Algorithmes de Résolution de Sudoku

Yves-Marie Le Gueut, Ronan Kerviche

Mai 2009

1					7		9	
	3			2				8
		9	6			5		
		5	3			9		
	1			8				2
6					4			
3							1	
	4							7
		7				3		

Table des matières

1 Qu'est ce qu'un Sudoku

Le Sudoku est un casse-tête d'origine américaine qui a été popularisé au Japon. Il consiste en une grille carrée de 81 chiffres, 9 colonnes et 9 lignes donc, séparée en 9 régions elle-même carrée et contenant 9 cases. Cette grille est initialement pré-remplie avec une vingtaine de chiffre compris entre 1 et 9 (ce nombre varie généralement avec la difficulté de la grille, qui sera abordée plus loin). Le joueur doit alors remplir les cases laissées vides en utilisant d'une part, uniquement les chiffres de 1 à 9 et d'autre part, en suivant ces quelques règles :

- Chaque case ne doit contenir qu'un seul chiffre parmi 1,...,9
- Chaque ligne ne doit contenir qu'une seul fois un chiffre donné parmi 1,...,9
- Chaque colonne ne doit contenir qu'une seul fois un chiffre donné parmi 1,...,9
- Chaque 'bloc' (ou sous-région) ne doit contenir qu'une seul fois un chiffre donné parmi 1,...,9

			3				5	
		5	4		6			2
2	7			1		3	6	
7		4	2	3				
5	1						3	7
				4	7	9		1
	4	6		9			1	5
1			6		8	7		
	5				4			

Fig. 1 – Un Sudoku initialement préparé

En appliquant les règles précédantes et en usant d'un peu de logique on obtient :

4	6	1	**3**	7	2	8	**5**	9
9	3	**5**	**4**	8	**6**	1	7	**2**
2	**7**	8	9	**1**	5	**3**	**6**	4
7	9	**4**	**2**	**3**	1	5	8	6
5	**1**	2	8	6	9	4	**3**	**7**
6	8	3	5	**4**	**7**	**9**	2	**1**
8	**4**	**6**	7	**9**	3	2	**1**	**5**
1	2	9	**6**	5	**8**	**7**	4	3
3	**5**	7	1	2	**4**	6	9	8

Fig. 2 – L'unique solution du sudoku précédent, les caractères en gras étaient déjà disposés

Signalons que, dans le sens commun, un sudoku proposé ne doit avoir qu'une unique solution. De plus, bien que tous les sudoku soient soumis aux mêmes règles ils n'ont pas tous la même difficulté. Nous reviendrons sur ce point.

2 Un premier algorithme de résolution

Pour résoudre un tel casse-tête de façon automatisée la première méthode envisagée et de calquer la stratégie de l'algorithme sur celle du joueur. C'est à dire que l'algorithme va utiliser la même logique. Nous pouvons donc définir trois stratégies simples.

Nous allons donc, tout d'abord, représenter un sudoku comme une matrice de $\mathbb{M}_9(\|1,9\|)$ lié à une matrice de listes de même taille, cette dernière ayant pour fonction de stocker les possibilités de chaque case.

Une première fonction va s'occuper de parcourir cette liste pour trouver les listes n'ayant qu'un unique élément, c'est à dire les cases où on l'on connait l'unique possibilité. En plaçant ce chiffre on répercute le changement sur les cases non encore déterminée qui se trouve sur la même ligne, la même colonne et le même bloc que la case courante et enlevant de leurs listes respectives la possibilité correspondante (si elle existe).

Cependant, cette fonction seule ne permet pas de résoudre tous les sudoku. Ce fait permet d'établir un premier niveau de difficulté. On définit donc trois nouvelles fonctions qui vont utiliser une nouvelle stratégie : elles vont, en considérant respectivement une ligne, une colonne ou un bloc, définir si un chiffre ne possède qu'une occurence possible parmi toutes les cases non encore fixées.

De même, ces nouvelles fonctions alliée à la première permettront de résoudre plus de sudoku que cette dernière seule, mais pas tous. Certains sudoku présentent la particularité de mener à un choix. C'est à dire, que pour une case donnée, les 4 stratégies précédentes ne permettent de décider la solution. On utilise alors un algorithme arborescent visant à tester toutes les possiblités offertes. Ainsi, lorsque les 4 stratégies basiques ne permettent plus d'avancer dans la résolution, on va prendre une des cases non encore fixées et possédant le moins de possibilités parmi toutes les cases. On construit un arbre par la racine en créant autant de branche contenant de nouveaux sudoku que de possibilités et plaçant dans chacun de ceux-ci une de ces possibilités. On résoud alors chaque sudoku en utilisant les 4 stratégies précédantes et éventuellement, en réopérant un ou plusieurs choix si nécessaire.

Cet algorithme arrivera à compléter n'impote quel sudoku, pourvu qu'il soit correct, en un temps fini. Il permettra, de plus, d'énumérer toutes les solutions possibles pour un sudoku. Cependant, il sera mis à rude épreuve face à certains sudoku de bon niveau. On notera que si l'on passe à cet algorithme un sudoku entièrement vide il retournera en un temps fini l'ensemble des sudoku possibles (car il se contente de les énumérer et il a été montré par Bertram Felgenhauer et Frazer Jarvis en 2005 que le nombre de sudoku était de $9! * 72^2 * 2^7 * 27704267971 \approx 6.67 * 10^{21}$).

Des stratégies alternatives existent pour avancer la résolution et repousser un choix (voire l'annihiler dans certains cas). Cependant ces stratégies sont utilisées de façon dynamique par les algorithmes les plus communs et sont plutôt mineure dans la résolution.

Notre problématique se transforme alors en : *peut-on donner une réponse à ce problème polynomial sans avoir à ne jamais énumérer et tester de pseudo-solutions ?*

3 La Programmation Linéaire

3.1 Un problème linéaire

Un problème linéaire est un problème posé sous cette forme :

$$n, m \in \mathbb{N}$$

$$f : \left| \begin{array}{l} \mathbb{R}^n \to \mathbb{R} \\ (x_1, x_1, \ldots, x_n) \to f(x_1, x_1, \ldots, x_n) \end{array} \right.$$

$$P : \begin{cases} \text{Maximiser :} & f(x_1, x_1, \ldots, x_n) = c_1 * x_1 + c_2 * x_2 + \ldots + c_n * x_n \\ \text{Contraintes :} & a_{1,1} * x_1 + a_{1,2} * x_2 + \ldots + a_{1,n} * x_n \leq b_1 \\ & a_{2,1} * x_1 + a_{2,2} * x_2 + \ldots + a_{2,n} * x_n \leq b_2 \\ & \vdots \\ & a_{m,1} * x_1 + a_{m,2} * x_2 + \ldots + a_{m,n} * x_n \leq b_m \\ \text{Limites :} & x_1 \geq 0 \\ & x_2 \geq 0 \\ & \vdots \\ & x_n \geq 0 \end{cases}$$

La fonction linéaire f est appelée fonctionnelle ou fonction objectif du problème. En écriture matricielle on peut donner :

$$P : \begin{cases} \textit{Maximiser :} & {}^tC * X \\ \textit{Contraintes :} & A * X \leq B \\ \textit{Limites :} & X \geq 0 \end{cases}$$

$$X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \quad C = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \quad B = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

3.2 Un premier exemple

Un agriculteur cultivant du maïs et du blé cherche maximiser son revenu. Il cherche combien de kilos de semences il doit prévoir. Chaque kilo de maïs lui rapportera 10€et chaque kilo de blé lui rapportera 7€. Chaque kilo de maïs occupe 7 ares et demande 3 litres de fertilisant et chaque kilo de blé occupe 3 ares et demande 5 litres de fertilisant.

Question : combien de kilos de maïs et de blé l'agriculteur doit-il prvoir pour maximiser son revenu ?

On peut mettre ce problème sous la forme d'un problème linéaire :

a : nombre de kilo de maïs

b : nombre de kilo de blé

$$P : \begin{cases} Maximiser : & 10 * a + 7 * b \\ Contraintes : & 7 * a + 3 * b \leq 300 \\ & 3 * a + 5 * b \leq 270 \\ Limites : & a \geq 0 \\ & b \geq 0 \end{cases}$$

Ou encore :

$$P : \begin{cases} Maximiser : & {}^t\begin{pmatrix} 10 \\ 7 \end{pmatrix} * \begin{pmatrix} a \\ b \end{pmatrix} \\ Contraintes : & \begin{pmatrix} 7 & 3 \\ 3 & 5 \end{pmatrix} * \begin{pmatrix} a \\ b \end{pmatrix} \leq \begin{pmatrix} 300 \\ 270 \end{pmatrix} \\ Limites : & \begin{pmatrix} a \\ b \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{cases}$$

Sous une forme graphique, on peut représenter les contraintes et le domaine des solutions dites "acceptables", c'est à dire : qui ne violent pas les contraintes posées (Figure 1 et Figure 2).

Dans ce cas précis on peut résoudre le problème en calculant l'intersection des deux droites délimitant le domaine. La solution est unique car, comme on peut le constater sur le graphique de la Figure 3 l'équation de la fonction objectif n'est pas parallèle à une des équations contraintes.

On trouve :

$$a = 26.5385, b = 38.0769$$

Dans ce cas

$$f(26.5385, 38.0769) = 531.923$$

Réponse au problème : *Il faut planter 26.53 kilos de maïs et 38.07 kilos de blé pour maximiser les revenus, en gagnant 531.92€.*

