

Sudoku : Rapport de projet

Université Montpellier II

Licence 2

Projet du semestre 2 de l'UE GLIN405

Équipe Sudoku

2013-2014

Membres de l'équipe de développement

[Abdoulaye Diallo](#)

[Redoine El Ouasti](#)

Assistant chef de projet : [Simon Galand](#)

Assistant chef de projet : [Adrien Lamant](#)

[Pierre-Louis Latour](#)

[Charly Maeder](#)

[Pierre Ruffin](#)

Chef de projet : [Stella Zevio](#)

Remerciements

Nous remercions notre référent, M. Meynard, pour ses conseils avisés et pour l'efficacité avec laquelle il nous a guidés tout au long du développement du projet.

Nous tenons également à remercier William Dyce, ingénieur R&D pour l'intérêt qu'il a manifesté envers notre projet, ainsi que Sofien Benharchache, étudiant en Licence 2, pour son précieux concours et son soutien indéfectible.

Table des matières

1	Introduction	6
1.1	Généralités	6
1.2	Sujet	6
1.3	Cahier des charges	6
2	Organisation du projet	19
2.1	Organisation du travail	19
2.2	Choix des outils de développement	20
3	Analyse du projet	21
4	Développement	23
4.1	Génération des grilles de sudoku	23
4.2	Expérience de jeu	30
5	Manuel d'utilisation	36
5.1	Terminal	36
5.2	Interface graphique	38
6	Perspectives et conclusions	39
6.1	Perspectives	39
6.2	Conclusion	39
7	Documents d'analyse	41
8	Listings	50

Table des figures

1	Une grille 4*4	24
2	Une grille 9*9	25
3	Une grille 16*16	25
4	Affichage d'une grille 4*4	26
5	Affichage d'une grille 9*9	26
6	Affichage d'une grille 16*16	27
7	Cross-hatching	31
8	Cross-hatching	31
9	Lone-number	32
10	Lone-number	32
11	Naked subset	32
12	Naked Subset	33

1 Introduction

1.1 Généralités

Ce projet s'inscrit dans la formation en Licence 2 d'Informatique à l'Université Montpellier II.

Il a lieu lors du second semestre et constitue une unité d'enseignement à 5 crédits. L'échéance du projet est fixée à la semaine 22 de l'année 2014. La première réunion ayant eu lieu à la semaine 5, nous disposons d'un délai de 17 semaines pour achever le projet.

L'objectif de cette unité d'enseignement est la collaboration des étudiants en Licence afin d'achever des projets informatiques donnés. Cette collaboration implique l'acquisition de compétences plus poussées en programmation, le partage des connaissances ainsi que l'initiation à la gestion de projet.

1.2 Sujet

Il s'agit d'écrire une application en ligne de commande (pas d'interface graphique) permettant de représenter et de résoudre des sudokus. Les sudokus seront de taille 4x4, 9x9, ou 16x16.

1.3 Cahier des charges

Ci-après, le cahier des charges que nous avons réalisé préalablement au développement du projet.

Sudoku : Cahier des charges

Université Montpellier II

Licence 2

Projet de fin de semestre

2013-2014

Table des matières

A	Introduction	4
A.1	Contexte	4
B	Demande	5
B.1	Description du projet	5
B.2	Fonctionnalités	5
C	Contraintes	6
C.1	De coût	6
C.2	De temps	6
D	Organisation du projet	7
D.1	Planification	7
D.2	Ressources	9
D.3	Différenciation	10

Table des figures

1	Diagramme de Gantt	7
2	Diagramme de Gantt	8
3	Tâches	9
4	Bilan de compétences	11

A Introduction

A.1 Contexte

Ce projet est un projet scolaire sans aucun but lucratif ni coût prévisionnel. De ce fait, les clauses juridiques et législatives pouvant viser un projet tel que le nôtre ne seront pas prises en compte, étant donné le but uniquement pédagogique de notre travail.

B Demande

B.1 Description du projet

L'objectif du programme est d'écrire une application en ligne de commande permettant de représenter et de résoudre des sudokus. Les sudokus seront de taille 4x4, 9x9, ou 16x16.

B.2 Fonctionnalités

Dans les faits, l'application permettra à l'utilisateur de choisir la taille de la grille de jeu (4x4, 9x9 ou 16x16) et initialisera une grille correspondant aux souhaits du joueur. Elle générera pseudo-aléatoirement la solution complète respectant les conditions de validité par méthode de backtracking, puis à partir de cette solution la grille de jeu incomplète.

L'utilisateur pourra alors jouer en affectant une valeur à une case vide (qu'il pourra sélectionner à partir de ses coordonnées).

Une aide sera proposée, affichant les possibilités d'une case en l'état actuel de la grille. Nous permettrons un nombre limité d'accès à cette aide éventuellement selon le niveau de difficulté choisi. Nous permettrons également un nombre limité d'essais de valeurs (suivant la taille de la grille) afin d'éliminer la possibilité de tester toutes les valeurs possibles pour l'ensemble des cases vides.

Selon l'avancée du projet, nous espérons implémenter une interface graphique.

C Contraintes

C.1 De coût

Aucune contrainte de coût. Les accès aux diverses documentations n'engendreront aucun frais.

Nous choisirons de toujours utiliser des outils de développement et de gestion de projet gratuits, et dans la mesure du possible, libres.


C.2 De temps

Ce projet se déroule durant le semestre 4 de la deuxième année de Licence Informatique à l'Université Montpellier II.

L'échéance est fixée à la semaine 22 de l'année 2014. La première réunion a eu lieu lors de la semaine 5 de la même année. Nous avons donc un délai de 17 semaines afin d'achever le projet.

D Organisation du projet

D.1 Planification



Nom	Date de début	Date de fin
• Début du projet	28/01/14	28/01/14
• Réunion membres de l'équipe	28/01/14	28/01/14
• Réunion officielle référent	30/01/14	30/01/14
• Cahier des charges	29/01/14	31/01/14
• Réunion officielle référent	13/02/14	13/02/14
• Réunion officielle référent	20/02/14	20/02/14
• Réunion officielle référent	27/02/14	27/02/14
• Fonction principale	28/01/14	28/02/14
• Réunion officielle référent	06/03/14	06/03/14
• Réunion officielle référent	13/03/14	13/03/14
• Réunion officielle référent	20/03/14	20/03/14
• Réunion officielle référent	27/03/14	27/03/14
• Réunion officielle référent	03/04/14	03/04/14
• Réunion officielle référent	10/04/14	10/04/14
• Fonctions auxiliaires	28/01/14	10/04/14
• Réunion officielle référent	17/04/14	17/04/14
• Réunion officielle référent	24/04/14	24/04/14
• Réunion officielle référent	01/05/14	01/05/14
• Réunion officielle référent	08/05/14	08/05/14
• Interface graphique	10/04/14	09/05/14
• Réunion officielle référent	15/05/14	15/05/14
• Support présentation	12/05/14	15/05/14
• Réunion officielle référent	22/05/14	22/05/14
• Rapport projet	03/02/14	22/05/14
• Réunion officielle référent	29/05/14	29/05/14
• Echéance	30/05/14	30/05/14

