MÉMOIRE DE STAGE

Résolution de Sudoku

Minatchy Jérôme
M1 Informatique
Année universitaire 2020/2021 à rendre pour le 3 Juin 2021

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

Université des Antilles
(Entreprise d'accueil)

Table des figures

Table des matières

Chapitre 1

Introduction

1.1 Présentation de la structure d'accueil

Durant la période de mon stage , j'ai été accueilli au **Laboratoire de Mathématiques Informatique et Application (LAMIA)** de l'Université des Antilles (UA).

Pour présenter cette structure, il me faut tout d'abord présenter l'université à laquelle il est rattaché.

1.1.1 L'université des Antilles

Bien que ce soit l'université dans laquelle j'ai fait toutes mes études, voici quelques chiffres que je ne connaissais pas et qui donnent la mesure de sa taille :

L'Université des Antilles s'organise autour deux pôles universitaires régionaux autonomes : le « Pôle Guadeloupe » et le « Pôle Martinique ».

Sur ces pôles, l'Université assure des missions d'*enseignement* et de *recherche*, assistées par des *administratifs et des techniciens.*

Administration et personnel technique

l'UA emploie 414 Administratifs et Techniciens (environ 200 personnes pour l'administation centrale et 100 répartis sur chaque pôle)

Enseignements

L'UA délivre des diplomes de la licence au doctorat dans de nombreux domaines. Au total, cela représente :

— 484 enseignants-chercheurs (environ 240 pour chaque pôle)

— 12 000 étudiants (environ 7000 pour la Guadeloupe , 5000 pour la Martinique)

Pour l'informatique, cela représente : - autour de 20 enseignants-chercheurs - autour de 120 étudiants

1.1.2 Le LAMIA

Le **Laboratoire de Mathématiques Informatique et Application (LAMIA)**, comme son nom l'indique, se concentre sur les recherches en informatiques et mathématiques.

Il compte une soixantaine de membres (Professeurs des Universités, Maitres de Conférences, ATER, Doctorants) répartis sur deux pôles (Guadeloupe et Martinique) au sein de trois équipes internes :

— Equipe **Mathématiques** (analyse variationnelle, analyse numérique, EDP, analyse statistique, mathématiques discrètes) ;

— Equipe Informatique **DANAIS** : Data analytics and big data gathering with sensors ;

— Equipe Informatique **AID** : Apprentissages Interactions Donnees ;

De plus, le LAMIA accueille en son sein un groupe de chercheurs associés travaillant en Epidémiologie clinique et médecine.

1.2 Contexte général

1.2.1 Qu'est ce que le sudoku

Le sudoku est un jeu représenté par une grille de 81 cases découpées en 9 lignes et 9 colonnes et 9 sous grilles 3 par 3. Le but du jeu est de remplir chaque ligne avec 9 chiffres allant de 1 à 9 en faisant en sorte qu'il n'y ai pas le même chiffre plusieurs fois sur la même ligne colonne ou dans la même sous grille.

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

FIGURE 1.1 – Exemple de Sudoku complet.

1.3 Contexte du problème

La résolution de sudoku est un sujet où plusieurs solutions existent et où c'est dans la complexité[1] et le temps de calcul des différentes solution que réside la difficulté. Nous pouvons aussi rendre compte de résolution de sudoku avec des règles spécifiques.
tel que :

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

FIGURE 1.2 – Sudoku où l'on applique la règle de l'unicité des chiffres sur les diagonales.

Dans ce mémoire nous allons éssentiellement parler de deux d'entre elles celle de la résolution de sudoku vu sous l'angle d'un problème d'optimisation linéaire grace à l'algorithme du simplex et une autre plus simple celle de l'algorithme du backtraking.

1. le nombre d'action réalisé durant la résolution

1.4 Méthodologie

1.4.1 Outils utilisés

Présentation de Python

FIGURE 1.3 – Logo de Python.

Python est un langage de programmation interprété [2] qui sera utilisé pour l'ensemble du projet. Nous avons choisi ce langage car il est très propice aux méthodes utilisées et contient toutes les librairies nécéssaires.

Présentation Qt

FIGURE 1.4 – Logo de Qt.

QT est une librairie [3] qui permet la création d'interface graphique en Python.Que nous emploierons pour créer l'interface graphique que l'on utilisera au cours du projet.

Présentation Cplex

FIGURE 1.5 – Logo de Cplex.

Cplex est une librairie[3] qui permet la modélisation et la résolution de problème d'optimisation linéaire [4].

2. Langage nécéssitant un programme informatique qui joue le rôle d'interface entre le projet et le processeur appellé interpréteur, pour exécuter du code.
3. Une librairie est un fichier contenant du code (généralement un ensemble de fonction et classes permettant de faciliter et/ou de réaliser certains programmes)
4. Terme que j'expliciterai plus tard dans mon rapport

Présentation de GitHub

FIGURE 1.6 – Logo de GitHub.

Nous pouvons définir GitHub comme une plateforme de développement de projet informatique en groupe. Elle simplifie grandement le développement de projets. Elle permet de versioner ses programmes et d'y apporter des modifications en temps réel à plusieurs. Ce rapport ainsi que le code généré durant le projet en plus de ce trouver en annexe, se trouverons aussi sur github à ces addresses :
Rapport de Stage : `https://github.com/MrMinatchy2/Rapport-de-stage-2021`
Code généré : `https://github.com/MrMinatchy2/Projet-de-Stage`

Présentation de LaTex

FIGURE 1.7 – Logo de LaTex

Nous pouvons dire que LaTex est un langage de traitement de texte tel que le markdown qui permet de mettre en forme notre texte de manière s̈cientifiquec̈ela veut dire que. LaTex permet une faciliter d'écriture des équations et de toutes les écriture mathématiques.Permet de par ses nombreux package une quasi-infinité de possibilitées. L'utilisation de cet outil permettra une synergie entre ceux-ci car LaTex peut-être utiliser avec un simple bloc-note c'est donc du texte ce qui permet une intéraction facilitée avec GitHub d'ailleurs ce rapport est écrit avec Latex et retrouvable sur GitHub.

