expression :

```
constraint forall (i, j in 1..nbpers where i < j)(
  couples[i] != couples[j] );
```

Ce dernier exemple est tout à fait équivalent au précédent. Observez l'emplacement des parenthèses. On peut également utiliser cette syntaxe pour réaliser des agrégats, portant sur des nombres entiers : sum, product, min et max, par exemple :

```
constraint total = sum (i in 1..nbpers) (couples[i]);
```

Cela correspond à la formule mathématique :

$$\text{total} = \sum_{i \in 1..\text{nbpers}} \text{couples}[i]$$

2.2.8 Contraintes portant sur des tableaux

Il est possible d'utiliser une variable comme indice de tableau ou de matrice. On pourra ainsi obtenir la satisfaction d'un couple dans le problème de la section 1.1.1 :

```
solve maximize sum(i in 1..(nbpers div 2)) (
  preferences[couples[2 * i - 1], couples[2 * i]] +
  preferences[couples[2 * i], couples[2 * i - 1]]);
```

D'autres fonctions sont disponibles :

Concaténation de deux tableaux : `x ++ y` (tableaux à une dimension uniquement)

Changement de dimension : `array1d(x)` si `x` est un tableau à une dimension, il est « aplati » en une dimension (pour une matrice à deux dimensions par exemple, les lignes sont mises bout-à-bout).

`array2d(x, 1..10, 1..20)` transforme le tableau `x` en matrice à deux dimensions de 10 lignes et 20 colonnes (on peut évidemment changer la taille et les indices)

Projection : `col(x, n)` renvoie la n^e colonne de la matrice `x`, `row(x, n)` renvoie la n^e ligne

Taille : `length(x)` renvoie le nombre d'éléments du tableau `x`

2.2.9 Contraintes globales

Les contraintes globales imposent des propriétés « complexes » sur des tableaux. Elles correspondent souvent à des sous-problèmes très généraux qui pourraient être modélisés séparément, mais de manière moins efficaces. Elles sont indispensables pour modéliser efficacement des problèmes de taille industrielle. Elles doivent être « importées » pour être utilisées dans MiniZinc avec la commande **include**.

`all_different(x)` : impose que toutes les valeurs du tableau `x` soient différentes,

`alldifferent_except_0(x)` : idem, mais la valeur 0 peut apparaitre plusieurs fois,

`nvalue(x)` : renvoie le nombre de valeurs différentes apparaissant dans le tableau `x`,

`count(x, n)` : renvoie le nombre de fois où la valeur `y` apparait dans le tableau `x`,

`member(x, e)` : impose que `e` soit un élément du tableau `x`,

`lex_less(x, y)` : impose que les valeurs du tableau `x` soient avant les valeurs du tableau `y` dans l'ordre lexicographique (i.e., « alphabétique »). Il y aussi les variantes `lex_lesseq` (les tableaux peuvent être égaux), `lex_greater` et `lex_greatereq`,

`maximum(x)` : renvoie la plus grande valeur du tableau `x` (il y a aussi `minimum`),

D'autre contraintes sont disponibles, n'hésitez pas à consulter la documentation officielle de MiniZinc [4]...

2.3 Modèle final pour l'affectation de colocataires

On reprend les extraits disséminés dans ce chapitre, en utilisant ici la contrainte globale `all_different` et quelques contraintes supplémentaires pour éliminer des symétries (cf section 2.4.3) :

```
include "all_different.mzn";

par int: nbpers;
array [1..nbpers, 1..nbpers] of par int: preferences;

array [1..nbpers] of var 1..nbpers: couples;

constraint all_different(couples);

% Élimination des symétries
constraint forall (i in 1..nbpers div 2)(
  couples[2*i-1] < couples[2*i]);

constraint forall (i in 2..nbpers div 2)(
  couples[2*i-3] < couples[2*i-1]);


solve maximize sum(i in 1..nbpers div 2)(
  preferences[couples[2 * i - 1], couples[2 * i]] +
  preferences[couples[2 * i], couples[2 * i - 1]]);

nbpers = 4;
preferences = [|
  0, 0, 6, 10,|
  3, 0, 7,  2,|
  3, 9, 0,  4,|
  7, 8, 4,  0 |];
```

2.4 Comment ça marche ?

En réalité, le problème de satisfaction de contraintes (CSP) est un problème combinatoire « générique », c'est-à-dire que n'importe quel problème combinatoire peut être converti en CSP. En théorie de la complexité algorithmique, on dit que le CSP est *NP-complet*. Si on arrive à résoudre efficacement le CSP, on aura résolu tous le problèmes combinatoires ! En attendant, les seules techniques connues pour résoudre le CSP utilisent soit des algorithmes à complexité exponentielle (c'est-à-dire qu'à chaque fois qu'on ajoute une variable au problème, on multiplie potentiellement le temps de calcul par le nombre de valeurs de son domaine), soit des algorithmes qui calculent des solutions approchées. Par exemple, on peut ne pas réussir à satisfaire toutes les contraintes, ou dans le cas des problèmes d'optimisation, s'arrêter sur une solution correcte sans aller jusqu'à l'optimal. Pour une coloration de graphe de très grande taille, on pourra chercher à le colorer en 5 couleurs, même si c'est théoriquement possible de le faire avec seulement 4.

2.4.1 L'algorithme d'exploration arborescente systématique

Pour résoudre un problème combinatoire de manière générique, la technique habituelle consiste à construire un arbre de recherche. On prend une variable du problème, et on lui affecte une valeur, puis on prend une deuxième variable, on lui affecte une valeur, et ainsi de suite. Si la valeur choisie ne satisfait pas une contrainte, on annule la dernière décision et on choisit une autre valeur. Dans l'exemple de la coloration de la carte de France (cf section 1.1.2), on peut suivre la démarche suivante :

1. on colore le Nord-Pas-de-Calais-Picardie avec la première couleur disponible, le blanc.
2. on colore la Normandie voisine également en blanc.
3. comme on ne satisfait pas la contrainte nord_picardie != normandie, on revient au point 2.
4. on colore la Normandie avec la deuxième couleur disponible, le rouge.
5. on colore l'Île-de-France en blanc.
6. comme on ne satisfait pas la contrainte nord_picardie != ile_de_france, on revient au point 5.
7. on colore l'Île-de-France en rouge.
8. comme on ne satisfait pas la contrainte normandie != ile_de_france, on revient au point 7.
9. etc.

2.4.2 Le filtrage

Pour améliorer les performances de l'algorithme d'exploration, l'idée centrale de la programmation par contraintes est de filtrer les valeurs des domaines : au fur et à mesure de la recherche, on supprime des valeurs des domaines des variables quand les contraintes permettent de se rendre facilement compte qu'elles ne peuvent plus être dans une solution. Reprenons l'exemple de la carte de France.

1. on colore le Nord-Pas-de-Calais-Picardie avec la première couleur disponible, le blanc.
2. par filtrage, on supprime le blanc des régions voisines : Normandie, Île de France et Est
3. on affecte alors directement le rouge à la Normandie
4. par filtrage, on supprime le rouge des régions voisines (Bretagne, Loire-Atlantique, Centre et Île de France ; le rouge n'est déjà plus disponible pour le Nord-Picardie).

5. on affecte alors directement la troisième couleur à l'Île-de-France, le vert.
6. on supprime le vert des régions voisines.
7. etc.

L'intérêt du filtrage, c'est qu'il est parfois possible de se rendre compte très vite que certaines affectations sont vouées à l'échec, ce qui permet d'éviter d'explorer de grandes parties de l'arbre. Une bonne modélisation d'un problème essaie d'utiliser au mieux les contraintes disponibles pour améliorer le filtrage. Une bonne connaissance des contraintes disponibles (et notamment des contraintes globales) est nécessaire. Par exemple, on peut utiliser une contrainte all_different(nord_picardie, ile_de_france, est) pour remplacer trois contraintes d'inégalité. Si un filtrage supprime le blanc et le noir de l'Île-de-France et de l'Est, on peut en déduire que l'une des deux sera nécessairement verte et l'autre rouge. On peut alors immédiatement supprimer ces deux couleurs du Nord.

2.4.3 Le problème des symétries

Dans un problème, une symétrie survient quand on peut déduire une solution à partir d'une autre. Pour la carte de France, si on trouve une solution, on peut simplement inverser deux couleurs pour retrouver une autre solution. Pour le problème des appariements de la section 1.1.1, si on a une solution en appariant A avec B, on a une autre solution simplement en appariant B avec A. Cependant, l'algorithme d'exploration systématique, même s'il n'a pas trouvé de solution en appariant A avec B, va quand même essayer d'associer B avec A. Cela génère de nombreux calculs inutile : chaque symétrie va potentiellement multiplier le temps de calcul par 2.

Pour éliminer les symétries, on fixe un ordre aux solutions : chaque couple dans l'ordre alphabétique, et l'ensemble des couples également dans l'ordre alphabétique. La contrainte globale lex_lesseq est beaucoup utilisée pour gérer de genre de cas. Dans le problème de coloration, on peut simplement fixer les couleurs de trois régions mutuellement adjacentes pour diviser par 12 les temps de calcul.

2.4.4 Pour aller plus loin...

Au delà des techniques de base, les chercheurs en IA continuent d'améliorer les solveurs. Par exemple, l'ordre dans lequel les variables sont affectées a beaucoup d'impact sur la taille de l'arbre de recherche : il vaut mieux commencer par les variables les plus contraintes. Si on ne trouve pas de solution au bout d'un certain temps, il peut être intéressant de recommencer la recherche depuis le début en changeant l'ordre d'affectation. On peut chercher à combiner les contraintes entre elles pour améliorer le filtrage. On peut cherche à paralléliser les algorithmes de recherche pour mieux exploiter les microprocesseurs modernes, etc.

Chapitre 3

La Programmation Linéaire

3.1 Comment lier mathématiques et optimisation ?

Tout le monde comprend en quoi consiste une optimisation. D'un point de vue mathématique, une maximisation peut se modéliser par la formulation 1

Problème 1 (Maximisation)

Pour une fonction $f : \mathbb{R}^n \to \mathbb{R}$ donnée, et un ensemble $M \subseteq \mathbb{R}^n$, trouver $\hat{x} \in M$ qui maximise f sur M tel que : $f(\hat{x}) \geq f(x), \quad \forall x \in M$.

Du point de vue de la terminologie :

- f est appelée **fonction-objectif**.
- M est appelé **espace admissible**.
- $\hat{x}$ est appelé **maximiseur de f sur M**.
- Tout $x \in M$ est appelé **solution réalisable ou admissible**.
- Les composantes x_i, $i = \{1, \ldots, n\}$ du vecteur x sont appelées **variables d'optimisation ou de décision**.

Il est possible de maximiser ou de minimiser f sur M. Pour ne pas faire de redondance, on ne travaillera que sur des maximisations. Nous verrons qu'il est toujours possible de transformer une minimisation en maximisation.

3.2 Inéquations linéaires

Partons sur de bonnes bases en rappelant quelques notions liées aux inéquations.

Définition 1 (Inéquation linéaire)

Une inéquation linéaire est une expression de la forme :

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq b,$$

avec x_i les variables, a_i les cœfficients des variables, b une constante et n le nombre d'inconnues.

On peut évidemment inverser le sens d'une inéquation en multipliant par un nombre négatif de chaque côté.

Définition 2 (Solution d'une inéquation linéaire)

On appelera solution de l'inéquation linéaire $a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq b$ tout n-uplet $(y_1, \ldots, y_n)$ tel que l'inégalité $a_1y_1 + a_2y_2 + \ldots + a_ny_n \leq b$ est vraie.