FIGURE 1 – Diagramme de Gantt

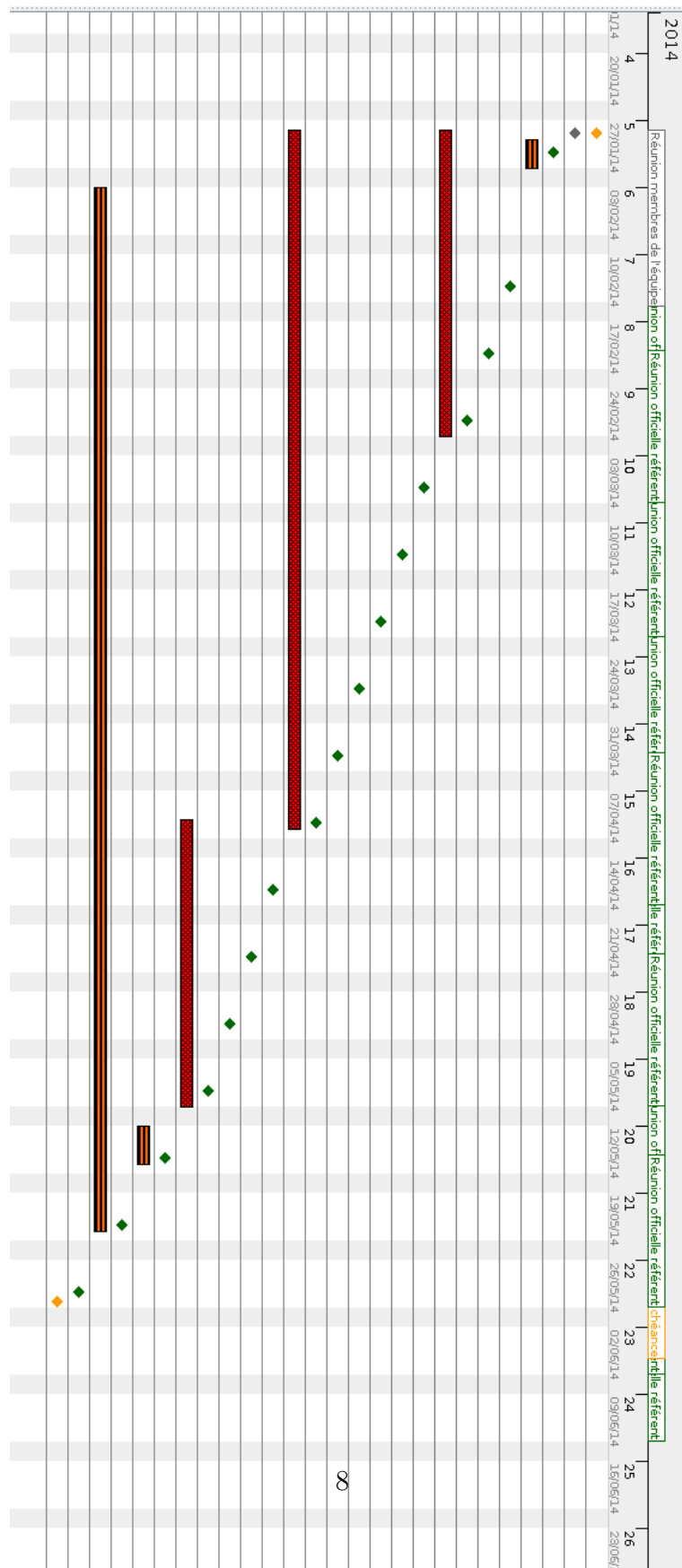


FIGURE 2 – Diagramme de Gantt

Nom	Date de début	Date de fin	Ressources
Début du projet	28/01/14	28/01/14	
Réunion membres de l'équipe Phase d'observation et de cohésion de l'équipe. Choix du chef de projet et mise en place de l'organisation. (formation des équipes et des outils, notamment groupware)	28/01/14	28/01/14	
Réunion officielle référent Réunion officielle en présence du référent.	30/01/14	30/01/14	
Cahier des charges Réalisation du cahier des charges du projet.	29/01/14	31/01/14	Stella Zevio
Réunion officielle référent Réunion officielle en présence du référent. Remise du cahier des charges.	13/02/14	13/02/14	
Réunion officielle référent	20/02/14	20/02/14	
Réunion officielle référent	27/02/14	27/02/14	
Fonction principale Recherche algorithmique et implémentation en C de la fonction principale de l'application.	28/01/14	28/02/14	Pierre-Louis Latour, Pierre Ruffin, Redoine El Ouasti, Simon Galand
Réunion officielle référent	06/03/14	06/03/14	
Réunion officielle référent	13/03/14	13/03/14	
Réunion officielle référent	20/03/14	20/03/14	
Réunion officielle référent	27/03/14	27/03/14	
Réunion officielle référent	03/04/14	03/04/14	
Réunion officielle référent	10/04/14	10/04/14	
Fonctions auxiliaires Recherche algorithmique et implémentation en C des fonctions auxiliaires de l'application.	28/01/14	10/04/14	Stella Zevio, Abdoulaye Diallo, Charly Maeder, Adrien Lamant
Réunion officielle référent	17/04/14	17/04/14	
Réunion officielle référent	24/04/14	24/04/14	
Réunion officielle référent	01/05/14	01/05/14	
Réunion officielle référent	08/05/14	08/05/14	
Interface graphique Implémentation de l'interface graphique (C++) et portage éventuel du code.	10/04/14	09/05/14	Simon Galand, Adrien Lamant, Abdoulaye Diallo, Stella Zevio
Réunion officielle référent	15/05/14	15/05/14	
Support présentation Réalisation du support de soutenance du projet. (class Beamer LaTeX)	12/05/14	15/05/14	Stella Zevio, Pierre-Louis Latour, Pierre Ruffin
Réunion officielle référent	22/05/14	22/05/14	
Rapport projet Réalisation du rapport de projet sous LaTeX.	03/02/14	22/05/14	Stella Zevio, Adrien Lamant, Abdoulaye Diallo, Redoine El Ouasti, Charly Maeder, Pierre Ruffin, Pierre-Louis Latour, Simon Galand
Réunion officielle référent	29/05/14	29/05/14	
Echéance Date de la soutenance.	30/05/14	30/05/14	

FIGURE 3 – Tâches

D.2 Ressources

Composition de l'équipe de projet (ressources humaines)

Stella Zevio (**Chef de projet**, équipe A)

Adrien Lamant (équipe A : **Co-responsable**)

Abdoulaye Diallo (équipe A)

Charly Maeder (équipe A)

Simon Galand (équipe B : **Co-responsable**)

Redoine El Ouasti (équipe B)

Pierre-Louis Latour (équipe B)

Pierre Ruffin (équipe B)

Le contexte de réalisation du projet va nous contraindre à adopter une démarche particulière. Nous effectuerons des mises en commun régulières au cours de réunion hebdomadaires, en dehors des réunions officielles de projet organisées par notre référent.

De plus, nous serons continuellement en lien par le biais de logiciels de type groupware (visioconférences, audioconférences, emails et gestions de documents). Cela nous permettra de contrôler et surveiller l'avancement du projet et de nous répartir efficacement les tâches.

Nous utiliserons également un logiciel de gestion de versions.

Nous utiliserons nos compétences en algorithmique afin de concevoir notre programme.

Nous choisissons le langage de programmation C pour l'implémentation, et nous espérons pouvoir implémenter une interface graphique à l'aide de GTK+.

Les différents supports de projet (cahier des charges, présentation, rapport de projet) seront réalisés à l'aide de L^AT_EX.

D.3 Différenciation

Après une phase d'observation et de cohésion du groupe de travail collaboratif, nous avons établi la synthèse du bilan de connaissances et compétences de l'équipe concernant les outils nécessaires à l'achèvement du projet.

Les compétences relevées et les choix des membres de l'équipe servent à la répartition des tâches et sont en possession du chef de projet. Elles ont permis de scinder les ressources humaines en deux équipes, afin de répartir efficacement les tâches.

Bilan de compétences de l'équipe de projet Sudoku

Niveau de l'équipe évalué de 1 à 4 (1 étant la valeur la plus basse, 4 la plus élevée, 2 correspondant à un niveau "vu à la fac")

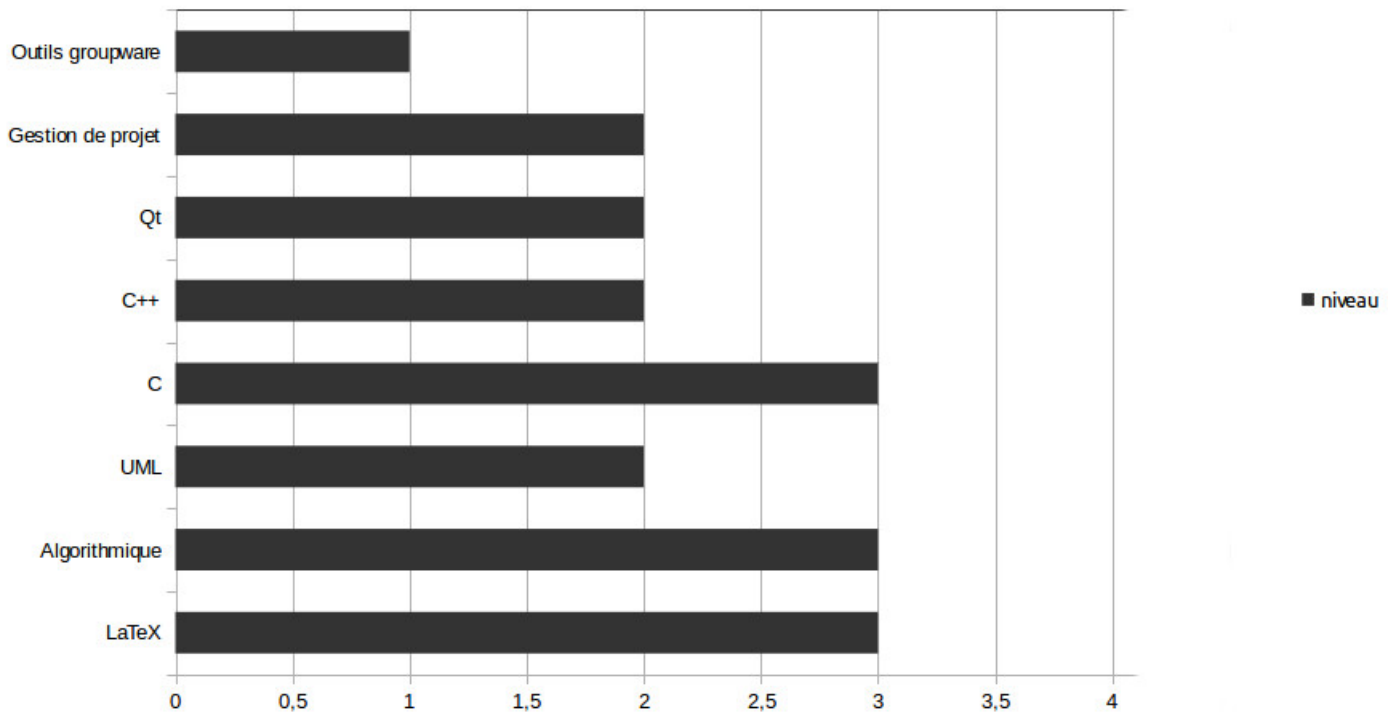


FIGURE 4 – Bilan de compétences

2 Organisation du projet

2.1 Organisation du travail

En plus des réunions officielles en présence de notre référent, nous nous réunissions de façon hebdomadaire, et régulièrement deux fois par semaine lorsque notre emploi du temps nous le permettait.

La première réunion avait lieu tous les mardi soirs et nous permettait de faire une mise au point concernant l'avancée de notre travail et la participation de chacun. Nous faisons le bilan de la semaine et nous récapitulons ce qu'il nous restait à achever. Les enjeux et les avancées étaient expliquées à tous les membres de l'équipe, afin de former un groupe le plus homogène possible. Chaque membre réalisait la tâche qui lui était assignée au cours de la semaine suivante.

La seconde réunion, facultative, avait lieu le jeudi après-midi dans une salle informatique de la Faculté des Sciences. Elle nous permettait de nous réunir pour réellement implémenter le projet en groupe, dans une ambiance de travail similaire à celle d'équipes de développement professionnelles.