1.5 Annonce du plan

1.5.1 Présentation des stratégies de résolution

Dans cette première partie je commencerais par vous présenter la première solution de résolution choisie qui est la résolution du sudoku en tant que problème d'optimisation linéaire.
En deuxième grande sous partie de cette section je vous présenterai ce qu'est le backtraking.
En dernière sous partie de cette présentation je vous expliquerai pourquoi le choix de ces deux solutions.

1.5.2 Implémentation des stratégies

La première sous partie de cette grande sous partie commencera avec la modélisation et l'implémentation d'un sudoku en python et l'implémentation de l'interface graphique.
La seconde sera l'implémentation de la méthode résolution utilisant cplex.
Pour finir nous ferons l'implémentation de la méthode utilisant l'algorithme du backtracking.

1.5.3 Test du résolveur

Nous commencerons par établir nos méthodes de test et expliquer la raison du choix de ces test.
Nous continuerons avec l'implémentation de ceux-ci.
Nous finirons par présenter et analyser nos résultats.

1.5.4 Conclusion

Nous terminerons ce rapport par une conclusion où nous rappelerons brièvement la problématique.
Nous ferons le bilan des produits du projet de stage.
Et ferons le bilan des apports du stage.

Chapitre 2

Déroulement

Ce chapitre, le plus volumineux du rapport, décrira l'ensemble des tâches que j'ai eu à effectuer au cours de ces deux mois.

2.1 Présentation des stratégies de résolution

2.1.1 Présentation de la résolution avec Cplex

Qu'est-ce qu'un problème d'optimisation linéaire

Nous allons dans cette première partie parler de la résolution de sudoku comme étant un problème d'optimisation linéaire. Mais tout d'abord nous devons définir ce qu'est un problème d'optimisation linéaire.

Par définition un problème d'optimisation linéaire est un problème dont la valeur à obtimiser ainsi que les contraintes qui y seront appliquées peuvent-êtres modélisées sous la forme d'une fonction linéaire cette description n'est pas des plus précises que l'on puisse donner, c'est pour cela que nous allons l'étoffer par un exemple de problème relativement connu.

Le problème du brasseur de bière peut être énoncer comme suit :

Un brasseur fabrique 2 types de bières : blonde et brune.
3 ingrédients : maïs , houblon , malt.
Quantités requises par unité de volume :
Bière blonde : 2,5 kg de maïs, 125 g de houblon, 17,5 kg de malt
Bière brune : 7,5 kg de maïs, 125 g de houblon, 10 kg de malt
Le brasseur dispose de 240 kg maïs, 5 kg houblon , 595 kg malt
Prix vente par u.v. : blonde 9 euros , brune 15 euros Le brasseur veut maximiser son revenu .
Quelle quantité de bières blondes et/ou brunes doit-il produire pour cela ?

Nous pouvons modéliser le revenu du brasseur comme étant une fonction linéaire tel que :
Soit x^1 le nombre de volume d'unité de bière blonde et x^2 le nombre d'unité de volume de bière brune
Nous avons :

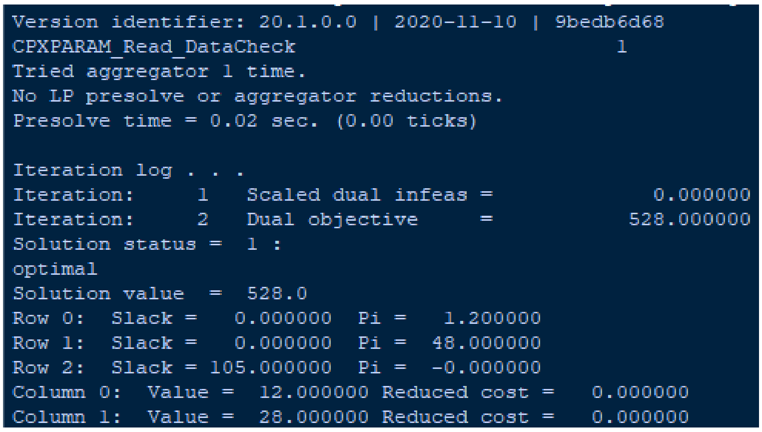
Le revenu que nous devons maximiser : $R = x^1 * 9 + x^2 * 15$

Les contraintes peuvent êtres représentées comme suit :

La quantité maximale de maïs : $M \geq x^1 * 2,5 + x^2 * 7,5$
La quantité maximale de houblon : $H \geq x^1 * 0,125 + x^2 * 0,125$
La quantité maximale de malt : $Ma \geq x^1 * 17,5 + x^2 * 10$

Comment résoudre un problème d'optimisation linéaire

Maintenant que nous avons vu ce qu'est un problème d'optimisation linéaire et illustré ceci par un exemple nous allons voir sa solution avec Cplex.
La résolution avec cplex est des plus simples il suffit de mettre en place nos contraintes et notre fonction à maximiser le code sera donner en annexe :



```
Version identifier: 20.1.0.0 | 2020-11-10 | 9bedb6d68
CPXPARAM_Read_DataCheck                          1
Tried aggregator 1 time.
No LP presolve or aggregator reductions.
Presolve time = 0.02 sec. (0.00 ticks)

Iteration log . . .
Iteration:     1   Scaled dual infeas =            0.000000
Iteration:     2   Dual objective     =          528.000000
Solution status =  1 :
optimal
Solution value  =  528.0
Row 0:  Slack =   0.000000  Pi =   1.200000
Row 1:  Slack =   0.000000  Pi =  48.000000
Row 2:  Slack = 105.000000  Pi =  -0.000000
Column 0:  Value =  12.000000 Reduced cost =   0.000000
Column 1:  Value =  28.000000 Reduced cost =   0.000000
```

FIGURE 2.1 – Résultat de l'optimisation du problème du brasseur

Nous voyons que nous avons pour revenu maximum 528 euro avec 12 unités de volume de bière blonde produite et 28 unités de volume de bière brune produites.