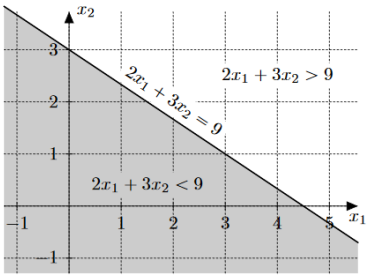


FIG. 3.1 : Exemple de résolution graphique d'une inéquation dans le plan. L'ensemble des solutions de l'inéquation $2x_1 + 3x_2 \leq 9$ est un demi-plan dans le système d'axes Ox_1x_2 (en gris). La frontière de ce demi-plan est la droite $2x_1 + 3x_2 = 9$.

Définition 3 (Système d'inéquations linéaires)

On appelle système de m inéquations linéaires à n inconnues un système de la forme :

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n \leq b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n \leq b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \ldots + a_{3n}x_n \leq b_3 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + a_{m3}x_3 + \ldots + a_{mn}x_n \leq b_m. \end{cases}$$

où x_j est une variable dans la colonne j, a_{ij} est le cœfficient de la variable x_j sur la ligne i, b_i est la constante de la ligne i, n est lenombre d'inconnues et m est le nombre d'inéquations.

3.3 De l'algèbre au programme linéaire

Faisons quelques rappels d'algèbre linéaire. Nous savons tous que cela est nécessaire ! (cf. module M1202)

Définition 4 (Fonction linéaire)

Une fonction $f : \mathbb{R}^n \to \mathbb{R}$ est linéaire si et seulement si $f(x+y) = f(x) + f(y)$ et $f(\lambda x) = \lambda f(x)$, avec $x, y \in \mathbb{R}^n$ et $\lambda \in \mathbb{R}$.

Exemple de fonctions linéaires :

- $f_1(x) = x$,
- $f_2(x) = 3x_1 - 5x_2$,
- $f_3(x) = Ax$.

Exemple de fonctions non-linéaires :

- $f_5(x) = 1$,
- $f_6(x) = x + 1$,
- $f_7(x) = x^2$,

- $f_7(x) = sin(x)$.

Remarque 1

Toute fonction linéaire $f : \mathbb{R}^n \to \mathbb{R}$ peut s'exprimer dans la forme $f(x) = Ax$, avec $A \in \mathbb{R}^{m \times n}$ une matrice.

Fournissons quelques détails maintenant sur les notations :

- $x \in \mathbb{R}^n$ est le vecteur colonne $\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$ avec $x_1, x_2, \ldots, x_n \in \mathbb{R}$.

- x^T correspond à la transposée du vecteur x, qui est alors le vecteur ligne $(x_1, x_2, \ldots, x_n)$.

- $A \in \mathbb{R}^{m \times n}$ est la matrice $\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}$.

- A^T est la transposée de la matrice A, soit $\begin{pmatrix} a_{11} & a_{21} & \ldots & a_{n1} \\ a_{12} & a_{22} & \ldots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \ldots & a_{nm} \end{pmatrix}$.

À partir de ces notations, nous allons définir ce qu'est notre problème d'optimisation linéaire. Il s'agit en effet d'un cas particulier d'optimisation, pour lequel la fonction f est linéaire et l'ensemble M est défini comme l'intersection d'un nombre fini de plans à partir des contraintes d'égalité et d'inégalités. Voyons de quoi cela a l'air.

Définition 5 (Programme Linéaire)

$$(LP)\begin{cases} \max c^T x = c_1x_1 + c_1x_1 + \ldots + c_nx_n = \sum_{i=1}^{n} c_ix_i \\ a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n \begin{pmatrix} \leq \\ = \\ \geq \end{pmatrix} b_i, \quad i = 1, \ldots, m \\ x_i \begin{pmatrix} \leq \\ \geq \end{pmatrix} 0, \quad i \in I \subseteq \{1, \ldots, m\} \end{cases}$$

où $x \in \mathbb{R}^n$ est le vecteur de variables inconnues.

La fonction $f(x) = c^T x$ est appelée la ***fonction-objectif*** ou ***fonction de coût*** ou ***fonction économique***. Le vecteur $c \in \mathbb{R}^n$ est le **vecteur de coût**. La matrice A (de terme général (a_{ij})) et le vecteur b collectent les informations des **contraintes**.

Remarque 2 (Résolution du (LP))

Généralement, la matrice A n'est pas carrée ($m \neq n$). Par conséquent, la résolution du (LP) est impossible par inversion matricielle. Habituellement, A a plus de colonnes que de lignes, signifiant qu'il y a plus d'inconnues que d'équations de contraintes. Le système est **sous-déterminé**. De ce fait, un grand choix de solutions potentielles maximisant $c^T x$ existe dans l'espace admissible M.

Remarque 3 (Maximisation et minimisation)

Dans la littérature, l'opérateur de minimisation remplace parfois celui de maximisation. Le passage de l'un à l'autre requiert une simple manipulation : la maximisation de $c^T x$ devient la minimisation de $-c^T x$. Le résultat de la fonction objectif sera du coup $-f(x)$.

3.4 Résolution d'un problème linéaire

La résolution du problème (LP) peut nécessiter une réécriture pour appliquer des algorithmes de résolution comme celui du simplexe (cf. section 3.4.3). Toutefois, dans

le cas où seules 2 voire 3 variables sont utilisées, une résolution graphique peut être effectuée.

3.4.1 Résolution graphique

Par souci de lecture et de construction des graphiques, nous nous limiterons au cas 2D. Nous avons évoqué au préalable la notion de demi-plan comme solution d'une inéquation, chaque inéquation correspondant à une contrainte.

L'idée est simple : construire la zone-solution en obtenant l'intersection de tous les demi-plans qui sont solutions des inéquations.

Puisqu'un dessin vaut mieux qu'un long discours, illustrons nos propos par un exemple. Par la suite, nous allons chercher à résoudre le (LP) suivant :

$$\begin{aligned} \max \quad & 100x_1 + 250x_2 \\ \text{sujet à :} \quad & x_1 + x_2 \leq 40, \\ & 40x_1 + 120x_2 \leq 2400, \\ & 6x_1 + 12x_2 \leq 312, \\ & x_1, x_2 \geq 0. \end{aligned}$$

Généralités sur les fonctions de $\mathbb{R}^2$

Chaque contrainte correspond permet d'établir une équation de droite. L'équation générale d'une droite dans $\mathbb{R}^2$ s'écrit :

$$h(x, y) = ax + by = c.$$

À partir de cette formulation, nous pouvons définir un vecteur normal à la droite, i.e. la perpendiculaire à la droite. Ce vecteur, noté n_h ici, s'écrit :

$$n_h = \begin{pmatrix} a \\ b \end{pmatrix}.$$

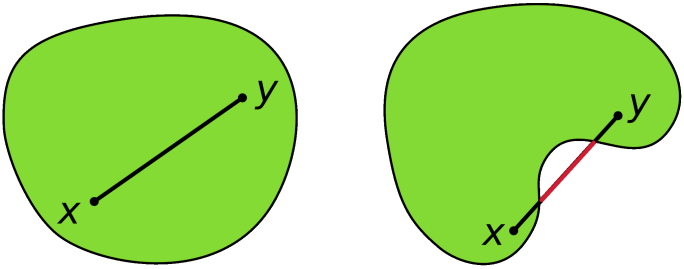
Il est orienté dans la direction des valeurs croissantes de h. S'il est orienté dans l'autre sens, il sera tourné vers les valeurs décroissantes de h. Cette notion est essentielle pour savoir quel demi-plan il nous faut récupérer avec les inéquations.

Généralités sur les polyèdres convexes

L'espace réalisable est construit par l'intersection de tous les *demi-espaces* (car soit on travaille avec $\leq$, soit avec $\geq$) à partir des contraintes. L'intersection de tous ces demi-espaces est ce que l'on appelle un **polyèdre**.

Définition 6 (Ensemble convexe)

Un ensemble M est dit convexe si l'intégralité d'une ligne connectant deux points de M appartient à M.



(a) Ensemble convexe

(b) Ensemble non-convexe

Théorème 1 (À propos des ensembles convexes)

- L'ensemble-solution d'une inéquation linéaire est un ensemble convexe.
- L'intersection de deux ou de plusieurs ensembles convexes est un ensemble convexe.
- L'ensemble-solution d'un système d'inéquations linéaires est un ensemble convexe.

Le théorème suivant illustre finalement l'importance de la relation entre la solution du problème linéaire et de l'ensemble convexe.

Théorème 2 (Solution sur un polyèdre)

Soit f une fonction linéaire définie sur un polyèdre convexe borné. Alors la fonction f atteint sa valeur maximale en au moins un des sommets du polyèdre convexe.

Au sujet des contraintes

Pour la résolution graphique, travaillons avec les 3 contraintes de notre problème. Pour cela, nous introduisons les 3 fonctions linéaires suivantes :

- $g_1(x_1, x_2) = x_1 + x_2$,
- $g_2(x_1, x_2) = 40x_1 + 120x_2$,
- $g_3(x_1, x_2) = 6x_1 + 12x_2$.

Les équations $g_1(x_1, x_2) = 0$, $g_2(x_1, x_2) = 0$, $g_3(x_1, x_2) = 0$, définissent des lignes droites dans $\mathbb{R}^2$.

Avec $g_1(x_1, x_2) = x_1 + x_2$, la normale est $n_{g_1} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. Le vecteur pointant vers les valeurs croissantes de g_1, il va mener à tout point $(x_1, x_2)^T$ avec $g(x_1, x_2) > 40$. Dans la direction opposée, on atteindra tous les points $(x_1, x_2)^T$ avec $g(x_1, x_2) < 40$. Puisque la contrainte du problème est $x_1 + x_2 \leq 40$, seuls les points avec $g(x_1, x_2) < 40$ sont dans l'espace réalisable.

Le même travail est effectuée avec les autres contraintes, ce qui nous fera donc 3 lignes au total (cf. figure 3.2).

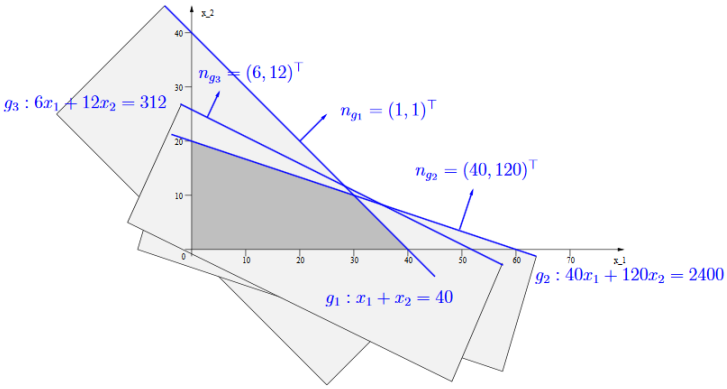


FIG. 3.2 : Illustration des contraintes avec les 3 fonctions g_1, g_2 et g_3.

Au sujet de la fonction-objectif

Maintenant que l'espace de solution admissible est obtenu, il nous reste à trouver le meilleur point y appartenant qui satisfera au mieux la fonction de coût.

Dans notre problème, l'objectif est de maximiser :

$$f(x_1, x_2) = 100x_1 + 250x_2.$$

Tous les points sur la droite $f(x_1, x_2) = d$ donnera d comme valeur de la fonction-objectif. La figure ci-dessous illustre deux déplacements de f :

1. Avec $d = 3500$,

2. Avec $d = 5500$.

Comme précédemment, les valeurs de la fonction vont augmenter dans la direction de la normale de f. Ici, sa normale est :

$$n_f(x_1, x_2) = \begin{pmatrix} 100 \\ 250 \end{pmatrix}.$$

Puisque nous devons maximiser f, nous allons déplacer la droite de la fonction-objectif dans la direction de n_f : le ou les derniers points de la ligne qui intersecteront l'espace réalisable seront nos optimums. La figure 3.3 nous montre que la solution optimale est trouvée en un point $(x_1, x_2)^T = (30, 10)^T$, avec $f(x_1, x_2) = 5500$.