Le chef de projet a été élu une semaine après le début de la formation de l'équipe, à l'unanimité. Elle a été choisie pour sa compréhension du sujet et son aisance à mener le groupe.

Elle a formé deux équipes distinctes de développement, après une étape de différenciation permettant d'évaluer le niveau de chaque membre de l'équipe. Un responsable a été nommé à la tête de chacun de ces deux groupes, choisi pour sa motivation et ses compétences pédagogiques.

Le groupe A a été formé par les personnes ayant initialement une plus grande expérience en développement, le groupe B étant celui de ceux qui n'avaient que très peu programmé en dehors des travaux pratiques et divers projets imposés durant notre formation.

Il a été assigné au groupe B des tâches plus aisées au début du projet, notamment l'implémentation de la fonction principale de l'application, afin que les membres progressent rapidement et soient plus à l'aise. Ils ont très vite atteint un niveau équivalent à celui du groupe A, qui avait directement

commencé à implémenter les fonctions auxiliaires.

Par la suite, l'implémentation de chaque fonction auxiliaire restante a été attribuée à un binôme ou trinôme hétérogène et dont la composition était régulièrement modifiée, afin que chacun puisse programmer avec la plupart des autres membres de l'équipe.

L'application nécessitant la maîtrise de deux concepts différents, le back-tracking (retour sur trace) et les heuristiques, les membres de l'équipe se sont individuellement et naturellement tournés vers l'un des deux axes qu'ils souhaitaient approfondir en particulier, et les binômes ou trinômes se sont naturellement figés avec le temps, selon les affinités.

Les divers documents exigés (cahier des charges, rapport, présentation) ont été réalisés par l'ensemble de l'équipe, sous la direction du chef de projet.

2.2 Choix des outils de développement

L'application a été implémentée en C, sous éditeur de texte (emacs) ou IDE (CodeBlocks) selon les préférences des membres de l'équipe.

Il a été compilé avec gcc, débogué à l'aide de l'option `-g` et de Valgrind pour l'allocation dynamique de mémoire.

L'API a été générée à l'aide des commentaires Doxygen des en-têtes des fonctions (`.h`).

Les divers documents (cahier des charges, présentation, rapport de projet) ont été générés à l'aide de \LaTeX .

3 Analyse du projet

Le projet consiste à développer une application générant des sudokus de taille 4×4 , 9×9 et 16×16 , et de pouvoir y jouer, nécessitant donc d'implémenter l'ensemble des fonctions permettant à l'utilisateur une expérience de jeu agréable (aide, sauvegarde, chargement).

La première partie du projet, la génération des sudokus, doit nous permettre de générer des grilles complètes de façon pseudo-aléatoire, en fonction de la taille choisie.

La seconde partie du projet, le jeu, doit permettre au joueur non seulement de pouvoir remplir la grille, mais également de faire appel à une aide affichant la liste des valeurs possibles pour une case donnée en cas de nécessité.

La génération des sudokus nécessite plusieurs fonctions :

- Une structure permettant de modéliser la grille de sudoku
- Une fonction permettant de choisir la taille de la grille (4×4 , 9×9 , 16×16 étant les seules tailles disponibles)
- Une fonction permettant d'initialiser cette grille
- Une fonction permettant d'afficher la grille
- Trois fonctions permettant de vérifier les conditions de validité de la grille (contrainte sur ligne, contrainte sur colonne, contrainte sur région)
- Une fonction générant la grille complète (la solution)
- Une fonction permettant de vider la grille complète en conservant l'unicité de la solution. On doit obtenir une grille de jeu avec un nombre de cases vides suffisant pour permettre une expérience de jeu agréable

L'expérience de jeu nécessite également plusieurs fonctions :

- Une fonction permettant de jouer, c'est-à-dire modifier la valeur associée à une case de la grille, si et seulement si la case est modifiable (non fixée, c'est-à-dire vide lors de la génération de la grille de jeu)
- Une fonction permettant de faire appel à l'aide en cas de nécessité

(c'est-à-dire permettant d'afficher la liste des valeurs possibles pour une case donnée en l'état actuel de la grille)

- Une fonction permettant de sauvegarder la partie
- Une fonction permettant de charger la partie
- Une fonction permettant de réinitialiser la partie
- Une fonction permettant de comparer deux grilles entre elles, afin de valider la grille remplie par l'utilisateur dans le cas où elle correspondrait à la solution de la grille
- Une fonction permettant de maintenir un classement selon la vitesse de résolution des sudoku, en fonction de la taille de la grille

L'aide doit permettre d'éliminer certaines valeurs de la liste des valeurs possibles, en appliquant les règles mathématiques du jeu.

4 Développement

4.1 Génération des grilles de sudoku

Modélisation de la grille

Nous avons choisi de modéliser une grille de sudoku par un tableau à deux dimensions de cases de grille, les cases de grille étant elles-mêmes modélisées par une structure.

Chaque case stocke :

- une valeur comprise entre 1 et la taille de la grille de sudoku, correspondant à la valeur de la case
- un pseudo-booléen (1 ou 0), qui vaut 1 si la case est modifiable, 0 si elle est non modifiable
- une suite de 16 bits (16 unités numériques les plus simples, de valeur 0 ou 1), correspondant à l'aide

Si la case est modifiable, cela signifie que le joueur pourra en modifier la valeur plus tard. C'est une case initialement vide (de valeur 0) lors de la génération de la grille de jeu. Le joueur doit lui attribuer une valeur qu'il pense correcte afin de résoudre le sudoku.

Si elle ne l'est pas, cela signifie que la case était initialement affectée à une valeur fixe (différente de 0) lors de la génération de la grille de jeu. Le joueur ne peut pas modifier cette valeur, puisqu'elle constitue un indice permettant de résoudre le sudoku, généré par la grille de jeu initiale.

L'aide correspond à la liste des valeurs possibles pour la case, c'est-à-dire la liste des valeurs encore affectables à la case en l'état actuel de la grille, en respectant les règles mathématiques du sudoku.

Chaque bit (unité de numération la plus simple) peut prendre les valeurs 0 ou 1.

Si le bit a pour valeur 1, cela signifie que la valeur correspondant à la place de ce bit dans la suite (de 1 à 16 au maximum pour les grilles 16*16) est possible.

Si le bit a pour valeur 0, cela signifie que la valeur correspondant à la

place de ce bit dans la suite est impossible, car elle ne respecterait pas une des règles mathématiques du sudoku.

Nous avons donc choisi d'utiliser une structure pour modéliser une case de grille afin de stocker chacune des informations précédentes au sein de chaque case correspondante. Cela nous sera utile plus tard.

Choix de la taille de la grille

L'utilisateur a le choix entre trois tailles de grilles possibles, que nous avons naïvement associées à trois niveaux de difficulté possibles. Il peut choisir le premier niveau de difficulté (grille 4*4), le deuxième niveau de difficulté (grille 9*9) ou le troisième niveau de difficulté (grille 16*16).

Nous avons implémenté une fonction permettant de choisir la taille de la grille.

Une sortie standard affiche les possibilités données à l'utilisateur, puis on récupère sa saisie.

Nous avons sécurisé la saisie afin de ne pas permettre à l'utilisateur d'entrer autre chose que l'information demandée. Pour cela, nous utilisons les codes ASCII (une norme de codage) de la saisie de l'utilisateur avant de les convertir dans un format correspondant à la taille choisie et manipulable par le programme.

Par abus de langage, on parlera d'une taille 4 pour une grille 4*4, d'une taille 9 pour une grille 9*9 et d'une taille 16 pour une grille 16*16.

FIGURE 1 – Une grille 4*4

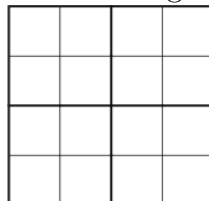


FIGURE 2 – Une grille 9*9

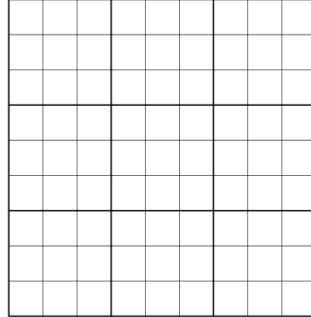
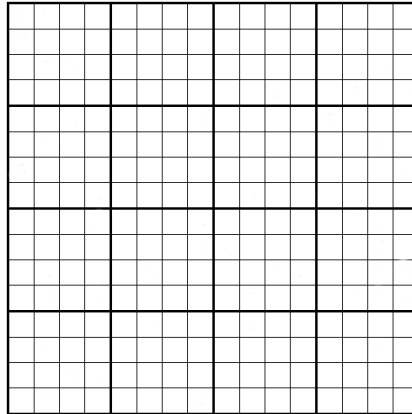


FIGURE 3 – Une grille 16*16



Initialisation de la grille

Après avoir choisi la taille de la grille, nous pouvons l'initialiser.

La grille étant un tableau à deux dimensions de cases de grille, il faut l'allouer dynamiquement dans la mémoire.

Nous avons implémenté une fonction permettant d'initialiser la grille.

On alloue d'abord la première dimension du tableau. Si l'allocation s'est bien déroulée, on peut allouer la seconde dimension. Sinon, on affiche un message d'erreur. Si l'allocation de la deuxième dimension ne s'est pas correctement déroulée, on libère cette dimension, puis on libère la grille en mémoire.

Si la grille a été correctement allouée en mémoire, on peut continuer l'initialisation.

Dans ce cas, on parcourt la grille et chaque case prend pour valeur 0, devient modifiable et peut prendre toutes les valeurs de 1 à la taille de la

grille.

Affichage de la grille

Pour afficher la grille dans un terminal, nous avons implémenté une fonction d'affichage.