2.1.2 Présentation de la résolution par backtracking

L'algorithme du backtracking est plus précisément une famille d'algorithme qui sont utilisés le plus souvent pour résoudre des problèmes de satisfaction de contrainte. Le backtracking est la solution la plus simple à comprendre il s'agit tout simplement de tester toutes les valeurs possible sur chaque case jusqu'a obtenir une sortie de sudoku remplie et valide.Cette algorithme agit comme un parcours d'arbre, illustrons cela avec une image :

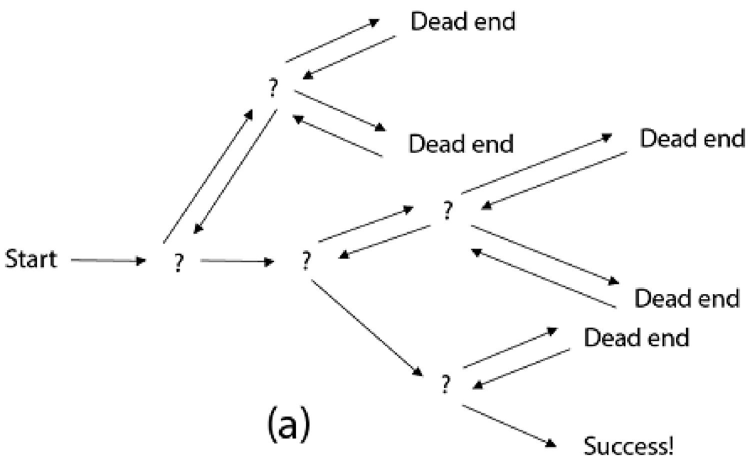


FIGURE 2.2 – Illustration de l'algorithme du backtracking.

Comme l'illustre cette image nous essayons toutes les possibilitées et retournons en arrière si nous tombons dans une impasse dans notre cas, nous passons de case vide [1] en case vide[1] en revenant en arrière si peu importe le chiffre essayer dans cette case allant de 1 à 9, l'ajout de celui-ci donne un sudoku invalide. Le problème de cet algorithme qui reviens le plus souvent est dù au fait que nous essayons toutes les valeurs possible ce qui nous donne une très grande quantité d'action à éffectué et donc ce qui augmente considérablement notre temps de calcul nous pouvons toutefois régler ce problème en évitant de tester les possibilitées qui nous amène de façon logique à une impasse ou éviter de tester des possibilitées d'une case si il est possible de déduire sa valeur de façon logique.

2.1.3 Pourquoi utiliser ces deux algorithme

Pourquoi utiliser l'algorithme du simplex

La résolution avec l'algorithme du simplex permet une implementation intuitive et facilité dù à l'utilisation de Cplex. Cplex nous permet aussi une meilleur étude de la complexité de notre algorithme car cous pouvons voir les différentes itération de recherche de notre outil de résolution. Cette méthode de résolution nous permet aussi de revoir le problème de plusieurs façons différentes de par le fait que la résolution ne dépends que de nos contraintes.

Pourquoi utiliser l'algorithme utilisant le backtracking

La résolution avec backtracking est l'algorithme le plus naturel qui viennent à l'esprit dans la résolution de problème avec contrainte ce qui permet une implémentation de la gestion des contraintes facilité malgré leur nombre.C'est un algorithme facilement améliorable car le but tout au long de son implémentation sera de tester le moins de valeur menant à une impasse possible. Cette algorithme a aussi pour avantage de n'avoir besoin d'aucune librairie extérieur ce qui permet une liberté totale au niveau du code et donne lieu à une certaine transparence quant à son fonctionnement contrairement au code que nous utilisons via cplex qui peut différé légèrement de son fonctionnement théorique.

1. Une case vide est comme dit plus haut une case contenant la valeur 0

2.1.4 Implémentation des Stratégies de résolution

2.1.5 Modélisation et implémentation d'un sudoku en Python et de l'interface graphique

Présentation de l'interface graphique

Nous allons tout d'abord vous présenter notre interface graphique qui nous servira tout au long de nos test :

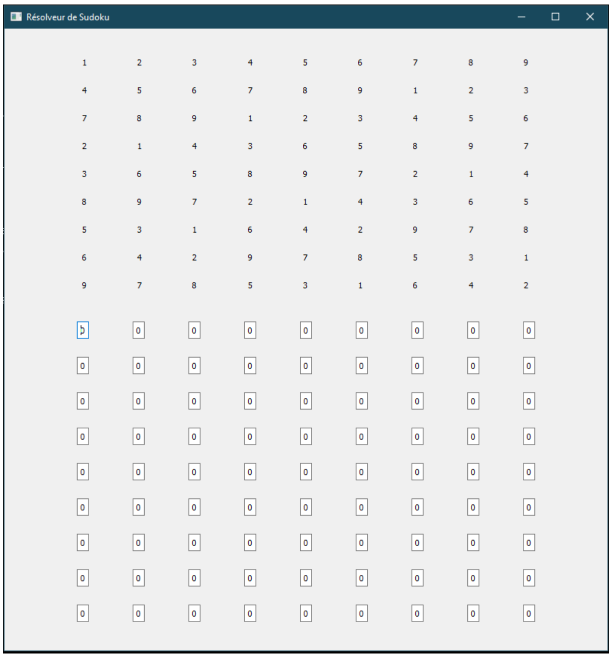


Figure 2.3 – Interface graphique Complete.

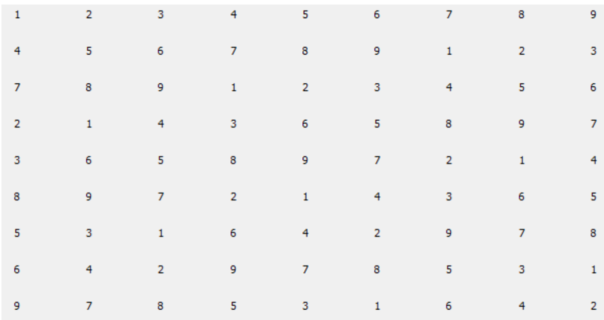


Figure 2.4 – Zone d'affichage.