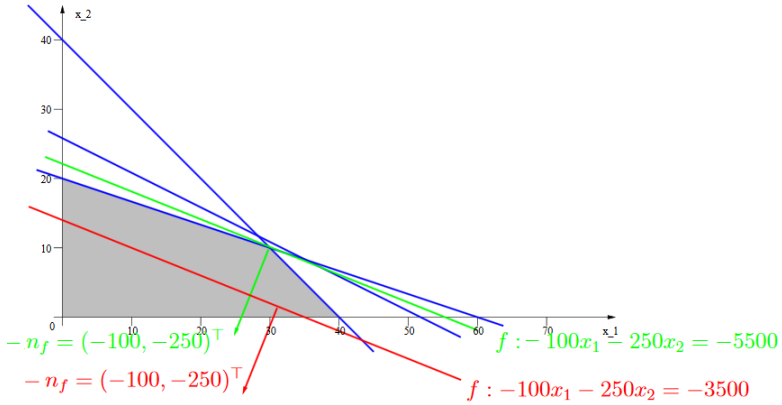


FIG. 3.3 : Déplacement de la fonction-objectif (en rouge) vers l'optimum (en vert).

Configurations possibles

Au final, l'intersection entre la fonction-objectif et l'ensemble des contraintes - noté M_c - peut conduire à 0, 1, ou à une infinité d'optimums. La figure 3.4 met en évidence les différentes configurations qu'il est possible de rencontrer en programmation linéaire.

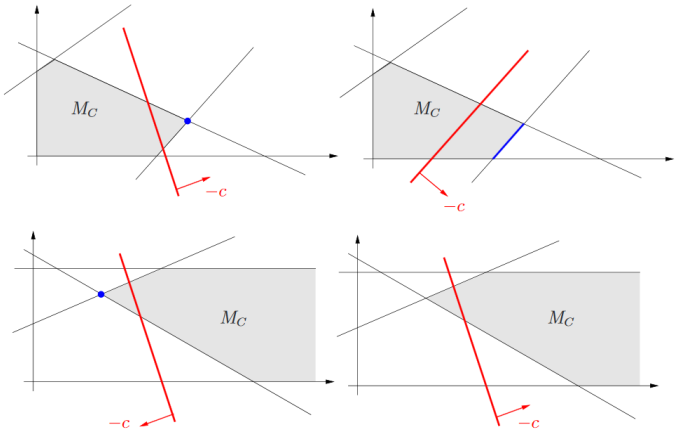


FIG. 3.4 : Différentes géométries de l'espace réalisable (en gris), de la fonction-objectif (en rouge) et de la solution optimale (en bleu).

Remarque 4

- L'ensemble M_c peut être borné (les 2 tracés du haut de la figure 3.4) ou non borné (les 2 tracés du bas de la figure 3.4).
- Si M_c est borné, il existera toujours une solution optimale.
- L'optimum peut être unique (les 2 tracés de gauche de la figure 3.4) ou non (les 2 tracés de droite de la figure 3.4).
- Même si M_c est non borné, une solution pourrait exister. Tout dépend du vecteur de coût c.
- Si une solution optimale existe, l'un des sommets de M_c en fera toujours partie.

Résumé des étapes à suivre :

1. Tracer l'ensemble réalisable M_c à partir des contraintes.
2. Tracer la fonction-objectif. Dans la pratique, on la fait initialement passer par $0_{\mathbb{R}^n}$ (par $(0,0)^T$ dans $\mathbb{R}^2$).

3. Déplacer la ligne de la fonction-objectif dans la direction définie par le vecteur c pour une maximisation.

4. La solution optimale est l'intersection la plus extrême entre M_c et la ligne de la fonction-objectif.

3.4.2 Du programme linéaire à la forme standard

La résolution du simplexe se fait par des tableaux. Vous aurez l'occasion de vous exercez à la main avec un nombre de contraintes et d'inconnues raisonnable. Dès que leur nombre devient conséquent, la programmation est de mise.

La résolution du problème (LP) peut nécessiter une réécriture pour appliquer des algorithmes de résolution. Détaillons la réécriture sous chacune de ces deux formes, qui sont la forme **canonique** et la forme **standard**.

La forme canonique

La forme canonique est décrite dans l'encadré suivant. Il s'agit de la formulation du problème linéaire avec des inéquations de même sens.

Définition 7 (Forme canonique (LPC))

Soient $c = (c_1, c_2, \ldots, c_n)^T \in \mathbb{R}^n$, $b = (b_1, b_2, \ldots, b_m)^T \in \mathbb{R}^m$ et $A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix} \in \mathbb{R}^{m \times n}$. Le programme linéaire canonique consiste à trouver $x = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n$ tel que la fonction-objectif $\sum_{i=1}^{n} c_i x_i$ soit maximal tout en étant sujet aux contraintes :

$$\begin{aligned} \sum_{j=1}^{n} a_{ij} x_j &\leq b_i, & i = 1, \ldots, m \\ x_j &\geq 0, & j = 1, \ldots, n. \end{aligned}$$

Remarque 5 (Au sujet du (LPC))

- L'ensemble $M_c = \{x \in \mathbb{R}^n | Ax \leq b, x \geq 0\}$ est l'espace admissible de (LPC).
- Si $x \in M_c$, x est une solution réalisable de (LPC).
- $\hat{x} \in M_c$ est la solution optimale si $\forall x \in M_c$, $c^T \hat{x} \geq c^T x$.

La transformation d'un problème (LP) en (LPC) se fait en mettant toutes les contraintes sous la forme $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$ et $x_j \geq 0, \quad j = 1, \ldots, n$. Un travail est à effectuer sur les contraintes et les variables :

1. Les contraintes d'inégalités :

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i$$

peut être mis sous la forme $\leq$ en multipliant les deux opérandes par -1 :

$$\sum_{j=1}^{n} (-a_{ij}) x_j \leq -b_i$$

2. Les contraintes d'égalités :

$$\sum_{j=1}^{n} a_{ij} x_j = b_i$$

peut être mis sous la forme $\leq$ en écrivant :

$$b_i \leq \sum_{j=1}^{n} a_{ij} x_j \leq b_i$$

On obtient alors deux inégalités plutôt qu'une seule égalité. En se servant du premier point, on obtient les deux inégalités sous la bonne forme :

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i$$

$$\sum_{j=1}^{n} (-a_{ij}) x_j \leq -b_i$$

3. Les variables libres :

   Si une variable n'est pas contrainte à être positive ou nulle, elle est dite **libre**. Tout nombre réel x_i libre est décomposé en $x_i = x_i^+ - x_i^-$, avec $x_i^+, x_i^- \geq 0$. Les contraintes $x_i^+ \geq 0$ et $x_i^- \geq 0$ doivent par ailleurs être ajoutées. Pour chaque x_i, nous avons maintenant deux variables supplémentaires x_i^+ et x_i^-.

Exemple :

Considérons le problème de maximisation suivant :

$$\begin{aligned} \max \quad & -1{,}2x_1 - 1{,}8x_2 - x_3 \\ \text{sujet à :} \quad & \\ & x_1 \geq -\frac{1}{3} \\ & x_1 - 2x_3 \leq 0 \\ & x_1 - 2x_2 \leq 0 \\ & x_2 - x_1 \leq 0 \\ & x_3 - 2x_2 \leq 0 \\ & x_1 + x_2 + x_3 = 1 \\ & x_2, x_3 \geq 0. \end{aligned}$$

La transformation en un (LPC) implique de changer certaines contraintes. La variable x_1 est une variable libre. Par conséquent, on remplace x_1 par $x_1^+ - x_1^-$, $x_i^+, x_i^- \geq 0$. La contrainte $x_1 + x_2 + x_3$ devient : $x_1^+ - x_1^- + x_2 + x_3 \leq 1$ et $-x_1^+ + x_1^- - x_2 - x_3 \leq -1$. La contrainte $x_1 \geq -\frac{1}{3}$ devient $-x_1^+ + x_1^- \leq \frac{1}{3}$.

On obtient alors le programme linéaire canonique suivant :

$$\begin{aligned} \max \quad & -1{,}2x_1 - 1{,}8x_2 - x_3 \\ \text{sujet à :} \quad & \end{aligned}$$

$$\begin{pmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & -2 \\ 1 & -1 & -2 & 0 \\ -1 & 1 & 1 & 0 \\ 0 & 0 & -2 & 1 \\ 1 & -1 & 1 & 1 \\ -1 & 1 & -1 & -1 \end{pmatrix} \begin{pmatrix} x_1^+ \\ x_1^- \\ x_2 \\ x_3 \end{pmatrix} \leq \begin{pmatrix} 1/3 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ -1 \end{pmatrix}, \text{ avec } \begin{pmatrix} x_1^+ \\ x_1^- \\ x_2 \\ x_3 \end{pmatrix} \geq 0.$$

La forme canonique est très utile pour visualiser les contraintes et résoudre le problème graphiquement. Comme indiqué précédemment, la visualisation sera d'autant plus simple que n est petit (l'idéal étant une dimension 2, voire 3). Toutefois, pour travailler avec des algorithmes de résolution, passer des inéquations aux équations se révèle nécessaire. Découvrons pour cela la forme **standard**.

La forme standard

En partant d'inéquations, il va falloir utiliser des variables supplémentaires pour transformer chaque contrainte en équations, sous la forme $\sum_{j=1}^{n} a_{ij}x_j = b_i$ avec $x_j \geq 0$. Cette étape nécessite quelques précautions.

Dans le cas simple, les contraintes sont toutes de type inférieur ou égal avec un second membre $b \geq 0$. La mise sous forme standard consiste, pour chaque contrainte, à introduire les **variables d'écart** représentant l'écart entre la quantité disponible de la ressource et la quantité effectivement utilisée par l'ensemble des x_j. Formalisons la forme standard.

Définition 8 (Forme standard (LPS))

Soient $c = (c_1, c_2, \ldots, c_n)^T \in \mathbb{R}^n$, $b = (b_1, b_2, \ldots, b_m)^T \in \mathbb{R}^m$ et $A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix} \in \mathbb{R}^{m \times n}$, avec $rang(A) = m$. Le programme linéaire standard consiste à trouver $x = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n$ tel que la fonction-objectif $\sum_{i=1}^{n} c_i x_i$ soit maximal tout en étant sujet aux contraintes :

$$\begin{aligned} \sum_{j=1}^{n} a_{ij}x_j &= b_i, & i = 1, \ldots, m \\ x_j &\geq 0, & j = 1, \ldots, n. \end{aligned}$$

Comment passer du (LPC) au (LPS) dans la pratique ? Soit un programme linéaire canonique comme défini plus haut. Nous définissons un vecteur $y = (y_1, \ldots, y_m)^T \in \mathbb{R}^m$ tel que :

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i,\ i = 1, \ldots, m \qquad \Rightarrow \qquad \sum_{j=1}^{n} a_{ij}x_j + y_i = b_i,\ i = 1, \ldots, m.$$

Ce vecteur y contient les **variables d'écart**. De même que pour x, $y \geq 0$.