En affichant les valeurs des cases, des séparateurs et des sauts de lignes, nous avons obtenu les résultats suivants :

— Pour la grille 4*4

FIGURE 4 – Affichage d'une grille 4*4

1	4		2	3	
2	3		1	4	

4	1		3	2	
3	2		4	1	

— Pour la grille 9*9

FIGURE 5 – Affichage d'une grille 9*9

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

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

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

— Pour la grille 16*16

FIGURE 6 – Affichage d’une grille 16*16

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

Pour chacune des tailles disponibles, nous avons utilisé des agencements de séparateurs et de sauts de lignes différents. Ainsi, la fonction est scindée en trois parties, chacune correspondant à une condition sur la taille différente (grille 4*4, grille 9*9, grille 16*16).

Conditions de validité de la grille

Un sudoku est un tableau à deux dimensions de cases de grilles. Il peut être scindé en lignes, colonnes ou régions.

Un sudoku est constitué de *taille* régions, c’est-à-dire constitué de *taille* groupes de *taille* cases.

Traditionnellement, un sudoku respecte trois règles principales, assurant sa validité.

Contrainte sur ligne

Deux cases situées sur la même ligne ne peuvent pas contenir la même valeur.

Nous avons implémenté une fonction permettant de vérifier qu’une valeur

est bien absente de la ligne de la case qu'on veut tester.

La fonction renvoie 0 si la valeur est présente (donc non absente), 1 sinon.

Contrainte sur colonne

Deux cases situées sur la même colonne ne peuvent pas contenir la même valeur.

Nous avons implémenté une fonction permettant de vérifier qu'une valeur est bien absente de la colonne de la case qu'on veut tester.

La fonction renvoie 0 si la valeur est présente (donc non absente), 1 sinon.

Contrainte sur région

Deux cases situées dans la même région ne peuvent pas contenir la même valeur.

Nous avons implémenté une fonction permettant de vérifier qu'une valeur est bien absente de la région de la case qu'on veut tester.

La fonction renvoie 0 si la valeur est présente (donc non absente), 1 sinon.

Génération de la grille complète (solution)

On souhaite générer une grille de sudoku complète (correspondant à la solution), de façon aléatoire.

Nous avons implémenté une fonction pour générer la grille complète.

Pour générer une telle grille, nous devons respecter les conditions de validité tout en remplissant la grille de manière aléatoire afin d'obtenir une grille différente à chaque appel de la fonction.

La génération de la grille complète repose sur le principe de retour sur trace (*backtracking*).

Le retour sur trace est un algorithme consistant à revenir en arrière sur des décisions prises, afin de sortir d'un blocage. C'est une stratégie permettant de trouver des solutions à des problèmes de satisfaction de contraintes, ce qui est l'idéal dans le cas de la génération d'un sudoku, le sudoku étant un parfait exemple de la programmation par contrainte.

En effet, générer une grille nécessite de satisfaire des conditions de validité.

Nous avons donc implémenté une fonction permettant de remplir pseudo-aléatoirement les cases de la grille initialement vide, et vérifiant à chaque

appel que le remplissage effectué est bien valide. Dès que la génération aboutit à une grille invalide, on revient en arrière et on ne prend pas l'opération qui nous a permis d'aboutir à cette grille invalide en compte.

Pour cela, il suffit d'utiliser des grilles temporaires qui nous permettent de tester aléatoirement des possibilités. On ne modifie la grille que l'on va renvoyer que lorsque l'étape de remplissage sur la grille temporaire s'est déroulée correctement. Sinon, il suffit d'effectuer une autre opération sur la grille précédente, chaque étape s'étant correctement déroulée étant enregistrée dans la grille finale.

Génération de la grille de jeu

Pour jouer, il est nécessaire de générer une grille de jeu, c'est-à-dire une grille dont certaines cases ont été vidées afin de permettre à l'utilisateur de la remplir.

Idéalement, cette grille doit permettre de trouver une unique solution (la solution générée précédemment), et elle doit comporter un nombre de cases vides suffisant pour assurer une expérience de jeu agréable.

Nous avons implémenté une fonction de génération de grille de jeu permettant de générer une grille à partir de la solution, en vidant un nombre suffisant de cases.

Néanmoins, on ne peut pas vider les cases au hasard ; il faut s'assurer de l'unicité de la solution. La seule solution possible doit être la grille complète à partir de laquelle on génère la grille de jeu.

Pour cela, il faut s'assurer qu'à chaque fois que l'on vide une case, on ne peut remplir l'ensemble des cases vides de la grille de jeu qu'avec les valeurs affectées aux mêmes cases de la grille complète.

Le problème posé se résout ainsi, en partant d'une grille complète :

1. Au départ, aucune case de la grille complète n'est nécessaire, c'est-à-dire qu'on peut effacer n'importe quelle case et toujours retrouver la solution complète initialement générée, et seulement cette solution
2. Ensuite, on choisit une case non nécessaire quelconque. Si la suppression de la case choisie conduit à plusieurs solutions, on la marque comme nécessaire sinon on la supprime

3. Si toutes les cases remplies sont nécessaires, la grille incomplète est un problème proposable, sinon on réitère l'étape précédente

On obtient ainsi une grille valide, en complexité exponentielle.

4.2 Expérience de jeu

Jeu

Nous devons ensuite permettre à l'utilisateur de jouer. Pour cela, il faut lui donner la possibilité de modifier les valeurs des cases initialement vides lors de la génération de la grille de jeu.

Nous avons implémenté une fonction permettant à l'utilisateur de jouer.

Lorsque l'utilisateur choisit l'option de jeu, il lui est possible de modifier la valeur d'une case (sous condition qu'elle soit bien modifiable). Le programme demande à l'utilisateur d'entrer la ligne de la case concernée, la colonne, puis la valeur qu'il souhaite lui affecter.

Nous avons choisi de permettre de pouvoir entrer n'importe quelle valeur de 1 à la taille de la grille.

Aide

L'utilisateur peut avoir besoin d'un indice pour progresser dans la résolution du sudoku.

Nous avons implémenté une fonction permettant à l'utilisateur de faire appel à une aide sur une case, indiquant la liste des valeurs possibles de la case en l'état actuel de la résolution.

Cette aide fait appel à trois heuristiques que nous avons également implémentées : *cross-hatching*, *lone-number* et *naked subset*.

Une heuristique est une méthode de calcul en optimisation combinatoire fournissant rapidement une solution réalisable, pas nécessairement optimale.

Nous avons choisi d'utiliser des heuristiques pour l'aide et de manipuler les bits (l'aide dans la structure de case de grille, une suite de 16 bits correspondant aux 16 valeurs possibles au maximum), afin d'obtenir des résultats de façon très rapide.

Cross-hatching

L'heuristique *cross-hatching* consiste à supprimer des listes de possibilités d'une sous-grille les valeurs qui sont déjà certaines.

FIGURE 7 – Cross-hatching

1	2	2
3		
1	2	1
3		4

FIGURE 8 – Cross-hatching

1		2
3	4	
1		1
3		4

En effet, si on regarde l'exemple précédent, le "2" étant une valeur certaine, on sait que l'on peut le retirer de toutes les autres listes de la sous-grille étudiée.

Nous avons implémenté cette heuristique en complexité $O(n^2)$ en utilisant deux boucles, l'une parcourant les listes pour trouver les singletons et la seconde pour les retirer des autres listes.

Lone-number

L'heuristique *lone-number* consiste à fixer une valeur à partir du moment où celle-ci n'appartient qu'à une seule des listes de possibilités d'une sous-grille.

FIGURE 9 – Lone-number

1	2	2
3		
1	2	1
3		4

FIGURE 10 – Lone-number

1	2	2
3		
1	2	
3		4

En effet, l'exemple qui suit nous montre que la dernière liste contient l'élément "4", celui-ci n'apparaissant pas ailleurs dans la sous-grille, on peut ainsi retirer les autres valeurs de la dite liste.

Nous avons aussi implémenté cette heuristique en $O(n^2)$ en utilisant deux boucles, l'une parcourant les listes pour trouver une valeur unique à la sous-grille, et une seconde permettant de retirer les autres valeurs.

Naked-subset

L'heuristique *naked subset* consiste à retirer de certaines listes de possibilités des valeurs appartenant à des N-uplets apparaissant N fois dans d'autres listes (N étant un entier non nul inférieur à la taille de la grille).

FIGURE 11 – Naked subset

1	5 6	5 6	2
	5 6		

FIGURE 12 – Naked Subset

1	5 6	2
5 6		

L'exemple précédent fait apparaître deux fois les 2-uplets "5 6". On sait donc qu'ils ne peuvent être présents que dans ces listes de possibilités et ainsi les retirer des autres listes.

Cette heuristique est codée en $O(n^3)$ en utilisant 3 boucles. La première parcourant toutes les listes de possibilités en cherchant si la liste en question apparait autant de fois dans la sous-grille que le nombre d'élément qu'elle contient. La seconde parcourant toutes les autres listes pour vérifier que ce n'est pas la même liste, et la troisième pour retirer les éléments du N-uplet.

Sauvegarde

Il peut être utile de sauvegarder sa progression pendant le déroulement de la partie afin de pouvoir la poursuivre plus tard.

Nous avons choisi d'utiliser trois fichiers de sauvegarde, un pour chaque taille de grille, dans le but de pouvoir jouer sur plusieurs grilles à la fois.

La fonction de sauvegarde enregistre tous les paramètres de la partie en cours.

Nous avons utilisé une sauvegarde dans un fichier binaire pour protéger la sauvegarde d'éventuelles modifications extérieures au programme.

Dans un premier temps on regarde la taille de la grille et on crée un fichier de sauvegarde correspondant à la taille (création d'un fichier *sudo[taille].bin*).