Notre interface se compose donc d'une zone de saisi qui contiendra le sudoku pas qui devra être résolu par notre programme et un zone d'affichage qui contiendra le sudoku une fois résolution par notre algorithme de résolution. L'affichage des nombre entrées dans la zone de saisi ce fait de façon dynamique dans la zone d'affichage et la résolution ce fait une fois que l'utilisateur a appuyé sur la touhe entrée. Notre algorithme devant être innovant nous avons pensez à la généralisation du sudoku c'est à dire que nous pouvons choisir la taille des sudokus résolu par notre interface toujours avec un nombre de case et de ligne égales pour une raison d'impossibilité autrement à cause de la règle des sous-grilles.

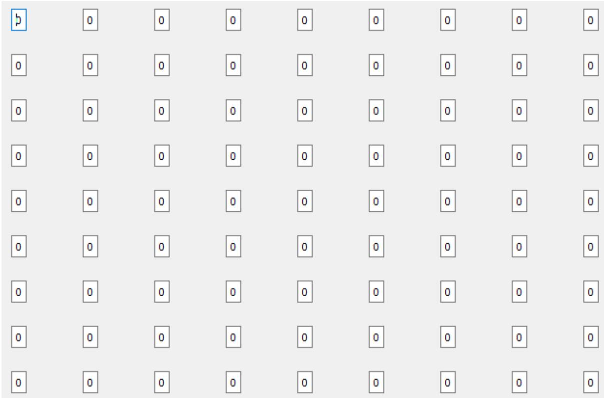


FIGURE 2.5 – Zone de saisi.

Voilà donc notre interface avec un sudoku de 8 cases et 8 colonnes :

FIGURE 2.6 – Interface graphique de résolution d'un sudoku 8 par 8.

Implémentation de l'interface graphique

Nous allons stocker dans une liste [2] contenant 9 listes[2] qui représenteront nos ligne qui elles contiendront 9 entiers allant de 0 à 9. Nous aurons donc besoin de coder des fonctions pour faire le lien entre l'interface et notre résolveur du fait de notre modélisation du sudoku nous pouvons découper cela en plusieurs fonction utilitaire :

— Une fonction qui nous permettra de stocker le contenu de notre interface graphique dans une liste comme celle décrite précédemment

2. structure de donnée incluse de base dans python qui permet de contenir plusieurs variables différentes

— Une fonction permettant de transférer le contenu d'une liste comme celle décrite plus haut dans la zone de saisi notre interface graphique
— Une fonction permettant de transférer le contenu d'une liste comme celle décrite plus haut dans la zone d'affichage notre interface graphique
— Une fonction permettant de copier le contenu de la zone de saisi de notre interface dans la zone d'affichage de notre interface
— Une fonction permettant de copier le contenu de la zone d'affichage de notre interface dans la zone de saisi de notre interface

Pour la création de notre interface nous avons eu besoin de 3 panel[3] un contenant nos deux grilles nous permettant leur superpositon et deux qui nous servent de grilles car nos grilles ne sont en réalité que deux panel contenant un nombre de zone de saisi de texte aussi appelé QTextEdit dans la librairie Qt égale à la $largeur^2$ et l'autre le même nombre de zone d'affichage de texte aussi appelé QLabel toujours dans la librairie Qt. Aussi nous avons eu besoin codés de quelques fonctions mineurs :

— Une fonction nous permettant de faire en sorte que la taille des zones de saisi de textes soit calculée par rapport à leurs contenu.
— Une fonction pour effectuer la résolution une fois la touche Entrée pressé.

2.1.6 Implémentation de la résolution avec Cplex

Comme nous l'avons fait dans notre exemple nous allons définir les contraintes et la fonction à maximiser : Nous avons pour contraintes :
— La somme de chaque ligne doit être égale à 45 pour permettre l'utilisation de chaque chiffre sur une ligne
— Les chiffres de chaque lignes doivent êtres différents
— les chiffre de chaque colonnes doivent êtres différents
— les chiffres de chaque sous grilles doivent être différents

Chaque inégalité peut être écrit sous la forme d'une fonction linéaire tel que :
$x - y = 0$
La fonction à maximiser peut-être décrite comme la maximisation de la somme de chaque case en sachant que celle ci donnera toujours $9 * 45$ c'est a dire 405.
les case remplies c'est à dire les cases indices de notre sudoku peuvent êtres modélisées par une contrainte d'égalité en plus sur la variable représentant la case concernée.
La généralisation du sudoku est implémenté telle qu'elle ne change pas la règle de l'unicité des chiffres sur les lignes et colonnes et ne fait que fixer le nombre de valeur possible comme étant égale à la largeur ou longueur de notre sudoku car comme dit plus haut nous avons toujours le nombre de case sur une ligne égale au nombre de case sur une colonne. Nous pouvons donc décelé 2 cas à notre génération, le pire cas où nous avons le nombre étant premier dans lequel nous avons décidé ne juste pas appliqué la règles des sous grilles et le meilleur cas où le nombre n'est pas premier nous n'avons pour appliqué la règle des sous grilles qu'a trouvé un diviseur en privilégiant un diviseur dont le résultat de la division donne lui même ainsi notre sous grille pour le cas d'une résolution 10 ligne par 10 colonnne par exemple donne des sous grilles qui ce composent de 2 élémments sur les lignes et 5 éléments sur les colonnes cequi donne un rectangle de 10 valeurs :

2.1.7 Implémentation de la résolution avec l'algorithme du backtracking

Pour cette algorithme nous avons besoin de 4 fonctions très simple :
— Une fonction pour copier notre liste.
— Une fonction pour vérifier si notre sudoku est toujours valide, autrement dit si il respect nos contraintes.
— Une fonction pour vérifier si notre sudoku est rempli autrement dit ne, contient pas de case contenant la valeur 0.
— Notre fonction principale qui utilisera toute les autres pour résoudre notre sudoku.