Avec $\hat{x} = (x, y)^T \in \mathbb{R}^{n+m}$, $\hat{c} = (c, 0)^T \in \mathbb{R}^{n+m}$ et $\hat{A} = (A|I) \in \mathbb{R}^{m \times (n+m)}$, on obtient le (LPS) suivant :

$$\begin{aligned} \max \quad & \hat{c}^T \hat{x} \\ \text{sujet à :} \quad & \hat{A}\hat{x} = b, \\ & \hat{x} \geq 0 \end{aligned}$$

Remarque 6 (Au sujet du (LPS))

- Le $rang(A) = m$ signifie que toutes les contraintes sont linéairement indépendantes (cf. module M1202 !).
- Le (LPS) n'a de sens que si $m < n$. Comme pour le (LP), il serait possible de résoudre le problème par inversion matricielle sinon, ce qui ne donnerait plus aucun sens à ce cours :).
- L'ensemble $M_s = \{x \in \mathbb{R}^n | Ax = b, x \geq 0\}$ est l'espace admissible du (LPS).
- Un élément x provenant de M_s est une solution réalisable du (LPS).
- $\hat{x} \in M_s$ est la solution optimale si $\forall x \in M_s$, $c^T \hat{x} \geq c^T x$.
- **Attention !** Les quantités c, b et A ne sont pas les mêmes dans le (LPC) et dans le (LPS), puisque nous allons ajouter de nouvelles variables.

Remarque 7 (Attention, cas simple !!)

Dans les problèmes plus proches de la vie réelle, la modélisation et leur résolution de problèmes n'est pas aussi simple. Certaines contraintes peuvent être toutes de type **inférieur**, **supérieur**, **égal**, avec $b \leq 0$ ou $b \geq 0$. Des précautions supplémentaires sont alors à prendre, car en appliquant le même principe que pour les variables d'écart, l'origine que l'on prend habituellement comme solution initiale réalisable n'est plus une solution réalisable. Il faut faire intervenir des **variables de surplus** et des **variables artificielles**.

Nous ne traiterons pas ce cas par la suite, mais sachez que nous ne travaillons que sur un cas précis de programmation linéaire.

3.4.3 La résolution par les tableaux

Notations et principe général

La méthode par tableaux permet de calculer une solution optimale au (LP). Avant d'aller plus loin, formulons le théorème fondamental de programmation linéaire.

Théorème 3 (Théorème fondamental de programmation linéaire)

Soit un (LPS), avec $M_s \neq \varnothing$. Alors :

- Soit la fonction-objectif n'est pas bornée et il n'y a pas de solution optimale, soit le problème a une solution optimale et au moins un sommet de M_s est parmi ces solutions.
- Si M_s est borné, une solution optimale existe, et $x \in M_s$ est optimal si et seulement si combinaison convexe de sommets optimaux.

Comme indiqué précédemment, il nous faut travailler sur le (LPS) :

$$\begin{array}{ll} \max & c^T x \\ \text{sujet à :} & Ax = b, \\ & x \geq 0. \end{array}$$

L'idée du simplexe est de se déplacer d'un sommet de l'ensemble réalisable à un voisin du même ensemble, puis de répéter la procédure jusquà ce que le sommet optimum soit atteint.

Introduisons les notations que nous utiliserons par la suite :

- Les lignes de A sont notées : $a_i^T = (a_{i1}, \ldots, a_{in}) \in \mathbb{R}^n$, $\forall i = 1, \ldots, m$. Soit :

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix} = \begin{pmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_m^T \end{pmatrix}.$$

- Les colonnes de A sont notées : $a^j = (a_{1j}, \ldots, a_{mj}) \in \mathbb{R}^m$, $\forall j = 1, \ldots, n$. Soit :

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix} = \begin{pmatrix} a^1 & a^2 & \ldots & a^n \end{pmatrix}.$$

- Soit $B = \{i \in \{1, \ldots, n\} \mid x_i > 0\}$ l'ensemble des indices des composantes positives de x.

- Soit $N = \{1, \ldots, n\} \backslash B = \{i \in \{1, \ldots, n\} \mid x_i = 0\}$ l'ensemble des indices des composantes nulles de x.

En considérant B, on peut écrire :

$$b = Ax = \sum_{j=1}^{n} a^j x_j = \sum_{j \in B} a^j x_j .$$

Cette équation linéaire de composantes de x a une unique solution si les vecteurs $a^j, j \in B$ sont linéairement indépendants (cf. module M1202). Dans ce cas, x est dit **solution réalisable de la base**.

Théorème 4

$x \in M_s$ est un sommet de M_s si et seulement si x est une solution réalisable de la base.

Le principe général du simplexe est :

1. En considérant toutes les contraintes d'un (LPC), nous obtenons un ensemble convexe. On sait en outre maintenant que la **solution optimale**, si elle existe, est **l'un des sommets de l'ensemble**.

2. Pour la calculer, nous allons **partir d'un de ces sommets**.

3. Nous allons **nous déplacer de sommet en sommet** le long des arêtes du polyèdre convexe jusqu'à ce que nous ayons trouver la solution optimale.

Il reste donc à répondre aux 3 questions suivantes :

1. Comment choisir l'un des sommets du polyèdre ?

2. Comment se déplacer de sommet en sommet ?

3. Quels sont les critères d'arrêt nous indiquant que nous avons atteint notre optimum ?

Choix d'un sommet

D'après le théorème 3, il est suffisant de calculer les sommets de l'ensemble réalisable pour obtenir au moins une solution optimale. Mais comment peut-on faire cela ?

Avant tout, grâce au théorème 4, nous savons qu'un sommet peut être caractérisé par les colonnes indépendantes de A. En effet, la solution doit vérifier les contraintes, i.e. qu'il faut trouver un vecteur x qui vérifie $Ax = b$. Appelons h l'application linéaire telle que $h(x) = Ax$.

Pour que b soit un élément de $Im(h)$ (les éléments Ax qui sont créés par h), il faut évidemment trouver au moins un vecteur x tel que $h(x) = b$. Or, si les vecteurs colonnes de A sont linéairement indépendantes, le système admet une solution unique (même principe qu'en M1202 ! Eh oui, ça vous sert !). Si on sélectionne des colonnes de la matrice A et que cette sélection conduit à $Ax = b$, alors x est une solution réalisable dans la base choisie.

Exemple de sélection de colonnes de A :

$$x = \begin{pmatrix} -1 \\ 10 \\ 3 \\ -4 \end{pmatrix}, A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}, B = \{2, 4\} \Rightarrow x_B = \begin{pmatrix} 10 \\ -4 \end{pmatrix}, A_B = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix}.$$

Remarque 8 (Nombre de colonnes et dimension de l'espace)

Le nombre de colonnes à choisir dépend évidemment de la dimension de x. Si x est de dimension n, n colonnes seront nécessaires. Ceci est logique, car pour déterminer un point dans $\mathbb{R}^2$, 2 lignes sont nécessaires ; dans $\mathbb{R}^3$, 3 plans sont nécessaires ; il est possible de généraliser cela dans $\mathbb{R}^n$, en travaillant avec n hyperplans (entités de dimension $(n-1)$).

L'algorithme 1 permet de calculer ces fameux sommets.

Algorithme 1 (Calcul d'un sommet du polyèdre)

1. Choisir de m colonnes indépendantes $a^j, j \in B, B \subseteq \{1, \ldots, n\}$ de A et construire N de sorte que $N = \{1, \ldots, n\} \backslash B$.
2. Fixer $x_N = 0$ et résoudre l'équation linéaire $A_B x_B = b$.
3. Si $x_B \geq 0$, x est une solution réalisable de la base. Nous avons un sommet ! S'il existe par contre un $i \in B$ tel que $x_i < 0$, alors x n'est pas réalisable, et la procédure est à réitérer dans un autre choix de colonnes indépendantes.

Trouver les sommets du polyèdre consiste donc "simplement" à résoudre un système d'équations linéaires.

Exemple de calcul de sommets :

Considérons les contraintes définies par $Ax = b, x \geq 0$, avec :

$$A = \begin{pmatrix} 2 & 3 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}, b = \begin{pmatrix} 6 \\ 2 \end{pmatrix}.$$

Il y a au plus $\binom{4}{2}$ combinaisons possibles de 2 colonnes de A :

1. $B_1 = \{1, 2\} \Rightarrow x_{B_1} = A_{B_1}^{-1} b = \begin{pmatrix} 2 \\ 2/3 \end{pmatrix}, x^1 = (2, 2/3, 0, 0)^T$.
2. $B_2 = \{1, 3\} \Rightarrow x_{B_2} = A_{B_2}^{-1} b = \begin{pmatrix} 2 \\ 2 \end{pmatrix}, x^2 = (2, 0, 2, 0)^T$.
3. $B_3 = \{1, 4\} \Rightarrow x_{B_3} = A_{B_3}^{-1} b = \begin{pmatrix} 3 \\ -1 \end{pmatrix}, x^3 = (3, 0, 0, -1)^T$.
4. $B_4 = \{2, 3\} \Rightarrow A_{B_4}$ est singulière (non inversible), donc pas de solution,
5. $B_5 = \{2, 4\} \Rightarrow x_{B_5} = A_{B_5}^{-1} b = \begin{pmatrix} 2 \\ 2 \end{pmatrix}, x^5 = (0, 2, 0, 2)^T$.
6. $B_6 = \{3, 4\} \Rightarrow x_{B_6} = A_{B_6}^{-1} b = \begin{pmatrix} 6 \\ 2 \end{pmatrix}, x^6 = (0, 0, 6, 2)^T$.

Remarque 9 (Approche naïve)

- Puisque l'on sait que la solution optimale est l'un des sommets du polyèdre, l'approche naïve serait de tous les calculer. Mais choisir m colonnes parmi n quand m et n sont très grands ne serait absolument pas efficace ! Le nombre de possibilités exploserait.
- Par exemple, en choisissant ne serait-ce que $m = 90$ (nombre de contraintes) et $n = 40$ (nombre d'inconnues), on aboutit déjà à $\binom{n}{m} \approx 6 * 10^{25}$ façons de choisir les colonnes.
- En outre, si M_s n'est pas borné, trouver une solution n'est pas non plus garanti.

Dans les paragraphes précédents, la notion de base est apparue. Nous allons formaliser cette définition car nous en aurons besoin par la suite.

Définition 9 (Base et éléments hors-base)

- Soit x une solution réalisable d'un (LPS), avec A la matrice de m contraintes. Tout système $\{a^j | j \in B\}$ de m colonnes linéairement indépendantes de A, qui inclut ces colonnes a^j telles que $x_j > 0$, est appelé **base** de x.
- Soit $\{a^j | j \in B\}$ une base de x. L'ensemble d'indices B est appelé **ensemble d'indices de base** et l'ensemble d'indices $N = \{1, \ldots, n\} \backslash B$ est appelé **ensemble d'indices hors-base**.
- La matrice $A_B = (a^j)_{j \in B}$ est appelée **matrice de la base** et la matrice $A_N = (a^j)_{j \in N}$ est appelée **matrice hors-base**.
- Le vecteur $x_B = (x_j)_{j \in B}$ est appelé **variable de la base** et le vecteur $x_N = (x_j)_{j \in N}$ est appelé **vecteur hors-base**.

Exemple de construction de la base et des éléments hors-base :

Considérons les contraintes définies par $Ax = b, x \geq 0$, avec :

$$A = \begin{pmatrix} 1 & 4 & 1 & 0 & 0 \\ 3 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix}, b = \begin{pmatrix} 24 \\ 21 \\ 9 \end{pmatrix}, x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}.$$

Considérons que $x = (6, 3, 6, 0, 0)^T \in M_s$ est une solution réalisable. Pour cela, les colonnes de A choisies sont : $\begin{pmatrix}1\\3\\1\end{pmatrix}$, $\begin{pmatrix}4\\1\\1\end{pmatrix}$ et $\begin{pmatrix}1\\0\\0\end{pmatrix}$. Les 3 colonnes ont linéairement indépendantes. D'après le théorème 4, il s'agit d'un sommet de l'ensemble convexe. La base est donc formée par ces 3 vecteurs. On peut écrire :

- $B = \{1, 2, 3\}$ et $N = \{4, 5\}$,
- $x_B = (6, 3, 6)^T$ et $x_N = (0, 0)^T$,
- $A_B = \begin{pmatrix}1 & 4 & 1\\3 & 1 & 0\\1 & 1 & 0\end{pmatrix}$ et $A_N = \begin{pmatrix}0 & 0\\1 & 0\\0 & 1\end{pmatrix}$.