On sauvegarde ensuite la grille complète et la grille de jeu (sauvegarde case par case).

Enfin on sauvegarde le temps déjà passé sur le jeu, afin de pouvoir maintenir un classement.

Chargement

Après sauvegarde d'une partie, l'utilisateur peut choisir de charger une

partie tout en étant dans le programme, ou de charger une partie au démarrage de l'application.

Dans le cas où le joueur est encore dans l'application, s'il choisit de charger une partie, la partie chargée sera la dernière partie sauvegardée.

Dans le cas où le joueur est encore dans l'application, il peut aussi choisir de revenir au menu de départ pour charger n'importe quelle partie préalablement sauvegardée (une maximum par taille de grille).

Dans le cas où le joueur vient de lancer l'application c'est le cas précédent qui s'applique.

Le programme demande alors à l'utilisateur quelle est la taille de la grille qu'il souhaite charger.

Après sélection de la taille, la grille complète et la grille de jeu sont initialisées grâce à la taille récupérée précédemment, puis ces grilles sont remplies case par case grâce au fichier de sauvegarde.

Enfin on charge le temps de jeu écoulé et on relance le compteur de temps. La grille jouable est affichée et le jeu reprend.

Réinitialisation

Il est parfois utile pour l'utilisateur de réinitialiser la grille de jeu pour recommencer le remplissage à zéro lorsqu'il se rend compte qu'il a fait trop d'erreurs.

Nous avons implémenté une fonction permettant de réinitialiser la grille de jeu à son état initial.

Il nous suffit de parcourir l'ensemble de la grille et de remettre à zéro chaque case modifiable, sans toucher aux cases non modifiables.

Cette opération se réalise très simplement puisque le caractère modifiable ou non modifiable d'une case est directement stocké dans sa structure.

Validation

Après avoir rempli la grille de jeu, lorsque l'utilisateur pense avoir trouvé la solution, il souhaite savoir s'il a correctement rempli la grille.

Nous avons implémenté une fonction permettant de comparer deux grilles de sudoku entre elles.

Si les valeurs des cases de la grille de jeu correspondent aux valeurs des

cases de la grille complète, alors elles sont bien identiques.

On utilise cette fonction lorsque l'utilisateur choisit l'option de validation du programme. Si la grille qu'il a remplie correspond à la solution de cette même grille, alors il a résolu le sudoku. Un message s'affiche, et il peut choisir de quitter le jeu. Sinon, un message lui indiquant qu'il n'a pas correctement rempli la grille s'affiche, et il peut continuer.

Classement

Nous avons implémenté une fonction permettant de mettre à jour un classement des meilleurs scores.

Le sudoku ne permettant pas de réaliser à proprement parler de scores, nous avons décidé de réaliser un classement relatif au temps de résolution des sudokus. Le classement est divisé en trois parties (un classement par taille de grille).

Solveur de grilles incomplètes

Nous avons également implémenté une fonction permettant de résoudre des grilles incomplètes entrées en énoncé.

Cette fonctionnalité est en supplément, et permet à l'utilisateur d'afficher la solution d'un énoncé, ce qui est utile par exemple lorsqu'il joue à un sudoku papier dont il souhaite vérifier la solution après remplissage.

L'utilisateur peut choisir de jouer avec cette grille, ou d'en afficher la solution.

5 Manuel d'utilisation

5.1 Terminal

Le sudoku peut se jouer sur console, via le terminal.

La ligne de compilation est la suivante : *gcc sudoku.c fonctions.c -lm -ansi -pedantic -Wall -O3 -std=c99 -o sudoku.*

Puis, on exécute à l'aide de la commande suivante : *./sudoku.*

Le terminal est alors vidé, afin de masquer les ordres de compilation.

L'utilisateur a alors le choix entre trois options :

1. Commencer une partie
2. Charger une partie
3. Top10
4. Recuperer une grille
5. Quitter

Il doit entrer le numéro correspondant à son choix.

S'il décide de quitter, l'exécution de l'application se termine.

Sinon, s'il décide de commencer une nouvelle partie, de charger une partie préalablement sauvegardée, d'accéder au top 10 ou de récupérer une grille, il a accès à un nouveau menu.

On lui demande alors quel niveau de difficulté il souhaite choisir, dans tous les cas. Le niveau de difficulté correspond naïvement à la taille de la grille de jeu (4*4, 9*9 ou 16*16).

Le top 10 est affiché dans le cas du choix 3.

Le nom du fichier où la grille doit être récupérée est demandé dans le cas du choix 4. Si ce fichier existe bien et correspond à la taille de grille choisie, la grille récupérée s'affiche et on propose à l'utilisateur de la résoudre. S'il accepte, la solution s'affiche, sinon il peut jouer avec cette grille.

Dans les deux autres choix, la grille de jeu est alors générée (dans le cas d'une nouvelle partie) ou récupérée (dans le cas d'un chargement).

L'utilisateur a ensuite accès à un nouveau menu (que l'on appellera par commodité *menu 2*) comportant les options suivantes :

1. Jouer
2. Aide
3. Vérification
4. Sauvegarde
5. Chargement
6. Réinitialisation
7. Nouvelle partie
8. Top10
9. Récupérer une grille
10. Quitter

Il doit entrer le numéro correspondant à son choix.

S'il décide de quitter, l'exécution de l'application se termine.

S'il décide de jouer, il pourra modifier la valeur d'une case en entrant le numéro de la ligne, le numéro de la colonne de la case concernée, puis sa nouvelle valeur. Si la case est modifiable, la grille sera affichée avec la nouvelle valeur modifiée puis on réitère le choix à partir du menu 2.

S'il décide de faire appel à l'aide, il pourra choisir la case sur laquelle il souhaite afficher la liste des valeurs possibles en entrant le numéro de la ligne puis le numéro de la colonne de la case concernée. L'aide est affichée, puis on réitère le choix à partir du menu 2.

S'il décide de vérifier sa résolution, il lui suffit de choisir l'option correspondante. S'il a trouvé la solution, on lui donnera la possibilité de quitter l'application (en tapant 'o' pour quitter, 'n' pour rester). S'il choisit de rester, on réitère le choix à partir du menu 2.

S'il décide de sauvegarder, la grille en cours est enregistrée, puis on réitère le choix à partir du menu 2.

S'il décide de charger une partie, on lui demande de choisir un niveau de difficulté et la grille correspondante préalablement sauvegardée est chargée. Elle s'affiche, puis on réitère le choix à partir du menu 2.

S'il décide de réinitialiser la partie, la grille actuelle est remise à son état originel, à la génération de la grille de jeu.

S'il décide de lancer une nouvelle partie, il est renvoyé au choix de la difficulté, puis la nouvelle grille de jeu s'affiche et on reitère le choix au menu 2.

S'il décide d'accéder au top10 ou de récupérer une grille, le déroulement des opérations suivantes est analogue à celles correspondantes dans le premier menu.

5.2 Interface graphique

Les menus sont analogues à ceux du terminal, sauf que l'interface graphique rend les choix cliquables à l'aide de boutons et permet un affichage plus élaboré.

6 Perspectives et conclusions

6.1 Perspectives

L'application pourrait être améliorée en ajoutant les fonctionnalités suivantes :

- La gestion du niveau de difficulté des grilles, relative à la méthode de résolution employée pour la résoudre (notamment la gestion des grilles diaboliques)
- La génération de grilles symétriques (pour une question d'esthétisme), suivant une symétrie axiale ou centrale

Nous aurions également souhaité mettre en place une interface graphique plus évoluée, avec un design plus travaillé. Néanmoins, nous avons réalisé une interface simple et fonctionnelle.

6.2 Conclusion

Nous avons rencontré des difficultés concernant la génération de la grille de jeu à partir de la grille complète. En effet, la complexité de l'algorithme réalisé est exponentielle, et le temps d'exécution pour les grilles 16×16 suffisamment long pour poser un problème dans le cadre d'une expérience de jeu agréable. Il n'est pas acceptable de faire patienter indéfiniment l'utilisateur.

Tout le reste est entièrement fonctionnel et suffisamment optimisé pour permettre une utilisation agréable.

Les outils choisis pour réaliser le projet ont été correctement choisis, facilitant grandement le travail de groupe, notamment l'élaboration d'un cahier des charges et d'un diagramme de Gantt.

Le choix du sujet n'a pas été anodin, la plupart des membres souhaitant approfondir l'apprentissage du langage C que nous ne voyions pas durant le semestre, et certains souhaitant participer à la réalisation d'un jeu.

L'analyse préalable du sujet a été une étape cruciale dans la réalisation. Même après une analyse poussée, il a fallu poursuivre l'analyse tout au long du développement, étant confrontés à des problèmes auxquels nous n'avions pas pensé au départ, ou qui nous avaient paru plus simples que ce qu'ils étaient en réalité. C'est l'étape principale de la réalisation du projet, loin

devant l'implémentation en elle-même.

En effet, même si nous avons abordés de nombreux concepts vus en cours, nous avons également approfondi l'enseignement reçu en Licence 2, notamment concernant les heuristiques et le retour sur trace.

Il n'a pas toujours été aisé de manipuler des concepts qui nous étaient au départ inconnus, notamment en ce qui concerne l'optimisation combinatoire. Néanmoins, nous avons su nous adapter grâce à l'analyse approfondie réalisée préalablement.

Le groupe a été très efficace. La plupart des membres de l'équipe se connaissant déjà, il a été facile d'élire un chef de projet compétent qui a su organiser le groupe, distribuer les tâches et motiver l'équipe.

L'entraide était de mise tout au long du projet. Malgré la grande hétérogénéité des membres de l'équipe au début du projet, les membres les plus aguerris ont aidé ceux qui l'étaient moins, et grâce à leur ténacité et à leur motivation, les membres les moins expérimentés ont très rapidement progressé.