3. Contenant des différents éléments de notre interface

1	2	3	4	5	6	7	8	9	10
3	4	1	2	7	8	9	10	5	6
5	6	7	8	9	10	1	2	3	4
7	8	9	10	3	4	5	6	1	2
9	10	5	6	1	2	3	4	7	8
2	1	4	3	6	5	8	7	10	9
4	3	2	1	8	7	10	9	6	5
6	5	8	7	10	9	2	1	4	3
8	7	10	9	4	3	6	5	2	1
10	9	6	5	2	1	4	3	8	7

FIGURE 2.7 – Sudoku 10 par 10 résolu montrant la première sous grille.

Première fonction

Cette fonction nous est utile en cas d'erreur quand nous testons les valeurs car nous devons pouvoir retourner en arrière et donc nous devons garder le sudoku de l'itération de test précédente, c'est pour cela que pour chaque test de valeur nous utiliserons une copie du sudoku de l'itération précédente.

Deuxième fonction

Cette fonction doit vérifier nos différentes règles d'unicité et si les valeur des différente cases sont belle et bien comprises entre 1 et la largeur/longueur du sudoku. Elle doit considérer un sudoku incomplet mais respectant toutes les contraintes comme étant un sudoku valide car dans le cas contraire aucune valeur testé sur la première case vide si il y en a d'autres ne serait valide ce qui nous empêcherait d'avancer dans nos test.

2.1.8 Troisième fonction

Comme dit précédemment nous considèrons un sudoku incomplet mais respectant nos contraintes d'unicité comme étant valide. Nous avons donc besoin d'une fonction pour savoir si notre sudoku est rempli ou non pour terminer notre résolution. Un sudoku rempli est un sudoku ne contenant pas la valeur zero et un sudoku résolu étant un sudoku rempli et valide.

2.1.9 Fonction principale

Cette fonction est la plus longue de notre programme de résolution mais n'en est pas moins simple cette fonction doit en premier lieu trouver la première case vide dans le sens de lecture de notre sudoku (de gauche à droite et de haut en bas).
Puis doit tester les valeurs de 1 à 9 en faisant attention avant de tester une valeur en vérifiant si le sudoku est toujours valide si elle est posé avant de l'essayer.
Une fois une valeur valide trouvée la fonction on répète cette opération de façon récursive[4] mais cette fois avec notre copie de sudoku avec notre nouvelle valeur ajoutée.
Si nous ne trouvons aucune valeur de 1 à 9 pour notre case nous retournons la liste sans ajouter de valeur afin de tester une autre valeur dans la case vide précédente.
Comme dit précédemment nous ne nous arrêtons que quand notre sudoku est rempli et valide.

4. Par un appel de la fonction dans cette même fonction

2.2 Test du résolveur

2.2.1 Choix de nos test

Test avec un sudoku standart complètement vide

Nous avons tout d'abord pensé à essayer notre résolveur sur un sudoku complètement vide ce sudoku ayant le plus grand nombre de solution car à ce sudoku est solution toutes les grilles de sudoku valide et remplie existantes, il s'agit donc d'un des meilleurs cas possible pour nos deux algorithmes.

Test avec un sudoku standart rempli de 17 valeurs

Maintenant que nous avons identifié notre meilleur des cas commun nous pouvons maintenant essayer de trouver notre pire des cas commun, c'est grace à un article que nous avons pu l'identifier en effet selon cette article : [1]
Nous avons appris que Le nombre minimal d'indice pour n'avoir qu'une seule solution à une grille de sudoku était 17. Ce qui en plus de n'admettre qu'une seule solution possible nous donne le moins de valeur déjà entrées possible.

2.2.2 Test de la généralisation

Quant aux sudokus de taille différente nous n'avons effectuer des test que sur sudoku sans indices de par le fait de notre méconnaissance des sudoku généralisé car contrairement au sudoku standart nous n'avons pas trouvé d'équivalent au nombre de dieu des sudoku standart et n'avons pas les moyens d'en effectuer la recherche comme il a été fait dans l'article.

2.2.3 Implémentation de nos test

Nous n'avons eu besoin pour l'implémentation de nos test que d'une liste contenant des 0 pour notre cas le plus simple. Et pour notre cas le plus compliqué nous avons choisi de télécharger un fichier contenant 9 000 000 de sudoku résolu et n'avons eu qu'à en enlever 64 valeurs et avons essayé notre sudoku sur quelques-uns d'entre eux.

2.2.4 Résultat de nos tests

Changement d'algorithme de cplex

Nos tests avec les sudokus standarts nous ayant mené à la conclusion que l'algorithme du simplex nous confronte à un temps de résolution bien trop long dans les deux cas nous avons finalement décidé d'utiliser l'optimiseur de cplex utilisant un autre algoritme de résolution, ne changeant pas l'implémentation mais changeant légèrement nos contraintes nous avons donc utilisé la contrainte Alldiff de Cplex pour les différentes règles d'unicité ayant une bien meilleur efficacité, et remplacer la contrainte de la somme des valeurs d'une ligne par une contrainte de valeur sur chaque variable pour qu'elles ne varient qu'entre 1 et 9.

2.2.5 résultat avec notre nouvelle algorithme

Nous avons des réponses quasiments instentanées dans les tout les cas, voilà le sudoku donnné en retour d'un sudoku complètement vide standart :

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

FIGURE 2.8 – Résolution de grille vide avec l'optimizer de Cplex.

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

FIGURE 2.9 – Résolution de grille avec 17 indices grace à l'optimizer de Cplex.

