

Solveurs et générateurs pour
des jeux de logique II
Mémoire

Martial DUVERNEIX, Florian GAUTIER,
Pierre LORSON, Teiki PEPIN

Client : Emmanuel Fleury
Chargé de TD : Celestin Lanterne

5 avril 2018

Table des matières

1	Contexte du projet	3
1.1	Introduction	3
1.1.1	Présentation du projet	3
1.1.2	Description des jeux de logiques	3
1.2	Description et analyse de l'existant	5
1.2.1	Analyse du Sudoku fourni	5
1.2.2	Références relatives à <i>Hashiwokakero</i>	5
1.2.3	Références relatives à <i>Inshi no heya</i>	6
2	Analyse des besoins	7
2.1	Description des besoins	7
2.1.1	Besoins fonctionnels	7
2.1.2	Besoins non fonctionnels	12
3	Inshi no Heya	15
3.1	Architecture et description du logiciel	15
3.1.1	Description de l'architecture	15
3.1.2	Fonctionnement du solveur	18
3.1.3	Exemples de bon fonctionnement du solveur	21
3.1.4	Fonctionnement du générateur	23
3.1.5	Exemples de bon fonctionnement du générateur	27
3.2	Analyse du fonctionnement et tests	29
3.2.1	Tests et interprétation de leur résultat	29
3.2.2	Analyse de fonctionnement et profilage du logiciel	36
3.2.3	Défauts du logiciel	47
4	Hashiwokakero	50
4.1	Architecture et description du logiciel	50

4.1.1	Description de l'architecture	50
4.1.2	Fonctionnement du solveur	52
4.1.3	Exemples de bon fonctionnement du solveur	55
4.1.4	Fonctionnement du générateur	57
4.1.5	Exemples de bon fonctionnement du générateur	58
4.2	Analyse du fonctionnement et test	59
4.2.1	Réponse aux besoins fonctionnels du logiciel	59
4.2.2	Réponse aux besoins non fonctionnels du logiciel	60
4.2.3	Tests et interprétation de leur résultat	62
4.2.4	Défauts et bugs du logiciel	64
5	Conclusion	66
5.1	Évolutions futures	66
5.1.1	Extensions possibles pour Inshi no Heya	66
5.1.2	Extensions possibles pour Hashiwokakero	67
5.2	Conclusion générale	69
6	Bibliographie	70

Chapitre 1

Contexte du projet

1.1 Introduction

1.1.1 Présentation du projet

Ce projet consiste à concevoir des programmes pouvant générer et résoudre des instances des jeux de logique *Hashiwokakero* et *Inshi no heya*. De manière similaire au *Sudoku*, ces deux jeux proviennent du journal japonais *Nikoli* et sont structurés sous forme d'une grille qui se complète à l'aide d'un crayon.

Dans le cadre de ce projet, on souhaite généraliser les méthodes implémentées pour la résolution et la génération de telle façon que la grille de jeu puisse avoir des dimensions variables.

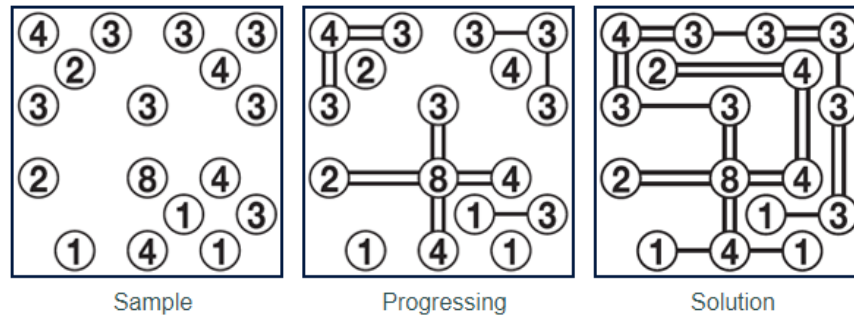
De plus, la résolution de ces puzzles devra être optimisée. Il sera nécessaire d'utiliser des méthodes proches de celles des solveurs SAT : il faut déterminer une valuation de variables booléennes exprimants les contraintes de la grille de jeu telle que toutes les contraintes soient satisfaites.

1.1.2 Description des jeux de logiques

Hashiwokakero [Nik18] est composé d'îles contenant des chiffres allant de 1 à 8. Il faut relier toutes les îles par des "ponts" pour former un seul groupe (figure 1.1). Les ponts doivent être en ligne droite, ne peuvent pas se croiser entre eux et ne peuvent pas survoler une île. Chaque île doit avoir exactement

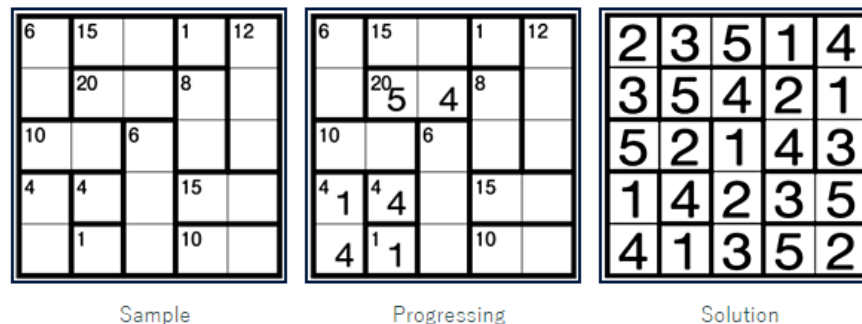
autant de ponts que le nombre qu'elle indique. Dans la variante originale du jeu, deux îles ne peuvent pas être connectées directement entre elles par plus de deux ponts.

FIGURE 1.1 – Étapes de résolution du puzzle *Hashiwokakero*.



Inshi no heya [Nik14] contient des "salles" délimitées par des traits épais avec un nombre écrit en haut à gauche. Ces salles ont l'une de leurs dimensions (la largeur ou la longueur de la salle) fixée à une case. Le but est de remplir la grille en insérant des nombres de 1 à N (où N est la taille de la grille) dans chaque case de façon à ce que le produit de tous les nombres d'une salle soit égal au nombre indiqué originalement dans cette salle (figure 1.2). Un nombre présent dans une case ne peut pas être répété dans une autre case de la même rangée ou de la même colonne.

FIGURE 1.2 – Étapes de résolution du puzzle *Inshi no Heya*.



Nous avons choisi deux jeux plutôt différents afin d'augmenter la diversité des structures de données et algorithmes mis en place au sein du projet. Cela

nous permet aussi de fournir au client deux exemples de générations et résolutions de problèmes distincts ce qui pourrait être utile pour de prochains projets similaires.

1.2 Description et analyse de l'existant

1.2.1 Analyse du Sudoku fourni

Nous disposons dans le cadre de ce projet de code source en C similaire à celui qui est attendu de nous pour la résolution et la génération de grilles de Sudoku. Ce code nous a été directement fourni par le client dans le but de nous guider. Il met en place la grande majorité des fonctionnalités que nous devons implémenter mais dans le contexte de la résolution du Sudoku.

On y observe notamment :

- L'utilisation de `uint64_t` pour stocker les valeurs des cellules. Ce type provenant de la bibliothèque standard `stdint.h` permet de fixer le nombre de bits qui seront utilisés par une variable.
- L'implémentation d'une liste chaînée d'une structure conservant chacune la grille d'origine et les modifications qui lui ont été apportées à chaque instant. Cela permet d'effectuer rapidement du backtracking lorsque nécessaire.
- L'application d'opérations bit à bit au sein d'algorithmes SWAR lors du traitement des données au sein de la grille. Ceci permet d'appliquer une instruction à de multiples données en parallèle afin de réduire le nombre d'opérations nécessaires.

1.2.2 Références relatives à *Hashiwokakero*

Nous pourrions nous servir d'un court document rédigé par D. Anderson [And09] qui démontre que le problème *Hashiwokakero* est un problème NP-complet. Nous savons donc qu'il n'existe pas de moyen de trouver une solution à une grille en un temps polynomial. Nous devons donc tester un ensemble de valeurs sur la grille et effectuer un backtracking lorsque la grille devient impossible à résoudre jusqu'à obtenir une solution, ou alors, jusqu'à que l'on puisse admettre l'absence de solution.

De même, nous disposons également d'un article de T. Morsink [Mor09] sur le puzzle *Hashiwokakero*. Celui-ci décrit :

- Les contraintes basiques qui nous permettent de connecter des îles entres elles sans avoir recours au backtracking.
- Les étapes de résolution effectuées par le solveur réalisé par l'auteur.
- Les étapes de génération pour implémenter un générateur de nouvelles grilles.
- Une comparaison des temps pris par le générateur de l'auteur, son solveur et un autre solveur externe pour générer et résoudre des grilles de taille allant de 7x7 à 15x15.

Afin de représenter les îles et les ponts du jeu *Hashiwokakero*, nous pourrions prendre appui sur une représentation sous forme d'arbre couvrant où chaque noeud représente une île et les ponts sont représentés par des arêtes. Cette implémentation est détaillé au sein de la publication "Rooted Tree and Spanning Tree Constraints" [PU06].

1.2.3 Références relatives à *Inshi no heya*

Inshi no heya étant un jeu bénéficiant de moins de popularité que *Hashiwokakero*, il existe peu d'ouvrages concernant les techniques de résolution ou d'autres particularités du puzzle. On peut néanmoins profiter d'un article de J. F. Crook [Cro09] qui décrit en détail l'utilisation d'ensembles préemptifs pour résoudre le Sudoku : des ensembles de valeurs sont attribués à chaque case afin de représenter les valeurs qui peuvent y être assignées. L'application de ceci est utile dans le cas de *Inshi no heya* car, tout comme le Sudoku, on peut déduire les valeurs de certaines cases en se servant des techniques de "naked pairs" et "hidden pairs" qui nous permettent de retrouver le nombre d'une case en fonction des valeurs permises par les autres cases d'une même région (rangée, colonne ou bloc).

Chapitre 2

Analyse des besoins

2.1 Description des besoins

2.1.1 Besoins fonctionnels

1. Permettre l’affichage d’une grille de jeu :
 - (a) Concevoir pour chaque puzzle un format d’affichage qui représente tous les éléments qui le compose et qui soit lisible et compréhensible par un utilisateur humain.
 - (b) Implémenter pour chaque programme une fonctionnalité d’affichage permettant d’afficher le puzzle en question sur la sortie standard.
 - (c) Ces fonctionnalités sont essentielles au bon fonctionnement du programme du fait de leur nécessité pour vérifier l’exécution des autres fonctionnalités. Cela sera assuré par l’utilisation de grilles de test dont les sorties sur console et sur fichier auront été prédéterminées.
2. Permettre la lecture de grilles enregistrées dans des fichiers :
 - (a) Définition d’un format de représentation des grilles pour chaque jeu optimisé pour la lecture par ordinateur (figure 2.1). Ce format peut donc être différent du format de représentation destiné à l’affichage.
 - (b) Permettre à chaque programme de lire les fichiers du format qui lui correspond et de convertir les données lues en une structure de

données utilisable par le reste du programme via un parseur que nous implémenterons.

- (c) De part la nature du programme, la lecture des fichiers est un besoin essentiel de ce projet. Le bon fonctionnement de cela sera aussi assuré par la lecture de grilles dont les valeurs qui devront être lues par le programmes seront prédéterminées.

FIGURE 2.1 – Prototype de format de fichier pour *Inshi no Heya* et sa grille correspondante.

6	4		5	40
	3		4	
	15	40		
4	10		6	
			3	

1	6b2
2	4r2
3	5
4	40b3
5	3r2
6	4b2
7	15r2
8	40b3
9	4b2
10	10b2
11	3r2
12	

3. Mettre en place, pour chaque jeu, un système de résolution de grilles :
 - (a) La résolution s'effectuera en étapes dans l'ordre suivant, similaire en exécution à un solveur SAT, afin d'améliorer sa rapidité :
 - Détermination des éléments évidents de la grille de jeu.
 - Construction des variables de la grille dont les valeurs restent à déterminer.
 - Recherche d'une valuation de ces variables qui valide la grille.
 - (b) La résolution devra proposer une solution à chaque grille qui lui est fournie, et ce, que la grille admette une unique solution ou plusieurs.
 - (c) La résolution devra se terminer et afficher un message correspondant si la grille fournie n'admet aucune solution.
 - (d) La résolution devra se terminer et afficher un message correspondant si la grille fournie est dans un format incorrect.
 - (e) Il sera possible d'afficher à chaque étape de la résolution l'état actuel de la grille.

- (f) La validité de la résolution sera vérifiée à l'aide de grilles de tests dont les résultats auront été prédéterminés.
 - (g) La résolution est un point essentiel du programme. Le suivi des étapes n'est pas nécessaire au bon fonctionnement du programme, il est considéré comme conditionnel.
4. Mettre en place, pour chaque jeu, un système de génération de grilles :
- (a) Le programme générera par défaut une grille ayant au moins une solution qu'il affichera à la sortie standard.
 - (b) L'utilisateur aura la possibilité de décider si la grille générée ne doit admettre qu'une unique solution.
 - (c) Permettre la sauvegarde des grilles générées dans des fichiers au même format que celui utilisé pour la lecture de grilles en entrée.
 - (d) Diverses grilles générées seront résolues à la main afin de vérifier la validité de la génération.
 - (e) La génération des grilles et leur sauvegarde étant réutilisées dans la résolution sont des fonctionnalités essentielles. La génération d'une grille avec une unique solution est une amélioration de la génération, elle est conditionnelle.
5. Mettre en place une taille de grille variable pour chaque jeu :
- (a) Définir les dimensions minimums et maximums qui seront supportées par les structures de données utilisées.
 - Pour *Hashiwokakero*, le minimum valide est une grille 3x1 (ou bien 1x3). Le maximum dépendra de notre implémentation mais une taille idéale serait de 64x64.
 - Pour *Inshi no heya*, le minimum valide est une grille 1x1. Le maximum sera fortement limité par les types utilisés au sein de l'implémentation car la valeur attribuée à une salle est, au pire des cas, $N!$ où N est la taille de la grille. On remarquera que un *unsigned long long* peut généralement stocker des valeurs allant jusqu'à $1.84 * 10^{19}$, ce qui est inférieur à $21!$. On essaiera donc à priori de limiter la taille des grilles, ou bien la taille des salles qui les composent, à 20 cases.
 - (b) Détecter automatiquement les dimensions des grilles fournies aux programmes.