FIG. 3 – Contraintes du problème P

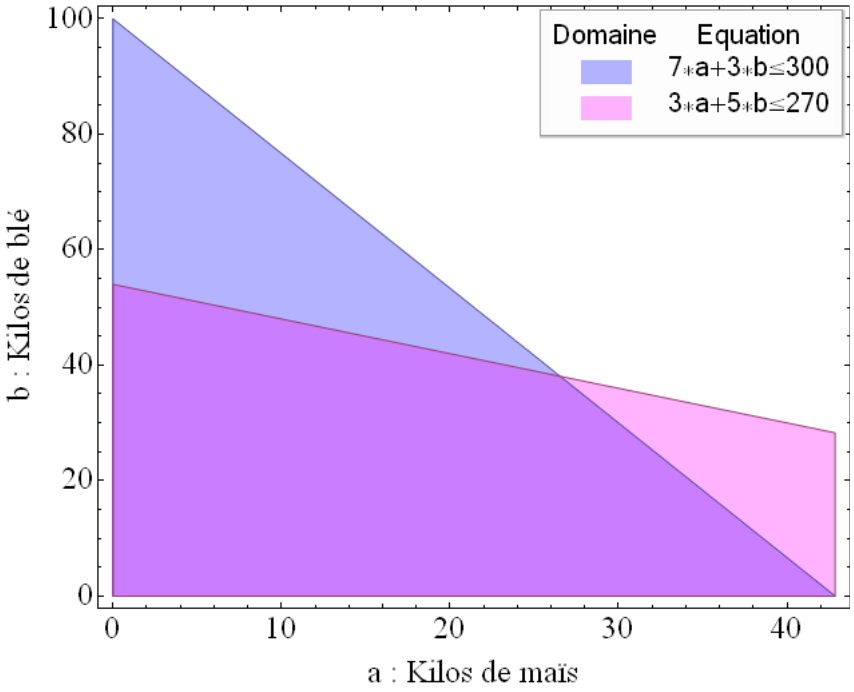


FIG. 4 – Domaine des solutions recevables du problème P

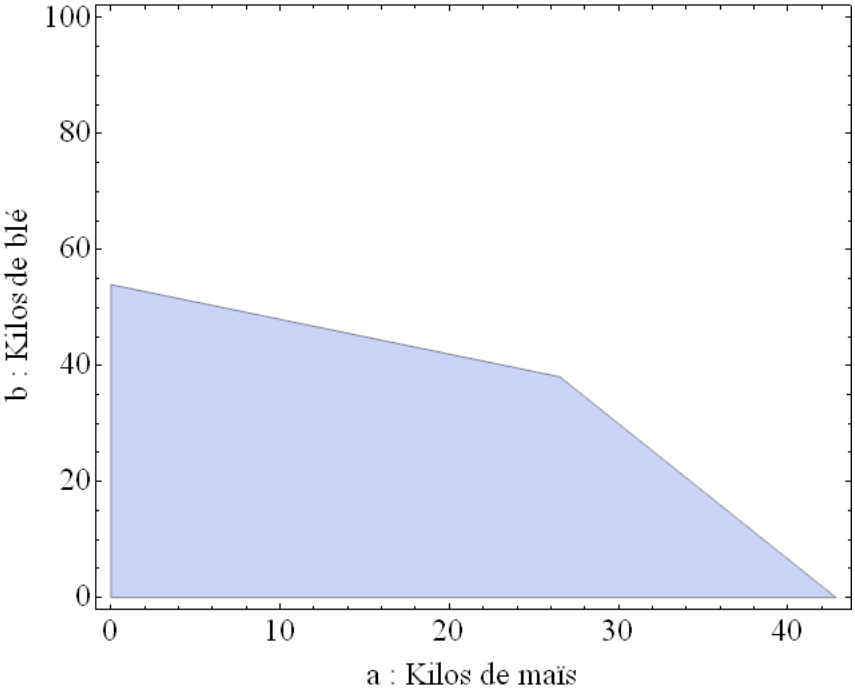
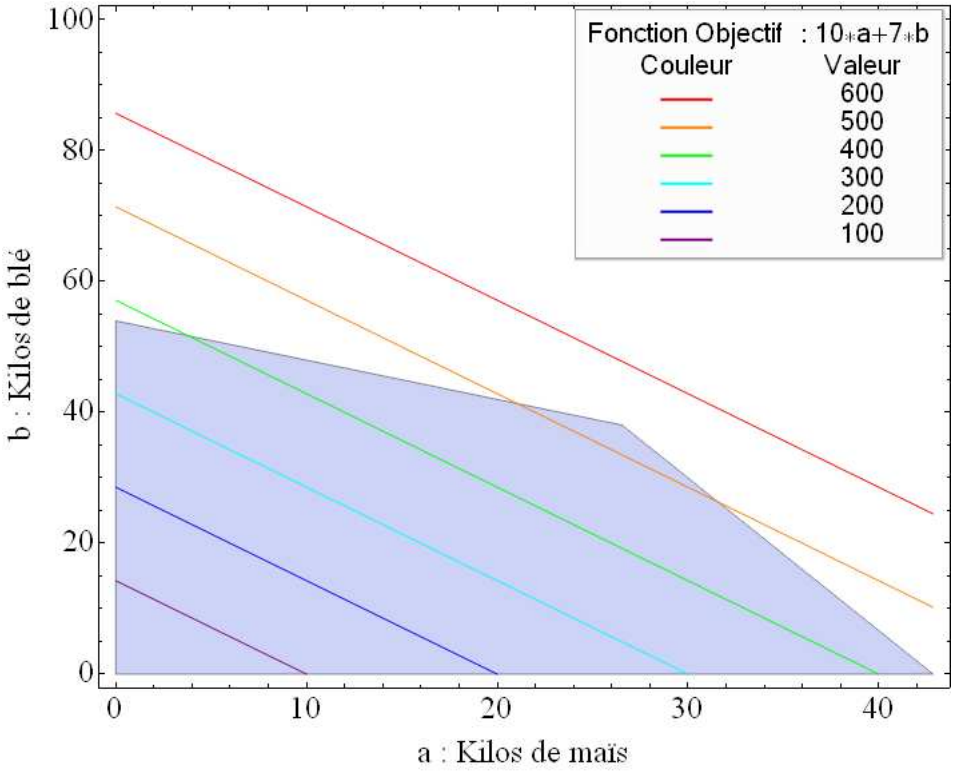


FIG. 5 – Superposition du domaine précédant avec certaines valeurs de la fonctionnelle



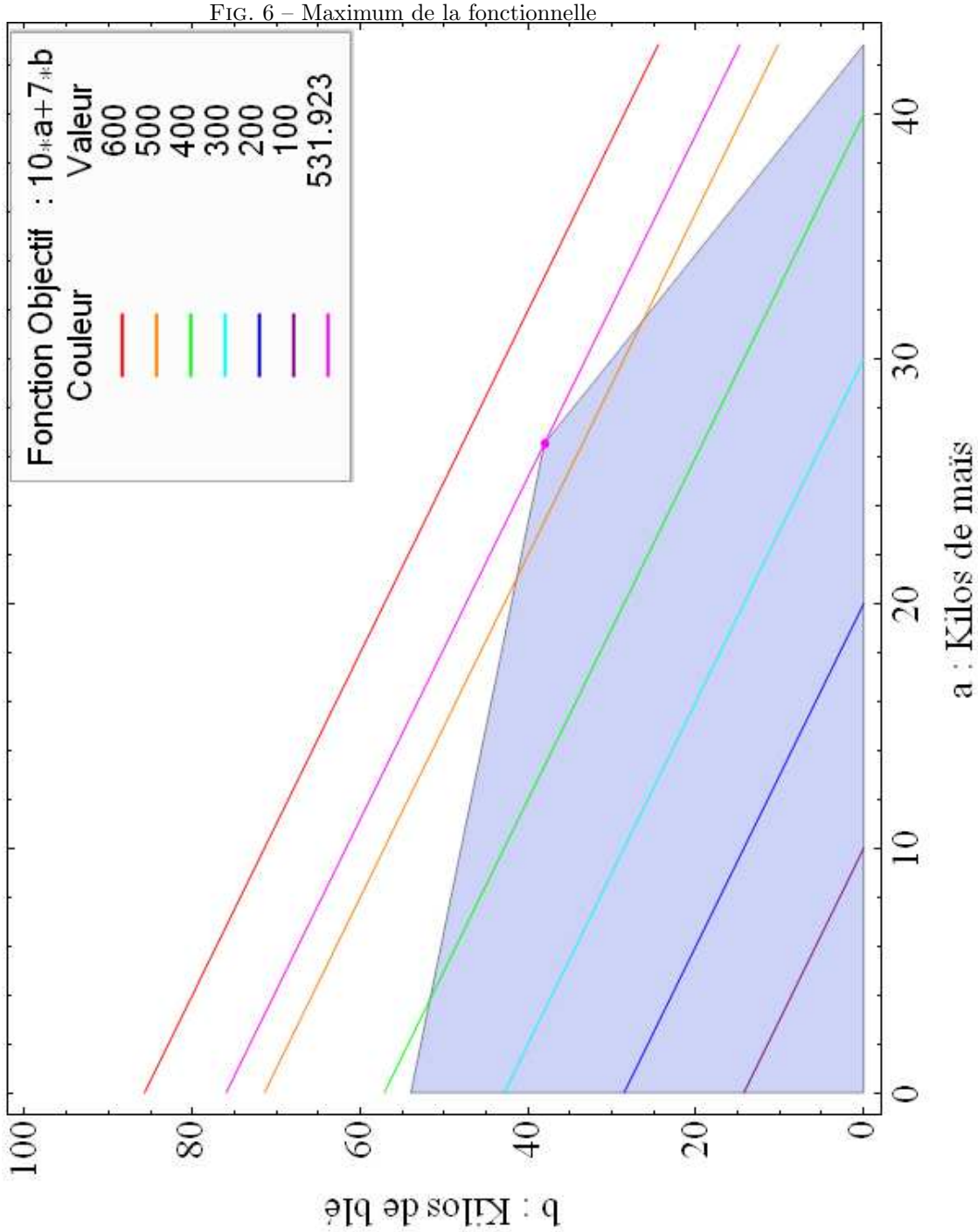


FIG. 6 – Maximum de la fonctionnelle

3.3 L'algorithme du Simplexe

On note E l'ensemble des situations accaptables, c'est à dire, l'ensemble des points dans $\mathbb{R}^n$ satisfaisant les équations contraintes. Avec les notations précédentes :

$$E = \{X \in \mathbb{R}^n / A * X \leq B\}$$

On peut facilement montrer que cet ensemble solution est une partie convexe de $\mathbb{R}^n$:

$$\forall X, Y \in E, \forall a \in [0,1], (1-a) * X + a * Y \in E$$

Démonstration :

$$\forall X, Y \in E, \forall a \in [0,1],$$

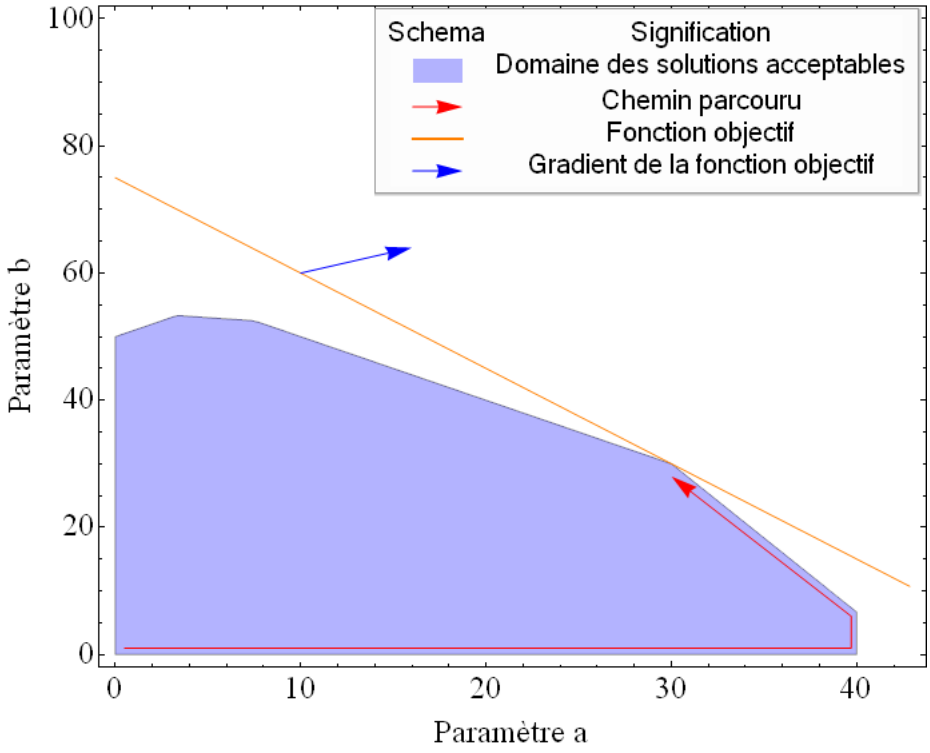
$$A * ((1-a) * X + a * Y) = (1-a) * A * X + a * A * Y \leq (1-a) * B + a * B = B$$

$$A * ((1-a) * X + a * Y) \leq B \Rightarrow ((1-a) * X + a * Y) \in E$$

De plus, l'ensemble des solutions forme un polyhèdre dans l'espace $\mathbb{R}^n$, comme espace convexe délimité par un nombre fini d'hyperplans.

La clef de l'algorithme du Simplexe telle que formulé par G. Dantzig est le simple parcours du polytope délimitant l'espace E des solutions comme suit :

FIG. 7 – Exemple de parcours de l'algorithme pour un problème à deux variables



L'algorithme part d'une solution acceptable (ici : $(0;0)$) qui est un sommet et choisit le prochain sommet comme ayant une arrête commune avec le sommet courant et permettant de maximiser la fonction objectif. Si l'algorithme ne peut déterminer un tel sommet alors une solution est trouvée.

On notera que si la droite fonction objective maximale est parallèle à la droite représentant l'équation d'une contrainte alors il existe une infinité de solutions, toutes réparties sur le segment formé par deux sommets pour les quels la maximisation est atteinte.

Enfin, l'algorithme utilise une expression matricielle du problème en adaptant à chaque sommet rencontré une nouvelle base de l'espace, permettant de traduire le prochain sommet cible permettant de maximiser la fonctionnelle (si besoin est).

3.4 La méthode du simplexe dans le cas du premier exemple

Rappel de la forme du problème :

$$P : \begin{cases} Maximiser : & 10 * a + 7 * b \\ Contraintes : & 7 * a + 3 * b \leq 300 \\ & 3 * a + 5 * b \leq 270 \\ Limites : & a \geq 0 \\ & b \geq 0 \end{cases}$$

On introduit alors deux variables d'écarts x_1 et x_2 dans la modélisation des contraintes et on utilise la forme *duale* du problème (la forme *primale* du problème étant liée à sa maximisation) :

$$P : \begin{cases} Minimiser : & p = 10 * a + 7 * b + 0 * x1 + 0 * x2 \\ Contraintes : & 7 * a + 3 * b + x_1 = 300 \\ & 3 * a + 5 * b + x_2 = 270 \\ Limites : & a \geq 0 \quad b \geq 0 \quad x_1 \geq 0 \quad x_2 \geq 0 \end{cases}$$

On peut alors formuler le problème sous une forme matricielle :

T_0 :

Variables	a	b	x1	x2	p	B
Contrainte 1 (L_1)	7	3	1	0	0	300
Contrainte 2 (L_2)	3	5	0	1	0	270
Fonctionnelle (L_3)	-10	-7	0	0	1	0

Les variables x_1 et x_2 sont dites en base, tandis que les variables a et b sont dites hors base.

Première itération :

Choix de la colonne pivot : on selectionne dans la dernière ligne, la ligne représentant la fonctionnelle, le nombre négatif dont la valeur absolue est la plus grande. Ici -10 satisfait ces conditions, on choisit donc la colonne de la variable a.