15	4	6	11	2	7	8	3	12	5	1	9	14	13	10	16
3	13	12	8	5	16	11	6	10	7	15	14	9	4	1	2
10	16	1	7	14	4	9	15	2	11	13	8	12	6	5	3
5	9	2	14	10	1	13	12	4	16	6	3	7	11	8	15
8	6	14	1	15	5	12	4	11	13	3	7	16	9	2	10
16	2	5	15	3	6	7	9	1	10	12	4	11	14	13	8
4	10	3	13	16	14	1	11	9	8	2	6	15	5	12	7
11	7	9	12	8	13	2	10	5	15	14	16	4	1	3	6
14	8	15	9	13	3	4	1	6	2	5	12	10	16	7	11
13	1	11	4	6	12	16	2	8	9	7	10	3	15	14	5
6	3	16	5	7	10	14	8	15	4	11	1	2	12	9	13
7	12	10	2	9	11	15	5	14	3	16	13	6	8	4	1
12	15	8	16	11	9	3	7	13	14	10	5	1	2	6	4
9	14	7	6	4	2	10	13	16	1	8	11	5	3	15	12
2	5	4	3	1	8	6	16	7	12	9	15	13	10	11	14
1	11	13	10	12	15	5	14	3	6	4	2	8	7	16	9

FIGURE 2.10 – Résolution de Grille de longueur et largeur égale à 16 grace à l'optimizer de Cplex.

2.2.6 Résultat avec notre algorithme de BackTracking

Pour ce qui est de l'algorithme utilisant le backtracking nous avons avec des sudoku standart des réponses instantanée. Pour ce qui est du résultat avec un sudoku généralisé de taille 16 par 16 nous avons un temps de réponse notable et assez génant voir beaucoup trop long pour certaine taille tel que 15.
Voila le résultat de notre résolution avec un sudoku de taille standart sans indices :

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

FIGURE 2.11 – Résolution de grille vide avec l'algorithme utilisant le backtracking.

2.2.7 Analyse de nos résultats

Résultats avec Cplex

Les résultats de l'utilisation de Cplex sont extrèmements positifs, notre résolvur utilisant Cplex peut réaliser d'extrèmes taille de sudoku sans aucun problème.

Résultat avec l'utilisation de BackTracking

Les résultats avec l'utilisation du backtracking permet parfaitement de résoudre un sudoku standart peu importe les indices données et leurs nombre. Mais la généralisation laissant à désiré au niveau du temps de calcul, nous pouvons réfléchir à plusieurs piste d'amélioration de cela nous pouvons par

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

FIGURE 2.12 – Résolution de grille avec 17 indices avec l'algorithme utilisant le backtracking.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
5	6	7	8	1	2	3	4	13	14	15	16	9	10	11	12
9	10	11	12	13	14	15	16	1	2	3	4	5	6	7	8
13	14	15	16	9	10	11	12	5	6	7	8	1	2	3	4
2	1	4	3	6	5	8	7	10	9	12	11	14	13	16	15
6	5	8	7	2	1	4	3	14	13	16	15	10	9	12	11
10	9	12	11	14	13	16	15	2	1	4	3	6	5	8	7
14	13	16	15	10	9	12	11	6	5	8	7	2	1	4	3
3	4	1	2	7	8	5	6	11	12	9	10	15	16	13	14
7	8	5	6	3	4	1	2	15	16	13	14	11	12	9	10
11	12	9	10	15	16	13	14	3	4	1	2	7	8	5	6
15	16	13	14	11	12	9	10	7	8	5	6	3	4	1	2
4	3	2	1	8	7	6	5	12	11	10	9	16	15	14	13
8	7	6	5	4	3	2	1	16	15	14	13	12	11	10	9
12	11	10	9	16	15	14	13	4	3	2	1	8	7	6	5
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

FIGURE 2.13 – Résolution de grille de longueur et largeur égale à 16 avec l'algorithme utilisant le backtracking.

exemple pour améliorer l'éfficacité de notre algorithme éviter les configurations où les valeurs peuvent êtres déduites de façon logique tel que celle-ci :

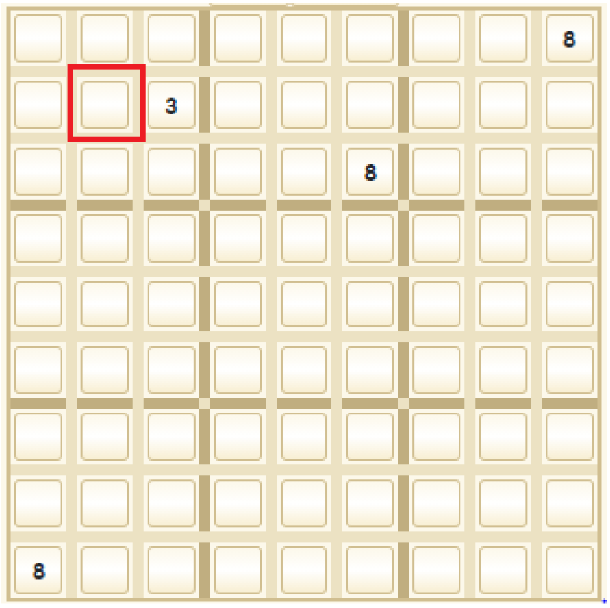


Figure 2.14 – Exemple de case dont la valeur peut-être déduite de façon logique, dans notre cas la valeur dans la case encadré est 8.

Chapitre 3

Conclusion

3.1 Rappel de la problèmatique

Notre problèmatique à la base était la résolution de sudoku Standart ce que nous avons reussi. Nous avons donc pour avoir une autre vision du problème nous avons choisi de généraliser celui-ci. Ce qui nous a ouvert à des pistes d'améliorations dont nous n'aurions pas eu besoin dans le cas de la résolution de sudoku standart.

3.2 Réponse apportées

Nous avons en ce qui concerne la résolution de sudoku standart deux programmes permettant leurs résolution quand aà la généralisation de ceux-ci nous avons un programes parfaitement fonctionnel permettant sa résolution via Cplex.