- (c) Permettre de choisir les dimensions désirées lors de la génération d'une nouvelle grille.
 - (d) La robustesse du programme doit être assurée : il ne doit pas être possible d'aller au delà des bornes déterminées pour les dimensions.
 - (e) Pour chacun des programmes, des grilles de test correspondant aux limites des dimensions seront utilisées afin de vérifier la correspondance de la résolution à des résultats préalablement déterminés.
 - (f) Pour chacun des programmes, la validité des grilles générées aux limites des dimensions sera vérifié par la résolution de ces grilles.
 - (g) Ces fonctionnalités de scalabilité sont essentielles.
6. Exécuter plusieurs threads afin de répartir les tâches effectuées lors de la résolution et la génération de grilles. Ceci a été vu comme une fonctionnalité supplémentaire pouvant être ajoutée au projet afin de l'améliorer au delà de ce qui est attendu.
7. Chaque programme devra disposer de plusieurs options d'exécution :
- (a) Comportement par défaut (3.1.3) :
 - Prend un paramètre : le nom du fichier contenant la grille à résoudre.
 - Lance la résolution d'une grille contenue dans un fichier et affiche en sortie standard la grille initiale puis la grille résolue.
 - Ceci est, bien évidemment, essentiel au projet.
 - (b) *-generate* (3.1.4) :
 - Prend un paramètre : la taille de la grille à générer.
 - Permet de générer une nouvelle grille avec au moins une solution dont la taille est déterminée par l'utilisateur.
 - Est compatible avec toutes les autres options d'exécution.
 - Cette fonctionnalité est un des points clés du programme et est donc essentielle.
 - (c) *-strict* (3.1.4b) :
 - Force la grille générée à n'avoir qu'une unique solution.
 - Cette option doit être utilisée avec l'option de génération et est compatible avec toutes les autres options d'exécution.

- Tel que vu précédemment, cette fonctionnalité est conditionnelle.
- (d) *-save*(3.1.4c) :
- Prend un paramètre : le nom du fichier de sortie.
 - Si le format de fichier d'entrée est différent de celui utilisé pour l'affichage de grilles, cette option permet d'enregistrer une grille générée au format d'entrée.
 - Cette option doit être utilisée avec l'option de génération et est compatible avec toutes les autres options d'exécution.
 - Cette fonctionnalité est essentielle à implémenter pour les tests de génération.
- (e) *-output* (3.1.1c) :
- Prend un paramètre : le nom du fichier de sortie.
 - Permet d'enregistrer un affichage compréhensible de la grille résolue ou générée dans un fichier sans l'afficher au préalable en sortie standard.
 - Est compatible avec toutes les autres options d'exécution.
 - Cette fonctionnalité est essentielle.
- (f) *-verbose* (3.1.3e) :
- Permet d'afficher chaque étape de la résolution ou de la génération de la grille.
 - Est compatible avec toutes les autres options d'exécution.
 - Tel que vu précédemment, cette fonctionnalité est conditionnelle.
- (g) *-help* :
- Permet d'afficher des informations sur le programme et ses options d'exécution.
 - Cette option peut être utilisée avec toute autre option.
 - Ceci apporte des informations supplémentaires non requis pour l'utilisation du programme, son implémentation est donc conditionnelle.
- (h) *-version* :

- Permet d’afficher la version courante du programme.
- Cette option peut être utilisée avec toute autre option.
- Ceci apporte des informations supplémentaires non nécessaires à l’utilisation du programme, son implémentation est donc conditionnel.

2.1.2 Besoins non fonctionnels

1. Un besoin essentiel du projet est que chacun des deux jeux de logique fera l’objet d’un fichier exécutable permettant de réaliser les opérations de résolution et de génération correspondant à son jeu.
2. Les grilles de jeu pouvant être de grande taille (3.1.5), la résolution (3.1.3) et la génération (3.1.4) doivent être optimisées :
 - (a) Utiliser un langage de programmation faible en consommation de mémoire et efficace en temps de calcul. Il a donc été convenu avec le client lors du premier entretien que les programmes seraient implémentés en C dans la mesure du possible.
 - (b) Mettre en place des algorithmes de résolution optimaux afin de déterminer tous les éléments certains d’une grille en moins d’une seconde.
 - (c) Utiliser de manière intensive des algorithmes SWAR. Cela désigne le fait d’appliquer une seule instruction sur de multiples données contenu dans un registre. Dans notre cas, on utilisera ces algorithmes pour réduire le temps de calcul en regroupant des données et en utilisant une seule instruction sur celles-ci ce qui est plus rapide que de répéter la même instruction au sein d’une boucle.
 - (d) Il faut se servir de structures de données concises utilisant des types optimisés pour du traitement bit à bit tels que *uint64_t* et *uint_fast32_t*.
 - (e) On doit nécessairement utiliser un langage de programmation efficace. Des comparaisons seront établies entre les différents algorithmes implémentés afin de déterminer leur efficacité.
 - (f) Il est essentiel d’utiliser des algorithmes de résolution optimaux et des algorithmes SWAR ainsi qu’utiliser des structures de données concises. L’utilisation de plusieurs threads est optionnelle.

3. S'assurer de la robustesse de la lecture des fichiers d'entrée (3.1.2) :
 - (a) En cas d'erreur mineure dans le fichier qui n'empêche pas sa compréhension, permettre de continuer la lecture en corrigeant le défaut.
 - (b) En cas d'erreur majeure dans le fichier qui rend la grille invalide, afficher un message d'erreur avant de mettre fin au programme.
 - (c) Ce besoin n'étant pas nécessaire à l'exécution du programme, il est conditionnel au projet.
4. Implémenter pour chaque programme la possibilité d'enregistrer la représentation de la grille dans un fichier, particulièrement pour les cas où la grille aurait des dimensions excessives pour s'afficher correctement en console. Ce besoin est conditionnel.
5. Chaque programme devra respecter un "coding style" demandé par le client :
 - L'affichage en général, les noms de variables, les noms de fonctions ainsi que les commentaires doivent être en anglais uniquement,
 - Utiliser des tabulations à 8 espaces,
 - Utiliser des indentations à 2 espaces,
 - Ne pas dépasser 80 colonnes par ligne,
 - Les noms de variable et de fonction doivent respecter un format avec des underscores tel que : `my_name`, `my_second_name`,
 - Commenter en utilisant le format `/* comment */` et non `// comment`,
 - Entourer les opérateurs binaires ou ternaires d'espaces : `+`, `-`, `*`, `=`, `<`, `>`, `...`,
 - Ne pas entourer les opérateurs unaires d'espaces : `!`, `-`, `++`, `...`,
 - Ne pas utiliser d'espace autour des opérateurs d'accès à une structure : `a.b` et `a->b`,
 - Mettre un espace après un mot clé : `if`, `else`, `for`, `while`, `...`,
 - Utiliser la même règle pour les marqueurs de bloc : `"` et `"`,
 - Les macros doivent être en majuscule : `#define MACRO 10`,
 - Déclarer les variables de boucle dans la définition de la boucle autant que possible tel que : `for (type i = 0; i < MAX_SIZE; ++i)`.

6. La réalisation du projet est tenue respecter le calendrier prévisionnel de l'UE Projet de Programmation :
- (a) Une première version du livrable a été fournie pour le 16 février.
 - (b) Un audit lors du 28 février.
 - (c) Une version finale du livrable ainsi que ce mémoire devront être fournis le 5 avril.
 - (d) Une soutenance aura lieu le 11 avril.

Chapitre 3

Inshi no Heya

3.1 Architecture et description du logiciel

3.1.1 Description de l'architecture

Le code source du programme permettant la résolution et la génération de grille pour Inshi no Heya est réparti sur trois modules : `inshiNoHeya.c`, `composites.c` et `composites.h`, et `bitset.c` et `bitset.h`.

Bitset

Le module `bitset` définit des fonctions permettant de faire des opérations bit à bit sur des `uint64_t`, des entiers de 64 bits définis dans la bibliothèque standard `stdint.h`. Il permet ainsi, pour un bit d'un rang spécifique, de mettre sa valeur à 1 ou à 0, d'inverser sa valeur, ou encore de tester et renvoyer sa valeur. Ce module nous permet aussi de faire la conjonction logique, la disjonction logique, la disjonction exclusive et la négation logique d'`uint64_t` qui lui sont fournis. Ces fonctionnalités basiques offre un niveau d'abstraction aux autres modules lorsqu'ils nécessitent l'usage d'opérations sur des bits spécifiques.

De plus, d'autres fonctions plus complexes sont présentes dans `bitset` : une fonction permettant de compter le nombre de bits à 1 parmi les 64 bits, une fonction permettant de trouver le rang du bit à 1 ayant le poids le plus faible, une fonction permettant de renvoyer le rang du seul bit à 1 dans un `uint64_t` fourni et cinq fonctions différentes permettant toutes, pour un en-

tier de 64 bits donné, de renvoyer le nombre de bits à 1. Le dénombrement de bits à 1 étant un processus utilisé régulièrement au sein de la résolution et de la génération de grilles, différents algorithmes avec des complexités de temps différents sont fournis afin de permettre un profilage de leur performance. Deux de ces cinq fonctions reposent sur des algorithmes SWAR et l'une des fonctions est tout simplement un wrapper pour une fonction interne de GCC. Ces cinq algorithmes renvoient des résultats identiques pour tout `uint64_t` valide.

Composites

Le module `composites` est responsable de la décomposition de nombres entiers en facteurs entiers. Il propose pour cela une structure de données `composites_set_t` qui est une liste simplement chaînée où chaque élément de la liste contient un `uint64_t` nommé `factors` et un pointeur `next` qui pointe sur le prochain élément de la liste ou sur `NULL` à défaut d'élément suivant.

Pour chaque instance de `composites_set_t`, les 64 bits de `factors` représentent une combinaison de facteurs distincts permettant de former un certain produit qui a été fourni en paramètre lors de sa construction. Le rang des bits à 1, allant de 1 à 64, indique les valeurs à multiplier afin d'obtenir le produit original. De cette manière, il est impossible d'avoir un facteur qui serait non-entier, supérieur à 64, ou inférieur à 1. Toutes les instances d'une même liste chaînée sont des décompositions différentes d'un même produit.

Le module s'occupe des toutes les opérations concernant cette structure de données : factorisation, modification, tri et suppression dans la liste chaînée, affichage des composants, constructeur et destructeur.

InshiNoHeya

Enfin, le module principal `inshiNoHeya` est en charge de l'aspect puzzle du programme. Il met en place les algorithmes de résolution et de génération de grilles, de lecture et d'écriture de fichiers, et gère au sein du `main` les arguments fournis aux programmes. Pour cela, il utilise deux structures de données : `grid_t` et `cell_t`.

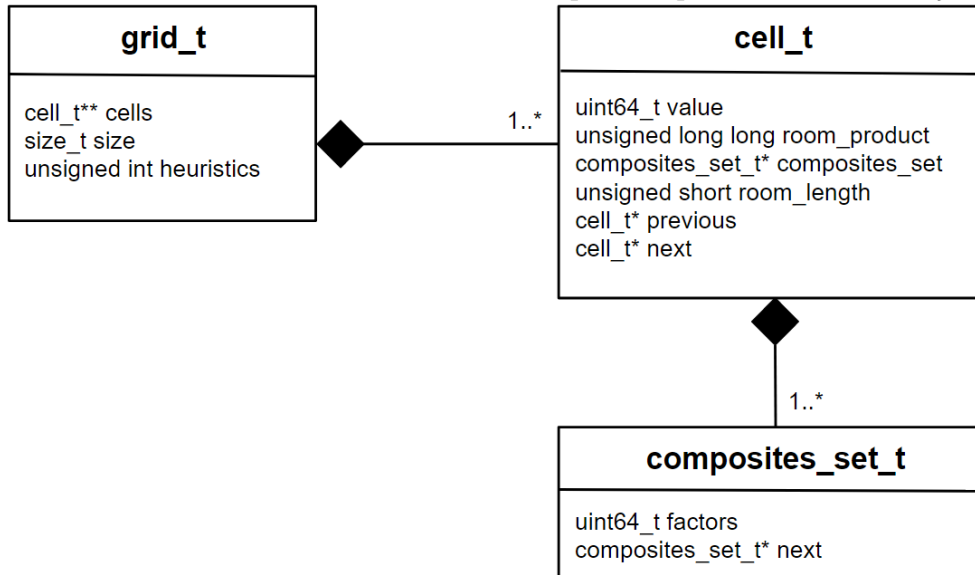
La première, `grid_t`, est la structure qui permet de représenter la grille sur laquelle le programme travaille. À tout instant lors de l'exécution du

programme, il n'existe qu'une seule instance de cette structure. Elle est composée des attributs `size`, qui désignent la taille de la grille, d'un entier non signé nommé `heuristics`, qui permet de compter le nombre de cases ayant été résolues sans utilisation de backtracking, et d'un tableau de pointeurs sur des `cell_t`, qui nous permet d'accéder à chaque case de la grille. Nous disposons donc au sein de la grille de `size * size` pointeurs sur des `cell_t`.

La deuxième structure, `cell_t`, stocke les informations correspondant à une case de la grille. Pour cela, elle utilise `value`, un `uint64_t` qui représente la valeur actuelle ou l'une des valeur potentielle de la case, `room_product`, un `unsigned long long` représentant le produit que la salle dont cette case fait partie doit former, `composites_set` un pointeur vers une structure `composites_set_t` qui permet de former ce produit, `room_length`, un `unsigned short` qui désigne la longueur de la salle, et enfin `previous` et `next`, des pointeurs sur des `cell_t` qui pointent respectivement vers la case précédente et la case suivante au sein de la même salle. Dans le cas où la case représenté se situe en extrémité de sa salle, le pointeur correspondant a alors la valeur `NULL`.

Les relations entre ces structures de données et `composites_set_t` sont illustrées au sein de la figure 3.1.

FIGURE 3.1 – Architecture des données pour le puzzle *Inshi no Heya*.



3.1.2 Fonctionnement du solveur

Exécution du solveur

Afin de faire fonctionner le solveur, il est nécessaire de lancer l'exécution du programme avec comme premier argument le nom d'un fichier contenant la grille à résoudre. On peut ensuite fournir d'autres paramètres parmi les suivants : "-o [fichier]" pour écrire la grille résolue dans un fichier au lieu de la console et "-v" pour activer la verbose ce qui affiche toutes les étapes de résolution de la grille dans la console ou le fichier de sortie.

Lecture de la grille

Le solveur va donc commencer par lire la grille à partir du fichier fourni via la fonction `read_file`. Ce fichier doit respecter un format que nous avons défini : la première ligne du fichier désigne la taille de la grille, soit "5x5" ou tout simplement "5", les lignes suivantes désignent chacune une salle composant la grille. Pour chaque salle, on y écrit le produit que la salle doit former, sa direction sous forme d'une lettre (on a "b" pour bottom qui désigne une salle verticale et "r" pour right qui désigne une salle horizontale) et sa longueur. Les salles sont listées en fonction de leur disposition dans la grille de haut en bas et de gauche à droite. Ainsi, pour la grille de taille 5 illustrée par la figure 3.2, on a le fichier représenté par la figure 3.3.

		15	1	
6		20		
	10		8	12
	4			15
4	1	6		10

FIGURE 3.2 – Grille représentée

```

1 5x5
2 6b2
3 15r2
4 1r1
5 12b3
6 20r2
7 8b2
8 10r2
9 6b3
10 4b2
11 4r1
12 15r2
13 1b1
14 10r2

```

FIGURE 3.3 – Format du fichier

Les valeurs lues sont ainsi enregistrées dans notre `grid_t` et la répartition des cases dans leur salles respectives est effectuée. Chaque case commence initialement avec une valeur par défaut de 0, soit une valeur qui sera absente de toute grille résolue. Pour chaque salle est alors généré un `composites_set_t` qui représente les factorisations possibles pour le produit de la salle en fonction de sa taille et de la valeur maximum que peut prendre une case, soit la taille de la grille.