Tous les membres de l'équipe ont un ressenti très positif concernant le projet, qui s'est déroulé sans accroche concernant l'entente, l'investissement et la motivation du groupe.

De nombreux membres de l'équipe n'avaient jamais programmé en dehors des travaux imposés lors de la formation, et ce projet représentait une grande source de progrès. Le niveau général en programmation a largement augmenté.

Nous avons donc réussi à acquérir de nouvelles compétences, à développer une application en commun et à s'entraider, afin de réaliser le projet dans les délais impartis.

Globalement, nous sommes satisfaits de notre projet, même s'il est toujours possible d'ajouter des améliorations ou d'optimiser certaines parties, nous avons été efficaces et nous avons abordé toutes les fonctionnalités demandées.

7 Documents d'analyse

Ci-après, un autre rapport de projet disponible sur internet, qui nous a aidés à réaliser nos propres heuristiques et que nous avons trouvé suffisamment clair pour vouloir l'inclure dans les documents d'analyse de notre propre rapport.

Projet de Programmation C

BESLAY Cyril

13 Novembre 20009

Règles du Sudoku et Ensembles préemptifs

Une grille de Sudoku est une grille de taille $n^2 \times n^2$. Cette grille est séparée en n^2 blocs de taille $n \times n$ (voir Figure 1).

Une solution, pour être valide, doit vérifier cette définition : “Chaque colonne, chaque ligne et chaque bloc doit contenir toutes les couleurs de l’ensemble.” Un ensemble est constitué de couleurs disponibles pour remplir la grille. Par exemple, l’ensemble d’une grille 9×9 est l’ensemble des chiffres $[1, 2, 3, 4, 5, 6, 7, 8, 9]$.

Dans le cas de grilles de taille supérieure à 9×9 , on introduit des lettres.

Pour résoudre les grilles, on utilise les ensembles préemptifs. Un ensemble préemptif est un ensemble composé d’éléments de l’ensemble de la grille. Seuls les éléments étant des solutions possibles sont dans cet ensemble. Chaque cellule de notre sudoku contient donc un ensemble préemptif.

Lorsque l’on applique des algorithmes qui suppriment les choix impossibles dans les ensembles préemptifs. Le sudoku est résolu quand tous les ensembles ne contiennent plus qu’une seule solution.

Pour notre projet, nous avons utilisé des ensembles préemptifs sous la forme d’entiers de 32 bits. Les 24 premiers bits correspondent aux couleurs, 1 pour possible, 0 pour impossible. Le bit 25 sert à définir si la couleur était définie dans le Sudoku de départ.

De cette manière, les calculs sur les ensembles préemptifs sont simplifiés et demandent peu d’opérations (calculs en binaire).

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

Figure 1

Heuristiques

Une heuristique est une méthode qui donne une solution réalisable mais pas optimale, contrairement aux algorithmes. Les heuristiques sont utilisées pour des problèmes NP-complets.

Nous divisons notre grille de Sudoku en sous-grilles (par ligne, colonne et bloc) pour exécuter les heuristiques sur toutes les sous-grilles.

Ici, le rôle des heuristiques est de supprimer de chaque sous-grille les solutions impossibles en appliquant des méthodes différentes que j'explique dans la suite.

Cross-hatching

L'heuristique du cross-hatching consiste juste à identifier toutes les cellules qui sont résolues (une seule couleur) et enlève la couleur de toutes les autres cellules de la sous-grille. Par exemple, si la sous-grille est $\{1234, 2, 123, 14\}$, alors cette heuristique retire tous les 2 de la sous-grille et donne $\{134, 2, 13, 14\}$.

J'ai codé cette heuristique en utilisant deux boucles *for*. La première parcourt la sous-grille à la recherche de singletons (une seule couleur dans le pset). Lorsqu'elle en trouve un, la deuxième boucle supprime cette couleur de tous les autres psets grâce à la fonction

Cette heuristique est en $O(n^2)$.

Lone-number

L'heuristique du lone-number veut que si une couleur est représentée une unique fois, alors c'est la bonne couleur. Par exemple, si la sous-grille est $\{123, 2, 123, 124\}$, alors cette heuristique valide l'ensemble qui contient la seule occurrence de 4 et donne la sous-grille $\{123, 2, 123, 4\}$.

J'ai codé cette heuristique en utilisant la fonction *exclusive_union*. Dès qu'on rencontre un pset, on le compare aux autres en enlevant à chaque fois les couleurs qui apparaissent. Si une couleur n'était présente que dans le pset initial, elle se retrouve comme singleton à la fin et on peut donc la valider.

Cette heuristique est en $O(n^2)$.

N-possible ou Naked subset

L'heuristique du N-possible implique que si un nombre N de couleurs est représenté N fois dans une sous-grille, alors ce sont les seuls endroits où ces couleurs sont possibles, et on peut donc les supprimer des autres ensembles. Par exemple la sous-grille $\{156, 56, 2, 56\}$ devient $\{1, 56, 2, 56\}$ car la paire 56 apparaît deux fois.

Pour implémenter cette heuristique, j'ai ajouté une nouvelle fonction aux psets, *pset_count*. Cette fonction compte le nombre de couleurs présentes dans un pset. Cela évite la répétition de codes, puisque cette fonction est souvent utilisée plus tard.

Naked subset compte le nombre N de couleurs dans un pset, et si ce pset apparaît N fois dans la sous-grille, on enlève ces couleurs des autres pset.

Cette heuristique est en $O(n^3)$.

Solveur

Cette partie explique le raisonnement utilisé pour résoudre les grilles.

Structure du solveur

La première partie de la fonction *grid solver* applique les heuristiques à toutes les sous-grilles. Les fonctions *enumerate...* servent à partager la grille en sous-parties. Une boucle *while* permet de résoudre la grille au maximum. Tant que des changements sont effectués sur la grille, on continue à appliquer les heuristiques.

La deuxième partie est le back-tracking expliqué ci-dessous. La combinaison des ces deux méthodes permet de résoudre toutes les grilles consistantes.

Back-tracking

Lorsque les heuristiques n'effectuent plus de changements, deux cas se présentent. Soit la grille est résolue et on peut sortir de la fonction. Soit il reste des cellules non résolues. Dans ce dernier cas, nous utilisons la technique de *back-tracking*. Cette méthode permet de finir de résoudre la grille en faisant des choix aléatoires.

Voici l'algorithme utilisé :

- sauvegarde de la grille
- choix du pset le plus petit non singleton
- choix d'une couleur
- trois cas se présentent alors :
 1. grille résolue \Rightarrow retourne *true* et la grille résolue
 2. grille non résolue \Rightarrow on refait un choix
 3. grille inconsistante \Rightarrow on remonte et on fait un autre choix

Le problème est que si toutes les couleurs ont été utilisées sans succès dans un pset, plus aucune solution n'est possible. On arrête donc la recherche et on retourne *false*.

Pour le choix du pset dans le backtracking, j'ai choisi de prendre le dernier plus petit. En effet, ce parcours demande un peu de temps, mais raccourci au final le temps global de résolution. En prenant le plus petit pset, on limite le nombre de tests pour chaque niveau de back-tracking. En prenant le dernier, on accélère la résolution, puisque si on tombe sur un mauvais choix, le solveur s'en rendra compte plus vite.

Lors du choix de la couleur dans le pset, deux choix se présentaient, soit choisir au hasard, soit choisir le premier. Pour avoir des résultats plus constants, j'ai choisi la deuxième option, les deux étant équivalentes en terme d'efficacité.

J'ai aussi choisi de m'arrêter la résolution à la première solution trouvée (sauf pour la génération stricte, que j'expliquerais plus tard).

Génération de grilles

Nous voici donc au projet proprement dit : la génération de grille. Le but est de générer des grilles de Sudoku avec un taux de cases remplies entre 25% et 50%.

Principe général

La fonction *generate_grid* commence donc par initialiser une grille vide (avec tous les psets pleins). On choisit alors une couleur au hasard pour la dernière cellule et on résoud la grille. Grâce au backtracking, une grille aléatoire de taille voulue est créée. On supprime ensuite un nombre de cellules suffisant pour que la grille soit vraisemblable. De même, le choix des cellules à supprimer doit être aléatoire.

Option -s (unique solution)

L'option -s doit permettre de générer des grilles de la même façon que précédemment mais avec une contrainte supplémentaire : la grille ne doit comporter qu'une seule solution.

Pour cela, je stocke toutes les cases de la grille dans un tableau. Puis je choisis au hasard une case dans le tableau. Je supprime cette case dans la grille (après l'avoir sauvegardée). Je lance la fonction de résolution en qui retourne *false* s'il existe plusieurs solutions. Si cette grille n'admet qu'une seule solution, je continue, sinon je remets la cellule supprimée précédemment.

Cet algorithme s'arrête quand plus de 50% des cases ont été supprimées ou quand toutes les cellules ont été testées de cette manière (le tableau est vide).

Voici l'algorithme général de génération résumé :

1. on génère une grille aléatoirement
2. on initialise une liste de cases à effacer avec toutes les cases du Sudoku
3. on choisit au hasard une case dans cette liste (et on l'enlève de la liste)
4. on efface, dans la grille, la valeur dans cette case
5. si le Sudoku n'a pas une solution unique, alors on remet la valeur effacée
6. si la liste est vide ou si assez de cellules sont vides, c'est fini, sinon on va en 3

Implémentation

J'avais choisi au début une liste chaînée pour stocker les cellules à choisir au hasard. Mais le parcours de la liste prenait du temps. La gestion grâce à un tableau s'est avérée plus rapide et moins coûteuse en mémoire. Par contre, deux sauvegardes de la grille sont nécessaires et cela prend beaucoup de place.