3.3 Piste d'amélioration

Nous pourrions pour voir les limites de l'algorithme du Cplex l'utiliser sur un grand volume de sudoku et voir le temps mis pour tous les résoudre en une seule fois. Nous pourrions essayer avec des sudokus d'une taille extrème pour en vérifier la consistence.
Comme dit précédemment en ce qui concerne la déduction des valeurs nous pouvons éviter beaucoup de test inutiles grace à cela pour amélioré notre algorithme utilisant le backtracking.
Nous pouvons aussi pour les tests sur les sudokus de taille standart tester nos algorithme sur les classes de sudoku décrite dans cette article :
[2]
Nous expliquant que tout les sudoku possibles peuvent être obtenu par plusieurs actions élémentaires à partir de 5472730538 grilles de sudoku que nous pouvons qualifier d'élémentaires. Cela nous permettrait d'affirmer de manière solide que nos algorithme fonctionnent dans tout les cas possibles.

3.4 Les apports du stage

3.4.1 les apports à l'entreprise

Grace à ce Stage les chercheurs auront une meilleur idée de l'utilisation de Cplex sur python pour la résolution de leurs problèmes et grace à notre algorithme utilisant le backtracking nous pouvons appuyer que l'utilisation des algorithmes de Cplex est bien plus efficaces que les méthodes que nous pouvons coder de façon artisanal sans y consacrer une très grande quantité de connaissances.

3.4.2 les apports personels

Avant ce stage je n'avais aucune connaissance en résolution de problème d'optimisation linéaire. Grace à ce stage je connais maintenant un de ses outils de résolution les plus puissant. J'ai aussi pu apprendre plusieurs termes de mathématique combinatoire que je n'aurais jamais connu autrement. J'ai maintenant la connaissance de plusieurs algorithme de résolution de problème de satisfaction de contrainte. J'ai nmaintenant aussi les connaissances basique nécéssaire quant à la programmation d'interface graphique avec Qt.

Chapitre 4

Remerciements

Je souhaiterais addresser mes remerciements :

— Monsieur Andrei Doncescu qui m'a proposé ce stage et qui a été mon maître de stage durant tout ce projet.

— Le LAMIA d'avoir pu m'accueillir en son sein.

— Mon collègue stagiaire stéphane qui m'a aidé dans la recherche de document

Bibliographie

[1] Pierre Barthélémy. 17 est le nombre de dieu au sudoku. *Le monde*, 2012.

[2] Ed Russell and Frazer Jarvis. There are 5472730538 essentially different sudoku grids. 2005.

Annexe A

Annexes

A.1 Code Générer durant le stage

```

import sys
import docplex
from docplex.cp.model import CpoModel
import random
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QFontMetrics
from PyQt5.QtWidgets import QApplication, QWidget, QHBoxLayout, QVBoxLayout, QLabel,
    QTextEdit
#Varible indiquand la m[U+FFFD]thode de resolution choisie
Res=2

#La classe de nos zones de saisi de texte
class Text(QTextEdit):
    def __init__(self):
        QTextEdit.__init__(self)
        self.setText("0")
        self.setFocusPolicy(Qt.StrongFocus)
        font = self.document().defaultFont()
        fontMetrics = QFontMetrics(font)
        textSize = fontMetrics.size(0, self.toPlainText())

        w = textSize.width() + 10
        h = textSize.height() + 10
        self.setMinimumSize(w, h)
        self.setMaximumSize(w, h)
        self.resize(w, h)
    def Test(self):
        font = self.document().defaultFont()
        fontMetrics = QFontMetrics(font)
        textSize = fontMetrics.size(0, self.toPlainText())

        w = textSize.width() + 10
        h = textSize.height() + 10
        self.setMinimumSize(w, h)
        self.setMaximumSize(w, h)
        self.resize(w, h)
    def keyPressEvent(self, event):
        if event.key() != Qt.Key_Enter:
            QTextEdit.keyPressEvent(self,event)
        refresh(fen.Grille,fen.Grille1,fen.taille)
        if event.key() == Qt.Key_Enter:
            if Res==1:
                fen.toGrille(myMap(lpexl(fen.toListeIntAff(),fen.taille),fen.taille))
            if Res==2:
                fen.toGrille(myMap(ResoudreArbre(fen.toListeIntAff(),fen.taille),fen.
    taille))

```

```python
#La classe de notre interface Graphique
class Fenetre(QWidget):
    WGrids= []
    WGridsLayout= []
    Lignes= []
    Grille= []
    LigneLayout= []
    Lignes1 = []
    Grille1= []
    Ligne1Layout= []
    layout = QHBoxLayout()
    layouts= []
    Panel = []
    taille=0

    def __init__(self,taille):
        QWidget.__init__(self)
        self.setWindowTitle("R[U+FFFD]solveur de Sudoku")
        self.resize(800,800)
        self.taille=taille
        for i in range(taille):
            self.Lignes.append(QWidget())
            self.Lignes1.append(QWidget())
            self.LigneLayout.append(QHBoxLayout())
            self.Ligne1Layout.append(QHBoxLayout())
            self.Lignes[i].setLayout(self.LigneLayout[i])
            self.Lignes1[i].setLayout(self.Ligne1Layout[i])
        for i in range(2):
            self.WGrids.append(QWidget())
            self.WGridsLayout.append(QVBoxLayout())
        for i in range(3):
            self.Panel.append(QWidget())
        self.layouts.append(QVBoxLayout())
        for i in range(3):
            self.layout.addWidget(self.Panel[i])
        self.setLayout(self.layout)
        self.Panel[1].setLayout(self.layouts[0])
        self.layouts[0].addWidget(self.WGrids[0])
        self.layouts[0].addWidget(self.WGrids[1])
        self.WGrids[0].setLayout(self.WGridsLayout[0])
        self.WGrids[1].setLayout(self.WGridsLayout[1])
        for i in range(taille):
            for j in range(taille):
                self.Grille.append(QLabel("0"))
                self.Grille1.append(Text())
                self.LigneLayout[i].addWidget(self.Grille[i*taille+j])
                self.Ligne1Layout[i].addWidget(self.Grille1[i*taille+j])
        for i in range(taille):
            self.WGridsLayout[0].addWidget(self.Lignes[i])
            self.WGridsLayout[1].addWidget(self.Lignes1[i])
        for i in range(taille):
            self.LigneLayout[i].setContentsMargins(68,9,9,9)

    def toListeInt(self):
        liste=[]
        for i in range(self.taille):
            liste.append([])
            for j in range(self.taille):
                liste[i].append(int(self.Grille[i*self.taille+j].text()))
        return liste


    def toListeIntAff(self):
        liste=[]
        for i in range(self.taille):
            liste.append([])
            for j in range(self.taille):
                liste[i].append(int(self.Grille1[i*self.taille+j].toPlainText()))
        return liste