Factorisation

Ce processus est effectué au sein de la fonction `find_composites` du module `composites`. Celle-ci utilise la technique des divisions successives afin de déterminer deux facteurs entier permettant de former le produit de la salle. Ces facteurs sont eux-mêmes factorisés en sous-composants. Cette méthode est répétée dessus chaque facteur obtenu jusqu'à que ceux-ci ne puissent plus être décomposés. On ajoute alors toutes les combinaisons possibles de facteurs à une liste chaîne de `composites_set_t`.

Le facteur '1' ne changeant pas le résultats des factorisations, celui-ci n'est ajouté qu'à la fin et uniquement aux ensembles pour lesquels le nombre de facteurs correspondra donc à la taille de la salle. De manière similaire, on supprime tous les ensembles pour lesquels le nombre de facteurs ne correspond pas à la taille de la salle. Ensuite, on trie les ensembles restants par un tri à bulles ce qui nous permet de facilement supprimer les doublons présents. On obtient donc une liste affinée des ensembles de facteurs correspondant à la salle.

Backtracking

Si la lecture s'est déroulée correctement, on exécute alors `solve_grid` qui entame alors le processus de résolution. Cette fonction fait d'abord appel à `fill_single_cells`. Il s'agit d'une fonction qui se contente de parcourir chaque case de la grille et si la salle à laquelle celle-ci appartient ne contient aucune autre case, alors on assigne le produit de la salle à la case, ceci étant la seule valeur que la case peut prendre.

Notre solveur fait ensuite appel à `solve_cell` sur la toute première case de la grille. Cette fonction représente le corps principal de la résolution. En effet, pour la case sur laquelle elle a été appelée, elle va parcourir tous les ensembles de facteurs possibles qui peuvent former le produit de sa salle et, pour chaque ensembles de facteurs, elle va parcourir les facteurs qui le composent. À chaque fois qu'elle trouve un nouveau facteur, elle vérifie alors si ce facteur est déjà présent sur la rangée ou sur la colonne. Dans un tel cas, le facteur est invalide et on passe donc au suivant. De même, on vérifie également que le facteur choisi s'accorde avec les autres facteurs déjà présents au sein de la salle pour former le produit désiré.

Lorsque l'on trouve un facteur validant la case, on avance alors à la prochaine case de la grille par un appel récursif et on répète les mêmes opérations. Si la fonction valide la dernière case de la grille, la grille est donc entièrement remplie et résolue. Si la fonction ne trouve aucun facteur permettant de valider la case, on peut en déduire qu'un facteur choisi préalablement dans la grille empêche la résolution, on retourne donc à la case précédente où on recherche d'autres facteurs permettant de valider la grille. Si la toute première case de la grille ne trouve pas ou plus de facteurs permettant de se valider, c'est donc que la grille n'est pas résoluble.

Affichage

Une fois le processus de résolution effectué, on affiche alors, en fonction du niveau de verbosité, la grille résolue ou, le cas échéant, un message approprié. En fonction des arguments fournis au programme, il est possible que au cours de la résolution et à la fin, la grille soit écrite au sein d'un fichier spécifié par l'utilisateur. Enfin, la grille et toutes les structures de données dont elle est composée sont libérées de la mémoire.

3.1.3 Exemples de bon fonctionnement du solveur

Exemple de résolution basique

FIGURE 3.4 – Exemple de résolution d'une grille de taille 8x8 valide.

```
tepepin@trelawney:~/PdP/logicgamespace/trunk/InshiNoHeya/src$ ./inshiNoHeya 8x8_Custom
 2 | 5 | 4 | 6 | 1 | 3 | 8 | 7 |
--|---|
 4 | 7 | 3 | 1 | 6 | 2 | 5 | 8 |
--|---|
 3 | 4 | 5 | 7 | 8 | 1 | 6 | 2 |
--|---|
 6 | 8 | 2 | 5 | 3 | 4 | 7 | 1 |
--|---|
 7 | 1 | 8 | 4 | 5 | 6 | 2 | 3 |
--|---|
 8 | 6 | 7 | 3 | 2 | 5 | 1 | 4 |
--|---|
 5 | 3 | 1 | 2 | 7 | 8 | 4 | 6 |
--|---|
 1 | 2 | 6 | 8 | 4 | 7 | 3 | 5 |
--|---|
The grid is valid.
tepepin@trelawney:~/PdP/logicgamespace/trunk/InshiNoHeya/src$
```

```
1 8x8
2 24b3
3 5r1
4 60b3
5 42b3
6 24r3
7 7r1
8 224b3
9 6r1
10 80r3
11 120b3
12 6r2
13 6b3
14 6r1
15 112b3
16 5r1
17 24b2
18 14b3
19 280b3
20 18b3
21 4r1
22 30r3
23 4r1
24 6b2
25 2r1
26 224r3
27 30b2
28 1r1
29 2r1
30 224r3
31 3r1
```

La figure 3.4 illustre un exemple d'utilisation du solveur. La commande `./inshiNoHeya 8x8_Custom` est utilisée dans le terminal afin de lancer l'exécution du programme sur le contenu du fichier "8x8_Custom" qui est présenté dans la partie droite de la figure. Une fois la résolution terminée, la grille résolue est alors affichée dans la sortie standard avec un message

indiquant que la grille est valide. Le contour de la grille et de chaque salle est représenté par les caractères '-' et '|'. Seule la valeur courante de chaque salle est affichée.

Exemple de résolution avec paramètres

FIGURE 3.5 – Exemple de résolution d'une grille de taille 5x5 valide.

```
tepepin@trellawney:~/PdP/logicgamesolver/trunk/InshiNoHeya/src$ ./inshiNoHeya ../test/5x5_Rapport -o resultats.txt -v
The grid is valid.
Heuristics made : 3
tepepin@trellawney:~/PdP/logicgamesolver/trunk/InshiNoHeya/src$
```

254	-----									
255	2		3		5		1		4	
256	---					---				
257	3		5		4		2		1	
258	---					---				
259	5		2		1		4		3	
260	---					---				
261	1		4		2		3		5	
262	---					---				
263	4		1		3		5		2	
264	-----									

1	5x5
2	6b2
3	15r2
4	1r1
5	12b3
6	20r2
7	8b2
8	10r2
9	6b3
10	4b2
11	4r1
12	15r2
13	1b1
14	10r2

La figure 3.5 illustre un exemple d'utilisation du solveur avec cette fois-ci des arguments supplémentaires. La commande `./inshiNoHeya 5x5_Rapport -o resultats.txt -v` est utilisée dans le terminal. En plus de la lecture du fichier `5x5_Rapport`, cette commande utilise l'option `-v` pour activer la verbosité et l'option `-o` pour faire sortir la grille résolue et toutes les grilles intermédiaires dans le fichier `resultats.txt`. La validité ou non de la grille ainsi que le nombre d'heuristiques employés au cours de la résolution sont affichés en sortie standard.

Exemple de résolution sur une grille invalide

FIGURE 3.6 – Exemple de résolution d'une grille de taille 4x4 invalide.

```
tepepin@trellawney:~/PdP/logicgamesolver/trunk/InshiNoHeya/src$ ./inshiNoHeya ../test/4x4_Rapport
Invalid
The grid is invalid.
tepepin@trellawney:~/PdP/logicgamesolver/trunk/InshiNoHeya/src$
```

1	4x4
2	24r4
3	23r4
4	24r4
5	24r4

La figure 3.6 illustre un exemple d'utilisation du solveur avec une grille qui n'admet aucune solution. Après avoir essayé de résoudre la grille, le programme affiche en sortie standard un message signalant que la grille n'est pas valide avant de se terminer.

3.1.4 Fonctionnement du générateur

Exécution du générateur

De manière similaire au solveur, le générateur est exécuté à l'aide de l'option de lancement : `"-g [taille]"`. L'option de verbosité `"-v"` peut ici aussi être utilisée afin d'obtenir des détails lors du fonctionnement du programme. De même, l'option `"-o [fichier]"` peut être employée afin de faire sortir la grille dans un fichier au lieu de la console. Des options supplémentaires peuvent y être ajoutées, celles-ci seront détaillées ci-dessous. La taille fournie en argument doit être inférieure ou égale à la limite définie dans le code source par `MAX_GRID_SIZE`.

Initialisation

La fonction `generate_grid` est en charge la génération d'une nouvelle grille. On y crée initialement une grille vide de la taille `N` désirée. Pour chacune des cases, on met pour sa valeur, qui est un entier sur 64 bits, les `N` premiers bits à 1 et tous les autres à 0. Le rang des bits à 1 indiquent ici les valeurs que chaque case peut prendre. Ainsi, les cases d'une grille 5x5 auront pour représentation bit à bit `"00000000 [...] 00011111"` car elles peuvent prendre pour valeur les entiers $\{1,2,3,4,5\}$.

Backtracking

Ensuite, la fonction `randomize_cell_value` est appelée sur la première case de la grille. Cette fonction compte, via le module `bitset`, le nombre de valeurs que la case peut prendre et en choisie au hasard une parmi celles disponible. Les bits correspondant à d'autres valeurs sont donc mis à zéro au sein de la valeur de la case et le bit correspondant à la valeur choisie est mis à zéro au sein des valeurs des autres cases de la rangée et de la colonne. En

effet, cette valeur devenant donc assignée à cette case, il devient impossible qu'elle réapparaisse dans la même rangée ou colonne.

On passe alors, par un appel récursif, à la case suivante de la grille et on effectue les mêmes opérations. Dans le cas où on arriverait sur une case qui n'aurait aucun bit à 1, cela impliquerait que la case ne peut prendre aucune valeur qui permettrait de générer une grille valide. Il est donc nécessaire de revenir à la case précédente afin de lui attribuer une valeur différente, cela autant de fois que nécessaire pour que la grille soit résoluble.

Lorsque l'on revient d'un appel récursif infructueux, on remet pour les cases de la rangée et de la colonne à 1 le bit correspondant à la valeur qui a été essayée au cours de cette itération et on tente, toujours de manière aléatoire, de lui attribuer une nouvelle valeur parmi celles qu'il peut assumer. Lorsqu'une case n'a plus aucune valeur qui puisse lui être attribuée, on retourne à la case précédente.

Eventuellement, une grille valide sera trouvée lorsque la dernière case de la grille aura validée sa valeur. À ce moment, l'`uint64_t` correspondant à la valeur de chaque case aura un seul et unique bit à 1 et le rang de ce bit correspondra à sa valeur. La fonction `convert_bits_to_values` est alors appelée, celle-ci remplace, pour chaque case, le bit unique à la valeur qui lui est équivalente grâce à des fonctions implémentées au sein de `bitset`.

Délimitation des salles

Chaque case de la grille ayant maintenant une valeur, on fait alors appel à la fonction `random_rooms`. Cette fonction parcourt chaque case de la grille et, pour chaque case n'ayant pas encore de salle qui lui soit attribuée, on lui génère une salle de direction et de longueur aléatoire. Pour cela, une constante, `"percent_to_not_expand"`, est ici utilisée. Elle vaut par défaut 20%.

Pour chaque case, on tire un nombre entre 1 et 2 de manière aléatoire afin de déterminer la direction (horizontale ou verticale) de la salle qui lui sera attribuée. Ensuite, un nombre tiré aléatoirement entre 1 et 100 est alors comparé à la constante mentionnée ci-dessus. Dans le cas où le nombre aléatoire est inférieur ou égale à la constante, la salle conserve sa longueur initiale qui

est de 1 case. Dans le cas où le nombre aléatoire est supérieur à la constante, la taille de la salle est incrémentée de 1 et on répète le tirage au sort.

Cette expansion aléatoire des salles continue jusqu'à que le nombre aléatoire soit inférieur à la constante, ou que la salle atteigne l'extrémité de la grille, ou que la salle atteigne une autre salle ayant déjà une salle qui lui est attribuée, ou que la salle atteigne la valeur max qu'elle puisse prendre telle que définie dans le code source.

Une fois la taille de la salle décidée, les pointeurs previous et next de chaque case sont utilisés pour connecter les cases appropriées entre elles et le produit correspondant à la salle est déduit des valeurs des cases qui la composent.

Affichage

Enfin, la grille générée est affichée en sortie standard telle qu'elle le serait si elle venait d'être résolue. Dans le cas où l'option "-o" a été employée, la grille générée est écrite dans le fichier spécifié au lieu de la sortie standard. La grille et ses données sont ensuite libérées de la mémoire avant que le programme ne se termine.

Sortie en format d'entrée

Lors du lancement du programme, il est possible d'utiliser l'option "-i [fichier]". Cette option est d'une forte importance pour le générateur car elle permet de sauvegarder la grille générée dans le format qui est utilisé par le solveur (voir figure 3.3). Ainsi, la grille résolue ne sera pas affichée mais son squelette sera sauvegardé dans le fichier désigné par l'utilisateur. Ceci correspond donc pour la première ligne aux dimensions de la grille, pour les lignes intermédiaires aux produits, direction et longueur des salles, et pour la dernière ligne un symbole de retour à la ligne ('\n') afin de faciliter la lecture.

Grille à solution unique

Il est également possible de forcer la grille générée à n'avoir qu'une seule et unique solution grâce à l'option d'exécution "-s". Ainsi, lorsque la grille

est générée, un appel à une fonction de résolution légèrement modifiée nous permet de déterminer si la grille admet une seule solution ou au moins deux solutions. Lorsque plus d'une solution à la grille est détectée, la grille courante est libérée de la mémoire et les étapes précédentes sont répétées jusqu'à que l'on obtienne une grille à solution unique.

Notamment, l'utilisation de la verbosité permet alors d'observer lorsque des grilles sont générées puis supprimées dû à leurs solutions.

Grille "linéaire"

On dispose également d'une option "-i" qui permet de faire appel aux fonctions de génération de grilles dites "linéaires". La génération de ces grilles suit les mêmes règles que pour une génération standard sauf pour l'attribution des valeurs des cases. En effet, la valeur des cases n'est plus déterminée par hasard et par backtracking mais prédéterminée. La case en haut à gauche de la grille a pour valeur '1' et toutes les autres cases ont comme valeur celle de la case précédente, en rangée et en colonne, incrémentée de 1. Lorsque l'on atteint une valeur supérieure à la taille de la grille, on reprend à partir de '1'. Une illustration d'une telle grille est disponible à la figure 3.10.

Cette fonctionnalité dispose de peu d'intérêt d'un point de vue de génération aléatoire de grille mais nous permet cependant de rapidement générer des grilles de taille importante sans que leur particularité ne fasse de différence pour notre solveur. De même, cela nous permet de tester de manière efficiente notre solveur et notre générateur car au moins l'une des solutions de la grille nous est connue d'avance. En fonction de la répartition des salles, d'autres solutions que celle initialement prévues par le générateur peuvent être possibles si celui-ci n'est pas désigné comme strict.