Choix du pivot : On observe les coefficients des contraintes de la colonne pour la variable a : ils sont 7 et 3. On cherche à augmenter la variable a dans les limites de ces contraintes :

$$C_0 : \begin{cases} 7 * a \leq 300 & \Leftrightarrow a \leq \frac{300}{7} \\ 3 * a \leq 270 & \Leftrightarrow a \leq \frac{270}{3} \\ Limites : & a \geq 0 \end{cases}$$

La contrainte la plus restrictive est bien entendu :

$$a \leq \frac{300}{7}$$

On choisit de ce fait 7 comme pivot et on applique les transformations suivantes :

$$L_2 \leftarrow 7 * L_2 - 3 * L_1$$

$$L_3 \leftarrow 7 * L_3 + 10 * L_1$$

On obtient le nouveau tableau :

T_1 :

Variables	a	b	x1	x2	p	B
Contrainte 1 (L_1)	7	3	1	0	0	300
Contrainte 2 (L_2)	0	26	-3	7	0	990
Fonctionnelle (L_3)	0	-19	10	0	7	3000

On fait sortir x_1 de la base et rentrer a en base.

Seconde itération :

On réitère ce même procédé : on choisit cette fois la colonne de la variable b, c'est la seule qui possède un coefficient strictement négatif (-19).

Les contraintes sont maintenant :

$$C_1 : \begin{cases} 3 * b \leq 300 & \Leftrightarrow b \leq 100 \\ 26 * b \leq 990 & \Leftrightarrow b \leq \frac{495}{13} \\ Limites : & b \geq 0 \end{cases}$$

La contrainte la plus restrictive est maintenant :

$$b \leq \frac{495}{13}$$

On choisit de ce fait 26 comme pivot et on applique les transformations suivantes :

$$L_1 \leftarrow 26 * L_1 - 3 * L_2$$

$$L_3 \leftarrow 26 * L_3 + 19 * L_1$$

On obtient le nouveau tableau :

T_2 :

Variables	a	b	x1	x2	p	B
Contrainte 1 (L_1)	182	0	35	-21	0	4830
Contrainte 2 (L_2)	0	26	-3	7	0	990
Fonctionnelle (L_3)	0	0	203	133	182	96810

On fait sortir x_2 de la base et rentrer b en base.

Il n'y a plus de coefficient strictement négatif dans l'expression de la fonctionnelle : la solution optimale a été atteinte. Les variables a et b sont en base, on trouve :

$$182 * a = 4830 \Leftrightarrow a = \frac{4830}{182} \approx 26.54$$

$$26 * b = 990 \Leftrightarrow b = \frac{990}{26} \approx 38.08$$

Les variables x_1 et x_2 sont hors base, on a :

$$x_1 = 0$$

$$x_2 = 0$$

De même, la fonctionnelle a pour valeur :

$$0 * a + 0 * b + 203 * x_1 + 133 * x_2 + 182 * p = 98610 \Rightarrow p = \frac{96810}{182} \approx 531.92$$

Il faut donc prévoir 26.54 kilos de maïs et 38.08 kilos de blé pour maximiser le revenu, qui vaudra 531.92€.

3.5 Second exemple et limites

Une usine produit deux sortes de jouets : des petits trains vendus 10€et des voitures vendues 7€. Chaque trains est fabriqué partir de 7 unités de bois et 3 unités de fer, chaque voiture est fabriquée partir de 3 unités de bois et 5 unités de fer. L'usine dispose de 300 unités de bois et 270 unités de fer.

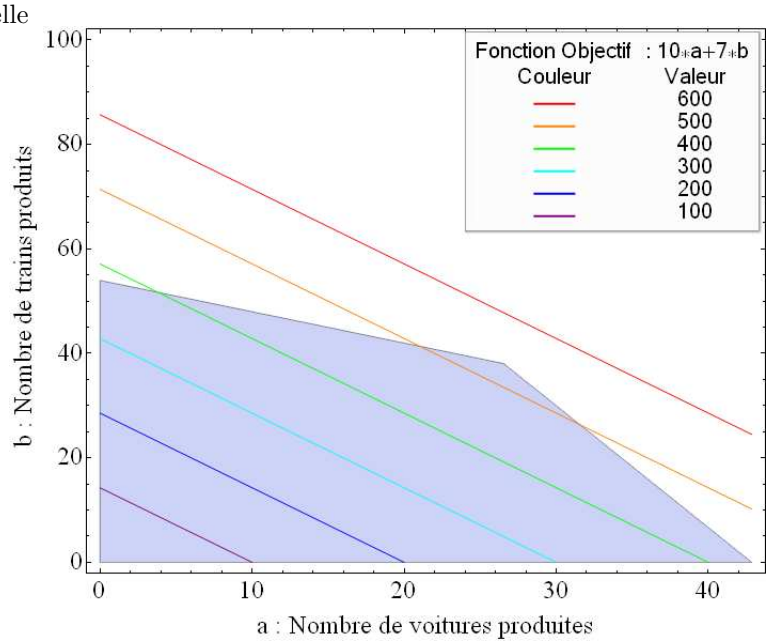
Question : combien de voitures et de trains doit fabriquer l'usine pour faire un profit maximum ?

On peut mettre ce problème sous la forme d'un problème linéaire :

a : quantité de trains produites

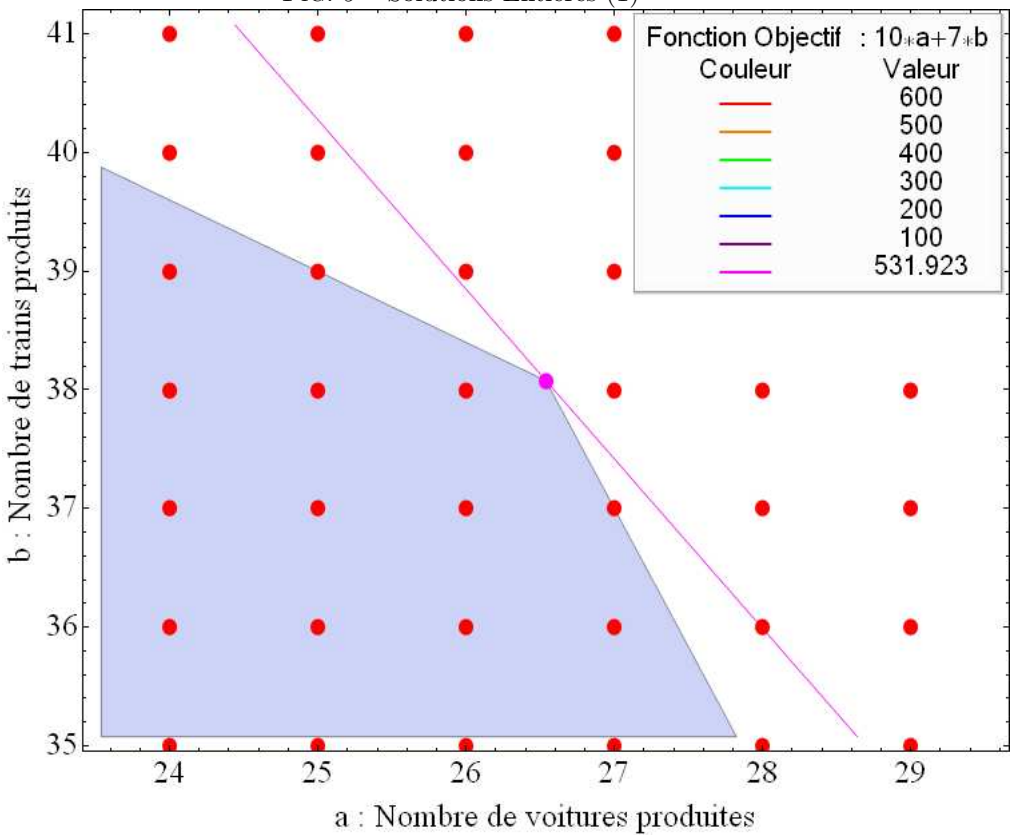
b : quantité de voitures produites

FIG. 8 – Superposition du domaine précédant avec certaines valeurs de la fonctionnelle



Ce problème est identique au problème précédant... mais la solution obtenue n'est pas acceptable : les nombres de jouets à produires ne sont pas des nombres entiers ! Représentons maintenant les solution entières :

FIG. 9 – Solutions Entières (1)



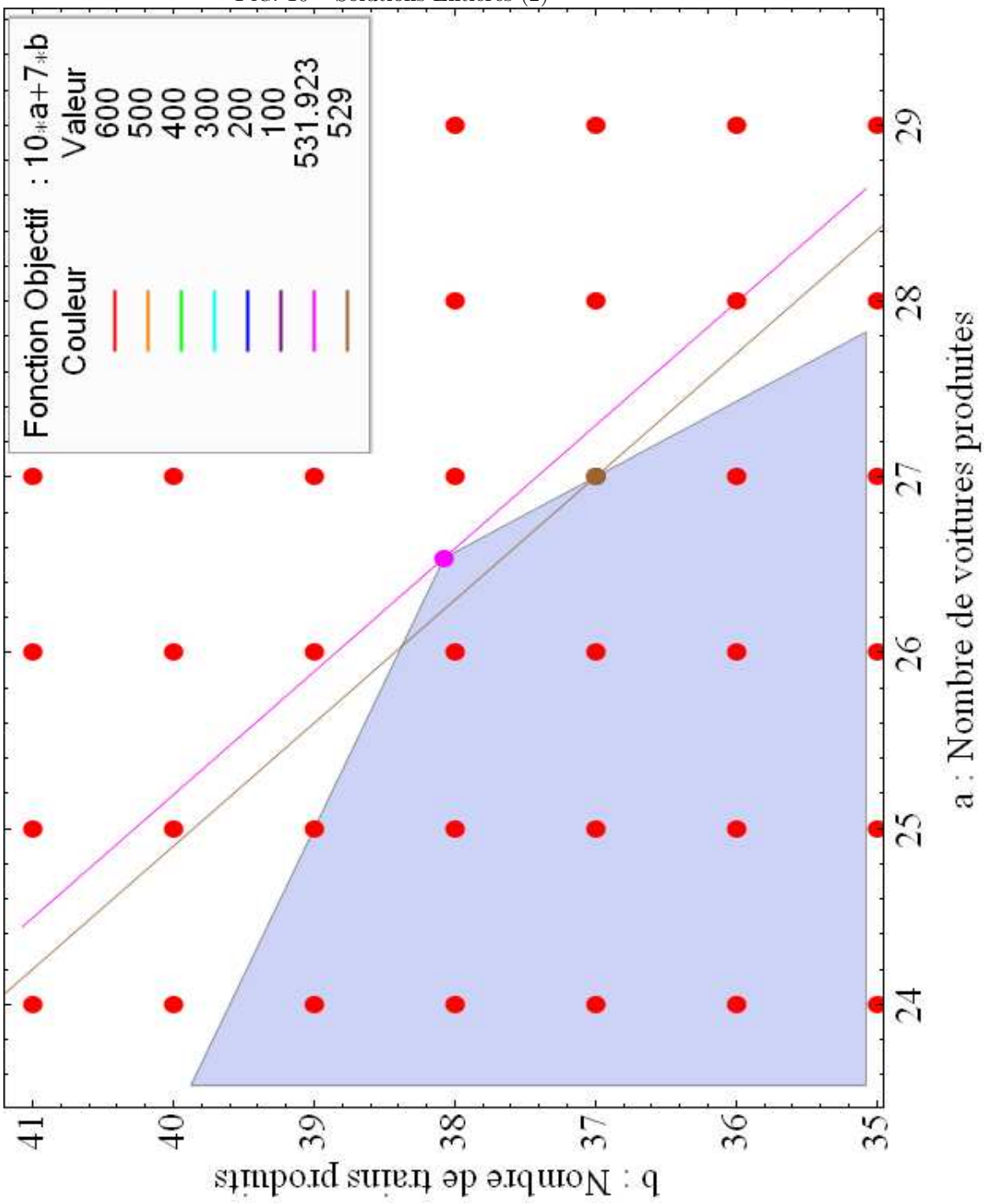
Dans cet exemple simple on remarque une solution sur une équation-contrainte. On trouve :

$$a = 27, b = 37$$

Dans ce cas :

$$f(27, 37) = 529$$

FIG. 10 – Solutions Entières (2)



On a donc montré que l'algorithme du Simplexe ne peut pas s'adapter dans sa forme première au problème en nombres entiers ou en booléens. On utilise donc un second algorithme pour trouver une solution entière !