```

```python

    def toListe(self):
        liste=[]
        for i in range(self.taille):
            liste.append([])
            for j in range(self.taille):
                liste[i].append(self.Grille[i*self.taille+j].text())
        return liste


    def toGrille(self,liste):
        for i in range(self.taille):
            for j in range(self.taille):
                self.Grille[self.taille*i+j].setText(liste[i][j])

    def reInput(self):
        for i in range(self.taille):
            for j in range(self.taille):
                self.Grille1[self.taille*i+j].setText(self.Grille[self.taille*i+j].text())
app = QApplication.instance()
if not app:
    app = QApplication(sys.argv)

#Fonction indiquant si un Sudoku est valide
def valideint(sudoku,taille):
    valide= True
    for i in range(taille):
        for j in range(taille):
            if sudoku[i][j]<0 or sudoku[i][j]>taille:
                valide=False

    for i in range(taille):
        l=sudoku[i]
        for j in range(taille+1):
            if l.count(j)>1 and j!=0:
                valide=False

    for i in range(taille):
        l=[]
        for j in range(taille):
            l.append(sudoku[j][i])
        for j in range(taille+1):
            if l.count(j)>1 and j!=0:
                valide=False

    divx=0
    divy=0
    for i in range(2,taille):
        if(taille%i==0 and i*i==taille):
            divx=i
    divy=divx
    if divx==0 and divy==0:
        for i in range(2,taille):
            if(taille%i==0 and divx==0):
                divx=i
        for i in range(2,taille):
            if(taille%i==0 and (divx*i)==taille and divy==0):
                divy=i
    for i in range(taille):
        l=[]
        for j in range(taille):
            l.append(sudoku[int(j/divx)+(int(i/divy)*divy)][(j%divx)+((i%divy)*divx)])
        for j in range(taille+1):
            if l.count(j)>1 and j!=0:
                valide=False

    return valide

#Fonction permettant d'avoir une liste repr[U+FFFD]sentant un sudoku d'entier sous la forme de
```

```
    liste de caract[U+FFFD]re
def myMap(l,taille):
    l1=l.copy()
    for i in range(taille):
        for j in range(taille):
            l1[i][j]=str(l1[i][j])
    return l1

#Fonction indiquand si un sudoku contient des cases vides
def Remplie(liste,taille):
    plein = True
    for i in range(taille):
        for j in range(taille):
            if (liste[i][j]==0):
                plein=False
    return plein

#Fonction permettant la copie de notre sudoku
def Copie(liste,y,x,val):
    l=liste.copy()
    l[y][x]=val
    return l

#Fonction de r[U+FFFD]solution utilisant le backtracking
def ResoudreArbre(liste,taille):
    c=1
    i=0
    j=0
    x=0
    y=0
    while x<taille:
        y=0
        while y<taille:
            if(liste[x][y]==0):
                i=x
                j=y
                x=taille
                y=taille
            y+=1
        x+=1
    while not Remplie(liste,taille):
        while c<taille and not valideint(Copie(liste,i,j,c),taille):
            c+=1
        if valideint(Copie(liste,i,j,c),taille):
            liste= ResoudreArbre(Copie(liste,i,j,c),taille)
            c+=1
        if c>=taille and not (Remplie(liste,taille) and valideint(liste,taille)):
            liste[i][j]=0
            return liste
    return liste

#Fonction pour actualiser notre affichage
def refresh(x,y,taille):
    for i in range(taille):
        for j in range(taille):
            x[i*taille+j].setText(y[i*taille+j].toPlainText())

#Fonction permettant de charger les cased[U+FFFD]j[U+FFFD]rempli de notre sudoku parmi nos
    contraintes
def chargez(var,liste,taille):
    for i in range(taille):
        for j in range(taille):
            if(liste[i][j]>0):
                var[i][j].set_domain((liste[i][j], liste[i][j]))

#Fonction de resolution utilisant Cplex
def lpexl(liste,taille):

    c=1
    M=CpoModel("Sudoku")
```

```python
    GRNG = range(taille)

    var = [[M.integer_var(min=1, max=taille, name="x" + str(l*taille+c)) for l in range(
    taille)] for c in range(taille)]

    # Ajout des contraintes sur les lignes
    for l in GRNG:
        M.add(M.all_diff([var[l][c] for c in GRNG]))

    # Ajout des contraintes sur les colonnes
    for c in GRNG:
        M.add(M.all_diff([var[l][c] for l in GRNG]))

    divx=0
    divy=0
    for i in range(2,taille):
        if(taille%i==0 and i*i==taille):
            divx=i
    divy=divx
    if divx==0 and divy==0:
        for i in range(2,taille):
            if(taille%i==0 and divx==0):
                divx=i
        for i in range(2,taille):
            if(taille%i==0 and (divx*i)==taille and divy==0):
                divy=i


    # Ajout des contraintes sur les sous grilles
    ssrng = range(0, taille, divy)
    for sl in ssrng:
        for sc in range(0, taille, divx):
            M.add(M.all_diff([var[l][c] for l in range(sl, sl + divy) for c in range(sc,
     sc + divx)]))
    chargez(var,liste,taille)
    msol=M.solve(TimeLimit=10)
    sol=[[msol[var[l][c]] for c in GRNG] for l in GRNG]
    return sol

fen = Fenetre(16)

fen.show()

app.exec_()
```