3.1.5 Exemples de bon fonctionnement du générateur

Génération basique

FIGURE 3.7 – Exemple de génération d’une grille de taille 5x5 avec son affichage en console.

```
tepepin@trelawney:~/PdP/logicgamesolver/trunk/InshiNoHeya/src$ ./inshiNoHeya -g 5
```

4	1	5	3	2
1	2	4	5	3
2	5	3	4	1
3	4	2	1	5
5	3	1	2	4

La figure 3.7 illustre un exemple de génération de grille sans paramètres supplémentaires. La commande `./inshiNoHeya -g 5` est utilisée dans le terminal afin de lancer l’exécution du programme. Une fois la génération terminée, la grille générée est alors affichée dans la sortie standard sous sa forme résolue. De manière similaire à la résolution, le contour de la grille et de chaque salle est représenté par les caractères ‘-’ et ‘|’.

Génération en format d’entrée

FIGURE 3.8 – Exemple de génération d’une grille de taille 5x5 au format d’entrée.

```
tepepin@trelawney:~/PdP/logicgamesolver/trunk/InshiNoHeya/src$ ./inshiNoHeya -g 5 -i grille
tepepin@trelawney:~/PdP/logicgamesolver/trunk/InshiNoHeya/src$ cat grille
5x5
4r1
6r2
8b3
5r1
6b3
5b2
5r1
3b2
8b3
3r1
20r2
5r1
4r1
6r2
tepepin@trelawney:~/PdP/logicgamesolver/trunk/InshiNoHeya/src$
```

La figure 3.8 représente la génération d'une grille de telle manière à ce quelle soit lue par la partie solveur du programme. On y ajoute pour cela "-i grille" à la commande précédente. On observe ensuite la génération correcte d'une nouvelle grille de taille 5x5.

Génération d'une grille à solution unique avec verbose

FIGURE 3.9 – Exemple de génération avec verbose d'une grille de taille 3x3 avec solution unique.

```
tepepin@trellawney:~/PdP/logicgamessolver/trunk/InshiNoHeya/src$ ./inshiNoHeya -g 3 -s -v
Generating a new 3x3 grid.
The grid has more than one solution.

Generating a new 3x3 grid.
The grid has more than one solution.

Generating a new 3x3 grid.
The grid has more than one solution.

Generating a new 3x3 grid.
Printing the grid.
|-----|
| 3   1 | 2 |
| ---  --- |
| 2 | 3 | 1 |
|   ---  --- |
| 1 | 2 | 3 |
|-----|
Finished generating the grid.
tepepin@trellawney:~/PdP/logicgamessolver/trunk/InshiNoHeya/src$
```

La figure 3.9 représente la génération d'une grille dite stricte. En effet, la grille ainsi générée n'admet qu'une unique solution. On constate également, grâce à la verbose, que plusieurs autres grilles ont été initialement générées puis supprimées car elles présentaient plus d'une solution.

Génération d'une grille "linéaire"

FIGURE 3.10 – Exemple de génération d'une grille "linéaire" de taille 9x9.

1									
2	1	2	3	4	5	6	7	8	9
3									
4	2	3	4	5	6	7	8	9	1
5									
6	3	4	5	6	7	8	9	1	2
7									
8	4	5	6	7	8	9	1	2	3
9									
10	5	6	7	8	9	1	2	3	4
11									
12	6	7	8	9	1	2	3	4	5
13									
14	7	8	9	1	2	3	4	5	6
15									
16	8	9	1	2	3	4	5	6	7
17									
18	9	1	2	3	4	5	6	7	8
19									

La figure 3.10 représente la sortie en fichier de la commande `./inshiNoHeya -g 9 -l -o grille`. On y observe ainsi la grille résolue d'une génération dite "linéaire" de taille 9x9. On constate ici que la répartition des valeurs des cases suit un motif correspondant à l'algorithme précédemment décrit : chaque case a pour valeur celle de la précédente plus un.

3.2 Analyse du fonctionnement et tests

3.2.1 Tests et interprétation de leur résultat

De nombreuses grilles de test sont fournies avec le programme afin de tester plusieurs aspects du solveur. Celles-ci sont disponibles dans un répertoire indépendant et sont accompagnées d'un fichier explicatif qui décrit l'utilité de chacune d'entre elles.

Tests de validité

FIGURE 3.11 – Extrait du makefile montrant les tests de validité effectués.

```
test_invalid: $(BIN)
    @./$(BIN) ../test/2x2_Invalid)
    @./$(BIN) ../test/3x3_Invalid)
    @./$(BIN) ../test/4x4_Invalid)
    @./$(BIN) ../test/5x5_Invalid)
    @./$(BIN) ../test/6x6_Invalid)
    @./$(BIN) ../test/7x7_Invalid)
    @./$(BIN) ../test/16x16_Invalid)
```

Plusieurs de ces grilles ne sont pas valides ou ne sont pas résolubles. On peut alors tester la robustesse et la validité du solveur à l'aide de ces grilles. Les makefiles fournis mettent à disposition une option qui nous permet d'automatiser l'exécution de ces tests tel que l'on peut le voir à la figure 3.11. Cela compile le programme puis passe au solveur sept grilles différentes sans aucun autre argument. On observe alors les résultats au sein de la figure 3.12.

FIGURE 3.12 – Résultats de l'exécution des tests de validité.

```
tepepin@trellawney:~/PdP/logicgamessolver/trunk/InshiNoHeya/src$ make test_invalid
cc -Wall -Wextra -std=gnu99 composites.c bitset.c inshiNoHeya.c -lm -o inshiNoHeya
The grid is invalid.
Error in input file format.
The grid is invalid.
Error in input file format.
The input file is invalid.
Error in input file format.
Error in input file format.
The input file is invalid.
tepepin@trellawney:~/PdP/logicgamessolver/trunk/InshiNoHeya/src$
```

En effet, la grille 2x2_Invalid présente deux salles à une seule case ayant le même produit sur la même rangée. Elle n'admet donc aucune solution qui permette de résoudre la grille. De manière similaire, la grille 4x4_Invalid contient une salle de quatre cases de long mais avec un nombre premier comme produit. La grille est donc non résoluble vu qu'elle n'admet aucune solution. Ces grilles nous permettent de tester l'efficacité de nos algorithmes de résolution. Cela prouve que le backtracking revient éventuellement à son

origine avant d'admettre l'absence de solutions.

La grille 5x5_Invalid présente le caractère 'c' là où un des caractères 'b' ou 'r' serait attendu pour diriger la direction de la grille. Ce caractère n'étant pas un caractère admis par notre parseur de fichier, la grille devient donc invalide. De même, la grille 7x7_Invalid présente des caractères aléatoires au lieu des produits et des longueurs de certaines des salles. La lecture correcte de la grille en devient donc impossible et la grille est déclarée invalide. Ceci nous permet de confirmer la robustesse de notre lecture de fichier.

La grille 3x3_Invalid est invalide vu qu'elle présente une salle de taille supérieure aux dimensions de la grille. La grille 6x6_Invalid dispose d'une salle ayant un produit valant zéro. Un tel produit serait impossible à atteindre avec des facteurs entiers strictement positifs. La grille est donc considérée comme invalide. Enfin, la grille 16x16_Invalid présente de nombreuses salles déclarées correctement. Cependant, elle dispose d'un nombre insuffisant de salles, et donc de cases, pour former une grille de taille 16x16. Elle est ainsi déclarée invalide. Cela nous permet de vérifier que les propriétés d'une grille correspondant au puzzle Inshi no Heya sont bien respectées sans avoir à appliquer inutilement l'algorithme de résolution.

Tests de résolution

FIGURE 3.13 – Extrait du makefile montrant les tests de résolution effectués.

```
inshiNoHeya: $(FILES)
    $(CC) $(CFLAGS) $(FILES) $(LDFLAGS) -o $(BIN)

test: $(BIN)
    @./$(BIN) ../test/3x3_FirstLine -o tmp)
    @./$(BIN) ../test/3x3_Generated -o tmp)
    @./$(BIN) ../test/3x3_Spacing -o tmp)
    @./$(BIN) ../test/4x4_Singles -o tmp)
    @./$(BIN) ../test/5x5_Columns -o tmp)
    @./$(BIN) ../test/5x5_Rows -o tmp)
    @./$(BIN) ../test/5x5_Rapport -o tmp)
    @./$(BIN) ../test/5x5_Wikipedia -o tmp)
    @./$(BIN) ../test/6x6_Linear -o tmp)
    @./$(BIN) ../test/8x8_Generated -o tmp)
    @./$(BIN) ../test/16x16_Linear -o tmp)
    @./$(BIN) ../test/16x16_Generated -o tmp)
    @./$(BIN) ../test/24x24_Linear -o tmp)
    @./$(BIN) ../test/24x24_Generated -o tmp)
    @./$(BIN) ../test/32x32_Linear -o tmp)
    @(rm tmp)
```

Les autres grilles de test fournies avec le programme sont toutes valides et résolubles. Le solveur devrait donc, pour chacune d'entre elles, offrir une solution. C'est ce que l'on cherche à tester avec le script de test présent au sein du makefile tel que présenté à la figure 3.13. Ce script compile initialement notre programme, puis l'exécute avec 15 grilles différentes fournies en paramètre. Les sorties de ces résolutions, c'est à dire les grilles résolues, sont écrites dans un fichier temporaire qui est supprimé à la fin de l'exécution du script. On obtient donc comme résultat la confirmation de la résolution pour chaque grille en console tel que montré par la figure 3.14.

FIGURE 3.14 – Résultats de l'exécution des tests de résolution.

```
tepepin@trelawney:~/PdP/logicgamesolver/trunk/InshiNoHeya/src$ make test
cc -Wall -Wextra -std=gnu99 composites.c bitset.c inshiNoHeya.c -lm -o inshiNoHeya
The grid is valid.
The grid is valid.
The grid is valid.
The grid is valid.
The grid is valid.
The grid is valid.
The grid is valid.
The grid is valid.
The grid is valid.
The grid is valid.
The grid is valid.
The grid is valid.
The grid is valid.
The grid is valid.
The grid is valid.
The grid is valid.
tepepin@trelawney:~/PdP/logicgamesolver/trunk/InshiNoHeya/src$
```

Les grilles 5x5_Wikipedia et 5x5_Rapport sont des grilles présentes dans le projet depuis son début dû à leur utilité pour des vérifications élémentaires et des démonstrations. Il s'agit de simples grilles 5x5 sans particularité autre que leur origine. La grille Wikipedia provient, comme son nom l'indique, de la page présentant Inshi no Heya sur le site Wikipedia. Ceci étant l'une des rares ressources concernant le puzzle disponible en ligne, il est plus facile pour un nouvel utilisateur de faire la liaison entre les ressources existant en ligne et la ressource équivalente au sein du programme. La grille Rapport est, quant à elle, la grille de démonstration utilisée au sein de l'analyse des besoins. Elle nous permet, de manière similaire, de faire la liaison entre une ressource existante dont on se sert et notre programme. Ces grilles nous permettent en outre de confirmer le fonctionnement du solveur pour des grilles de taille 5.

La grille 3x3_FirstLine présente une forme raccourcie de la spécification des dimensions de la grille sur la première ligne du fichier : "3x3" devient simplement "3". Ceci est quelque chose d'accepté par notre parseur et le reste de la grille est lu normalement. La grille 3x3_Spacing présente plusieurs sauts de lignes à divers endroits au sein du fichier. Ceci est aussi pris en compte lors de la lecture du fichier et la grille est résolue comme prévu. Ceci nous permet de confirmer la validité de notre parseur de fichier dans des situations où certaines libertés seraient prises par l'utilisateur.

La grille 4x4_Singles est une grille qui présente le cas particulier où chaque salle de la grille ne serait composée que d'une seule cellule. Au contraire, les grilles 5x5_Rows et 5x5_Columns présentent des grilles où

chaque salle a pour longueur la taille de la grille. Comme indiqué par leur noms, Rows présente uniquement des salles horizontales tandis que Columns n'en présente que des verticales. Ceci nous permet de vérifier le bon fonctionnement de notre solveur dans des situations extrêmes.

Les grilles 6x6_Linear, 16x16_Linear, 24x24_Linear et 32x32_Linear sont fournis afin d'effectuer des tests considérés comme rapides sur le fonctionnement du solveur à différentes tailles de grilles. Ces grilles ont été générées par le solveur "linéaire" tel que défini précédemment dans ce document. Elles permettent ainsi de tester le solveur avec des grilles de taille plus importante que les précédents tests. On peut ainsi observer le bon fonctionnement du solveur sur chacune de ces grilles.

Cependant, les grilles "linéaires" étant un cas particulier de grilles dû à la nature de leur construction, il est nécessaire d'appliquer le solveur à des grilles ayant une disposition plus aléatoire. C'est pour cela que l'on teste aussi les grilles 3x3_Generated, 8x8_Generated, 16x16_Generated et 24x24_Generated. Ces grilles sont générées par notre générateur sans particularités. Cela nous permet donc de tester des cas quelconques pour notre solveur.

Autres grilles de test

Trois autres grilles sont fournies avec le programme mais ne sont pas incluses dans les scripts de tests : 32x32_Generated, 48x48_Linear et 64x64_Linear. Ces grilles ne sont pas incluses aux tests automatisés à cause du temps de résolution important engendré par leur taille. La première de ces grille se résout en moins d'une heure sur un poste disposant d'un AMD Opteron Processor 6168 à 1900MHZ tandis que les deux autres sont toujours en cours de résolution une fois le délai d'une heure écoulé. Ceci nous permet donc d'évaluer l'efficacité de notre programme en fonction de la taille des données.

On constate ainsi que la complexité en temps de notre algorithme de résolution ne nous permet pas de résoudre efficacement, dans son état actuel, des grilles de taille importante, où "importante" désigne une valeur comprise entre 32 et 48. Cette valeur n'est pas quantifiable de manière précise dû à la durée nécessaire à chaque test et au besoin d'appliquer ces tests à un échantillon important de grilles pour chaque taille possible.

Tests de génération de grille

FIGURE 3.15 – Extrait du makefile montrant les tests de génération effectués.

```
test_generate: $(BIN)
    @./$(BIN) -g 4 -i tmp_input)
    @./$(BIN) tmp_input -o tmp_output)
    @./$(BIN) -g 16 -l -i tmp_input)
    @./$(BIN) tmp_input -o tmp_output)
    @./$(BIN) -g 8 -i tmp_input)
    @./$(BIN) tmp_input -o tmp_output)
    @./$(BIN) -g 3 -s -i tmp_input)
    @./$(BIN) tmp_input -o tmp_output)
    @(rm tmp_input)
    @(rm tmp_output)
```

Afin de tester la partie générateur du programme, on fournit également une automatisation de quelques tests basiques de génération dans le makefile illustré par la figure 3.15. En admettant que les tests de résolution précédents sont effectués correctement, on peut donc utiliser le solveur pour vérifier l'intégrité des grilles que nous générons au cours du test. On appelle ainsi quatre fois le générateur avec des arguments différents, et après chaque génération, on résout la grille générée pour vérifier qu'elle est correcte. On obtient alors en sortie standard, si les tests se sont déroulés correctement, quatres messages confirmant la validité des grilles générées (voir figure 3.16).