3.6 Programmation linéaire en nombre entiers

3.6.1 Représentation du problème

On représente en écriture matricielle la répartition discrète de solutions par :

$$P : \begin{cases} Maximiser : & {}^tC * X \\ Contraintes : & A * X \leq B \\ Limites : & X \geq 0 \end{cases}$$

Avec :

$$X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{N}^n \quad C = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \in \mathbb{R}^n \quad B = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \in \mathbb{R}^m$$

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \in M_{m,n}(\mathbb{R})$$

3.6.2 Relaxation continue du problème

On considère tout d'abord l'ensemble des variables comme un ensemble continu :

$$\forall i \in [|1, n|], \quad x_i \in \mathbb{R}$$

On résoud alors le système grâce à l'algorithme du simplexe pour trouver une solution acceptable, sommet du polytope délimité par l'ensemble des hyperplans-contraintes. On dispose d'un vecteur solution de P_s :

$$X_s \in \mathbb{R}^n$$

On cherche à présent à obtenir une solution dans $\mathbb{N}^n$.

3.6.3 Algorithme "Branch and Bound"

Ce premier algorithme consiste à construire un arbre de solution dans lequel on évolue en vérifiant la maximisation du problème et en utilisant la relaxation continue précédante. Nous allons donc décrire rapidement cet algorithme dans le cas d'un problème en nombres binaires :

- Étape Initiale : on considère un problème : on le note P_0, il est identique à P_s au domaine discret de solution près. On note V_0 l'ensemble des variables de ce problème, elles n'ont pas de valeurs assignée.
- On construit un arbre par la racine et on le dote de deux fils : un pour lequel une variable x de V_0 se voit assigner la valeur 0, l'autre pour lequel la même variable x de V_0 se voit assigner la valeur 1. On note le problème associé : P_1 ; et l'ensemble des variables "libres" : V_1 (ie. $V_1 = V_0 \setminus \{x\}$.

– Pour chaque noeud de cet arbre, de problème P_i et d'ensemble de variables V_i :
  – Si la solution de la relaxation continue du problème associé à ce noeud est pire, en terme de maximisation de la solution objectif, que la meilleure solution entière trouvée jusqu'ici alors on remonte au dernier noeud non totalement exploré et on supprime la branche que l'on vient de quitter (en anglais *backtracking*).
  – Si la solution est entièrement définie dans $\mathbb{B}^n$ on la compare à la meilleure solution entière trouvée jusqu'ici et on conserve celle des deux qui est la meilleure.
  – Sinon, on considère une variable x de l'ensemble V_i et on ajoute au noeud courant les deux fils pour les quels x à soit pris la valeur 0, soit la valeur 1. Donc ces deux noeuds fils sont muni de P_{i+1} et de $V_{i+1} = V_i \setminus \{x\}$.
– La solution entière est donc la solution trouvée.

3.6.4 Application de l'algorithme "Branch and Bound"

On reprend le problème linéaire précédent :

$$P : \begin{cases} Maximiser : & 10 * a + 7 * b \\ Contraintes : & 7 * a + 3 * b \leq 300 \\ & 3 * a + 5 * b \leq 270 \\ Limites : & a \in \mathbb{N}, \quad b \in \mathbb{N} \end{cases}$$

Par la relaxation continue on avait obtenu cette solution :

$$a = 26.5385, \quad b = 38.0769$$

Dans ce cas, la valeur de la fonctionnelle est :

$$f(26.5385, 38.0769) = 531.923$$

On choisit d'abord la variable, parmi a et b, dont la valeur est la plus éloignée d'un entier. Ici c'est a qui remplit ce rôle. On va donc proposer deux nouveaux problèmes en ajoutant à chacun une contrainte découlant du caractère entier que doit respecter a. On note :

$$P_1 : \begin{cases} Maximiser : & 10 * a + 7 * b \\ Contraintes : & 7 * a + 3 * b \leq 300 \\ & 3 * a + 5 * b \leq 270 \\ & 1 * a + 0 * b \leq 26 \quad (C_1) \\ Limites : & a \in \mathbb{N}, \quad b \in \mathbb{N} \end{cases}$$

$$P_2 : \begin{cases} Maximiser : & 10 * a + 7 * b \\ Contraintes : & 7 * a + 3 * b \leq 300 \\ & 3 * a + 5 * b \leq 270 \\ & 1 * a + 0 * b \geq 27 \quad (C_2) \\ Limites : & a \in \mathbb{N}, \quad b \in \mathbb{N} \end{cases}$$

Les problèmes P_1 et P_2 vont être résolus séparément par l'algorithme du Simplexe. Les contraintes C_1 et C_2 traduisent le fait que a ne peut être dans l'intervalle ouvert $]\,26, 27\,[$.

En résolvant P_1, on obtient :

$$a = 26, \quad b = 38.4, \quad f(26, 38.4) = 528.8$$

On remarque b n'est pas entier. On choisit cette variable et on applique de nouveau la méthode précedente :

$$P_3 : \begin{cases} Maximiser : & 10 * a + 7 * b \\ Contraintes : & 7 * a + 3 * b \leq 300 \\ & 3 * a + 5 * b \leq 270 \\ & 1 * a + 0 * b \leq 26 \quad (C_1) \\ & 0 * a + 1 * b \leq 38 \quad (C_3) \\ Limites : & a \in \mathbb{N}, \quad b \in \mathbb{N} \end{cases}$$

$$P_4 : \begin{cases} Maximiser : & 10 * a + 7 * b \\ Contraintes : & 7 * a + 3 * b \leq 300 \\ & 3 * a + 5 * b \leq 270 \\ & 1 * a + 0 * b \geq 27 \quad (C_2) \\ & 0 * a + 1 * b \geq 39 \quad (C_4) \\ Limites : & a \in \mathbb{N}, \quad b \in \mathbb{N} \end{cases}$$

En résolvant P_3, on obtient :

$$a = 26, \quad b = 38, \quad f(26, 38.4) = 526$$

On constate que P_3 a une solution entière.
En résolvant P_4, on obtient :

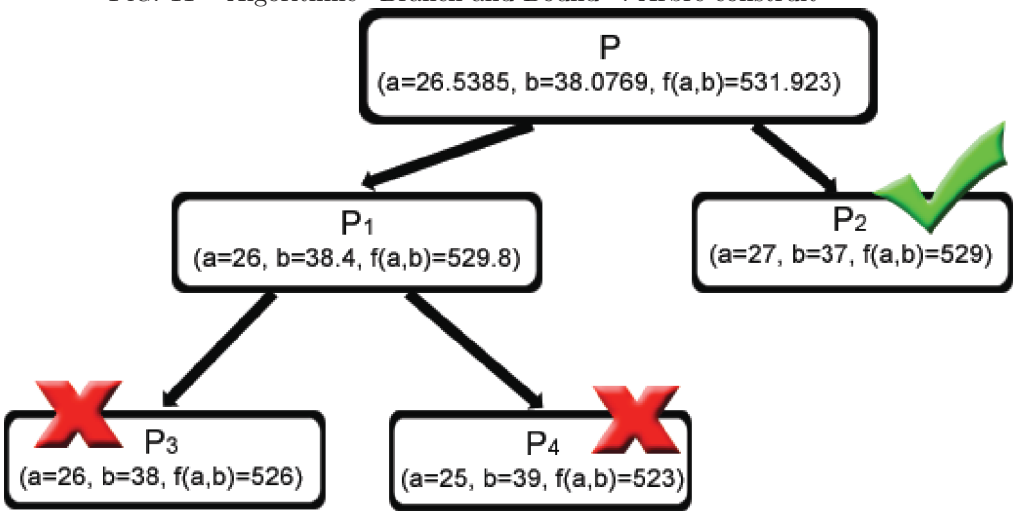
$$a = 25, \quad b = 39, \quad f(26, 38.4) = 523$$

On constate que P_4 a une solution entière. De plus, la fonctionnelle est maximum pour la solution du problème P_3 par rapport à la solution du problème P_4. On peut donc conserver celle-ci.
En résolvant P_2, on obtient :

$$a = 27, \quad b = 37, \quad f(27, 37) = 529$$

On constate que P_2 est une solution entière au problème. De plus, la fonctionnelle est maximale pour la solution du problème P_2 par rapport à la solution du problème P_3. On peut donc conserver la solution du problème P_2 qui est la solution entière optimale du problème P. On vérifie, bien évidemment, que cette solution correspond à celle trouvée "*graphiquement*".
On peut représenter l'arbre construit pendant la recherche :

FIG. 11 – Algorithme "Branch and Bound" : Arbre construit



3.6.5 Algorithme "Cutting Planes"

Cet algorithme réside dans l'ajout de nouvelle contraintes par la procédure suivante :

- Résoudre la relaxation du problème linéaire
- Si la solution est entière : terminer le processus
- Sinon : Ajouter une contrainte linéaire qui permet de retirer la solution de la relaxation *SANS* retirer de solution entière.

Il existe pour cela, des méthodes permettant de déterminer de tels plans sécants à partir des contraintes déjà utilisées : les Coupes de Gomory.

3.6.6 Dans la pratique

En fait, ces deux techniques sont utilisées conjointement car : les coupes de Gomory ne permettent pas de converger *rapidement* vers la solution (en comparaison de la méthode arborescente seule). Cependant leur combianaison permet de ne pas avoir à explorer certaines branches ouvertes par l'algorithme "Branch and Bound".

4 La Résolution d'un sudoku comme un problème linéaire

4.1 La modélisation du problème

On a vu qu'un Sudoku pouvait être représenter par une matrice carrée de 81 éléments. On peut aussi le représenter sous la forme :

$$\forall (i,j,k) \in [|1;9|]^3, \quad \psi_{i,j,k} \in \mathbb{B}$$

$$\psi_{i,j,k} = \begin{vmatrix} 1 & \text{si la case de coordonnées } (i,j) \text{ contient le nombre } k \\ 0 & \text{sinon} \end{vmatrix}$$

Avec la représentation précédante on peut écrire l'ensembles des contraintes d'un sudoku comme sujet à ensemble de contraintes :

$$S : \begin{cases} \textit{Contraintes} : & \forall (i,j) \in [|1,9|]^2, \quad \sum_{k=1}^{9} \psi_{i,j,k} = 1 \\ & \forall (i,k) \in [|1,9|]^2, \quad \sum_{j=1}^{9} \psi_{i,j,k} = 1 \\ & \forall (j,k) \in [|1,9|]^2, \quad \sum_{i=1}^{9} \psi_{i,j,k} = 1 \\ & \forall (i_0,j_0) \in [|0,2|]^2, \quad \sum_{i=1}^{3}\sum_{j=1}^{3} \psi_{3i_0+i,3j_0+j,k} = 1 \\ \textit{Limites} : & \forall (i,j,k) \in [|1;9|]^3, \quad \psi_{i,j,k} \in \mathbb{B} \end{cases}$$

Puis, on peut alors décrire un problème linéaire associé :

$$S_l : \begin{cases} \textit{Maximiser} : & \sum_{(i,j,k)\in[|1;9|]^3} \psi_{i,j,k} \quad (1) \\ \textit{Contraintes} : & \forall (i,j) \in [|1,9|]^2, \quad \sum_{k=1}^{9} \psi_{i,j,k} \leq 1 \quad (2) \\ & \forall (i,k) \in [|1,9|]^2, \quad \sum_{j=1}^{9} \psi_{i,j,k} \leq 1 \quad (3) \\ & \forall (j,k) \in [|1,9|]^2, \quad \sum_{i=1}^{9} \psi_{i,j,k} \leq 1 \quad (4) \\ & \forall (i_0,j_0) \in [|0,2|]^2, \quad \sum_{i=1}^{3}\sum_{j=1}^{3} \psi_{3i_0+i,3j_0+j,k} \leq 1 \quad (5) \\ \textit{Limites} : & \forall (i,j,k) \in [|1;9|]^3, \quad \psi_{i,j,k} \in \mathbb{B} \quad (6) \end{cases}$$

Clarifions les inégalités apparaîssant ci-dessus :

1. Fonctionnelle à maximiser : il doit y avoir un maximum de '1' dans le Sudoku
2. Contrainte : il ne doit y avoir qu'un seul chiffre dans chaque case.
3. Contrainte : il ne doit y avoir qu'un seul chiffre donné parmi $[|1,9|]$ sur chaque ligne.
4. Contrainte : il ne doit y avoir qu'un seul chiffre donné parmi $[|1,9|]$ sur chaque colonne.

5. Contrainte : il ne doit y avoir qu'un seul chiffre donné parmi $[|1, 9|]$ dans chaque bloc de 3x3.
6. Limite : chaque variable est binaire.