Tests

J'ai effectué de nombreux tests notamment avec *time* et *gprof* pour l'efficacité du programme et *valgrind* pour la gestion de la mémoire.

J'ai analysé avec *gprof* les fonctions qui prenaient le plus de temps à l'exécution. J'ai donc séparé mes heuristiques dans des fonctions pour pouvoir analyser leur efficacité.

C'est logiquement Cross-hatching qui prend le plus de temps. En effet, c'est l'heuristique qui agit le plus souvent. On rentre donc dans les boucles incluses plus souvent que pour les autres heuristiques. C'est ensuite Lone-number puis Naked-subset dans l'ordre de temps d'exécution.

Bugs

L'exécution avec l'outil *valgrind* m'a permis de chasser les fuites mémoires. Elle se situaient principalement au niveau des grilles temporaires. Ces grilles étaient allouées en mémoire, mais je ne libérais pas l'espace attribué. Une fois ce problème résolu, aucun bloc mémoire n'était perdu.

Certaines grilles (notamment des grilles de taille 25) sont longues à résoudre. Il suffit qu'un choix de backtracking soit mauvais au début et le nombre de tests à faire explose.

Suivant le choix de la couleur dans le back-tracking, le temps de résolution change pour certaines grilles. Par exemple pour la grille 25x25-02 fournie avec les tests, le fait de prendre la dernière couleur du pset au lieu de la première fait passer le temps de résolution de quelques minutes à 20 sec. Mais d'autres grilles sont du coup plus longues à résoudre. Cela dépend de la structure de la grille initiale.

Optimisation

J'ai rajouté un test d'inconsistance à chaque tour d'heuristiques, ce qui évite de faire tourner le programme pour rien et fait gagner du temps.

J'ai aussi essayé d'améliorer les heuristiques en rajoutant quelques tests qui évitent des boucles pour rien.

Lors de la génération stricte, la fonction de résolution s'arrête dès qu'elle trouve une deuxième solution, ce qui raccourcit le temps de génération.

Conclusion

Ce projet nous a permis d'apprendre le developpement de A à z d'un programme. Nous avons construit le solveur en implémentant les parties au fur et à mesure, en pouvant améliorer chaque partie en fonction des demandes.

Le projet en lui-même nous a forcé à chercher la méthode à utiliser, les algorithmes et l'implémentation les plus efficaces. Chacun a donc construit un générateur différent, alors que l'ensemble du programme reste semblable pour tout le monde.

8 Listings

Ci-après, les listings.

Référence du fichier fonctions.h

Sudoku. [Plus de détails...](#)

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <math.h>
#include <time.h>
#include <stdbool.h>
#include <string.h>
#include "color.h"
```

[Aller au code source de ce fichier.](#)

Structures de données

struct **caseGrille**

La structure "caseGrille" modelise une case d'une grille de sudoku. On utilise un uint16_t code sur 16 bits. [Plus de détails...](#)

Macros

```
#define COULEUR_BLEU printf(BLUE);
#define COULEUR_VERT printf(GREEN);
#define COULEUR_ORIGINE printf(DEFAULT_COLOR);
#define COULEUR_ROUGE fprintf(stderr, RED);
#define FLUSH_COULEUR_ORIGINE fprintf(stderr, DEFAULT_COLOR);
```

Fonctions

size_t **initTaille** ()

La fonction "initTaille" initialise la taille de la grille de sudoku. [Plus de détails...](#)

caseGrille ** **initGrille** (size_t taille)

La fonction "initGrille" initialise la grille de sudoku (qui est un tableau a deux dimensions de cases de grille de sudoku), en fonction de la taille

passee en parametre. [Plus de détails...](#)

void **afficheGrille** (**caseGrille** **grille, size_t taille)

La fonction "afficheGrille" affiche la grille de sudoku passee en parametre.

[Plus de détails...](#)

int **absentSurLigne** (uint8_t valeur, **caseGrille** **grille, size_t ligne, size_t taille)

int **absentSurColonne** (uint8_t valeur, **caseGrille** **grille, size_t colonne, size_t taille)

int **absentSurRegion** (uint8_t valeur, **caseGrille** **grille, size_t ligne, size_t colonne, size_t taille)

void **echange** (uint8_t *a, uint8_t *b)

La fonction "echange" permet d'echanger deux valeurs. [Plus de détails...](#)

void **melangerTableauValeurs** (uint8_t *tableauValeurs, size_t taille)

bool **genererGrilleComplete** (**caseGrille** **grille, size_t position, size_t taille, uint8_t *tableauValeurs)

void **genererGrilleJeu** (**caseGrille** **grille, size_t taille, uint8_t *tableauValeurs)

void **jouer** (**caseGrille** **grille, size_t taille)

La fonction "jouer" permet de modifier les valeurs associees aux cases de grille. [Plus de détails...](#)

int **comparaisonGrilles** (**caseGrille** **premiereGrille, **caseGrille** **deuxiemeGrille, size_t taille)

La fonction "comparaisonGrilles" permet de comparer deux grilles de sudoku. [Plus de détails...](#)

void **sauvegarde** (**caseGrille** **grille, **caseGrille** **grilleSolution, size_t taille, float temps)

La fonction "sauvegarde" permet de sauvegarder une partie. [Plus de détails...](#)

bool **chargerPartie** (size_t taille, **caseGrille** **grilleSolution, **caseGrille** **grille, float *temps)

La fonction "chargerPartie" permet de charger une partie. [Plus de détails...](#)

void **reinitialiserPartie** (**caseGrille** **grille, size_t taille)

La fonction "reinitialiserPartie" permet de reinitialiser la grille de jeu comme en debut de partie. [Plus de détails...](#)

bool **estInclus** (uint16_t singleton, uint16_t possibilitesCase, size_t taille)

bool **crossHatching** (**caseGrille** *sousGrille, size_t taille)

La fonction "crossHatching" permet d'eliminer un singleton des possibilites des autres cases de sa sous-grille. [Plus de détails...](#)

bool **loneNumber** (**caseGrille** *sousGrille, size_t taille)

La fonction "loneNumber" est une heuristique qui permet de transformer une aide en singleton si une de ces valeurs n'est pas presente dans les autres cases. [Plus de détails...](#)

int **nbValeurs** (unsigned short **aide**)

La fonction "nbValeurs" permet de connaitre le nombre de bits a 1 pour un unsigned short representant le nombre de valeurs d'une liste de possibilites. [Plus de détails...](#)

int **nbOccurence** (**caseGrille** *sousGrille, int indice, size_t taille)

La fonction "nbOccurence" permet de connaitre le nombre de fois qu'une liste de possibilites apparait. Si elle n'apparait qu'une fois, cela renvoie 0. [Plus de détails...](#)

bool **nakedSubset** (**caseGrille** *sousGrille, size_t taille)

La fonction "nakedSubset" permet de retirer une liste de possibilite si elle repond au Naked Subset, et de renvoyer un booleen temoignant ou non de l'application de cette regle. [Plus de détails...](#)

caseGrille ** **convertirZoneEnTableau** (**caseGrille** **grille, size_t ligne, size_t colonne, size_t taille)

La fonction "convertirZoneEnTableau" permet de convertir une zone de la grille en tableau. [Plus de détails...](#)

caseGrille ** **convertirColonneEnTableau** (**caseGrille** **grille, size_t ligne, size_t colonne, size_t taille)

La fonction "convertirColonneEnTableau" permet de convertir une colonne de la grille en tableau. [Plus de détails...](#)

void **initAide** (**caseGrille** **grille, size_t taille)

La fonction "initAide" permet d'initialiser l'aide en fonction des valeurs des cases. [Plus de détails...](#)

void **aide** (**caseGrille** **grille, const size_t taille)

La fonction "aide" permet de creer l'aide pour l'ensemble de la grille. [Plus de détails...](#)

void **parcoursGrilleAide** (**caseGrille** **grille, size_t taille, size_t ligne, size_t colonne)

La fonction "parcoursGrilleAide" permet d'appliquer les heuristiques sur l'ensemble de la grille. [Plus de détails...](#)

void **echangeInt** (int *a, int *b)

La fonction "echangeInt" permet d'echanger deux entiers. [Plus de détails...](#)

int **insérerPetiteValeur** (int *T, char noms[10][30], int val, char nom[30])

La fonction "insérerPetiteValeur" permet d'insérer une valeur val si elle est la plus petite parmi tous les elements du tableau T. [Plus de détails...](#)

void **demanderNom** (char *nom)

La fonction "demanderNom" permet de demander et de renvoyer une chaine de caracteres. [Plus de détails...](#)

void **score** (int temps, size_t taille)

La fonction "score" permet de verifier s'il y a un nouveau score. [Plus de détails...](#)

void **reinitialiserTop10** (size_t taille)

La fonction "reinitialiserTop10" permet de reinitialiser le score dans les fichiers Top10_4.txt, Top10_9.txt, Top10_16.txt. [Plus de détails...](#)

void **afficherTop10** (size_t taille)

La fonction "afficherTop10" permet d'afficher le Top 10 d'une taille de grille donnee. [Plus de détails...](#)

void **recupeGrille** (**caseGrille** **grille, size_t taille)

La fonction "recupeGrille" permet de recuperer une grille ecrite dans un fichier. [Plus de détails...](#)

void **libereGrille** (**caseGrille** **grille, size_t taille)

La fonction "libereGrille" permet de liberer en memoire les grilles allouees dynamiquement. [Plus de détails...](#)

void **purger** (void)

La fonction "purger" permet de vider le buffer. [Plus de détails...](#)

void **clean** (char *chaine)

La fonction "clean" permet d'eliminer un retour a la ligne a la fin de chaine passee en parametre et vide le buffer. [Plus de détails...](#)

size_t **saisie** (void)

La fonction "saisie" permet de renvoyer la valeur saisie par l'utilisateur. [Plus de détails...](#)

Description détaillée

Sudoku.

Auteur

Abdoulaye Diallo, Redoine El Ouasti, Simon Galand, Adrien Lamant, Pierre-Louis Latour, Charly Maeder, Pierre Ruffin, Stella Zevio

Version

0.2

Date

2014-03-26

Sudoku en C (resolution des grilles 4x4, 9x9, 16x16 et jeu).

Documentation des fonctions

```
void afficheGrille ( caseGrille ** grille,  
                   size_t      taille  
                   )
```

La fonction "afficheGrille" affiche la grille de sudoku passee en parametre.

Auteur

Abdoulaye Diallo, Stella Zevio

Paramètres

grille - la grille, qui est un tableau a deux dimensions.

taille - la taille de la grille.