FIGURE 3.16 – Résultats de l'exécution des tests de génération.

```
tepepin@trellawney:~/PdP/logicgamessolver/trunk/InshiNoHeya/src$ make test_generate
cc -Wall -Wextra -std=gnu99 composites.c bitset.c inshiNoHeya.c -lm -o inshiNoHeya
The grid is valid.
The grid is valid.
The grid is valid.
The grid is valid.
tepepin@trellawney:~/PdP/logicgamessolver/trunk/InshiNoHeya/src$
```

La première grille générée est une grille standard de taille 4x4. Ceci nous permet de tester la génération d'une grille de petite taille, évitant ainsi les problèmes et la lenteur qui peuvent survenir à une taille importante, sans

argument changeant le comportement du générateur. On observe que la grille est correctement générée et résolue.

La deuxième grille générée est une grille de taille 16x16 dite "linéaire". Le générateur employé dans ce cas-ci étant différent du générateur standard, on peut ainsi observer la présence d'erreurs dans le cas des grilles "linéaires". On observe que la grille est correctement générée et résolue.

La grille suivante est une grille de taille 8x8 standard, similaire à la première grille générée. Ceci nous permet de constater si des défauts au sein du générateur apparaissent lorsque la taille de la grille augmente. On observe que la grille est correctement générée et résolue.

Enfin, la dernière grille générée est une grille de taille 3x3 avec une solution unique. Ceci nous permet de vérifier le bon fonctionnement de l'algorithme en charge de la génération de grilles strictes. On observe que la grille est correctement générée et résolue.

3.2.2 Analyse de fonctionnement et profilage du logiciel

L'optimisation étant un point important de ce projet, on procède à l'analyse du temps d'exécution, de la répartition de l'activité au sein des fonctions, et de l'utilisation de la mémoire. Cette analyse est appliquée au solveur ainsi qu'au générateur pour des grilles de tailles variables.

Utilisation de la mémoire du solveur

Afin d'observer la mémoire utilisée par le logiciel, on utilise l'outil valgrind. On exécute alors notre programme avec la commande "valgrind ./inshiNoHeya [fichier] [options]" et des informations concernant l'exécution du programmes nous sont données lorsque celui-ci se termine. On y observe notamment l'utilisation de la mémoire en fonction du nombre d'allocation faite, du nombre de données libérées, et du nombre d'octets utilisés tel qu'illustré par la figure 3.17.

FIGURE 3.17 – Extrait de l’affichage de valgrind avec la commande `./inshi-NoHeya -h`.

```
HEAP SUMMARY:
  in use at exit: 0 bytes in 0 blocks
  total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated

All heap blocks were freed -- no leaks are possible
```

Cependant, l’utilisation de valgrind ralentit fortement les performances du programme qui souffre déjà d’une certaine lenteur sur des grilles de tailles conséquentes. On se contente donc d’analyser la résolution de grilles standards de taille 1 à 11. Il est important de noter que les grilles utilisées pour ces tests sont générées par notre générateur. Elles sont générées de manière aléatoires mais leur génération n’est pas distribuée de manière uniforme sur toutes les possibilités qu’elles peuvent assumer.

On observe que :

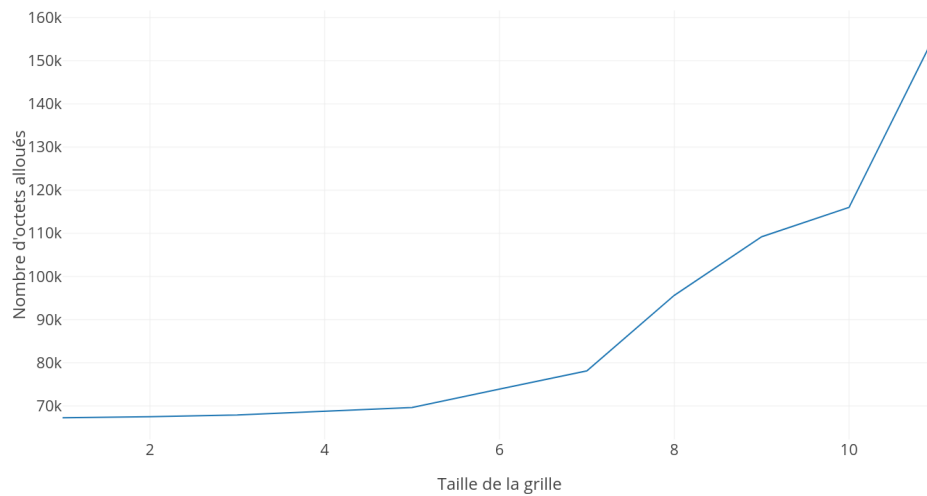
- une grille de taille 1x1 fait 8 allocations et 8 libérations,
- une grille de taille 2x2 fait 17 allocations et 17 libérations,
- une grille de taille 3x3 fait 30 allocations et 30 libérations,
- une grille de taille 4x4 fait 67 allocations et 67 libérations,
- une grille de taille 5x5 fait 98 allocations et 98 libérations,
- une grille de taille 6x6 fait 337 allocations et 337 libérations,
- une grille de taille 7x7 fait 569 allocations et 569 libérations,
- une grille de taille 8x8 fait 1625 allocations et 1625 libérations,
- une grille de taille 9x9 fait 2432 allocations et 2432 libérations,
- une grille de taille 10x10 fait 2811 allocations et 2811 libérations,
- une grille de taille 11x11 fait 5300 allocations et 5300 libérations.

On en déduit qu’aucune fuite de mémoire n’est présente dans notre algorithme de résolution et on observe de plus que valgrind ne relève aucune erreur d’accès en mémoire. Dans le contexte de ce projet où l’utilisation de la mémoire augmente exponentiellement avec la taille des grilles, il est important d’assurer une bonne gestion de la mémoire et ceci nous permet de le confirmer.

On observe aussi grâce à valgrind le nombre d’octets alloués pour chaque résolution. Ces données sont collectées et mises dans un graphe que l’on retrouve à la figure 3.18. On remarque que la résolution utilise initialement

près de 67ko et que la taille de la grille a peu d'impact sur la mémoire utilisée pour des grilles de tailles inférieures à 6 mais que une fois ce palier dépassé, la quantité de mémoire alloué au programme augmente rapidement.

FIGURE 3.18 – Courbe du nombre d'octets utilisés pour la résolution d'une grille en fonction de sa taille.



Temps d'exécution du solveur

Pour étudier l'efficacité en temps de notre solveur, on utilise la commande `"/usr/bin/time -v ./inshiNoHeya [fichier]"` qui nous permet de lancer notre programme comme on le ferait habituellement tout en affichant, à sa fin, des informations relatives à sa durée d'exécution. Une démonstration sur la résolution d'une grille est disponible à la figure 3.19.

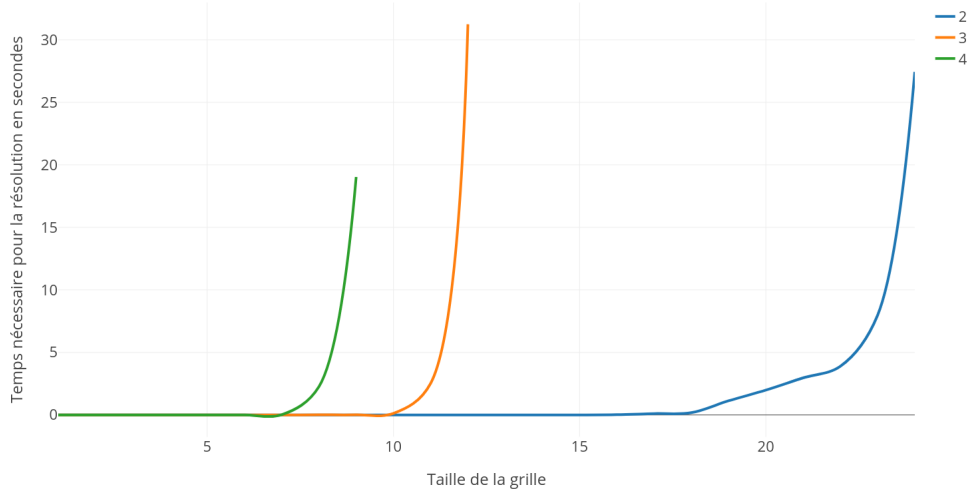
FIGURE 3.19 – Extrait de la sortie de la commande `"/usr/bin/time -v ./inshiNoHeya"` sur une grille de taille 11x11.

```
Command being timed: "./inshiNoHeya 11.grid"
User time (seconds): 2.54
System time (seconds): 0.00
Percent of CPU this job got: 99%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:02.55
```

On applique ce procédé à des grilles de tailles diverses. Il est nécessaire de remarquer que le temps de résolution d'une grille dépend principalement de deux facteurs : la taille de la grille et la taille maximum qu'une salle peut atteindre. Ces deux paramètres influencent directement le nombre d'itérations effectuées par le backtracking de l'algorithme de résolution.

On effectue donc nos tests sur de multiples grilles pour chaque taille et avec différentes limites pour la taille de la grille. Pour chaque cas, on applique le test à dix grilles correspondants aux paramètres générées par notre programme. Ces résultats sont utilisés pour générer le graphe 3.20.

FIGURE 3.20 – Graphe représentant le temps moyen nécessaire à la résolution en fonction de la taille de la grille et de la limite de taille des salles, sur un AMD Opteron Processor 6168 à 1900MHZ.



On remarque que la résolution se fait instantanément pour un certain nombre de tailles de grille en fonction de la taille maximum des salles, avant de croître exponentiellement. Cela correspond à l'augmentation du nombre d'itérations au sein du backtracking.

De même, on constate que la taille maximum qu'une salle peut atteindre influence fortement les performances du programme. Cela limite donc le choix des grilles que notre programme pourra espérer résoudre.

Pour comparaison, la figure 3.21 présente le temps de résolution nécessaire

à la résolution d'une grille de taille 32x32 ayant 2 pour longueur maximum de ses salles. Il a fallu au programme seize minutes pour arriver à résoudre une telle grille.

FIGURE 3.21 – Temps de résolution affiché en console pour une grille standard 32x32 avec pour taille limite de salle 2, sur un AMD Opteron Processor 6168 à 1900MHZ.

```
Command being timed: "./inshiNoHeya ../test/32x32_Generated"  
User time (seconds): 966.67  
System time (seconds): 0.00  
Percent of CPU this job got: 99%  
Elapsed (wall clock) time (h:mm:ss or m:ss): 16:06.96
```

Repartition de la charge de travail du solveur

Afin de déterminer où est ce que tout ce temps est consommé au sein de notre programme, on utilise l'outil gprof qui nous permet de déterminer la distribution du travail au sein des fonctions appelées par notre programme. On emploie celui-ci sur la résolution d'une grille 12x12 standard générée par notre programme et on obtient le profil présenté par la figure 3.22 et le graphe d'appels de la figure 3.23.

FIGURE 3.22 – Profil des fonctions généré par gprof pour la résolution d'une grille 12x12.

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
21.66	7.99	7.99	76632229	0.00	0.00	check_col
20.37	15.50	7.51	1	7.51	35.68	solve_cell
20.01	22.88	7.38	1643792236	0.00	0.00	bits_and
18.98	29.88	7.00	76632229	0.00	0.00	check_row
12.19	34.38	4.50	76632229	0.00	0.00	check_room
2.77	35.40	1.02	76632229	0.00	0.00	check_cell
1.19	35.84	0.44				print_bits
0.85	36.15	0.32	1982	0.00	0.00	bits_or
0.54	36.35	0.20	25686026	0.00	0.00	get_set_length
0.49	36.53	0.18				_popcountdi2
0.22	36.61	0.08	25686026	0.00	0.00	popcount_preprocessor
0.11	36.65	0.04	1	0.04	0.04	check_grid
0.08	36.68	0.03				popcount_kernighan
0.05	36.70	0.02				main
0.04	36.72	0.02	1	0.02	0.02	print_grid
0.03	36.73	0.01	5236	0.00	0.00	free_composites_set
0.00	36.73	0.00	9125	0.00	0.00	generate_set
0.00	36.73	0.00	6799	0.00	0.00	set_bit
0.00	36.73	0.00	3889	0.00	0.00	step_next_set
0.00	36.73	0.00	2583	0.00	0.00	merge_and_append
0.00	36.73	0.00	404	0.00	0.00	find_subsets
0.00	36.73	0.00	70	0.00	0.00	add_one_to_sets
0.00	36.73	0.00	70	0.00	0.00	delete_duplicate_sets
0.00	36.73	0.00	70	0.00	0.00	filter_sets_by_length
0.00	36.73	0.00	70	0.00	0.00	find_composites
0.00	36.73	0.00	70	0.00	0.00	sort_sets
0.00	36.73	0.00	42	0.00	0.00	set_rightward_room
0.00	36.73	0.00	28	0.00	0.00	set_downward_room
0.00	36.73	0.00	1	0.00	0.00	check_grid_size
0.00	36.73	0.00	1	0.00	0.00	fill_single_cells
0.00	36.73	0.00	1	0.00	0.00	free_grid
0.00	36.73	0.00	1	0.00	0.00	initialize_grid
0.00	36.73	0.00	1	0.00	36.06	process_solve
0.00	36.73	0.00	1	0.00	0.33	read_file
0.00	36.73	0.00	1	0.00	35.68	solve_grid

FIGURE 3.23 – Extrait du graphe d’appels des fonctions généré par gprof pour la résolution d’une grille 12x12.

[4]	97.1	0.00	35.68	1/1	process_solve [2]
		0.00	35.68	1	solve_grid [4]
		7.51	28.16	1/1	solve_cell [3]
		0.00	0.00	1/1	fill_single_cells [34]

		0.00	0.00	144/76632229	check_grid [21]
		1.02	19.48	76632085/76632229	solve_cell [3]
[5]	55.8	1.02	19.48	76632229	check_cell [5]
		7.99	0.00	76632229/76632229	check_col [6]
		7.00	0.00	76632229/76632229	check_row [8]
		4.50	0.00	76632229/76632229	check_room [9]

		7.99	0.00	76632229/76632229	check_cell [5]
[6]	21.8	7.99	0.00	76632229	check_col [6]

		0.00	0.00	3318/1643792236	merge_and_append [13]
		7.38	0.00	1643788918/1643792236	solve_cell [3]
[7]	20.1	7.38	0.00	1643792236	bits_and [7]

		7.00	0.00	76632229/76632229	check_cell [5]
[8]	19.1	7.00	0.00	76632229	check_row [8]

		4.50	0.00	76632229/76632229	check_cell [5]
[9]	12.2	4.50	0.00	76632229	check_room [9]

On observe dans le profil que la grande majorité du temps de calcul est dédié aux fonction solve_cell, check_cell, check_col, check_row, check_room et bits_and. Le graphe d’appel nous confirme en effet que 97% du temps de résolution est passé au sein de la fonction de backtracking de solve_cell et que plus de la moitié de ce temps y est passé au sein de la fonction check_cell qui est en charge de vérifier la validité d’une attribution de valeur à une case en faisant elle-même appel aux fonctions check_col, check_row et check_room.