Maintenant que nous savons claculer un sommet, il faut voir comment nous déplacer de sommet en sommet.

Changement de sommet

Dans la plupart des problèmes, cette partie est répétée plusieurs fois. Lidée est de modifier les ensembles B et N pour calculer un nouveau sommet. En effet, un sommet du polyèdre est calculé en choisissant des colonnes linéairement indépendantes de A. Donc si nous choisissons une autre combinaison de colonnes, un autre sommet sera obtenu (cf. l'exemple sur les calculs des sommets du polyèdre).

Par la suite, nous cherchons à calculer une meilleure solution réalisable x^+ de la base. Les deux ensembles d'indices suivants sont utilisés :

$$B^+ = (B \backslash \{p\}) \cup \{q\}$$
$$N^+ = (N \backslash \{q\}) \cup \{p\}$$

Quelles sont les différences par rapport à B et N ? Nous avons un changement entre les indices p et q. L'indice p sort de B et rentre dans N, alors que l'indice q sort de N et rentre dans B. Cette procédure s'appelle le **changement de base**.

Formalisation :

Peut-être vous en doutez-vous, mais le choix des indices p et q n'est pas fait au hasard (on pourrait, mais cela ressemblerait beaucoup à l'approche naïve). Certains critères doivent être respectés :

1. Si x est réalisable, x^+ doit rester réalisable : $Ax = b, x \geq 0 \Rightarrow Ax^+ = b, x^+ \geq 0$.
2. Pour une maximisation, la valeur de la fonction-objectif doit augmenter : $c^T x^+ \geq c^T x$.

Les calculs doivent maintenant nous fournir la manière dont cette augmentation doit être faite, i.e. quels choix formuler sur B^+ et N^+, i.e. quels choix effectuer pour les pivots p et q. Le principe est illustré par la figure 3.5. On considère un rayon z partant de x dans une direction s en parcourant une longueur de t :

$$z(t) = x + ts,\ t \geq 0.$$

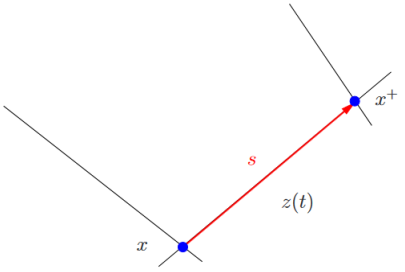


FIG. 3.5 : Idée du changement de base dans la méthode du simplexe. Il faut trouver la direction s telle que la fonction-objectif croisse le long de cette direction.

Afin de ne pas surcharger le document, nous omettrons les calculs par la suite. Il vous faut tout de même chercher à comprendre ce qu'il en est. En effet, sans rentrer dans les détails, il est possible de choisir la direction s de sorte que seulement la variable hors-base x_q soit modifiée sans toucher aux autres composantes $(x_j)_{j \in N, j \neq q}$ qui restent alors à 0. En choisissant q ainsi, x_q quitte les variables hors-base et devient une variable de la base. Le pivot q est déterminé. Nous verrons dans l'exemple pratique plus loin que si le coût associé à une variable hors-base est positif, alors la solution de base courante n'est pas encore optimale.

Dans la même idée, une fois le pivot q déterminé, nous pouvons trouver p de sorte qu'il existe un t_{min} tel que $z(t_{min}) = x^+$ soit réalisable, avec $x_p = 0$. De ce fait, x_p quitte les variables de la base et devient une variable hors-base.

D'un point de vue pratique :

Pour le choix du pivot q, le principe consiste à maximiser la fonction-objectif. De ce fait, choisir une variable hors-base contribuant à ce besoin est de rigueur. Nous allons donc sélectionner une variable hors-base dont le coefficient dans la fonction-objectif est **positif**.

Concernant le pivot p, la conceptualisation est plus délicate, mais possible. La relation que les solutions réalisables doivent vérifier est $Ax = b$ avec $x \geq 0$. Soit x_q la variable entrante. L'arrivée de x_q va augmenter la valeur de la fonction-objectif. Pour

que la relation continue à être vérifier, il faut qu'une variable sorte pour réduire la valeur de la fonction. Afin de savoir laquelle sortir, nous allons augmenter x_q jusqu'à ce que l'une des variables s'annule. La première variable de la base à s'annuler sera notre variable sortante.

$$\begin{aligned} Ax = b &\Leftrightarrow A_B x_B + a^q x_q = b \text{ où } a^q \text{ désigne la } q\text{-ième colonne de } A \\ &\Leftrightarrow x_B = A_B^{-1}(b - a^q x_q) \\ &\Leftrightarrow x_B = \underline{x}_B - A_B^{-1} a^q x_q \\ &\Leftrightarrow x_B = \underline{x}_B - r x_q \end{aligned}$$

avec $r = A_B^{-1} a^q \in \mathbb{R}^m$, $\underline{x}_B$ la solution réalisable précédente. Pour que la solution soit réalisable, il faut que $x_B \geq 0$, et donc $\underline{x}_B - r x_q \geq 0$.

Plusieurs cas de figure se présentent alors à nous :

- Si $r \leq 0$ (toutes les composantes sont négatives), on peut augmenter x_q autant que l'on veut, on aura toujours la positivité de la variable x_B. Le critère n'est pas majoré, on obtient max $f(x) = +\infty$: arrêt de l'algorithme.
- Sinon, il existe au moins une composante de r telle que $r_i > 0$. Pour avoir la positivité $(\underline{x}_B)_i - r_i x_q \geq 0$ pour tout i, on choisit la variable sortante x_p pour laquelle le rapport $\frac{(\underline{x}_B)_i}{r_i}$ pour $i = 1, \ldots, m$ (avec $r_i > 0$) est le plus petit possible (correspond à trouver la première variable qui s'annulera).

On remarque l'apparition de A_B^{-1}. Afin de pouvoir résoudre le simplexe par tableaux, nous allons appliquer le **pivot de Gauss** sur la matrice A_B. Ainsi, A_B^{-1} sera une matrice identité. De ce fait, r_i est la i^{me} composante de a_q, et $(\underline{x}_B)_i$ devient simplement b_i. Le rapport $\frac{(\underline{x}_B)_i}{r_i}$ devient finalement $\frac{b_i}{a_{iq}}$. L'indice du minimum sera l'indice p.

Critères d'arrêt

Soit c le vecteur de coûts qui évolue tout au long de l'application du simplexe, constitué initialement des cœfficients de la fonction-objectif. L'arrêt de l'algorithme dépend de ce vecteur et de la solution réalisable. Plus spécifiquement, cela dépend si la solution de base réalisable est dégénérée ou non.

Définition 10 (Sommet dégénéré)

On dit qu'un sommet x est ***dégénéré*** si l'une de ses composantes est nulle.

1. Si tous les coûts sont négatifs (meilleur cas de figure), alors la solution de base réalisable courante est l'unique optimum.

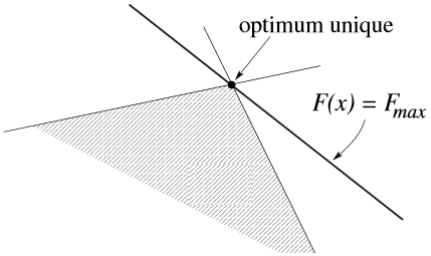


FIG. 3.6 : Optimum unique.

2. Si les coûts sont négatifs ou nuls, deux cas sont à prendre en compte :

   (a) Si $c_e = 0$ et $x_e > 0$, alors l'optimum n'est pas unique :

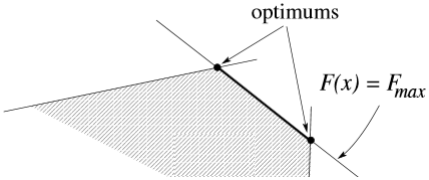


FIG. 3.7 : Optimum non-unique.

   (b) Si $c_e = 0$ et $x_e = 0$, alors l'optimum est unique (a priori).

   Dans ce cas, la base est dite dégénérée : il existe une variable de base nulle.

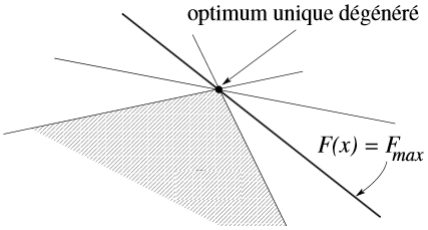


FIG. 3.8 : Optimum à base dégénérée.

3. Si $c_e > 0$ et x_e est non-bornée, alors la fonction-objectif n'est pas majorée.

Théorème 5

Si au cours de l'algorithme du simplexe, aucune base rencontrée n'est dégénérée, alors l'algorithme se termine en un nombre fini d'itérations.

Choix et problèmes calculatoires

Au cours de l'application du simplexe, certains points peuvent être surprenants, suscitant quelques interrogations de votre côté. Mais rassurez-vous, il y a une solution à chaque problème ! Énumérons les soucis ou interrogations que vous pourriez avoir.

1. Si le pivot p ou q n'est pas unique, comment faire notre choix ?

   Lorsque l'on permute des colonnes, une variable rentre dans la base et une autre en sort. Dantzig a défini plusieurs critères pour nous aider à répondre à la question.

Remarque 10 (Premier critère de Dantzig : choix du pivot q)

- En principe, n'importe quelle composante de x_q (avec $c_q > 0$) peut être choisie comme variable entrante dans la base.
- S'il n'existe pas de pivot q tel que $c_q > 0$, la solution optimale est trouvée et l'algorithme s'arrête.
- Dans le cas contraire, la stratégie la plus courante est de choisir le pivot q correspondant à **la composante la plus grande du vecteur de coûts**. Ce choix garantit la plus grande croissance de la fonction-objectif.

Comme dit précédemment, à partir du pivot q nous pouvons calculer le meilleur choix pour le pivot p.

Remarque 11 (Second critère de Dantzig : choix du pivot p)

Dans la pratique, la stratégie pour choisir le pivot p (et donc la variable sortant de la base) est de prendre le minimum des rapports $\frac{b_i}{a_{iq}}$ pour $i = 1, \ldots, m$, avec q la colonne de la variable entrante.

2. Est-ce-que l'algorithme se termine toujours ?

Selon les pivots choisis, il reste possible de voir l'apparition de cycles, notamment dans le cas d'une solution de base réalisable dégénérée. Ce que l'on appelle **cycle** est la réapparition des ensembles d'indices B et N initiaux pendant une étape du simplexe. Heureusement, ces cycles peuvent être évités grâce à la **règle de Bland**.

Définition 11 (Règle de Bland)

Lors d'un pivotage, la variable qui entre est celle d'indice minimal parmi celles qui peuvent rentrer et la variable qui sort est celle d'indice minimal parmi celles qui peuvent sortir :

(a) Choix de q : $q = \min\{j \in N | c_j > 0\}$.

(b) Choix de p : $p = \min\left\{k \in B | \frac{b_k}{a_{kq}} = \min\{\frac{b_i}{a_{iq}} | a_{ik>0, i \in B}\}\right\}$.

3. Bien que le problème soit borné, je n'obtiens pas de solution. Pourquoi ?

Deux cas de figure sont possibles :

(a) Soit les pivots ont été mal choisis.

(b) Soit une erreur s'est glissée dans le pivot de Gauss.