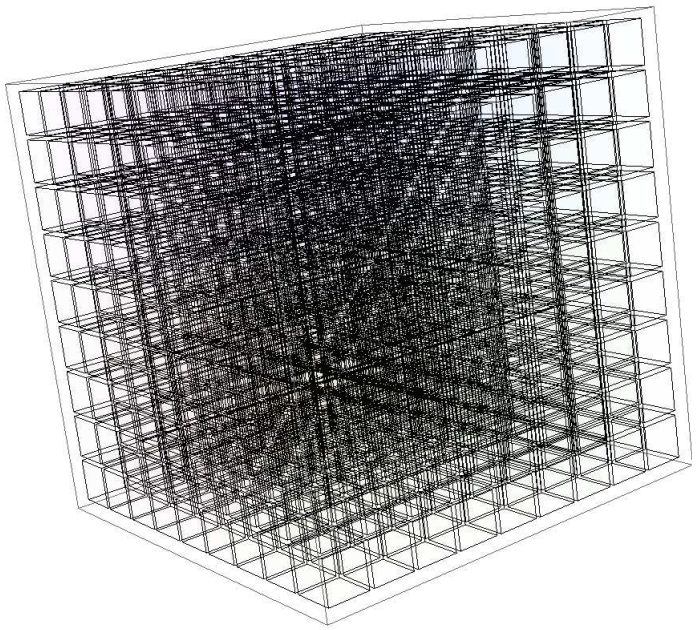
4.2 Le programme

4.2.1 Le principe général

Le programme chargé de résoudre les sudoku utilise la bibliothèque GLPK, acronyme de *GNU Linear Programming Kit*, qui implémente la méthode du Simplexe et permet la résolution de problèmes linéaires en nombres entiers.

Le programme doit charger un fichier texte contenant le sudoku à résoudre et construire la structure de donnée adéquate. Nous pouvons représenter celle-ci par un tableau tri-dimensionnel :

Fig. 12 – Structure des données



Pour chaque chiffre connu, le programme doit *"fermer"* les possibilités similaires. C'est à dire qu'il doit mettre à *FAUX* les possibilités du chiffre dans les zones suivantes : la ligne, la colonne et le bloc auxquels appartient le chiffre. De plus, il doit aussi fermer la possibilité de mettre un autre chiffre dans la case. Enfin il met le chiffre considéré à *VRAI*. Il recommence ce procédé pour chaque chiffre lu.

FIG. 13 – Structure des données : possibilités supprimées au chargement (1)

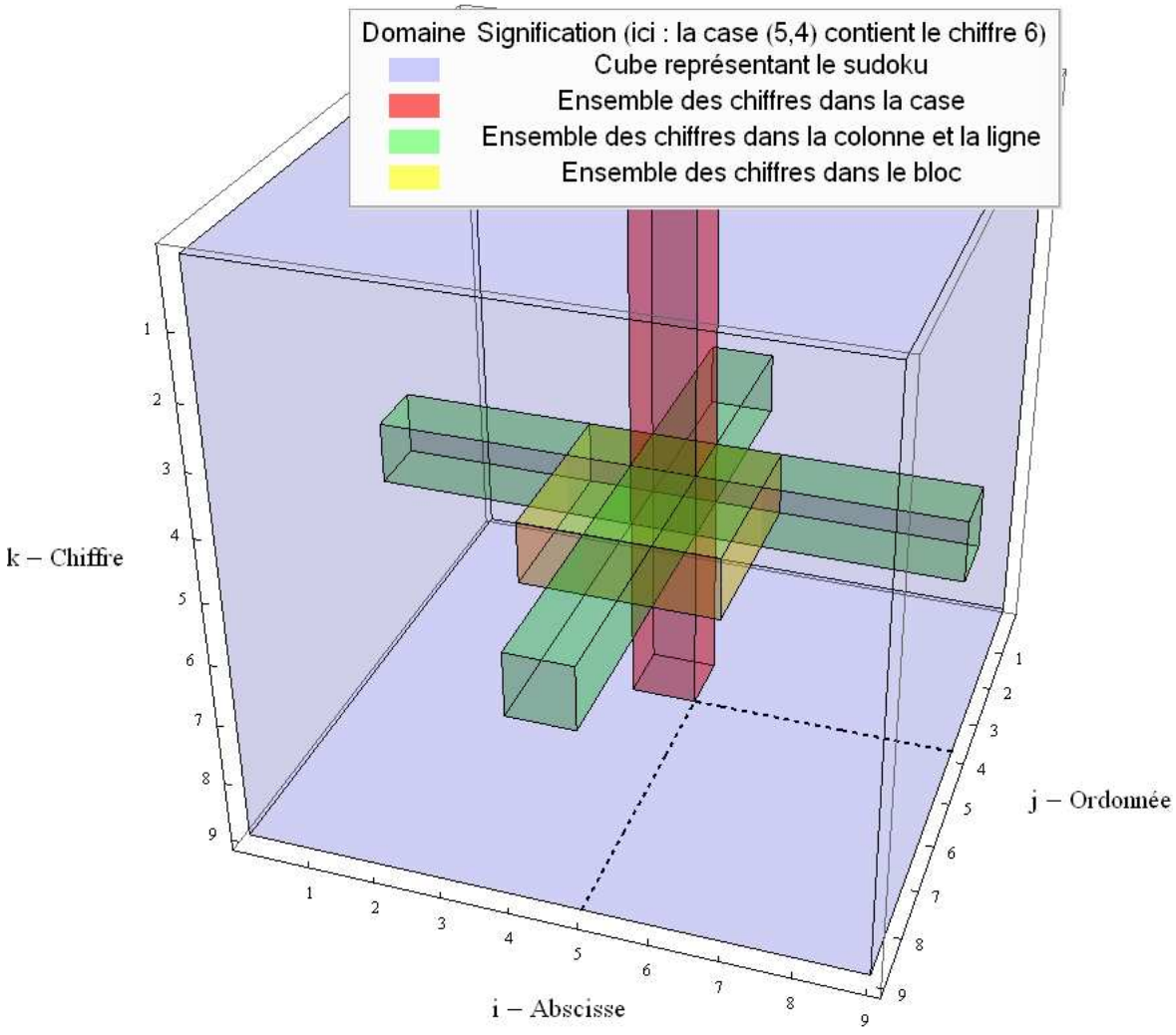
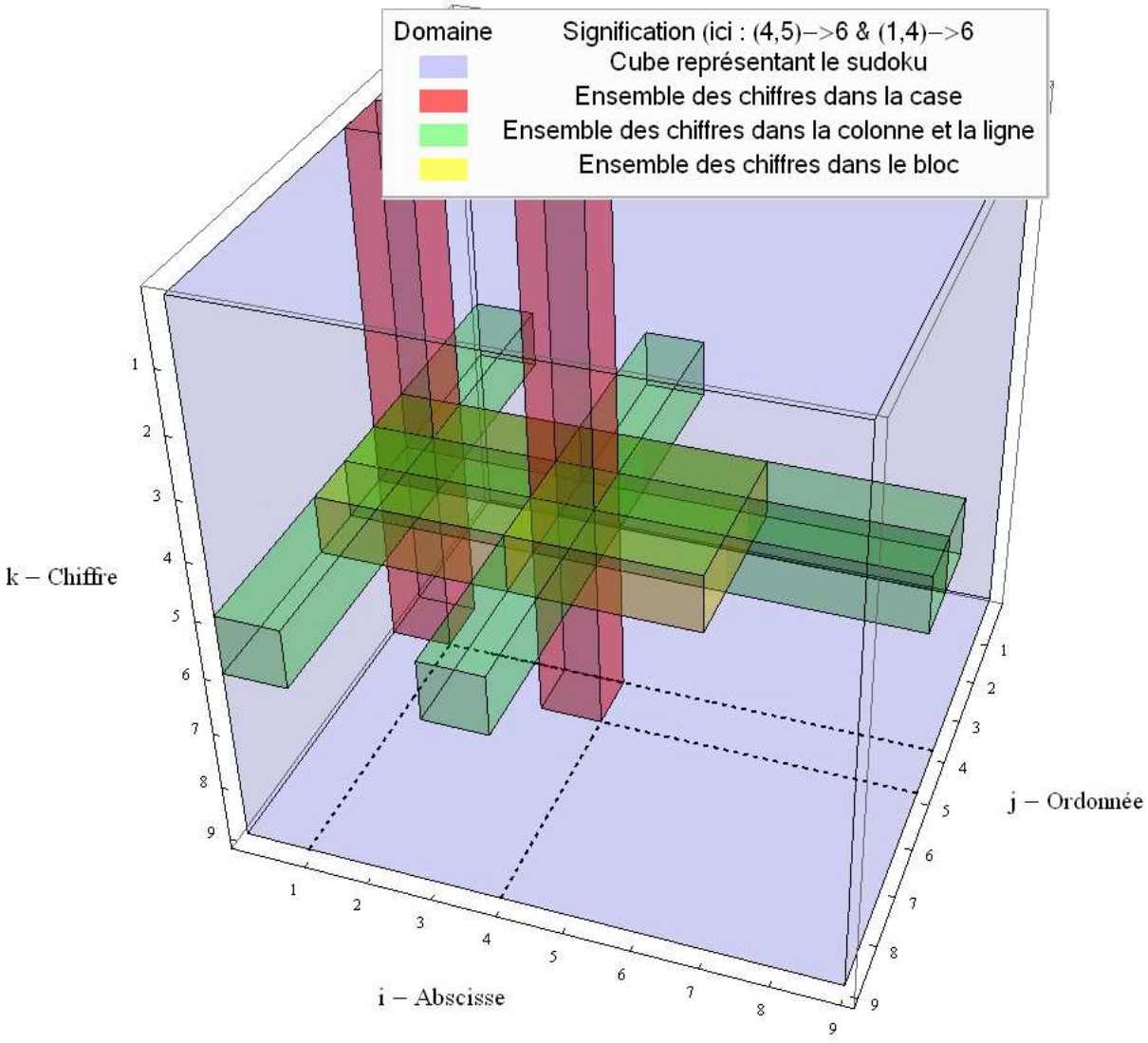


FIG. 14 – Structure des données : possibilités supprimées au chargement (2)



Enfin il construit le problème lié aux cases inconnues et aux contraintes s'y appliquant en specifiant la nature binaire des variables. Il faut ensuite lancer la résolution puis récupérer la solution trouvée et la remettre en forme selon le passage de la structure binaire à la forme d'un sudoku conventionnel.

4.2.2 Quelques résultats

Pour faciliter l'étude des résultats, le programme permet de lire une suite de sudoku disposé en ligne dans un fichier pour plus d'aisance. Les premiers résultats donnent, sur une machine commune (processeur à 1.2Ghz) :

Type de sudoku	Nombres	moyen (s)	maxi. (s)	échec(s)
niveaux : *"Facile", "Moyen" et "Difficile"*	50	0.04	≈ 1	0
générés aléatoirement - série 1	1000	0.084	≈ 1	0
générés aléatoirement - série 2	1000	0.087	≈ 1	0
générés aléatoirement - série 3	1000	0.08	≈ 1	0
générés aléatoirement - série 4	1000	0.08	≈ 1	0
générés aléatoirement - série 5	1000	0.08	≈ 1	0
niveau : *"Très Difficile"*	10000	0.303	5	0
Parmi les plus difficile connus	1400	1.67	8	0

Remarque 1 : Les temps mini. ne sont pas précisés car l'outil de mesure ne permet pas de les distinguer d'un temps nul.

Remarque 2 : Les niveaux de difficultés présentés sont déduit par rapport à un certains nombre de critères directement liés à la stratégie "humaine". Parmi lesquels on trouve : le nombre de chiffre que l'on peut placer dès le début en utilisant des techniques d'élimination classique, le nombre de choix successifs et obligatoires permettant d'arriver à la solution ou encore l'utilisation de techniques dites à reconnaissance de motifs.

Remarque 3 : Les sudoku générés *aléatoirement* l'on été grâce au programme **suexg**. Il fournit un nombre de sudoku de difficultés aléatoires à partir d'une graine lui servant pour la génération de nombres aléatoires.

On peut d'ores-et-déjà remarquer que le temps de résolution est en accord avec la difficulté des sudoku proposés. Et ce, même si la stratégie employée n'est pas celle d'une logique humaine.

Enfin, ce programme ne permet pas d'énumérer toutes les solutions possibles d'un sudoku. Mais il a la particularité de donner une réponse au sudoku vide !

1	8	4	6	5	9	2	3	7
2	7	6	3	8	1	9	5	4
3	9	5	4	7	2	1	8	6
4	1	7	2	6	3	8	9	5
5	2	8	9	1	7	6	4	3
6	3	9	5	4	8	7	1	2
7	6	1	8	3	4	5	2	9
8	4	2	7	9	5	3	6	1
9	5	3	1	2	6	4	7	8

FIG. 15 – La réponse du programme au sudoku vide

5 Conclusion

Nous pouvons rappeler la problématique : peut-on créer un algorithme de résolution de sudoku n'ayant pas à formuler de choix, donc à effectuer de test sur les solutions trouvées. La réponse s'étale, malheuresement, en deux parties. Tout d'abord, avec les techniques de programmation linéaire étalées ci-dessus, il nous est possible de considérer le sudoku comme un problème linéaire en nombres entiers (binaires) et de le solutionner en utilisant, par exemple, les coupes de Gomory. Celles-ci nous autorisent à chaque itération du programme de nous rapprocher de plus en plus de la solution, jusqu'à l'atteindre et sans "tester" des solutions fausses.