Cela représente un total de 56% du temps de résolution qui est utilisé pour effectuer ces vérifications. Pour toute amélioration à apporter au programme par la suite, réduire la charge de travail de ces fonctions serait donc une priorité.

On remarque aussi dans le profil qu’une fonction nommée popcount_preprocessor est appelée 25 686 026 fois. Cette fonction est une fonction fournie par le module Bitset qui est utilisée à la fois lors de la résolution de grille et lors de la génération de grille pour compter le nombre de bits à 1

dans un entier de 64 bits. Le module Bitset fourni cinq variantes de cette fonction qui utilise chacune un algorithme différent pour renvoyer le résultat attendu.

Un objectif majeur de ce projet étant l’optimisation, un soin à été apporté à la réalisation de ces cinq fonction et à leur comparaison afin de déterminer celle qui offre les meilleures performances à notre projet. Les résultats de cette comparaison ont été obtenus en utilisant chacune de ces fonction dans des situations identiques en tout point et en comparant le profil obtenu par gprof que l’on peut voir à la figure 3.24.

FIGURE 3.24 – Comparaison des profils des cinq popcounts lors de la résolution d’une grille 12x12.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
0.22	36.61	0.08	25686026	0.00	0.00	popcount_preprocessor
1.37	35.99	0.51	25686026	0.00	0.00	popcount_kernighan
1.53	35.76	0.57	25686026	0.00	0.00	popcount_lookup_table
1.54	35.92	0.57	25686026	0.00	0.00	popcount_mit_hakmem
3.16	35.85	1.20	25686026	0.00	0.00	popcount_naive

On observe que, de manière prévisible, l’algorithme naïf est le moins performant, ayant passé 1,20 seconde au sein de la fonction. Les deux algorithmes SWAR, `popcount_mit_hakmem` et `popcount_lookup_table` utilisent tous les deux 0.57 seconde pour leur tâche. L’algorithme de Brian Kernighan, qui est décrit à la référence [And05] ainsi que deux des autres algorithmes employés, semble en effet être légèrement plus performant que ces deux derniers mais cela est probablement dû au fait qu’il s’agit d’un algorithme qui fait autant d’itération sur notre entier que celui-ci contient de bits à 1. On peut donc s’attendre à une baisse de performance lorsque celui-ci serait utilisé sur des grilles de grandes tailles où le nombre de bits à 1 serait plus important.

Enfin, on remarque le `popcount_preprocessor` est la plus efficiente des cinq variantes avec notablement moins de temps utilisé que les prochains algorithmes. Cette fonction est tout simplement un wrapper pour une fonction de GCC nommée `__builtin_popcountll`. GCC fait autant d’optimisations que possible en fonction de l’architecture de l’ordinateur afin de permettre d’atteindre de meilleurs résultats. On va donc, là où le choix est offert au

sein de notre programme, utiliser de préférence la fonction `popcount_preprocessor` afin d'améliorer ses performances. Les autres variantes restent tout de même disponibles afin de permettre d'autres tests par la suite et d'offrir une alternative à la fonction de GCC.

Utilisation de la mémoire du générateur

De la même manière que pour le solveur, on utilise `valgrind` en générant des exemples de grilles typiques et on constate pour chacune des grilles :

- une grille standard de taille 4x4 fait 19 allocations et 19 libérations,
- une grille standard de taille 8x8 fait 67 allocations et 67 libérations,
- une grille standard de taille 16x16 fait 259 allocations et 259 libérations,
- une grille standard de taille 32x32 fait 1027 allocations et 1027 libérations,
- une grille "linéaire" de taille 4x4 fait 19 allocations et 19 libérations,
- une grille "linéaire" de taille 8x8 fait 67 allocations et 67 libérations,
- une grille "linéaire" de taille 16x16 fait 259 allocations et 259 libérations,
- une grille "linéaire" de taille 32x32 fait 1027 allocations et 1027 libérations,
- une grille standard stricte de taille 4x4 fait 63 allocations et 63 libération.

On observe donc que le nombre d'allocations est, sans aucune surprise, proportionnelle à la taille de la grille tel qu'il vaut le nombre de cases présentes dans la grille auquel s'additionne trois autres allocations. Aucune fuite de mémoire n'est présente dans le programme et aucun accès en mémoire invalide n'est effectué au cours de son exécution.

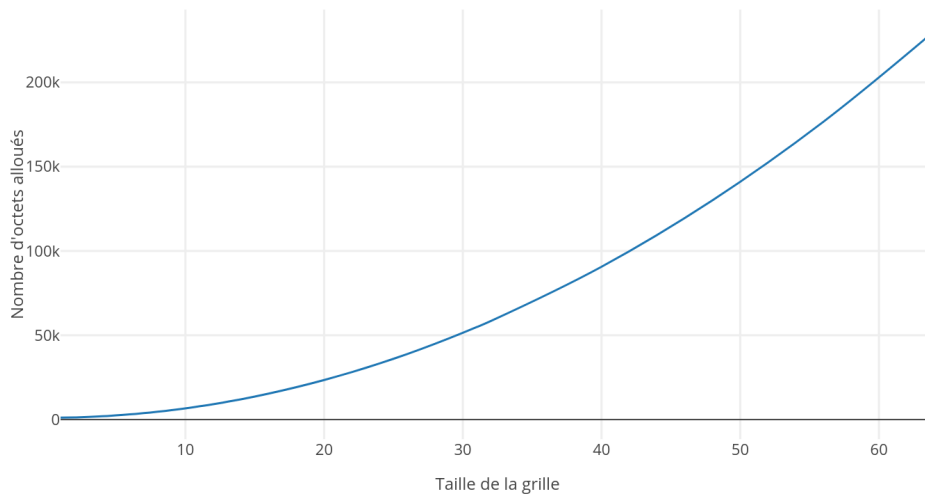
La génération d'une grille à solution unique étant uniquement une répétition du processus de génération jusqu'à l'obtention d'une grille stricte. Il est normale que celle-ci s'effectue aussi correctement et avec un nombre d'allocations supérieur à une grille régulière de dimensions équivalentes.

On observe aussi, par le même procédé, le nombre d'octets utilisés par le programme lors de la génération des grilles. On constate que le nombre d'octets alloués lors d'une génération étant le même pour une grille standard et une grille "linéaire". On porte donc notre analyse sur le nombre d'octets

utilisés par la génération en fonction des dimensions de la grille. Pour cela, on analyse la génération de grilles de taille 1 à 32, et de grilles de tailles 40, 48, 56 et 64.

Les résultats obtenus sont mis à disposition dans le graphe 3.25. On y observe que le nombre d'octets est évidemment proportionnel à la taille de la grille et que, pour générer une grille de taille 64, on utilise 230 424 octets. Ceci est une valeur parfaitement raisonnable qui ne causera aucun soucis pour l'utilisation du programme, y compris si celui-ci venait à être modifié de manière à pouvoir générer des grilles de taille légèrement supérieure à 64.

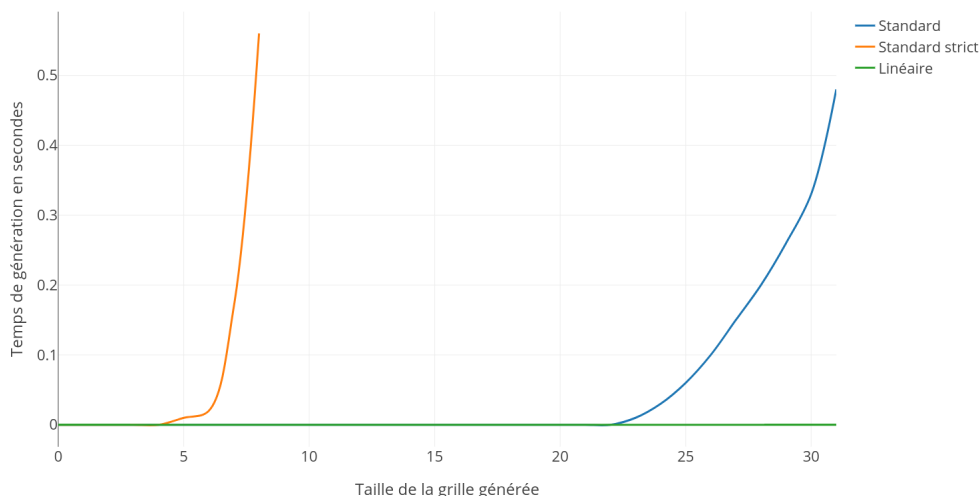
FIGURE 3.25 – Courbe du nombre d'octets utilisés pour la génération d'une grille en fonction de sa taille.



Temps d'exécution du générateur

Contrairement au solveur, la taille des salles d'une grille générée a peu d'influence sur le temps de génération de la grille, vu que les salles sont composées indépendamment du backtracking. On effectue à nouveau des tests de durée similaire à ceux effectués pour le solveur afin de déterminer le temps de génération de grilles en fonction de leur taille et des paramètres de génération. On construit alors le graphe 3.26.

FIGURE 3.26 – Graphe représentant le temps moyen nécessaire à la génération en fonction de la taille de la grille et de son type, sur un AMD Opteron Processor 6168 à 1900MHZ.



On constate que, sans aucune surprise, la génération de grilles "linéaires" se faisant sans backtracking, toutes les grilles linéaires sont générées instantanément, et ce jusqu'à une taille de 64. On remarque que les grilles standards sont générées instantanément jusqu'à une taille de 24x24, à partir d'où le temps de génération croît de manière exponentielle. La génération stricte étant une répétition de générations standards et de résolutions, il est attendu que ce type de génération devienne rapidement plus lent.

Repartition de la charge de travail du générateur

On répète pour le générateur le traitement par gprof afin d'obtenir les informations qui nous sont nécessaires pour évaluer la répartition du temps de calcul. On obtient alors le profil 3.27 et le graphe d'appels 3.28.

On constate que la grande majorité du temps est passé à attribuer des valeurs aléatoires aux cases de la grille avec le backtracking au sein de la fonction `randomize_cell_value`. Celle-ci fait de nombreux appels à la fonction `check_bit` au sein de son exécution, ce qui explique sa forte présence.

FIGURE 3.27 – Profil des fonctions généré par gprof pour la génération d’une grille standard 32x32.

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
58.05	0.47	0.47	1	470.23	770.37	randomize_cell_value
25.94	0.68	0.21	26513251	0.00	0.00	check_bit
8.65	0.75	0.07	2773836	0.00	0.00	set_bit
3.71	0.78	0.03				toggle_bit
2.47	0.80	0.02	2797579	0.00	0.00	clear_bit
1.24	0.81	0.01	1	10.00	10.00	print_grid
0.00	0.81	0.00	333827	0.00	0.00	popcount_preprocessor
0.00	0.81	0.00	1024	0.00	0.00	find_first_set
0.00	0.81	0.00	1024	0.00	0.00	find_singleton_position
0.00	0.81	0.00	1024	0.00	0.00	is_singleton
0.00	0.81	0.00	1	0.00	0.00	convert_bits_to_values
0.00	0.81	0.00	1	0.00	0.00	free_grid
0.00	0.81	0.00	1	0.00	770.37	generate_grid
0.00	0.81	0.00	1	0.00	0.00	initialize_grid
0.00	0.81	0.00	1	0.00	780.38	process_generate
0.00	0.81	0.00	1	0.00	0.00	random_rooms

FIGURE 3.28 – Extrait du graphe d’appels des fonctions généré par gprof pour la génération d’une grille standard 32x32.

				333826	randomize_cell_value [4]
		0.47	0.30	1/1	generate_grid [3]
[4]	95.1	0.47	0.30	1+333826	randomize_cell_value [4]
		0.21	0.00	26513251/26513251	check_bit [5]
		0.07	0.00	2773804/2773836	set_bit [6]
		0.02	0.00	2797579/2797579	clear_bit [8]
		0.00	0.00	333827/333827	popcount_preprocessor [10]
				333826	randomize_cell_value [4]

		0.21	0.00	26513251/26513251	randomize_cell_value [4]
[5]	25.9	0.21	0.00	26513251	check_bit [5]

3.2.3 Défauts du logiciel

Limitations de tailles

Un des défauts les plus évidents présents dans notre programme est la présence de limites fixées en dur au sein du code source (voir figure 3.29). En effet, dû à l’utilisation du type `uint64_t` pour toutes nos opérations bits à bits, une grille ne peut pas admettre de valeur supérieur à 64, et par conséquent ne peut pas avoir une taille supérieure à 64. Cette limite pourrait être éliminée en déclarant une nouvelle structure de données dont la taille serait

équivalente à la taille de la grille sur laquelle le programme opère. Il serait alors nécessaire d'adapter l'architecture existante et remanier le module bit-set qui est prévu uniquement pour des entiers de 64 bits.

De manière similaire, la longueur d'une salle d'une grille lue par le solveur ne peut pas être supérieure à 20. Ceci est la taille maximum qu'une salle peut atteindre dû au fait que le produit d'une telle salle serait ainsi de $20!$. Ce produit serait sauvegardé dans un unsigned long long. Or, un unsigned long long ne peut pas sauvegarder la valeur $21!$, il est donc impossible de sauvegarder un tel produit sur ce type de données. Il faudrait donc utiliser un autre type de données pour remplacer l'unsigned long long pour éliminer ce soucis.

FIGURE 3.29 – Extrait du code source représentant les limites fixées pour les grilles.

```
#define MAX_GRID_SIZE 64
#define MAX_ROOM_SIZE_SOLVER 20
#define MAX_ROOM_SIZE_GENERATOR 3
```

Performances du backtracking

Comme nous l'avons remarqué au chapitre précédent, le logiciel souffre beaucoup en termes de performance lorsque la taille de la grille devient trop large ou lorsque les salles peuvent se décomposer en de trop nombreuses salles. Trop de temps est utilisé par le backtracking dû à un manque d'heuristiques. Le logiciel pourrait être amélioré en améliorant les algorithmes actuellement présents au sein de la résolution et de la génération ainsi que en ajoutant des heuristiques à la résolution de la grille.

On pourrait notamment utiliser dans la résolution la technique de mark-up qui est employée à la génération pour garder trace des valeurs possibles pour une case. La difficulté de son implémentation au sein de la résolution consiste à combiner son utilisation aux ensembles de facteurs possibles qui sont déjà utilisés.

De manière similaire, ces ensembles de facteurs possibles pour chaque case sont actuellement utilisés pour déduire rapidement la valeur qu'une case est supposée prendre, mais une fois cette valeur trouvée, les autres cases de la salle ne sont pas mises à jour afin d'utiliser en priorité ce même ensemble de facteurs. Un des intérêts majeurs de cette approche est ainsi perdu. Ceci permettrait d'augmenter le nombre d'heuristiques employées, notamment lorsque la taille des salles augmente.

Factorisation des produits des salles

Pour rester sur le sujet précédent, la factorisation des produits des salles en ensembles de facteurs est un point majeur du programme. Pour cela, on fait usage d'un algorithme dit de divisions successives. Cet algorithme est particulièrement connu et simple d'utilisation mais n'est pas le plus performant. En sachant que l'on peut avoir à factoriser des nombres aussi large que $20!$, l'amélioration de l'algorithme de factorisation est critique à l'évolution du projet.