4. Comment démarrer le simplexe ? Faut-il calculer au hasard une solution de base réalisable ?

Non, dans la pratique, vous considérerez que la base ne contient que les variables d'écart. Ainsi, A_B sera la matrice identité, et le second membre sera votre sommet initial. Ce sommet est réalisable, car les colonnes de A_B sont effectivemt linéairement indépendantes.

Application du simplexe par tableaux

Résolvons le problème suivant :

$$\begin{array}{ll} \max & f(x_1, x_2) = 6x_1 + 4x_2 \\ \text{Soumis à :} & 3x_1 + 9x_2 \leq 81 \\ & 4x_1 + 5x_2 \leq 55 \\ & 2x_1 + x_2 \leq 20 \\ & x_1, x_2 \geq 0 \end{array}$$

Le problème est déjà écrit sous forme d'un (LPC). Sous sa forme standard, nous introduisons les variables d'écart afin d'obtenir des contraintes d'égalité :

$$\begin{array}{ll} \max & f(x_1, x_2) = 6x_1 + 4x_2 \\ \text{Soumis à :} & 3x_1 + 9x_2 + e_1 = 81 \\ & 4x_1 + 5x_2 + e_2 = 55 \\ & 2x_1 + x_2 + e_3 = 20 \\ & x_1, x_2, e_1, e_2, e_3 \geq 0 \end{array}$$

L'algorithme du simplexe se déroule ensuite ainsi.

Itération 1

Variables de la base			Variables hors-base		
e_1	e_2	e_3	x_1	x_2	b
1	0	0	3	9	81
0	1	0	4	5	55
0	0	1	2	1	20
0	0	0	6	4	$f = 0$

TAB. 3.1 : Variable entrante : $x_e^{(1)} = \max(6, 4) = x_1$. Variable sortante : $\min\left(\frac{81}{3}, \frac{55}{4}, \frac{20}{2}\right) \Rightarrow x_s^{(1)} = e_3$.

Itération 2

Après l'étape 1, on remarque que les colonnes de x_1 et e_3 ont été interverties.

Pour obtenir ce résultat, un pivot de Gauss a été appliqué en prenant comme pivot l'élément à l'intersection de la colonne de x_1 et de la ligne e_3. Tous les autres éléments de la colonne x_1 sont alors éliminés.

Rappelons la méthode du pivot de Gauss. Soit a_{se} le pivot (obligatoirement non nul), cœfficient de la ligne s colonne e :

Variables de la base			Variables hors-base		
e_1	e_2	x_1	e_3	x_2	b
1	0	0	-3/2	15/2	51
0	1	0	-2	3	15
0	0	1	1/2	1/2	10
0	0	0	-3	1	$f = 60$

TAB. 3.2 : Variable entrante : $x_e^{(2)} = x_2$. Variable sortante : $\min\left(\frac{51}{15/2}, \frac{15}{3}, \frac{10}{1/2}\right) \Rightarrow x_s^{(2)} = e_2$.

1. Les éléments de la ligne s sont divisés par le pivot :

$$a'_{sj} = \frac{a_{sj}}{a_{se}}, \quad \forall j \text{ et } b'_s = \frac{b_s}{a_{se}}.$$

2. Les éléments des autres lignes ($i \neq s$) sont des combinaisons linéaires de lignes pour annuler le cœfficient dans la même colonne que le pivot :

$$a'_{ij} = a_{ij} - a_{ie}a'_{sj} \text{ et } b'_i = b_i - a_{ie}b'_s$$

Ainsi, la solution à chaque itération peut se lire dans la colonne de b.

Itération 3

Variables de la base			Variables hors-base		
e_1	x_2	x_1	e_3	e_2	b
1	0	0	7/2	-5/2	27/2
0	1	0	-2/3	1/3	5
0	0	1	5/6	-1/6	15/2
0	0	0	-11/3	-1/3	$f = 65$

TAB. 3.3 : Tous les coûts réduits sont négatifs. L'optimum est atteint avec $f = 65$.

Au final, la solution rélisable optimale est obtenu en résolvant un système similaire à $A_B x_B = b$. En appliquant le pivot de Gauss, nous avons transformé le système de sorte à isoler x_B en faisant apparaître eune matrice identité. La solution optimale s'obient en lisant tout simplement la colonne correspondant au vecteur b :

$$\begin{cases} e_1^* & = 27/2 \\ x_1^* & = 15/2 \\ x_2^* & = 5 \\ e_2^* & = 0 \\ e_3^* & = 0 \end{cases}$$

L'optimum de la fonction-objectif est : $f(x_1^*, x_2^*) = 6x_1^* + 4x_2^* = 6 * 15/2 + 4 * 5 = 65$.

Chapitre 4

Optimisation discrète

En optimisation discrète, certaines variables du modèle appartiennent à un ensemble discret. Dans ce chapitre, deux domaines de l'optimisation discrète sont présentées : la programmation en nombres entiers, et l'optimisation combinatoire.

4.1 Programmation en nombres entiers

4.1.1 Principe général

Dans un programme en nombres entiers - aussi appelé (PLNE) - les fonctions-objectif et les contraintes sont toujours linéaires, mais une partie ou toutes les variables sont des entiers. La programmation en nombres entiers a l'avantage d'être plus réaliste que les problèmes de programmation linéaire sans la contrainte d'intégrité, mais ont le désavantage d'être plus difficile à résoudre. La méthode usuellement employée consiste à résoudre une série de (LP) associés (le (PLNE) où la contrainte d'intégrité est relaxée) issue de la recherche d'une solution entière.

Dans un programme en nombres entiers linéaire, nous cherchons à optimiser une fonction linéaire soumise à un ensemble de contraintes linéaires et d'intégrité sur un espace n-dimensionnel :

$$\begin{array}{ll} \max & c^T x \\ \text{Soumis à :} & Ax = b \\ & x \geq 0 \\ & x \in \mathbb{Z}^n \end{array}$$

Si seulement certaines composantes de x appartiennent à $\mathbb{Z}$, alors le problème est un problème linéaire en nombres entiers mixte. Si toutes les variables prennent les valeurs 0 ou 1, alors il s'agit d'un problème d'optimisation binaire.

Les problèmes de gestion de stock comme celui du *Sac à dos* rentrent clairement dans cette catégorie. Par contre, il est important de remarquer qu'il ne s'agit plus d'un simple problème linéaire, puisque l'ensemble de solutions admissibles n'est plus représenté par un polyèdre mais par un ensemble discret de points. Il faut ajouter la contrainte d'intégrité. Dans ce cas, comment déterminer l'optimum ? Quatre familles de méthodes répondent à la question :

1. Les méthodes de recherches arborescentes : *Branch and Bound*, algorithme de Little pour le problème du voyageur de commerce, de Dakin pour les (PLNE).

2. Les *Méthodes de Coupe* : aussi appelées méthodes de troncature, comme la troncature de Gomory.

3. La programmation dynamique : pour les problèmes de plus court chemin ou de sac à dos.

4. Les méthodes approchées : algorithme tabou, recuit simulé, algorithme génétique, colonie de fourmis.

4.1.2 Résolution par le *Branch and Bound*

Nous n'avons évidemment pas le temps de tout présenter. Nous avons opté par la suite par une explication de la méthode du *Branch and Bound*.

Principe général

Comme son nom l'indique, la procédure se découpe en deux étapes : la séparation et l'évaluation. C'est le principe du *Diviser pour régner*. Les bornes sur le coût optimal vont être utilisées pour éviter d'explorer certaines parties de l'ensemble des solutions admissibles.

Les deux étapes sont :

- **La séparation** - L'ensemble des solutions admissibles d'un problème F est partitionné en une collection finie de sous-ensembles $\{F_i\}$. Chaque problème est ensuite résolu séparément.

- **L'évaluation** - Pour chaque sous-problème, une borne inférieure est calculée sur le coût optimal du (LP) associé. En effet, la relaxation linéaire est employée. Si la solution d'un sous-problème est entière, le partitionnement est inutile. Sinon, deux sous-problèmes sont créés en ajoutant comme contraintes les bornes supérieure et inférieure de la solution non-entière.

Exemple

$$
\begin{array}{ll}
\max & f(x_1, x_2) = 10x_1 + 50x_2 \\
\text{Soumis à :} & -x_1 + 2x_2 \leq 5 \\
& x_1 + 2x_2 \leq 14 \\
& x_1 \leq 8 \\
& x_1, x_2 \geq 0 \\
& x_1, x_2 \in \mathbb{Z}
\end{array}
$$

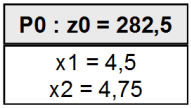


FIG. 4.1 : Étape 1. Résolution du PL. La solution optimale est calculée.

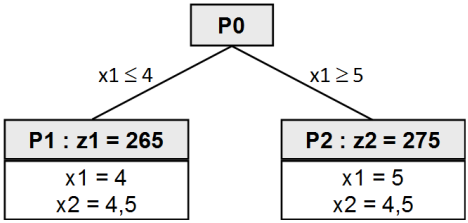


FIG. 4.2 : Étape 2. Séparation en deux sous-problèmes.

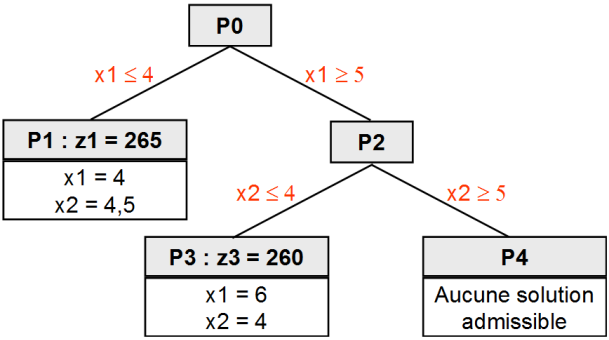


FIG. 4.3 : Étape 3. Seconde séparation en sous-problèmes. Aucune solution admissible pour P_4.

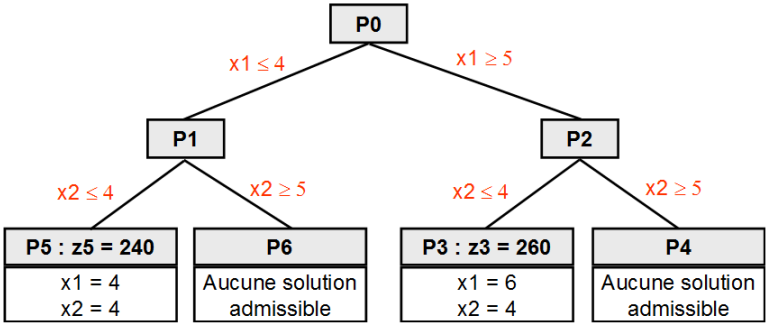


FIG. 4.4 : Étape 4. Solution optimale trouvée en P_3.

Bibliographie

[1] O. Bonaventure et al. *APP0*. École Polytechnique de Louvain, sept. 2008.

[2] Z. Galil. "Efficient Algorithms for Finding Maximum Matching in Graphs". In : *ACM Comput. Surv.* 18.1 (mar. 1986), p. 23-38. ISSN : 0360-0300. DOI : 10.1145/6462.6502. URL : http://doi.acm.org/10.1145/6462.6502.

[3] Ministère de l'Enseignement Supérieur et de la Recherche. *DUT Informatique. Programme Pédagogique National*. http://www.enseignementsup-recherche.gouv.fr/cid53575/programmes-pedagogiques-nationaux-d.u.t.html. 2013.

[4] The G12 Team. *MiniZinc*. 2007-2014. URL : http://minizinc.org/.

Annexe A

Consignes de travail

Texte repris du livret APP0 de l'École Polytechnique de Louvain [1].