```
void afficherTop10 ( size_t taille )
```

La fonction "afficherTop10" permet d'afficher le Top 10 d'une taille de grille donnee.

Auteur

Stella Zevio

Paramètres

taille - la taille de la grille.

```
void aide ( caseGrille ** grille,  
           const size_t  taille  
           )
```

La fonction "aide" permet de creer l'aide pour l'ensemble de la grille.

Auteur

Charly Maeder, Pierre Ruffin

Paramètres

grille - la grille de jeu.

taille - la taille de la grille.

```
bool chargerPartie ( size_t      taille,  
                    caseGrille ** grilleSolution,  
                    caseGrille ** grille,  
                    float *      temps  
                    )
```

La fonction "chargerPartie" permet de charger une partie.

Auteur

Abdoulaye Diallo, Adrien Lamant

Paramètres

taille - la taille des grilles.

grilleSolution - la solution de la grille de jeu (la grille complete).

grille - la grille de jeu.

temps - le temps passe a resoudre le sudoku.

Renvoie

true si la grille est chargeable

false sinon.

void clean (char * chaine)

La fonction "clean" permet d'eliminer un retour a la ligne a la fin de chaine passee en parametre et vide le buffer.

Auteur

Stella Zevio

Paramètres

chaine - la chaine passee en parametre.

```
int comparaisonGrilles ( caseGrille ** premiereGrille,  
                        caseGrille ** deuxiemeGrille,  
                        size_t      taille  
                        )
```

La fonction "comparaisonGrilles" permet de comparer deux grilles de sudoku.

Auteur

Redoine El Ouasti

Paramètres

premiereGrille - l'une des grilles que l'on veut comparer.

deuxiemeGrille - l'autre grille que l'on veut comparer.

taille - la taille des grilles.

Renvoie

1 si les grilles sont identiques.

0 si les grilles sont differentes.

```
caseGrille** convertirColonneEnTableau ( caseGrille ** grille,  
                                         size_t      ligne,  
                                         size_t      colonne,  
                                         size_t      taille  
                                         )
```

La fonction "convertirColonneEnTableau" permet de convertir une colonne de la grille en tableau.

Auteur

Charly Maeder

Paramètres

- grille** - la grille de jeu.
- ligne** - la ligne de la grille où se trouve la case où on veut l'aide.
- colonne** - la colonne de la grille où se trouve la case où on veut l'aide.
- taille** - la taille de la grille.

Renvoie

tableau - tableau correspondant a la colonne que l'on voulait convertir.

```
caseGrille** convertirZoneEnTableau ( caseGrille ** grille,  
                                     size_t      ligne,  
                                     size_t      colonne,  
                                     size_t      taille  
                                     )
```

La fonction "convertirZoneEnTableau" permet de convertir une zone de la grille en tableau.

Auteur

Charly Maeder

Paramètres

- grille** - la grille de jeu.
- ligne** - la ligne de la grille où se trouve la case où on veut l'aide.
- colonne** - la colonne de la grille où se trouve la case où on veut l'aide.
- taille** - la taille de la grille.

Renvoie

tableau - tableau correspondant à la zone que l'on voulait convertir.

```
bool crossHatching ( caseGrille * sousGrille,  
                      size_t      taille  
                      )
```

La fonction "crossHatching" permet d'eliminer un singleton des possibilites des autres cases de sa sous-grille.

Auteur

Charly Maeder, Pierre Ruffin, Stella Zevio

Paramètres

sousGrille - la sous-grille a tester.

taille - la taille des grilles.

Renvoie

true - si cross-hatching s'applique.

false - sinon.

```
void demanderNom ( char * nom )
```

La fonction "demanderNom" permet de demander et de renvoyer une chaine de caracteres.

Auteur

Stella Zevio

Paramètres

nom - Une chaine de caractere.

```
void echange ( uint8_t * a,  
              uint8_t * b  
              )
```

La fonction "echange" permet d'echanger deux valeurs.

Auteur

Stella Zevio

Paramètres

a - la premiere valeur a echanger.

b - la deuxieme valeur a echanger.

```
void echangeInt ( int * a,  
                 int * b  
                 )
```

La fonction "echangeInt" permet d'echanger deux entiers.

Auteur

Stella Zevio

Paramètres

a - le premier entier que l'on souhaite echanger.

b - le deuxieme entier que l'on souhaite echanger.

```
void initAide ( caseGrille ** grille,  
               size_t      taille  
             )
```

La fonction "initAide" permet d'initialiser l'aide en fonction des valeurs des cases.

Auteur

Charly Maeder, Pierre Ruffin

Paramètres

grille - la grille de jeu.

taille - la taille de la grille.

```
caseGrille** initGrille ( size_t taille )
```

La fonction "initGrille" initialise la grille de sudoku (qui est un tableau a deux dimensions de cases de grille de sudoku), en fonction de la taille passee en parametre.

Auteur

Stella Zevio

Paramètres

taille - la taille de la grille.

Renvoie

grille - la grille de sudoku, qui est un tableau a deux dimensions.

```
size_t initTaille ( void )
```

La fonction "initTaille" initialise la taille de la grille de sudoku.

Auteur

Abdoulaye Diallo, Stella Zevio

Renvoie

taille - la taille de la grille de sudoku.

```
int insererPetiteValeur ( int * T,  
                        char noms[10][30],  
                        int  val,  
                        char nom[30]  
                        )
```

La fonction "insererPetiteValeur" permet d'insérer une valeur val si elle est la plus petite parmi tous les éléments du tableau T.

Auteur

Stella Zevio

Paramètres

- T** - un tableau d'entier.
- noms** - Un tableau de char[30].
- val** - Un entier qui est la valeur à insérer.
- nom** - une chaîne de caractères qui est le nom à insérer.

Renvoie

- trouve - la position de la valeur à modifier
- 1 - s'il n'y a pas de valeur à modifier

```
void jouer ( caseGrille ** grille,  
            size_t      taille  
            )
```

La fonction "jouer" permet de modifier les valeurs associées aux cases de grille.

Auteur

Redoine El Ouasti, Stella Zevio

Paramètres

- grille** - la grille de jeu.
- taille** - la taille de la grille.

```
void libereGrille ( caseGrille ** grille,  
                  size_t      taille  
                  )
```

La fonction "libereGrille" permet de liberer en memoire les grilles allouees dynamiquement.

Auteur

Redoine El Ouasti, Stella Zevio

Paramètres

grille - la grille a liberer.

taille - La taille de la grille.

```
bool loneNumber ( caseGrille * sousGrille,  
                 size_t      taille  
                 )
```

La fonction "loneNumber" est une heuristique qui permet de transformer une aide en singleton si une de ces valeurs n'est pas presente dans les autres cases.

Auteur

Charly Maeder, Pierre Ruffin

Paramètres

sousGrille - la sous-grille a tester.

taille - la taille des grilles.

Renvoie

true - si lone-number s'applique.

false - sinon.


```
bool nakedSubset ( caseGrille * sousGrille,  
                  size_t      taille  
                  )
```

La fonction "nakedSubset" permet de retirer une liste de possibilité si elle répond au Naked Subset, et de renvoyer un booléen témoignant ou non de l'application de cette règle.

Auteur

Simon Galand, Pierre-Louis Latour

Paramètres

***sousGrille** - l'ensemble des listes de possibilités.

taille - la taille des grilles.

Renvoie

true - si naked subset s'applique

false - sinon

```
int nbOccurence ( caseGrille * sousGrille,  
                 int         indice,  
                 size_t      taille  
                 )
```

La fonction "nbOccurence" permet de connaître le nombre de fois qu'une liste de possibilités apparaît. Si elle n'apparaît qu'une fois, cela renvoie 0.

Auteur

Simon Galand, Pierre-Louis Latour

Paramètres

***sousGrille** - l'ensemble de listes de possibilités.

indice - l'indice de la case qu'on cherche à dénombrer.

taille - la taille des grilles.

Renvoie

nbocc - le nombre d'occurrences d'une liste de possibilités.

int nbValeurs (unsigned short **aide)**

La fonction "nbValeurs" permet de connaître le nombre de bits a 1 pour un unsigned short représentant le nombre de valeurs d'une liste de possibilites.

Auteur

Simon Galand, Pierre-Louis Latour

Paramètres

aide - la liste de possibilite a denommer.

Renvoie

nbval - le nombre de valeurs d'une liste de possibilites.

```
void parcoursGrilleAide ( caseGrille ** grille,  
                           size_t      taille,  
                           size_t      ligne,  
                           size_t      colonne  
                           )
```

La fonction "parcoursGrilleAide" permet d'appliquer les heuristiques sur l'ensemble de la grille.

Auteur

Pierre Ruffin, Charly Maeder

Paramètres

grille - la grille de jeu.

taille - la taille de la grille.

ligne - la ligne de la case où on veut l'aide.

colonne - la colonne de la case où on veut l'aide.