Cependant, comme il a été noté plus haut, cette méthode ne permet pas une convergence rapide de l'algorithme vers la (ou en fait, une) solution. Il est couplé à une recherche arborescente, laquelle peut être amenée à explorer des voies non valides.

Enfin, on rappellera que le problème de la résolution d'un sudoku est reconnu comme ayant une complexité polynomiale. On peut, bien entendue, le résoudre en testant toutes les possibilités qui sont en nombre fini.

6 Bibliographie

- Sudoku - Article Wikipedia (http://en.wikipedia.org/wiki/Sudoku)
- Programmation Linéaire - Article Wikipedia (http://en.wikipedia.org/wiki/Linear_programming)
- Simplexe - Article Wikipedia (http://en.wikipedia.org/wiki/Simplex_algorithm)
- "Programmation Mathématiques, Théorie d'algorithmes 2eme ed.", Michel MINOUX Tech&Doc Lavoisier, 2008
- "Introduction à l'analyse numérique matricielle et à l'optimisation" P.G. CIARLET, 1994, Ed. Masson
- Librairie GLPK (GNU Linear Programming Kit), partie du projet GNU, librairie sous license GPL
- Documentation de la librairie GLPK
- "Generating Gomory's Cuts for linear integer programming problems : the HOW and WHY" - Présentation publiée sur le site Internet de l'Université de Melbourne - code 620-362
- "Solving Mixed Integer Linear Programs Using Branch and Cut Algorithm", Shon Albert, Faculty of North Carolina State University

7 Annexes

```cpp
/*****************************************************************************/
/**   SudSol                                                                **/
/**     File : MainHeader.hpp                                               **/
/**     -> Includes and global linking                                      **/
/*****************************************************************************/

/*****************************************************************************/
/**   Project's files                                                       **/
/**       -> MainHeader.hpp (this)                                          **/
/**       -> Sudoku.hpp                                                     **/
/**       -> Sudoku.cpp                                                     **/
/**       -> main.cpp                                                       **/
/*****************************************************************************/

#ifndef MAINHEADER_HPP_INCLUDED
#define MAINHEADER_HPP_INCLUDED

    // External includes
        #include <iostream> // Input/Output
        #include <sstream>  // Streams/Strings
        #include <fstream>  // File IO
        #include <glpk.h>   // GNU LINEAR PROGRAMMING KIT
        #include <time.h>   // Clock routines

    // Rnternal includes
        #include "./Sudoku.hpp"

    // Colors allowed only under Unix-like platforms
        #define RESET COLOR                 "\33[0;m"
        #define COL( color )                "\33[0;"##color##"m"
        #define COL ATT( color, attribute)  "\33["##attribute##";"##color##"m"

        // Attributes
            #define ResetCol                "\33[0;m"
            #define Bright                  "\33[1;m"
            #define Dim                     "\33[2;m"
            #define Underscore              "\33[4;m"
            #define Blink                   "\33[5;m"
            #define Reverse                 "\33[7;m"
            #define Hidden                  "\33[8;m"

        // ForeGround Colors
            #define Black                   "\33[0;30m"
            #define Red                     "\33[0;31m"
            #define Green                   "\33[0;32m"
            #define Yellow                  "\33[0;33m"
            #define Blue                    "\33[0;34m"
            #define Magenta                 "\33[0;35m"
            #define Cyan                    "\33[0;36m"
            #define White                   "\33[0;37m"

        // Background Colors
            #define B Black                 "\33[0;40m"
            #define B Red                   "\33[0;41m"
            #define B Green                 "\33[0;42m"
            #define B Yellow                "\33[0;43m"
            #define B Blue                  "\33[0;44m"
            #define B Magenta               "\33[0;45m"
            #define B Cyan                  "\33[0;46m"
            #define B White                 "\33[0;47m"

    /// TEMPLATE TOOLS
    template<typename TYPE>
        TYPE num from string(const char* str)
        {
            std::stringstream s(str);  //creating a stream
            TYPE num;
            s >> num;
            return num;
        }

#endif // MAINHEADER_HPP_INCLUDED