A.1 Travailler en groupe

La formation se fait sous la forme d'APP (apprentissage par problèmes). Une des caractéristiques de cette méthode est d'optimiser la participation active de chaque étudiant. Individuellement, chacun d'entre vous contribue selon son style et ses ressources à la progression efficace de la rencontre et au climat constructif des échanges. De plus, pour faciliter le déroulement d'un tutorial, il est conseillé aux étudiants de remplir 3 rôles spécifiques :

Animateur

- S'assure que le groupe suit les étapes prévues,
- Veille à ce que le contenu de la discussion soit noté par le secrétaire,
- Anime la discussion :
  - distribue la parole, suscite/sollicite la participation ou modère les interventions,
  - amène le groupe à clarifier les idées développées,
  - réalise des synthèses au besoin ;
- S'assure du respect du timing : informe le groupe régulièrement (« il nous reste 30 minutes pour cette séance »…)

Scribe

- Note au tableau l'essentiel des échanges (support et mémoire de la discussion du groupe),
- Ne filtre pas les informations notées,
- Organise le tableau en fonction des étapes (de manière à garder la trace de toute la réflexion → ne pas effacer).

Secrétaire

- Garde une trace écrite et complète de la production du groupe,
- Transmet cette trace à tous les membres du groupe et au tuteur.

Lors des séances, l'enseignant fait office de *tuteur* :

- Il ne fait pas partie du groupe d'apprenants,
- Il guide le groupe :
  - l'empêche de s'égarer !
  - l'incite à aller plus loin…
- Il n'est pas là pour vous donner un cours (si c'était le cas, vous seriez tous regroupés en auditoire),
- Il connait la réponse au problème mais c'est à vous, étudiants, de faire le travail. Vous ne serez donc pas étonné qu'il refuse parfois de répondre directement aux questions que vous vous posez. Ce sera le cas notamment s'il estime que cette question n'a pas été débattue préalablement au sein du groupe.

A.2 Travail individuel

Pourquoi faire du travail individuel ?

- Le vrai but est que tout le monde apprenne, pas uniquement que le problème soit bien résolu !
- Ce n'est pas le groupe qui doit devenir compétent mais bien chacun de ses membres !
- Le travail collectif est certes important mais l'APP vise à rendre chaque étudiant compétent.
- C'est la raison pour laquelle chaque APP fait l'objet d'une évaluation individuelle. Le travail réalisé entre les séances de groupe est la manière la plus efficace et la plus simple de se préparer à cette évaluation. Avant la fin de chaque problème, chaque étudiant sera amené à présenter sa solution individuelle aux autres membres du groupe.

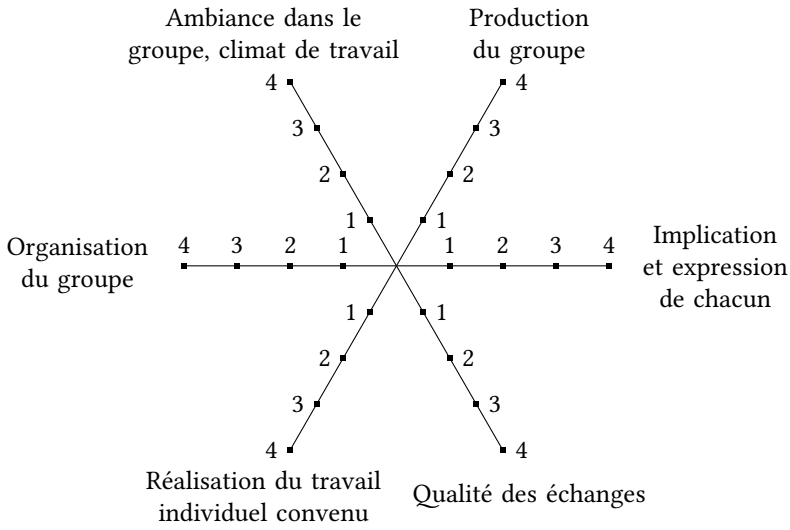


FIG. A.1 : Étoile d'évaluation

A.3 Évaluation du travail en groupe (questionnaire)

A.3.1 Les axes (quelques critères d'évaluation)

Indiquez sur chacun des 6 axes figurant sur l'étoile (figure A.1) votre niveau d'appréciation générale entre 0 (« très insatisfaisant ») et 4 (« très satisfaisant »). Ensuite, reliez les points.

Production du groupe. Le groupe a produit quelque chose de satisfaisant et cette production est réellement le résultat d'un effort *collectif*.

Implication et expression de chacun. Chacun des participants a contribué de manière significative à l'efficacité du groupe, le groupe a donné l'occasion à chacun de ses membres d'exprimer son point de vue, les participants en retrait ont été sollicités.

Qualité des échanges. Il y eu suffisamment d'interactions entres les différents membres du groupe, ces échanges ont permis de faire émerger des points de vue différents pour traiter le problème, les temps de mises en commun ont permis à chacun de confronter sa compréhension du problème et des notions travaillées…

Réalisation du travail individuel convenu. Les membres du groupe ont fait leur part de travail individuel entre les séances, tous les membres du groupe ont mené à bien leurs responsabilités…

Organisation du travail. Le groupe est parvenu à coordonner ses activités, les réunions étaient efficaces, le groupe est resté centré sur la tâche à accomplir, le groupe a fait suffisamment usage du tableau, le groupe s'est réparti des rôles : un secrétaire a gardé des traces des échanges, un animateur a joué son rôle, le timing a été respecté…

Ambiance dans le groupe, climat de travail. Bonne entente entre les membres du groupe, les participants s'aident et s'encouragent mutuellement, le groupe est arrivé à surmonter ses divergences de vue, personne n'est arrivé à imposer son point de vue…

A.3.2 Questions ouvertes

1. Déterminez deux points qui ont bien fonctionné pour le travail en groupe

2. Déterminez deux points qui ont mal fonctionné pour le travail en groupe

3. Quelles sont les leçons à tirer de cette expérience ? Si c'était à refaire, que feriez-vous – quel engagement prendriez-vous – pour que cela fonctionne mieux ? Pensez aux travaux de groupe qui se présenteront prochainement durant votre formation.

4. Êtes-vous satisfait des connaissances ou des compétences acquises lors de la résolution de ce problème ? Commentaires à propos de ce que vous avez appris en informatique.

5. Autres commentaires et suggestions à propos de ce problème.

Annexe B

Problème 1 : Optimisation de production de gâteaux

Votre équipe va confectionner des gâteaux pour la fête annuelle de l'école. On dispose de trois recettes de gâteaux qui seront réalisés, et d'un stock d'ingrédients. Combien de gâteaux de chaque doit-on préparer pour réaliser le plus de gâteaux au total ?

On dispose en tout de 3 kg de farine, 2 kg de beurre, 3 kg de sucre, 60 œufs, 1 kg de chocolat, 15 citrons et 30 sachets de levure.

Voici les trois recettes :

Moelleux au chocolat	**Moelleux au citron**	**Tarte au citron**
Préparation : 10 min Cuisson : 35 min	Préparation : 15 min Cuisson : 25 min	Préparation : 30 min Cuisson : 25 min
125 g de farine 125 g de beurre 250 g de sucre 4 œufs 200 g de chocolat ½ sachet de levure	180 g de farine 120 g de beurre 200 g de sucre 6 œufs 1 citron ½ 1 sachet de levure	200 g de farine 90 g de beurre 250 g de sucre 4 œufs 3 citrons

B.1 Travail demandé

B.1.1 Préparation

Il ne vous est pas demandé de résoudre le problème, mais de le *modéliser*. Pour cela, vous devez tout d'abord déterminer les *variables* du problème (que vous pouvez nommer comme vous le souhaitez), et le *domaine* de chacune d'elles. Un domaine peut être l'ensemble des entiers naturels $\mathbb{N}$, des entiers $\mathbb{Z}$, des réels $\mathbb{R}$, un intervalle (e.g., $[0..10]$) d'entiers ou de réels, un intervalle semi-ouvert (e.g., $[1..\infty]$) d'entiers ou de réels, ou encore un ensemble de valeurs (e.g., $\{10, 15, 20\}$).

Ensuite, déterminez s'il s'agit d'un problème de *décision* ou d'*optimisation*. S'il s'agit d'un problème d'optimisation, il faut déterminer *une* expression numérique à maximiser ou à minimiser (e.g., maximiser $(x + y)$, minimiser $(|x - y|)$, etc.)

Enfin, déterminez les *contraintes* de problème. Ici, une contrainte peut être n'importe quelle expression booléenne mettant en jeu une ou plusieurs variables (e.g., $x + y \leq 10$, $x \neq y$, $x \times y = 2$, $|x - y| + z^2 > 20$, etc.)

S'il vous reste du temps, regardez les variantes dans la section suivante.

B.1.2 Réalisation

Réécrivez votre modèle à l'aide du langage *MiniZinc* et utilisez un des solveurs fournis pour le résoudre.

B.2 Avec les courses

On reprend le problème précédent, les questions sont les mêmes mais on ne dispose pas du stock d'ingrédients. On dispose de 150 € pour faire les courses. On ne peut acheter que des portions entières d'un ingrédient (par exemple la farine s'achète par kilogrammes) ! Voici le prix des ingrédients :

Farine : 0,44 €/kg

Beurre : 2,44 € pour une plaquette de 500 g

Sucre : 0,89 €/kg

Œufs : 1,85 € les 12 ou 0,99 € les 6

Chocolat : 1,29 € la plaquette de 200 g

Citrons : 1,49 € les 4

Levure : 0,20 € les 6 sachets

On peut imaginer d'autres extensions au problème :

- prendre en compte un stock existant (e.g., il reste 3 kg de farine de la dernière vente),

- on va vendre le moelleux au chocolat à 3,70 € la part, le moelleux au citron à 3,10 € la part et la tarte à 3,20 € la part (les gâteaux font tous 6 parts), on veut maintenant maximiser le profit,
- prendre en compte le temps de préparation et de cuisson des gâteaux : on ne dispose que de 6 h à trois personnes, dont une seule qui sait faire la tarte au citron, et deux fours (il faudra peut-être ajuster les prix de vente en conséquence),
- on veut imposer une variété dans les préparations (maximum 50 % d'écart entre deux types de gâteaux),
- etc.

Testez chacune de ces variantes. Pouvez-vous réaliser un modèle commun, en permettant de paramétrer à part le stock, le prix des ingrédients ? Les recettes ? Essayez d'augmenter fortement le budget : quel est l'impact sur le temps de calcul ? Essayez d'ajouter ou supprimer une recette : quel est l'impact ?

Annexe C

Problème 2 : Création d'emplois du temps

Votre objectif est de construire un emploi du temps pour une semaine d'enseignement dans un collège miniature, disposant d'une classe par niveau (de la 6e à la 3e).

Chaque classe suit des cours, par séances d'1 h 30. Les séances ont lieu de 9 h à 10 h 30 et de 11 h à 12 h 30 du lundi au samedi inclus, ainsi qu'une séance de 14 h à 15 h 30 les lundi, mardi, jeudi et vendredi, soit 16 séances par semaine au maximum pour chaque classe. Voici la répartition :

Pour les 6e et les 5e :

- Français : 4 séances par semaine
- Anglais : 3 séances par semaine
- Mathématiques : 3 séances par semaine
- Arts : 2 séances par semaine
- Sport : 2 séances par semaine

Pour les 4e et les 3e :

- Français : 4 séances par semaine
- Anglais : 3 séances par semaine
- Mathématiques : 3 séances par semaine

- Sciences : 2 séances par semaine
- Arts : 2 séance par semaine
- Sport : 2 séances par semaine

Chaque matière n'est enseignée que par un seul enseignant. Le collège dispose de cinq salles.

Proposez un modèle pour réaliser l'emploi du temps : organisez les séances de cours en associant une classe avec un enseignant et une salle à un horaire donné.