De même, une fois ces ensembles de facteurs générés, on procède ensuite à plusieurs opérations sur ces ensembles dont un tri. Le tri utilisé sur ces listes chaînées d'entiers est un tri à bulles dont la complexité au pire cas est $O(n^2)$. Ce tri pourrait être remplacé par un tri plus efficace afin d'améliorer les performances du programme lorsque celui-ci est amené à traiter de grands ensembles de facteurs.

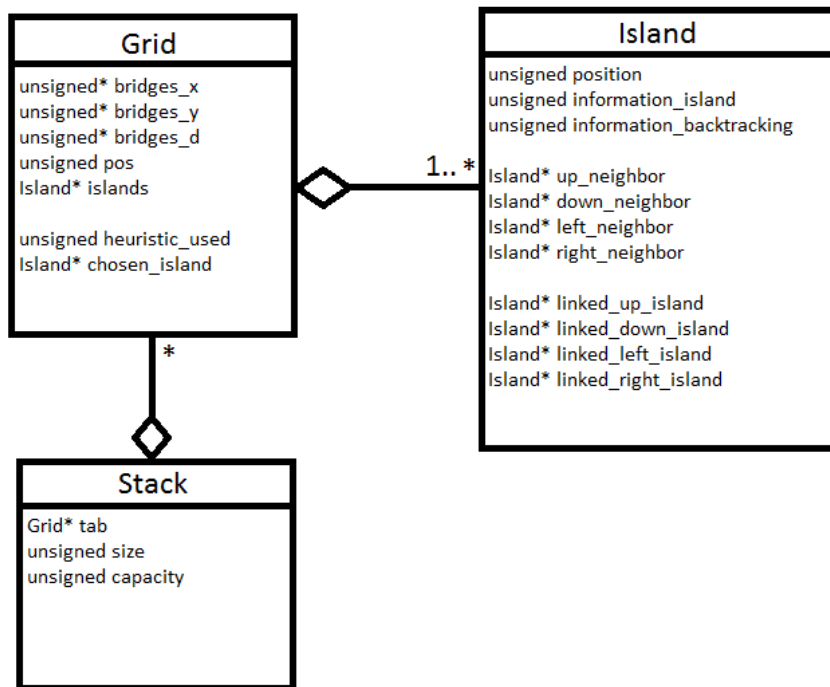
Chapitre 4

Hashiwokakero

4.1 Architecture et description du logiciel

4.1.1 Description de l'architecture

FIGURE 4.1 – Architecture pour le puzzle *Hashiwokakero*.



Voici l'architecture utilisée pour le puzzle hashiwokakero. Il y a 3 structures qui représentent la pile, les grilles et les îles.

Les îles utilisent 3 registres : position, information_island et information_backtracking. Le registre position utilise les 8 bits de poids faible pour stocker la position en x de l'île et les 8 bits au dessus pour stocker la position en y.

Le registre information_island utilise les 4 bits de poids faible pour stocker la valeur de l'île, le bit au dessus pour le statut finished indiquant si l'île a trouvé tout ses ponts ou pas. Les 2 bits au dessus servent à stocker le nombre de pont à droite de l'île. De même, les 3 paires de bits au dessus servent à stocker le nombre de pont successivement à gauche, en bas et en haut de l'île.

Le registre information_backtracking utilise les 4 bits de poids faible pour dire si un pont a déjà été tracé ou pas successivement à droite, à gauche, en bas et en haut. Le bit au dessus sert à indiquer si l'île bloque les autres îles lors du backtracking. Ce registre est uniquement utilisé lors du backtracking. Le registre reste à 0 jusqu'à cette étape. On lit ou change une valeur présente dans les registres à l'aide de masques définis dans les fonctions get et set.

L'île comporte également des pointeurs vers ces voisins dans les 4 directions qui sont utilisés lorsqu'on veut tracer un pont que ce soit avec une heuristique ou dans le backtracking.

Enfin, on utilise également des pointeurs vers l'île à laquelle on est reliée dans toutes les directions pour différencier les voisins des îles avec celles auxquelles on est reliés. On peut bien sûr n'avoir aucun voisin ou aucune île reliée dans une ou plusieurs directions. Dans ce cas, le pointeur reste à NULL.

Les grilles utilisent trois tableaux pour stocker la position des ponts. bridges_x stocke la position en x de chaque pont. bridges_y stocke la position en y de chaque pont. bridges_d stocke le type de pont : simple ou double. pos sert à indiquer la position du dernier pont dans les tableaux. Les grilles stockent également une liste d'îles : islands.

heuristic_used sert à stopper l'utilisation des heuristiques lorsqu'ils ne tracent plus de ponts. Il est présent pour arrêter d'appliquer les heuristiques quand plus aucun n'est utilisé.

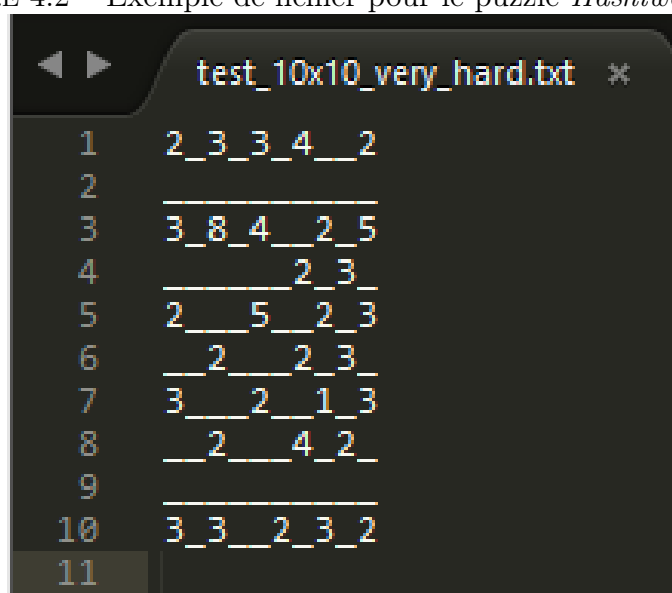
Le pointeur chosen_island n'est utilisé que lors du backtracking. Il sert à sauvegarder quelle île a été choisie par la fonction choose_and_draw_bridge.

La pile stocke un tableau de grilles : `tab`. Elle a également une taille : `size` et une capacité : `capacity`. La capacité représente le nombre maximale de grilles qu'on peut stocker. Elle peut être augmentée en réallouant la mémoire. La taille représente le nombre total de grilles qu'on stocke à l'instant présent. On peut l'utiliser notamment pour accéder à la dernière grille de la pile.

4.1.2 Fonctionnement du solveur

Le solveur de Hashiwokakero nécessite 1 argument qui est le fichier de la grille que l'on veut résoudre. On ne peut résoudre que des grilles de taille carré et de taille inférieure ou égale à 255x255.

FIGURE 4.2 – Exemple de fichier pour le puzzle *Hashiwokakero*.



Les fichiers de grille doivent respecter un format particulier. Les îles sont représentées par leur valeur allant de 1 à 8 (inclus). Le vide entre les îles est représenté par underscore ('_'). Il est essentiel de ne pas ajouter d'autres caractères même des espaces et de respecter une taille carré tel que le nombre de colonnes 'n' est égal au nombre de lignes 'n'. Nous refusons également de résoudre des fichiers sans aucune île.

Le solveur qui est utilisé est découpé en 2 étapes. La première étape se résume à utiliser le solveur simple : `simple_solve` et la deuxième étape consiste à utiliser du backtracking pour trouver le reste des ponts à tracer.

Le solveur simple est en fait une utilisation successive d'heuristiques sur chaque île de la grille afin de tracer un maximum de ponts. Actuellement le nombre d'heuristiques est au nombre de quatre. Trois d'entre eux sont des heuristiques de base qui ne sont utilisés qu'une seule et unique fois sur chaque île tandis que la dernière heuristique peut être utilisée plusieurs fois. Ce dernier est utilisé jusqu'à ce qu'on remarque qu'il ne trace plus de ponts sur une itération sur chaque île.

La première heuristique est la suivante : si une île n'a qu'un seul voisin alors on trace le nombre de ponts égal à sa valeur entre cette île et son voisin. Par exemple, si une île de valeur 2 n'a qu'un seul voisin alors on peut tracer 2 ponts entre cette île et son voisin. L'heuristique peut être utilisée sur les îles de valeur 1 et 2 seulement étant donné qu'on ne peut pas tracer plus de 2 ponts entre deux îles.

La deuxième heuristique regarde la valeur de l'île. Si l'île a une valeur paire et que son nombre de voisins est égale à sa valeur divisée par 2 alors on trace 2 ponts vers chacun des voisins. Par exemple, si une île de valeur 6 a 3 voisins alors on peut tracer 2 ponts entre cette île et chacun de ses voisins. L'heuristique peut ainsi être utilisée sur les îles de valeur 2, 4, 6 et 8.

La troisième heuristique regarde également la valeur de l'île. Si l'île a une valeur impaire et que son nombre de voisins est égale à sa valeur + 1 divisée par 2 alors on peut tracer 1 pont vers chacun des voisins. Cependant, si un pont a déjà été tracé vers l'un des voisins, on le laisse alors en place au lieu de le remplacer afin de ne pas compromettre le travail d'un autre heuristique. Par exemple, si une île a valeur 5 a 3 voisins alors on peut tracer un pont vers chacun des voisins ($3 = (5+1)/2$). On prend évidemment la condition énoncé précédemment pour ne pas remplacer un pont. L'heuristique peut donc être utilisé pour les îles de valeur 1, 3, 5 et 7.

La dernière heuristique peut être utilisée plusieurs fois. Elle consiste à regarder le nombre de voisins de l'île qui ont le statut `finished` à 1 : c'est à dire les voisins qui ont tracé tout leurs ponts et qui ont donc finis. S'il ne

reste qu'un seul voisin qui n'a pas fini à l'île alors on peut tracer le nombre de ponts que l'île a encore besoin de tracer entre elle et ce voisin. Par exemple, si une île de valeur 3 a deux voisins de valeur 1 et 2, alors la troisième heuristique permettra de tracer un pont entre l'île et ces deux voisins. L'île de valeur 1 prendra le statut 'finie' et il ne restera qu'un seul voisin non fini à l'île de valeur 3. Ainsi, on tracera son dernier pont vers l'île de valeur 2. L'heuristique peut être utilisée sur toutes les îles.

Le vrai solveur utilise d'abord ce solveur simple afin de tracer un maximum de ponts. Si on a trouvé les ponts à tracer on peut arrêter là. Sinon on passe dans la partie backtracking.

Nous utilisons un système de pile qui stocke des grilles dans le backtracking. La première chose faite est de stocker la grille complétée par le solveur simple afin de toujours pouvoir revenir à cette version qui est toujours juste.

Le backtracking repose principalement sur une fonction `choose_and_draw_bridge`. Cette fonction choisit toujours l'île qui a le moins de ponts à tracer et qui a le plus de voisins finis afin de limiter les erreurs. Ensuite elle trace le pont dans la première direction qui est disponible. Chaque pont tracé aléatoirement se voit marqué selon sa direction grâce à l'un des bits `up_chosen`, `down_chosen`, `left_chosen` et `right_chosen` présent dans le registre `information_backtracking` de chaque île. La fonction ne choisira pas cette direction à nouveau si le bit correspondant est à 1. Si tout les bits de backtracking d'une île sont à 1 et/ou que l'absence de voisin disponible empêche de tracer un pont alors on arrête de là choisir et on en choisit une nouvelle.

On vérifie constamment dans le backtracking si la grille reste correcte (cohérente). Ainsi, on vérifie pour toutes les îles que le nombre de ponts tracés par l'île ne soit pas supérieur à sa valeur, que le nombre de ponts tracés chaque côté soit inférieur ou égale à 2 et enfin on vérifie que le nombre de ponts qu'elle peut tracer avec ses voisins soit égale ou supérieure au nombre de ponts qu'il lui reste à tracer.

Dans le cas où le backtracking réussit à tracer un pont et que la grille reste correcte, on utilise l'heuristique utilisable plusieurs fois afin de tracer encore plus de ponts. Si la grille reste correcte, on l'insère dans la pile du backtracking.

Dans le cas où le pont tracé amène à une grille non correcte, alors on ne

l'insère pas et on restaure la dernière grille présente dans la pile afin d'effacer le mauvais pont. On garde néanmoins les registres `information_backtracking` de chaque île. On réessayera ensuite de tracer un nouveau pont à chaque fois. En gardant le registre on s'assure que le pont tracé à chaque fois ne sera pas le même.

Dans le cas où la fonction n'arrive pas à tracer de ponts alors on a épuisé les cas possibles avec la dernière grille insérée. Cela signifie qu'elle est partiellement fausse. On revient donc à son état et on efface la grille de la pile ainsi que le pont qui avait été tracé afin d'en tracer un nouveau.

On continue le processus jusqu'à trouver tout les ponts de la grille. Plus le backtracking a de ponts à tracer plus l'arbre des possibilités est grand et plus le backtracking mettra du temps. Ainsi plus le solveur simple trouve de ponts, plus le travail pour le backtracking sera simple.

4.1.3 Exemples de bon fonctionnement du solveur

Pour utiliser le solveur, il faut utiliser un argument comme expliqué au dessus. On peut utiliser l'option `'v'` pour afficher les îles successivement choisies par le backtracking ainsi que la taille de la pile et la grille. L'option `'o'` permet d'afficher la grille résolue dans un fichier à l'emplacement choisi par l'utilisateur. On peut également utiliser l'option `'t'` pour donner le temps maximum en secondes pendant lequel le backtracking a le droit de tourner. On peut ensuite voir dans la console le résultat de la résolution.