```

```cpp
/*****************************************************************************/
/**   SudSol                                                                **/
/**     File : Sudoku.hpp                                                   **/
/**     -> Sudoku's structure definition                                    **/
/*****************************************************************************/

#ifndef SUDOKU_HPP_INCLUDED
#define SUDOKU_HPP_INCLUDED

    //external includes
        #include <iostream>
        #include <sstream>
        #include <fstream>
        #include <glpk.h>

    //Constants
        const int BLOC_X[] = {0, 3, 6, 0, 3, 6, 0, 3, 6};
        const int BLOC_Y[] = {0, 0, 0, 3, 3, 3, 6, 6, 6};
         #define SSB_TRUE    2
         #define SSB_FALSE   0
         #define SSB_UNDEF   1

    //Objects
        struct SUDOKU
        {
            // Data
                char val[729];
                glp_prob* problem;

            // Functions (Constructors, Destructor)
                    SUDOKU(void);
                    SUDOKU(const char* filename);
                    ~SUDOKU(void);

                // Basic tools
                    int get_pos( int i, int j, int nbr);
                    int idsquare(int i,int j);
                    int get( int i, int j);
                    int getline( int j, int k);
                    int getcolumn( int i, int k);
                    int getblock( int b, int k);
                    int put( int i, int j, int c);
                    void clear(void);
                    bool complete(void);

                // IO tools
                    bool load(const char* filename);
                    bool load_from_string(const char* str);
                    void print(const char* filename);
                    void print_screen(void);
                    void raw_print_screen(void);
                    void valid(void);

                // Problem tools
                    void build_problem(bool verbose);
                    void retrieve_information(void);
                    void retrieve_information2(void);
        };

#endif // SUDOKU_HPP_INCLUDED
```

```cpp
/*****************************************************************************/
/**   SudSol                                                                **/
/**     File : Sudoku.cpp                                                   **/
/**     -> Sudoku's method code                                             **/
/*****************************************************************************/

#include "./Sudoku.hpp"
#include "./MainHeader.hpp" //colors

/// Functions
    // Constructor
        SUDOKU::SUDOKU(void)
        {
            clear();
            problem = NULL;
        }

    // Constructor : load from a file
        SUDOKU::SUDOKU(const char* filename)
        {
            clear();
            problem = NULL;
            load(filename);
            return ;
        }

    // Destructor
        SUDOKU::~SUDOKU(void)
        {
            if(problem!=NULL)
                glp_delete_prob(problem);
        }

    ///Basic Tools
        int SUDOKU::get_pos( int i, int j, int nbr)
        {
            ///WARNING nbr E [|1;9|]
            return i*81+j*9+(nbr-1); //OK
        }

        int SUDOKU::idsquare(int i,int j)
        {
            int ln = static_cast<int>( (float)(i)/3.0);
            int cl = static_cast<int>( (float)(j)/3.0);

            return cl+3*ln;
        }

        int SUDOKU::get( int i, int j)
        {
            ///WARNING nb E [|1;9|]
            int k;
            int m = 0;

            for( k=1; k<=9; k++)
            {
                if( val[ get_pos(i,j,k) ]==SSB_TRUE && m!=0 )
                    return -1; //ERROR
                if( val[ get_pos(i,j,k) ]==SSB_TRUE && m==0 )
                    m = k;
                if( val[ get_pos(i,j,k) ]==SSB_UNDEF )
                    return 0; //UNDEFINED|UNKNOWN
            }
            if(m!=0)
                return m;
            else
                return -1;
        }

        int SUDOKU::getline( int j, int k)
        {
            ///WARNING nb E [|1;9|]
            int i;

            for( i=0; i<9; i++)
            {
                if( val[ get_pos(i,j,k) ]==SSB_TRUE )
                    return i;
                if( val[ get_pos(i,j,k) ]==SSB_UNDEF )
                    return 0; // UNDEFINED|UNKNOWN
            }
            return -1; //error
        }

        int SUDOKU::getcolumn( int i, int k)
        {
            ///WARNING nb E [|1;9|]
            int j;

            for( j=0; j<9; j++)
            {
                if( val[ get_pos(i,j,k) ]==SSB_TRUE )
                    return j;
                if( val[ get_pos(i,j,k) ]==SSB_UNDEF )
                    return 0; // UNDEFINED|UNKNOWN
            }
            return -1; //error
        }

        int SUDOKU::getblock( int b, int k)
        {
            ///WARNING nb E [|1;9|]
            int  x = BLOC_X[b],
                 y = BLOC_Y[b];
            int  i,  j;

            for(  i=0;  i<3;  i++)
            {
                for(  j=0;  j<3;  j++)
                {
                    if( val[ get_pos( x+ i, y+ j,k) ]==SSB_TRUE )
                        return 1; // known
```

```
                    if( val[ get_pos(_x+_i,_y+_j,k) ]==SSB_UNDEF )
                        return 0; // UNDEFINED|UNKNOWN
                }
            }
            return -1; // error
        }

        int SUDOKU::put( int i, int j, int nb)
        {
            ///WARNING nb E [|1;9|]
            int k;

            if(nb<1 || nb>9) std::cout << "ERROR PUT" << std::endl;

            // erase all the elements from the case
            for(k=1; k<=9; k++)
                val[ get_pos(i,j,k) ]=SSB_FALSE;

            // erase all the elements from the line
            for(k=0; k<9; k++)
                val[ get_pos(i,k,nb) ]=SSB_FALSE;

            // erase all the elements from the column
            for(k=0; k<9; k++)
                val[ get pos(k,j,nb) ]=SSB FALSE;

            // erase all the elements from the block
            int x = static_cast<int>( ((float)i)/3.0 )*3;
            int y = static_cast<int>( ((float)j)/3.0 )*3;
            int i , j ;

            for( i =0; i <3; i ++)
            {
                for( j =0; j <3; j ++)
                    val[ get pos(x+i ,y+j ,nb) ]=SSB FALSE;
            }

            // Just set the element
            val[ get pos(i,j,nb) ]=SSB TRUE;

            return nb;
        }

        void SUDOKU::clear(void)
        {
            int i;

            for( i=0; i<729; i++)
            {
                val[i] = SSB UNDEF;
            }

            return ;
        }

        bool SUDOKU::complete(void)
        {
            int i,j;

            for( i=0; i<9; i++)
            {
                for( j=0; j<9; j++)
                {
                    char p = get(i,j);
                    if( p==-1 || p==0) //Error or undef
                        return false;
                }
            }

            return true;
        }

    ///IO Tools
        bool SUDOKU::load(const char* filename)
        {
            int i = 0,
                j = 0,
                l, length;
            std::fstream file;
            char *buffer, c;
            bool tst = true;

            clear();

            file.open(filename, std::fstream::in);

            if(!file.is open())
            {
                std::cout << Red << " -> Unable to read the input file : " << filename << ResetCol << std::endl;
                return false;
            }

            // get length of file:
            file.seekg (0, std::ios::end);
            length = file.tellg();
            file.seekg (0, std::ios::beg);

            // allocate memory:
            buffer = new char [length];

            // read data as a block:
            file.read (buffer,length);
            file.close();

            // read data
            for( l=0; l<length && tst; l++)
            {
                c = *(buffer+l);
                if( ('0'<=c && c<= '9') || c==' ' || c=='X' || c=='-' || c=='.')
                {
                    if('0'<c && c<= '9')
                        put(i, j, (c-'0'));
```

```
                    j++;
                    if(j>=9)
                    {
                        j = 0;
                        i++;
                        if( i>=9) tst = false; // we reached the end
                    }

                }
            }

            delete[] buffer;

            return true;
        }

        bool SUDOKU::load_from_string(const char* str)
        {
            /** WARNING : length(str)>81 **/
            int i = 0,
                j = 0,
                l;
            char c;
            bool tst = true;

            clear();

            // read data
            for( l=0; l<81 && tst; l++)
            {
                c = *(str+l);
                if( ('0'<=c && c<= '9') || c==' ' || c=='X' || c=='-' || c=='.')
                {
                    if('0'<c && c<= '9')
                        put(i, j, (c-'0'));
                    j++;
                    if(j>=9)
                    {
                        j = 0;
                        i++;
                        if( i>=9) tst = false; // we reached the end
                    }

                }
            }

            return true;
        }

        // Print on a File
        void SUDOKU::print(const char* filename)
        {
            int i, j;
            std::fstream file;

            file.open(filename, std::fstream::out|std::fstream::ate|std::fstream::app);

            if(!file.is_open())
            {
                std::cout << Red << " -> Unable to create the result file" << ResetCol << std::endl;
                return ;
            }

            for( i=0; i<9; i++)
            {
                std::cout << "  ";
                for( j=0; j<9; j++)
                {
                    char p = get(i,j);
                    if( p==0 ) // Undef
                        std::cout << Yellow << ' ' << ResetCol << ' ';
                    if( p==-1 ) // Error
                        std::cout << Red << 'X' << ResetCol << ' ';
                    if( p > 0 ) // a number
                        std::cout << Black << static_cast<int>(p) << ResetCol <<' ';
                    if(j==2 || j==5)
                        std::cout << Blue <<  "| " << ResetCol;
                }
                std::cout << std::endl;
                if(i==2 || i==5)
                    std::cout << Blue << "  - - - * - - - * - - - " << ResetCol << std::endl;
            }

            file << std::endl;
            file << std::endl;

            file.close();
            return ;
        }

        // Print on SCREEN
        void SUDOKU::print screen(void)
        {
            int i, j;

            std::cout << Blue << " -> Printing the Sudoku : " << ResetCol << std::endl;
            for( i=0; i<9; i++)
            {
                std::cout << "  ";
                for( j=0; j<9; j++)
                {
                    char p = get(i,j);
                    if( p==0 ) //Undef
                        std::cout << Yellow << ' ' << ResetCol << ' ';
                    if( p==-1 ) //Error
                        std::cout << Red << 'X' << ResetCol << ' ';
                    if( p > 0 ) //a number
                        std::cout << Black << static_cast<int>(p) << ResetCol <<' ';
                    if(j==2 || j==5)
                        std::cout << Blue <<  "| " << ResetCol;
                }
                std::cout << std::endl;
```

```
                if(i==2 || i==5)
                    std::cout << Blue << "  - - - * - - - * - - - " << ResetCol << std::endl;
            }

            return ;
        }

        // Layer printing
        void SUDOKU::raw_print_screen(void)
        {
            int i, j, k;

            std::cout << Blue << " -> -RAW- Printing the Sudoku : " << ResetCol << std::endl;
            for( k=1; k<=9; k++)
            {
                std::cout << "Layer n°" << k << std::endl;
                for( i=0; i<9; i++)
                {
                    std::cout << "  ";
                    for( j=0; j<9; j++)
                    {
                        char p = val[get pos(i, j, k)];
                        std::cout << Black;
                        switch(p)
                        {
                            case SSB TRUE : std::cout << "1";
                                            break;
                            case SSB FALSE : std::cout << "0";
                                            break;
                            case SSB UNDEF : std::cout << " ";
                                            break;
                        }
                        std::cout << ResetCol << ' ';
                        if(j==2 || j==5)
                            std::cout << Blue <<  "| " << ResetCol;
                    }
                    std::cout << std::endl;
                    if(i==2 || i==5)
                        std::cout << Blue << "  - - - * - - - * - - - " << ResetCol << std::endl;
                }
            }

            return ;
        }

        // Printing validation
        void SUDOKU::valid(void)
        {
            //Pretty Way to verify a sudoku
            int i, j, k;

            std::cout << " -> Verifying (TRUE,FALSE,UNDEF) : " << std::endl;
            for( i=0; i<9; i++)
            {
                for( j=0; j<9; j++)
                {
                    int tmp = 0;
                    std::cout << '(';

                    for(k=1; k<=9; k++)
                        if( val[ get pos(i, j, k) ]==SSB TRUE ) tmp++;
                    std::cout << tmp << ',';

                    tmp = 0;
                    for(k=1; k<=9; k++)
                        if( val[ get pos(i, j, k) ]==SSB FALSE ) tmp++;
                    std::cout << tmp << ',';

                    tmp = 0;
                    for(k=1; k<=9; k++)
                        if( val[ get pos(i, j, k) ]==SSB UNDEF ) tmp++;
                    std::cout << tmp << ") ";
                }
                std::cout << std::endl;
            }

            std::cout << std::endl;

            return ;
        }

    ///Problems Tools
        void SUDOKU::build problem(bool verbose)
        {
            /**
                Constraints
                V i,j E [|1;n|]² Sum( v[i,j, k], k, 1, n ) = 1 //only one number per case
                V k,j E [|1;n|]² Sum( v[i,j, k], k, 1, n ) = 1 //only one number per line
                V k,i E [|1;n|]² Sum( v[i,j, k], k, 1, n ) = 1 //only one number per column
            **/

            /**
                CONSTRAINTS MATRIX
                Initialzing 3 tables of numbers
                First and Second represent coordinates of the value
                Third represent the coefficient

                Number of coefficients : 729*4 = 2916 (each var appears 3 times)

                Ai     -> coordinate 1
                Aj     -> coordinate 2
                Acoeff -> coefficient in the constraints' matrix
            **/

            //Methods & Macro
                #define VERBOSE               if(verbose)   //To provide output on screen during the processing
                //#define AFFINFO
                #define SEARCHMODE DB    //Choice between inequalities and equalities
                //#define SEARCHMODE FX  //
                #ifdef SEARCHMODE_DB
                    VERBOSE std::cout << " -> Using Double Bounded Method" << std::endl;
                #else
```

```
                    #ifdef SEARCHMODE_FX
                        VERBOSE std::cout << " -> Using Fixed Method" << std::endl;
                    #else
                        VERBOSE std::cout << " -> ERROR : NO SEARCHMODE DEFINED" << std::endl;
                    #endif
                #endif

            /**
                Ai -> index of constraint
                Aj -> index of var
            **/
            int Ai[2916], Aj[2916];
            double Acoeff[2916];

            /**
                IndGP -> get_pos -> index of var
            **/
            int IndGP[729];

            // declare all variables
            // first we create an empty problem
            problem = glp create prob();

            // then we give him a name
            glp set prob name(problem, "SudoKu Problem Binary");

            // set MAXIMIZE  for the objective function
            glp set obj dir(problem, GLP MAX);

            ///1 - Find the number of unknows and constraints
            int nvar = 0,
                ncns = 0;
            int IndMat = 1; // number of point in the matrix

            { /// BUILDING BLOC
                int i, j, k, b;

                /// Case Constraints
                for(j=0; j<9; j++)
                {
                    for(i=0; i<9; i++)
                    {
                        if( get(i, j)==0 ) //Case undefined
                        {
                            ncns++;
                            ///Create the constraint
                                glp add rows(problem, 1);
                                std::stringstream s;       //creating a stream / string
                                s << "C0";
                                if( ncns<10 )  s << '0';
                                s << ncns << " case" ;     //set it to the name of the case
                                #ifdef AFFINFO
                                    std::cout << "        -> New Constraint : " << s.str() << std::endl;
                                #endif
                                glp set row name(problem, ncns, (s.str()).c str() );  // set the name of a slack variables
                                #ifdef SEARCHMODE DB
                                    lpx set row bnds(problem, ncns, LPX DB, 0.0, 1.0); // set the value to 0.0<=X<=1.0
                                #else
                                    #ifdef SEARCHMODE FX
                                        lpx set row bnds(problem, ncns, LPX FX, 1.0, 1.0); // set the value to X=1.0
                                    #endif
                                #endif

                            for(k=1; k<=9; k++) // +variables
                            {
                                if( val[ get pos(i, j, k) ] == SSB UNDEF ) //Value undefined
                                {
                                    nvar++;
                                    ///create the var
                                        glp add cols(problem, 1);
                                        std::stringstream s;  // creating a stream / string
                                        s << 'X';
                                        if( nvar<10 ) s << '0';
                                        if( nvar<100 ) s << '0';
                                        s << nvar << ' ' << i << '-' << j << ' ' << k; //set it to the name of the case
                                        #ifdef AFFINFO
                                            std::cout << "        -> New Unknown : " << s.str() << std::endl;
                                        #endif
                                        glp set col name(problem, nvar, (s.str()).c str()); // set the name of the variable
                                        glp set obj coef(problem, nvar, 1.0);               // set the coefficient in the
objective function
                                        glp set col kind(problem, nvar, GLP BV); /// Variables are binary
                                    /// Matrix update
                                        IndGP[get pos(i, j, k)] = nvar;
                                    /// Update 2
                                        Ai[IndMat] = ncns;
                                        Aj[IndMat] = nvar;
                                        Acoeff[IndMat] = 1.0;
                                        IndMat++;
                                }
                            }
                        }
                    }
                }

                /// Line Constraints
                for(j=0; j<9; j++)
                {
                    for(k=1; k<=9; k++)
                    {
                        if( getline(j, k)==0 ) //line undefined
                        {
                            ncns++;
                            ///Create the constraint
                                glp add rows(problem, 1);
                                std::stringstream s;       //creating a stream / string
                                s << "C";
                                if( ncns<100 )  s << '0';
                                s << ncns << " line" ;   //set it to the name of the case
                                #ifdef AFFINFO
                                    std::cout << "        -> New Constraint : " << s.str() << std::endl;
                                #endif
```

```
                                glp_set_row_name(problem, ncns, (s.str()).c_str() );  // set the name of a slack variables
                                #ifdef SEARCHMODE DB
                                    lpx set row bnds(problem, ncns, LPX DB, 0.0, 1.0); // set the value to 0.0<=X<=1.0
                                #else
                                    #ifdef SEARCHMODE_FX
                                        lpx set row bnds(problem, ncns, LPX FX, 1.0, 1.0); // set the value to X=1.0
                                    #endif
                                #endif ///A REVOIR
                            ///Update
                                for(i=0; i<9; i++)
                                {
                                    if( val[ get pos(i, j, k) ] == SSB UNDEF ) //Value undefined
                                    {
                                        Ai[IndMat] = ncns;
                                        Aj[IndMat] = IndGP[ get pos(i, j, k) ];
                                        Acoeff[IndMat] = 1.0;
                                        IndMat++;
                                    }
                                }
                        }
                    }
                }

                /// Column Constraints
                for(i=0; i<9; i++)
                {
                    for(k=1; k<=9; k++)
                    {
                        if( getcolumn(i, k)==0 ) //column undefined
                        {
                            ncns++;
                            ///Create the constraint
                                glp add rows(problem, 1);
                                std::stringstream s;       //creating a stream / string
                                s << "C";
                                s << ncns << " column" ;  //set it to the name of the case
                                #ifdef AFFINFO
                                    std::cout << "        -> New Constraint : " << s.str() << std::endl;
                                #endif
                                glp set row name(problem, ncns, (s.str()).c str() );  // set the name of a slack variables
                                #ifdef SEARCHMODE DB
                                    lpx set row bnds(problem, ncns, LPX DB, 0.0, 1.0); // set the value to 0.0<=X<=1.0
                                #else
                                    #ifdef SEARCHMODE FX
                                        lpx set row bnds(problem, ncns, LPX FX, 1.0, 1.0); // set the value to X=1.0
                                    #endif
                                #endif
                            ///Update
                                for(j=0; j<9; j++)
                                {
                                    if( val[ get pos(i, j, k) ] == SSB UNDEF ) //Value undefined
                                    {
                                        Ai[IndMat] = ncns;
                                        Aj[IndMat] = IndGP[ get pos(i, j, k) ];
                                        Acoeff[IndMat] = 1.0;
                                        IndMat++;
                                    }
                                }
                        }
                    }
                }

                /// Bloc Constraints
                for(b=0; b<9; b++)
                {
                    for(k=1; k<=9; k++)
                    {
                        if( getblock(b, k)==0 )
                        {
                            ncns++;
                            ///Create the constraint
                                glp add rows(problem, 1);
                                std::stringstream s;       //creating a stream / string
                                s << "C";
                                s << ncns << " block" ;    //set it to the name of the case
                                #ifdef AFFINFO
                                    std::cout << "        -> New Constraint : " << s.str() << std::endl;
                                #endif
                                glp set row name(problem, ncns, (s.str()).c str() );  // set the name of a slack variables
                                #ifdef SEARCHMODE DB
                                    lpx set row bnds(problem, ncns, LPX DB, 0.0, 1.0); // set the value to 0.0<=X<=1.0
                                #else
                                    #ifdef SEARCHMODE FX
                                        lpx set row bnds(problem, ncns, LPX FX, 1.0, 1.0); // set the value to X=1.0
                                    #endif
                                #endif
                            ///Update
                                int  i,  j;
                                int  x = BLOC X[b],
                                     y = BLOC Y[b];
                                for( i=0;  i<3;  i++)
                                {
                                    for( j=0;  j<3;  j++)
                                    {
                                        if( val[ get pos( x+ i,  y+ j, k) ] == SSB UNDEF ) //Value undefined
                                        {
                                            Ai[IndMat] = ncns;
                                            Aj[IndMat] = IndGP[ get pos( x+ i,  y+ j, k) ];
                                            Acoeff[IndMat] = 1.0;
                                            IndMat++;
                                        }
                                    }
                                }
                        }
                    }
                }

                VERBOSE std::cout << "    -> There are " << nvar << " variables" << std::endl;
                VERBOSE std::cout << "    -> There are " << ncns << " constraints" << std::endl;
                VERBOSE std::cout << "    -> There are " << IndMat << " links in the matrix" << std::endl;
            }
```

```
672             /// Load / Give to GLPK
673             VERBOSE std::cout << " -> Loading Constraints Matrix" << std::endl;
674             glp_load_matrix(problem, IndMat-1, Ai, Aj, Acoeff);
675 
676             VERBOSE std::cout << " -> Problem built" << std::endl;
677             return ;
678         }
679 
680     /// After solving, we must read the information returned by GLPK
681     /// /!\ DEPRECATED
682         void SUDOKU::retrieve_information(void)
683         {
684             int i,j,k, ivar = 1;
685 
686             std::cout << " -> Retreiving information" << std::endl;
687 
688             //loops
689             for(j=0; j<9; j++)
690             {
691                 for(i=0; i<9; i++)
692                 {
693                     if( get(i,j)==0 ) // unknown case
694                     {
695                         std::cout << "    -> At " << i << '-' << j << std::endl;
696                         for(k=1; k<=9; k++)
697                         {
698                             if( val[ get_pos( i, j, k) ] == SSB_UNDEF ) //is unknown
699                             {
700                                 int res = static_cast<int>( lpx_get_col_prim(problem, ivar) );  // get the value found
701                                 std::cout << "         -> Val : " << k << " ~ res : " << res << std::endl;
702                                 if( res>0 )
703                                     val[ get_pos( i, j, k) ] = SSB_TRUE;
704                                 else
705                                     val[ get_pos( i, j, k) ] = SSB_FALSE;
706                                 ivar++;
707                             }
708 
709                         }
710                     }
711                 }
712             }
713             return ;
714         }
715 
716     /// After solving, we must read the information returned by GLPK
717         void SUDOKU::retrieve_information2(void)
718         {
719             int i,j,k, ivar = 1;
720 
721             #ifdef AFFINFO
722                 std::cout << " -> Retreiving information (Objective Function : " << glp_mip_obj_val(problem) << ")" << std::
    endl;
723             #endif
724 
725             //loops
726             for(j=0; j<9; j++)
727             {
728                 for(i=0; i<9; i++)
729                 {
730                     if( get(i,j)==0 ) // case inconnue
731                     {
732                         #ifdef AFFINFO
733                             std::cout << "    -> At " << i << '-' << j << std::endl;
734                         #endif
735                         for(k=1; k<=9; k++)
736                         {
737                             if( val[ get_pos( i, j, k) ] == SSB_UNDEF ) //is unknown
738                             {
739                                 double res = glp_mip_col_val(problem, ivar); // get the value found
740                                 #ifdef AFFINFO
741                                     std::cout << "         -> Val : " << k << " ~ res : " << res << std::endl;
742                                 #endif
743                                 if( res>0.5)
744                                     val[ get_pos( i, j, k) ] = SSB_TRUE;
745                                 else
746                                     val[ get_pos( i, j, k) ] = SSB_FALSE;
747                                 ivar++;
748                             }
749                         }
750                     }
751                 }
752             }
753             return ;
754         }
755 
756 
```

```cpp
/*****************************************************************************/
/**   SudSol                                                                **/
/**     File : main.cpp                                                     **/
/**     -> Main funcion                                                     **/
/*****************************************************************************/

#include "./MainHeader.hpp"

/** Tools to process a list of Sudoku **/
bool process continue(const char* filename, int beg, int l);

int main(int argc, char *argv[])
{
    std::cout << "SudSol <Binary Version>" << std::endl;

    //Read argument given to the program
    if( argc==2 )
    {
        std::cout << " -> Loading Sudoku File : " << argv[1] << "..." << std::endl;

        SUDOKU* s = new SUDOKU();

        if( s->load(argv[1]) )
        {
            // Print the Sudokus loaded
            s->print screen();

            // Build the problem
            std::cout << " -> Building the problem..." << std::endl;
            s->build problem(true);
            std::cout << " -> Solving the problem..." << std::endl;
            std::cout << Cyan << "[Lib GLPK] say :" << ResetCol << std::endl;
            /** SOLVING PART **/
                time_t start,end;
                double dif s;
                time (&start);
                int code = lpx intopt(s->problem);
                time (&end);
                dif s = difftime (end,start);
            /** END **/
            std::cout << Cyan << "said [Lib GLPK]" << ResetCol << std::endl;
            std::cout << " -> Solving time : " << dif s << " second(s)" <<  std::endl;
            std::cout << " -> IntOpt Returned : " << std::endl;
            switch(code)
            {
                //Successes
                case LPX E OK:
                    std::cout << Green << "      OPTIMAL SOLUTION" << ResetCol << std::endl;
                    break;
                case LPX FEAS:
                    std::cout << "FEASIBLE SOLUTION" << std::endl;
                    break;
                //Problems :
                case LPX INFEAS:
                    std::cout << "INFEASIBLE PROBLEM" << std::endl;
                    break;
                case LPX NOFEAS:
                    std::cout << "NO FEASIBLE SOLUTION" << std::endl;
                    break;
                case LPX UNBND:
                    std::cout << "UNBOUNDED SOLUTION" << std::endl;
                    break;
                case LPX UNDEF:
                    std::cout << "UNDEFINED SOLUTION" << std::endl;
                    break;
                default:
                    std::cout << "NO CODE" << std::endl;
                    break;*/
            }
            std::cout << " -> Writing solution..." << std::endl;
            // GLPK write a text file with a description of the LP
            lpx write cpxlp(s->problem, "SUDSOL SEARCHERROR.lp");
            // Use the solution to build the entire Sudoku
            s->retrieve information2();
            // Prnt it on screen
            s->print screen();
            if( !(s->complete()) ) //Verification
            {
                std::cout << Red << " -> ERROR : Bad Solution..." << ResetCol << std::endl;
                s->raw print screen();
            }
            std::cout << " -> Done." << std::endl;

        }

        delete s;
    }
    if( argc==1 || argc==3 )
        std::cout << Red << " -> ERROR : You must give a Sudoku (or a list and an index) to this program" << ResetCol << std::endl;
    if( argc==4 )
    {
        std::cout << Red << "TEST" << ResetCol << "/ From : " << num from string<int>(argv[2]) << ", Length : " << num from string<int>(argv[3]) << std::endl;
        process continue(argv[1], num from string<int>(argv[2]), num from string<int>(argv[3]));
        std::cout << Red << "END TEST" << ResetCol << std::endl;
    }
    if( argc>4 )
        std::cout << Red << " -> ERROR : There are too many argument [only one or two] required]" << ResetCol << std::endl;

    std::cout << Yellow << " -> Press Enter to quit..." << ResetCol << std::endl;
    return 0;
}

/** Another little tool **/
bool process_continue(const char* filename, int beg, int l)
{
    /**
    Format :
        Sudokus must begin at the first character of the line
    **/
```

```
111         std::fstream file;
112         time_t start,end;
113         double dif s;
114         double average = 0;
115         char *buffer;
116         int index = 0,
117             failure = 0;
118 
119         // Load the file that contains the Sudokus
120         file.open(filename, std::fstream::in);
121 
122         if(!file.is open())
123         {
124             std::cout << Red << " -> Unable to read the input file : " << filename << ResetCol << std::endl;
125             return false;
126         }
127 
128         // go to the beginning of the file
129         file.seekg (0, std::ios::beg);
130 
131         // allocate memory:
132         buffer = new char [100];
133 
134         // stop output mode of GLPK
135         glp term out(GLP OFF);
136 
137         // read data as a block:
138         std::cout << " -> Processing... (Beginning : " << beg << ", Length : " << l << ')' << std::endl;
139         while( file.getline(buffer, 100) )
140         {
141             if( (index>=beg) && (index<beg+l) )
142             {
143                 // process on the line
144                 SUDOKU* sud = new SUDOKU();
145                 if( sud->load from string(buffer) )
146                 {
147                     sud->build problem(false);
148                     /** SOLVING PART **/
149                         time (&start);
150                         int code = lpx intopt(sud->problem);
151                         time (&end);
152                         dif s = difftime (end,start);
153                         average += dif s;
154                     /** END **/
155                     if( code==LPX E OK )
156                     {
157                         sud->retrieve information2();
158                         if( sud->complete() ) // the sudokju was entirely solved
159                             std::cout << Blue << "    -> Sudoku n°" << index << " - Optimal Solution Found [" << dif s << "
    second(s)]" ResetCol << std::endl;
160                         else
161                         {
162                             failure++;
163                             std::cout << Red << "    -> Sudoku n°" << index << " - Optimal Solution Found but NOT COMPLETE [" <<
    dif s << " second(s)]" ResetCol << std::endl;
164                         }
165                     }
166                     else
167                     {
168                         std::cout << Red << "    -> Sudoku n°" << index << " - Optimal Solution NOT Found [Code :" << code << "] ["
     << dif s << " second(s)]" << ResetCol << std::endl;
169                         failure++;
170                     }
171                 }
172                 else
173                     std::cout << Yellow << "Unable to read entry n°" << index << std::endl;
174                 delete sud;
175             }
176             index++;
177         }
178         std::cout << " -> Done" << std::endl;
179         std::cout << " -> Failures : " << failure << std::endl;
180         std::cout << " -> Average solving time : " << average/static_cast<double>(l) << std::endl;
181 
182         //restart output mode of GLPK
183         glp term out(GLP ON);
184 
185         file.close();
186         delete buffer;
187 
188         return true;
189     }
190 
```