Extensions

Essayez d'ajouter les paramètres suivants au modèle (dans le désordre) :

- On ne propose pas deux cours identiques la même journée,
- Les cours de sciences ne peuvent avoir lieu que dans une salle spécifiquement équipée,
- Le sport se fait uniquement sur le terrain de football du collège (que l'on considère comme une salle spécifique, où aucun autre cours ne peut avoir lieu),
- Les enseignants d'anglais et d'arts sont à temps partiel et n'enseignent pas le mercredi,
- Comme les 6^{e} et les 5^{e} n'ont que 14 séances de cours, on voudrait libérer leur samedi matin,
- Pouvez-vous également éliminer les « trous » dans les emplois du temps des enseigants ? Limiter le nombre de jours de présence de chaque enseignant ?
- Peut-on développer le collège en doublant le nombre de classes ? Faut-il ajouter des enseignants, des salles ?

Annexe D

Problème 3 : Un étudiant de l'IUT de Maubeuge au supermarché. Problèmes nutritionnels en perspective ?

D.1 Sujet

Tout le monde le sait, l'arrivée à l'âge adulte conduit inexorablement à aller faire ses courses sans maman. Cependant, votre santé n'en a que faire de votre nouvelle liberté, requérant un certain apport nutritionnel à chaque repas. L'objectif du *problème de rationnement* consiste à sélectionner un ensemble d'aliments satisfaisant les Apports Journaliers Recommandés à un coût minimum. Les contraintes du problème ne prennent évidemment pas en compte l'intégralité des nutriments existants, mais le principe reste identique. Il vous faudra composer un menu à moindre coût, dans lequel le nombre de calories et l'ensemble des nutriments seront présents en quantité satisfaisante. Vous vous servirez des données fournies dans l'annexe.

D.2 Préparation

Votre travail de préparation consiste à formuler mathématiquement le problème :

1. identifiez les variables du problème,
2. définissez la fonction-objectif,
3. formulez les contraintes à satisfaire,

4. mettez la formulation sous forme matricielle.

Descriptif des contraintes : *pour bien fonctionner, un corps humain doit disposer d'assez d'énergie, mais a contrario ne peut assimiler une ration de calories supérieure à un seuil sous peine d'augmenter sa masse adipeuse. Le principe est identique pour les nutriments : au-delà d'une certaines dose, même les vitamines peuvent se révéler toxiques.*

1. Tout d'abord, vous assimilerez la méthode du simplexe en l'appliquant sur ce problème en considérant :

   - Parmi les aliments : chips, pizza, céréales ;
   - Parmi les contraintes : calories, vitamine A.

   Quelle solution obtenez-vous (rations et coût) ?

2. Vous vous rendez compte en faisant vos courses que vous ne pourrez pas forcément tout stocker. Modifiez le modèle en tenant compte du fait que vous ne pouvez stocker plus de 5 exemplaires de chaque aliment.

3. Cette fois-ci, toutes les contraintes et les aliments à votre disposition sont à considérer. Adaptez votre formulation pour en tenir compte.

4. Jusqu'à présent, vous avez optimisé le coût, ce qui peut sembler raisonnable du point de vue du portefeuille, mais mauvais pour votre santé. Proposez une fonction permettant d'améliorer votre menu en diversifiant au maximum les produits.

D.3 Réalisation individuelle

Reprenez les formulations mathématiques des questions précédentes, et implémentez-les avec Minizinc, puis Scilab. Quels régimes obtenez-vous ? Pour quels coûts ?

D.4 Annexe

Dans ses toolboxes, Scilab fournit différentes méthodes d'optimisation. Celles qui nous intéressent ici concerne les problèmes d'optimisation linéaire et quadratique. Les fonction *linpro* et *quapro* nous aideront dans la résolution. Pour connaître le prototype de la fonction linpro par exemple, écrivez dans la console `help('linpro')`.
Si le module n'est pas installé, lancez dans la console Scilab : `atomsInstall('linpro')`.

	Unité	Minimum	Maximum
Calories	cal	2 000	2 500
Cholesterol	mg	0	300
Matière grasse	g	40	90
Protéine	g	50	100
Vitamine A	UI	2 000	50 000
Vitamine C	UI	50	20 000
Calcium	mg	800	1 600
Fer	mg	10	30

Tab. D.1 : Apports nutritionnels

	Coût par ration
Brocolis	0,16
Carottes rapées	0,07
Salade	0,02
Pommes de terre	0,06
Poulet	0,84
Bananes	0,15
Raisins	0,32
Oranges	0,15
Pain	0,06
Beurre	0,05
Camembert	0,25
Bœuf	0,27
Jambon	0,33
Céréales	0,28
Pizza	0,44
Couscous	0,39
Riz blanc	0,08
Côtelettes de porc	0,81
Sardines à l'huile	0,45
Chips	0,19
Yaourts	0,20
Haricots verts	0,75

Tab. D.2 : Coût et capacité de stockage des aliments

	Calories	Cholesterol	Matière grasse	Protéine	Vitamine A	Vitamine C	Calcium	Fer
Brocolis	73,8	0,0	0,8	8,0	5 867,4	160,2	159,0	2,3
Carottes râpées	23,7	0,0	0,1	0,6	15 471,0	5,1	14,9	0,3
Salade	2,6	0,0	0,0	0,2	66,0	0,8	3,8	0,1
Pommes de terre	171,5	0,0	0,2	3,7	0,0	15,6	22,7	4,3
Poulet	277,4	129,9	10,8	42,2	77,4	0,0	21,9	1,8
Bananes	104,9	0,0	0,5	1,2	92,3	10,4	6,8	0,4
Raisins	15,1	0,0	0,1	0,2	24,0	1,0	3,4	0,1
Oranges	61,6	0,0	0,2	1,2	268,6	69,7	52,4	0,1
Pain	65,0	0,0	1,0	2,3	0,0	0,0	26,2	0,8
Beurre	725,0	250,0	83,0	0,7	2 499,0	0,0	15,0	0,1
Camembert	112,7	29,4	9,3	7,0	296,5	0,0	202,0	0,2
Boeuf	141,8	27,4	12,8	5,4	0,0	10,8	9,0	0,6
Jambon	37,1	13,3	1,4	5,5	0,0	7,4	2,0	0,2
Céréales	110,5	0,0	0,1	2,3	1 252,2	15,1	0,9	1,8
Pizza	181,0	14,2	7,0	10,1	281,9	1,6	64,6	0,9
Couscous	100,8	0,0	0,1	3,4	0,0	0,0	7,2	0,3
Riz blanc	102,7	0,0	0,0	0,3	2,1	0,0	0,0	7,9
Côtelettes	710,8	105,1	72,2	13,8	14,7	0,0	59,9	0,4
Sardines	49,9	34,1	2,7	5,9	53,8	0,0	91,7	0,7
Chips	139,2	0,0	9,2	2,2	61,5	9,6	14,2	0,5
Yaourts	70,0	5,0	3,5	4,1	4,0	0,0	151,0	0,1
Haricots verts	31,0	0,0	0,1	1,8	108,0	24,3	307,0	1,0

Tab. D.3 : Information nutritionnelle par aliment

Annexe E

Problème 4 : Location de sites

E.1 Sujet

Considérons un autre problème bien connu de l'optimisation combinatoire : l'étude *d'emplacement de sites*. Une entreprise d'informatique possède 4 usines qui produisent des cartes-mères. La compagnie fonctionnant bien, elle aimerait aller plus loin en concevant dorénavant son propre ordinateur. Le tableau E.1 ci-dessous contient : les coûts fixes, variables, et les capacités hebdomadaires pour chaque site, ainsi que les productions hebdomadaires de chaque usine. Les coûts variables sont en euros par semaine, et incluent les frais de transport. Les coûts fixes sont en euros par an. Les quantités de production et de capacité sont en tonnes par semaine.

Vous devez faire face à deux problèmes distincts :

- combien de tonnes de matériel chaque usine doit envoyer à chaque site par semaine ?
- chaque site a-t-il son utilité ?

L'objectif est clair : minimiser les coûts. La première question correspond aux problèmes de programmation linéaire à variables continues que vous avez déjà rencontré. La seconde fait intervenir des variables entières, d'où l'appelation de programmation linéaire en nombres entiers.

E.2 Préparation

Votre travail de préparation consiste à formuler mathématiquement le problème :

1. identifiez les variables du problème,
2. définissez la fonction-objectif,

	Site 1	Site 2	Site 3	Production
Coûts variables – Usine 1	25	20	15	1 000
Coûts variables – Usine 2	15	25	20	1 000
Coûts variables – Usine 3	20	15	25	500
Coûts variables – Usine 4	25	15	15	500
Coûts fixes	500 000	500 000	500 000	
Capacité	1 500	1 500	1 500	

TAB. E.1 : Données sur les capacités et les coûts des sites

3. formulez les contraintes à satisfaire,
4. mettez la formulation sous forme matricielle.

Descriptif des contraintes : *chaque usine est sujette à une certaine production. Cette production devra bien aller quelque part. En outre, il faut prendre en compte la possibilité qu'un site soit ou non ouvert.*

1. Tout d'abord, appliquez la méthode du simplexe pour résoudre ce problème. Vous relaxerez la contrainte d'intégritté, obtenant de ce fait un problème linéaire (PL) et répondrez aux questions ci-dessous. On considère que la production arrivant sur un site ne provient que d'une seule usine.
   (a) Quel est la valeur de la fonction-objectif et quels sites doivent être construits ?
   (b) Cette solution est-elle directement utilisable comme solution optimale pour le PLNE ?
2. L'idée consiste à appliquer un *Branch & Bound* pour trouver la solution entière optimale. Tout d'abord, entraînez-vous en cherchant la solution optimale entière sur le problème indépendant suivant :

$$\begin{array}{ll} \textit{max} & f(x_1, x_2) = 5 \cdot x_1 + 4 \cdot x_2 \\ \text{Soumis à} & x_1 + x_2 \leq 5 \\ & 10 \cdot x_1 + 6 \cdot x_2 \leq 45 \\ & x_1, x_2 \geq 0 \end{array}$$

   Vous pourrez vous appuyez sur un graphique pour trouver la solution du programme linéaire et éliminer les branches n'apportant pas de solution.
3. Considérons un PLNE et son PL obtenu en relâchant la contrainte d'intégrité. Répondez par VRAI ou FAUX aux questions suivantes en justifiant par du texte ou un graphique :
   (a) Si le PLNE est un problème de minimisation, la valeur de la fonction-objectif est supérieure ou égale à la valeur de la fonction-objectif du PL.

(b) Si le PL est insatisfaisable, le PLNE aussi.

(c) Si toutes les variables du PL sont des entiers, il s'agit de la solution optimale du PLNE.

4. Dans cette question, vous allez chercher la solution du PLNE du problème initial en vous appuyant sur la programmation par contraintes. Vous minimiserez tout d'abord le nombre de sites ouverts, puis les coûts engendrés (Généralisez le programme pour pouvoir modifier aisément le nombre de sites, d'usines et les coûts).

   (a) Comment évolue le coût ?

   (b) Quels sites doivent être construits dans ce cas ?

E.3 Réalisation individuelle

- L'ensemble des formulations prédédemment établies sont à implémenter avec Scilab. Donnez les valeurs des coûts et des variables dans chaque cas.
- L'algorithme du Branch & Bound n'existe pas par défaut dans Scilab. Utilisez la programmation par contraintes avec Minizinc pour répondre à la question du PLNE. Vérifiez vos résultats obtenus par le calcul.

E.4 Annexe

Dans ses toolboxes, Scilab fournit différentes méthodes d'optimisation. Celle qui nous intéresse ici concerne les problèmes d'optimisation linéaire. La fonction linpro nous aidera dans la résolution. Pour connaître le prototype de la fonction, écrivez dans la console `help('linpro')`.