(Exemple de Résolution de 50 sudoku parmi les plus difficiles)

```
SudSol <Binary Version>
TEST/ From : 0, Length : 50
 -> Processing... (Beginning : 0, Length : 50)
    -> Sudoku n°0 - Optimal Solution Found [0 second(s)]
    -> Sudoku n°1 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°2 - Optimal Solution Found [6 second(s)]
    -> Sudoku n°3 - Optimal Solution Found [2 second(s)]
    -> Sudoku n°4 - Optimal Solution Found [0 second(s)]
    -> Sudoku n°5 - Optimal Solution Found [2 second(s)]
    -> Sudoku n°6 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°7 - Optimal Solution Found [2 second(s)]
    -> Sudoku n°8 - Optimal Solution Found [3 second(s)]
    -> Sudoku n°9 - Optimal Solution Found [2 second(s)]
    -> Sudoku n°10 - Optimal Solution Found [3 second(s)]
    -> Sudoku n°11 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°12 - Optimal Solution Found [4 second(s)]
    -> Sudoku n°13 - Optimal Solution Found [4 second(s)]
    -> Sudoku n°14 - Optimal Solution Found [3 second(s)]
    -> Sudoku n°15 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°16 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°17 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°18 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°19 - Optimal Solution Found [5 second(s)]
    -> Sudoku n°20 - Optimal Solution Found [3 second(s)]
    -> Sudoku n°21 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°22 - Optimal Solution Found [3 second(s)]
    -> Sudoku n°23 - Optimal Solution Found [8 second(s)]
    -> Sudoku n°24 - Optimal Solution Found [2 second(s)]
    -> Sudoku n°25 - Optimal Solution Found [4 second(s)]
    -> Sudoku n°26 - Optimal Solution Found [3 second(s)]
    -> Sudoku n°27 - Optimal Solution Found [4 second(s)]
    -> Sudoku n°28 - Optimal Solution Found [4 second(s)]
    -> Sudoku n°29 - Optimal Solution Found [3 second(s)]
    -> Sudoku n°30 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°31 - Optimal Solution Found [5 second(s)]
    -> Sudoku n°32 - Optimal Solution Found [5 second(s)]
    -> Sudoku n°33 - Optimal Solution Found [3 second(s)]
    -> Sudoku n°34 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°35 - Optimal Solution Found [5 second(s)]
    -> Sudoku n°36 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°37 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°38 - Optimal Solution Found [3 second(s)]
    -> Sudoku n°39 - Optimal Solution Found [3 second(s)]
    -> Sudoku n°40 - Optimal Solution Found [3 second(s)]
    -> Sudoku n°41 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°42 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°43 - Optimal Solution Found [1 second(s)]
    -> Sudoku n°44 - Optimal Solution Found [2 second(s)]
    -> Sudoku n°45 - Optimal Solution Found [0 second(s)]
    -> Sudoku n°46 - Optimal Solution Found [2 second(s)]
    -> Sudoku n°47 - Optimal Solution Found [0 second(s)]
    -> Sudoku n°48 - Optimal Solution Found [0 second(s)]
    -> Sudoku n°49 - Optimal Solution Found [2 second(s)]
 -> Done
 -> Failures : 0
 -> Average solving time : 2.36
END TEST
```