FIGURE 4.3 – Exemple de bon fonctionnement du solveur pour le puzzle *Hashiwokakero*.

```

test_10x10_hard.txt
1 2 _ _ 3 _
2 _ _ _ 1
3 _ 1 _ 3 _
4 4 _ _ _ 2
5
6 4 _ _ 3 _ 3
7 _ 2 _ 2 _
8 _ 2 _ _ 5
9 _ 1 _ _ _
10 2 _ 2 _ 1 _ 3
11

result.txt
1 2SSSSSS3_
2 S _ _ D 1
3 S _ 1SSSS3_S
4 4SSSSSSSS2
5 D _ _ _ _
6 4SSSSSS3D3
7 S2S2 _ _ S
8 SS_S2DDDD5
9 S1_S _ _ D
10 2SS2 _ 1SS3
11

"C:\Users\Martial D\Desktop\PDP\PDP\bin\Debug\PDP.exe" -f logicgamessolv...
Island[0], position 0 0: right bridges: 1, down bridges: 1,
Island[1], position 7 0: left bridges: 1, down bridges: 2,
Island[2], position 9 1: down bridges: 1,
Island[3], position 3 2: right bridges: 1,
Island[4], position 7 2: left bridges: 1, up bridges: 2,
Island[5], position 0 3: right bridges: 1, up bridges: 1, down bridges: 2,
Island[6], position 9 3: left bridges: 1, up bridges: 1,
Island[7], position 0 5: right bridges: 1, up bridges: 2, down bridges: 1,
Island[8], position 7 5: left bridges: 1, right bridges: 2,
Island[9], position 9 5: left bridges: 2, down bridges: 1,
Island[10], position 1 6: right bridges: 1, down bridges: 1,
Island[11], position 3 6: left bridges: 1, down bridges: 1,
Island[12], position 4 7: right bridges: 2,
Island[13], position 9 7: left bridges: 2, up bridges: 1, down bridges: 2,
Island[14], position 1 8: up bridges: 1,
Island[15], position 0 9: right bridges: 1, up bridges: 1,
Island[16], position 3 9: left bridges: 1, up bridges: 1,
Island[17], position 6 9: right bridges: 1,
Island[18], position 9 9: left bridges: 1, up bridges: 2,
Total number of islands : 19
Number of finished islands : 19
Solved.
Is the grid correct : 1
Process returned 0 (0x0)   execution time : 0.301 s

```

Voici un exemple du solveur utilisé dans ce cas avec la commande `./hashiwokakero -f ../test/test_10x10_hard.txt -o ../test/result.txt`. On demande donc au solveur de résoudre la grille `test_10x10_hard.txt` située dans le répertoire `test` qui est au même endroit que le répertoire `src`. On demande également d'afficher la solution dans le fichier `result.txt` situé au même endroit que la grille source.

En haut à gauche, on peut voir la grille source que l'on donne au solveur. En bas, il y a le résultat dans la console. On affiche le numéro de l'île dans la structure, sa position en x et en y ainsi que son nombre de ponts dans chaque direction s'il y en a au moins un. On affiche également le nombre d'îles au total et combien ont le statut "finie". On affiche ensuite le statut "Solved" ou "Unsolved". La résolution s'étant bien passée, elle a trouvé tout les ponts à tracer et le statut "Solved" apparaît. En cas de problème, le statut

"Unsolved" indique que le solveur n'a pas réussi à tracer tout les ponts ou qu'il ne les a pas tracés correctement. Enfin, on affiche si la grille est correcte ou pas (veuillez vous référer au fonctionnement du solveur pour l'explication du statut "correcte"). En haut à droite, on peut observer le fichier contenant la solution généré par l'option 'o'. Il est semblable au fichier source mais on affiche les ponts tracés par le solveur. 'S' correspond à un pont simple (un seul) et 'D' correspond à un double pont. Il faut prêter attention entre quelles îles le pont est tracé en suivant le parcours entre les deux. Dans ce cas, l'île en haut à gauche de valeur '2' est reliée à l'île tout à droite de valeur '3' par un pont simple 'S'. Cette dernière île de valeur '3' est reliée en dessous par un pont double 'D' à l'autre île de valeur '3'. Attention, le fichier de solution ne peut pas être utilisé par le solveur. Il est uniquement fait pour donner un autre type de solution "visuelle" pour l'utilisateur.

4.1.4 Fonctionnement du générateur

Le générateur de Hashiwokakero nécessite 3 arguments qui sont la taille de la grille que l'on veut générer, le nombre d'îles qu'on veut générer ainsi que l'emplacement du fichier où l'on veut stocker la grille. On génère uniquement des grilles de type carré. On ne peut pas générer de grille de taille supérieur à 255x255. Une grille d'hashiwokakero est normalement constitué d'îles séparées l'une de l'autre par au moins une case vide donc on ne peut générer des îles que sur la moitié des cases de la grille. Dans le cas où ces conditions sont respectés, alors la génération peut fonctionner.

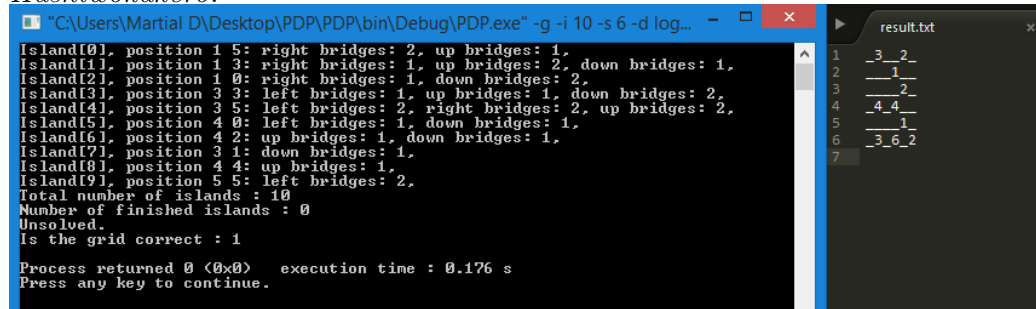
La génération marche totalement aléatoirement. Le processus consiste à choisir l'emplacement de la première île aléatoirement. On choisit ensuite aléatoirement une île dans la liste des îles déjà tracées ainsi qu'une direction à prendre et une distance. S'il est possible de tracer une nouvelle île à cet endroit (pas de blocage par un pont par exemple), alors on trace un ou deux ponts vers cet endroit et la nouvelle île. Si une île était déjà présente à cet endroit alors on trace un ou deux ponts entre les deux îles. On continue de tracer ces nouvelles îles jusqu'à qu'on en ait trouvé autant que l'utilisateur demande. A chaque fois qu'on n'arrive pas à tracer de nouveaux ponts ou de nouvelles îles, on incrémente un compteur d'échec. S'il y a trop d'échec on arrête d'essayer de tracer et on recommence le processus depuis le début. Le processus est effectué un certain nombre de fois.

Si on arrive à trouver le nombre d'îles voulu on arrête tout et on écrit la grille dans le fichier en effaçant les ponts tracés. Si on échoue à tous les trouver alors on écrit quand même la grille de la même manière mais on précise à l'utilisateur que le générateur a échoué et on propose de relancer avec une taille de grille plus élevée si on veut ce nombre d'îles précisément.

4.1.5 Exemples de bon fonctionnement du générateur

Pour utiliser le générateur, il faut utiliser 3 arguments comme expliqué au dessus. On peut également utiliser l'option 'v' pour afficher la solution de la grille générée. On peut ensuite retrouver à l'emplacement du fichier choisi, la grille générée conforme au format utilisé par le solveur et sans pont.

FIGURE 4.4 – Exemple de bon fonctionnement du générateur pour le puzzle *Hashiwokakero*.



```

C:\Users\Martial D\Desktop\PDP\bin\Debug\PDP.exe -g -i 10 -s 6 -d log...
Island[0], position 1 5: right bridges: 2, up bridges: 1,
Island[1], position 1 3: right bridges: 1, up bridges: 2, down bridges: 1,
Island[2], position 1 0: right bridges: 1, down bridges: 2,
Island[3], position 3 3: left bridges: 1, up bridges: 1, down bridges: 2,
Island[4], position 3 5: left bridges: 2, right bridges: 2, up bridges: 2,
Island[5], position 4 0: left bridges: 1, down bridges: 1,
Island[6], position 4 2: up bridges: 1, down bridges: 1,
Island[7], position 3 1: down bridges: 1,
Island[8], position 4 4: up bridges: 1,
Island[9], position 5 5: left bridges: 2,
Total number of islands : 10
Number of finished islands : 0
Unsolved.
Is the grid correct : 1
Process returned 0 (0x0)   execution time : 0.176 s
Press any key to continue.

```

```

result.txt
1  _3_2_
2  _1_
3  _2_
4  _4_4_
5  _1_
6  _3_6_2_
7

```

Voici un exemple du générateur utilisé dans ce cas avec la commande `"/hashiwokakero -g -i 10 -s 6 -d ../test/result.txt -v"`. On demande donc au générateur de tracer 10 îles dans une grille de taille 6x6 et de stocker la grille dans un fichier `result.txt` dans le répertoire `test` situé au même endroit que le répertoire `src`. On demande également d'afficher la solution de la grille générée.

A gauche, on peut voir le résultat dans la console. On affiche le numéro de l'île dans la structure, sa position en x et en y ainsi que son nombre de ponts dans chaque direction s'il y en a au moins un. On affiche également le nombre d'îles au total et combien ont le statut "finie". On ne donne pas de statut "finie" dans le générateur car il n'a aucune utilité dedans, c'est pourquoi il sera toujours à 0. On affiche ensuite le statut "Solved" ou "Unsolved".

Etant donnée qu'aucune île n'a le statut "finie", on a le statut "Unsolved" dans le générateur ce qui est cohérent étant donné que son but n'est pas de résoudre. Enfin, on affiche si la grille est correcte ou pas (veuillez vous référer au fonctionnement du solveur pour l'explication du statut "correcte"). A droite, on peut observer le fichier contenant la grille. Il peut ensuite être utilisé dans le solveur pour le résoudre.

4.2 Analyse du fonctionnement et test

4.2.1 Réponse aux besoins fonctionnels du logiciel

Analysons les réponses aux besoins fonctionnels avancés lors de la réalisation de cahier des besoins.

Les deux premiers besoins fonctionnels venant à l'esprit sont bien entendu la création d'une grille et son affichage. Nous avons donc cherché à développer un encodage le plus simple et lisible possible pour un utilisateur humain. Nous avons choisi l'encodage expliqué lors de la précédente partie. Dans un souci de clarté, d'optimisation et de facilité d'utilisation du programme, nous nous appuyons massivement sur des options lors du lancement : ainsi, il est possible (et nécessaire dans le cas d'une exécution en mode solveur) de spécifier un fichier à lire grâce à l'option -f (comme file) suivie du chemin vers le fichier. Si ce format est utilisé lors de la conception d'une grille, il n'a pas été retenu lors de l'affichage sur la sortie standard : nous avons préféré un affichage plus textuel, où l'on décrit simplement le numéro de la grille par ordre croissant et sa position. Si la grille est résolue, les ponts tracés et leur direction sont également affichés.

Venons en au coeur de notre logiciel : les deux fonctionnalités principales (et donc besoins fonctionnels) sont la résolution et la génération automatique d'une grille de Hashiwokakero. Commençons par le solveur : les souhaits émis par le clients sont très majoritairement remplis. La résolution résout une bonne quantité de grilles (voir catégorie résultats pour les exceptions). Un message est affiché pour renseigner l'utilisateur. Il est possible d'afficher l'état de résolution de la grille à chaque étape, La taille maximale de la grille d'entrée est fixée à 255x255. Il est cependant nécessaire de rester sur des grilles de cotés égaux. Un fichier contenant un tracé de la grille finale peut être généré à l'aide de la commande -o chemin-vers-le-fichier. Passons main-

tenant au générateur. La quasi totalité des besoins est effective : ici aussi, il est possible de générer des grilles de taille 255x255 maximum. Le nombre d'îles maximum est la moitié du nombre de cases. Ces deux paramètres sont spécifiés par la commande -s (comme size) puis -i (comme islands). Il est cependant nécessaire de rajouter l'option -g pour passer le programme en mode générateur. Afin de réutiliser les grilles générées dans le solveur, il faut les enregistrer dans un fichier grâce à l'argument -d (comme done) chemin-vers-le-fichier. A noter qu'il est possible de résoudre un fichier généré lors de la même exécution. Nous n'avons cependant pas implémenté la possibilité d'avoir une unique solution, cela nous paraissant très difficile dans notre cas.

Pour chacun du mode résolution et génération, il est possible d'effectuer une liste de tests automatiquement grâce au Makefile inclus avec les sources. Il suffit de lancer `make testsolve` pour des tests sur la résolution ou `make testgen` pour des tests sur la génération.

4.2.2 Réponse aux besoins non fonctionnels du logiciel

Ici encore, une grande partie des besoins à été comblée : le langage de programmation choisi est le C, qui est le langage le plus bas niveau mais compréhensible pour un humain dont nous disposons de par nos connaissances, comme convenu avec le client lors du premier entretien. Ce choix a pu nous causer des soucis : il a été nécessaire de définir de grosses structures pour mener à bien notre projet. Nous avons bien entendu pris en compte les remarques émises par notre jury lors de la présentation du prototype du projet, et avons fait de notre mieux pour réduire l'espace pris par notre structure : comme expliqué dans l'architecture, des masques ont été utilisés.

Un grand nombre d'algorithmes de résolution optimaux ont été utilisés, que nous appellerons heuristiques. Ces éléments certains sont trouvées de manière quasi instantanée, puisque l'on se situe très largement en dessous de la seconde (ordre du dixième de seconde). Les algorithmes utilisés étant très dépendants de grilles et au final jamais de calculs, les algorithmes SWAR n'ont malheureusement pas été utilisés dans notre programme. De même, nous n'avons pas intégré de multithreading, cela n'étant pas possible dans nos délais.

D'une manière générale, la robustesse du programme semble assurée : la taille de la grille est détectée automatiquement à la lecture du fichier. Il n'est

pas possible de donner au solveur une grille non carrée ou contenant de mauvais caractères (un message d'erreur correspondant est affiché).

Venons en aux options d'exécution : les options de génération, enregistrement et lecture dans un fichier et le mode verbeux ont été implémentés. En voici les caractéristiques :

1. *-f* :
 - Active le mode résolution.
 - Prend un paramètre : le nom du fichier d'entrée.
 - Permet de résoudre la grille contenue dans le fichier d'entrée
 - Est compatible avec toutes les autres options d'exécution.
2. *-g* :
 - Active le mode génération.
 - Permet de générer une nouvelle grille avec au moins une solution dont la taille est déterminée par l'utilisateur.
 - Est compatible avec toutes les autres options d'exécution.
3. *-s* :
 - Spécifie la taille de la grille à générer.
 - Cette option doit être utilisée avec l'option de génération et est compatible avec toutes les autres options d'exécution.
4. *-i* :
 - Spécifie le nombre d'îles à générer.
 - Cette option doit être utilisée avec l'option de génération et est compatible avec toutes les autres options d'exécution.
5. *-d* :
 - Prend un paramètre : le nom du fichier de sortie.
 - Cette option doit être utilisée avec l'option de génération et est compatible avec toutes les autres options d'exécution.
6. *-o* :
 - Prend un paramètre : le nom du fichier de sortie.
 - Permet d'enregistrer un affichage compréhensible de la grille résolue dans un fichier sans l'afficher au préalable en sortie standard.
 - Cette fonctionnalité doit être utilisée avec l'option de résolution et est compatible avec toutes les autres options d'exécution.

7. *-v* :
 - Permet d’afficher plus de détails lors de la résolution ou de la génération de la grille.
 - Est compatible avec toutes les autres options d’exécution.
8. *-t* :
 - Permet de définir un temps au bout duquel le programme se fermera automatiquement même si la résolution n’est pas terminée.
 - Est compatible avec toutes les autres options d’exécution mais nécessite de résoudre une grille.

Autrement, le coding style imposé par le client a bien entendu été respecté le mieux possible.

Le même exécutable a été utilisé pour la génération et la résolution.

4.2.3 Tests et interprétation de leur résultat

Nous avons automatisé les tests de résolution et de génération à l’aide du Makefile.

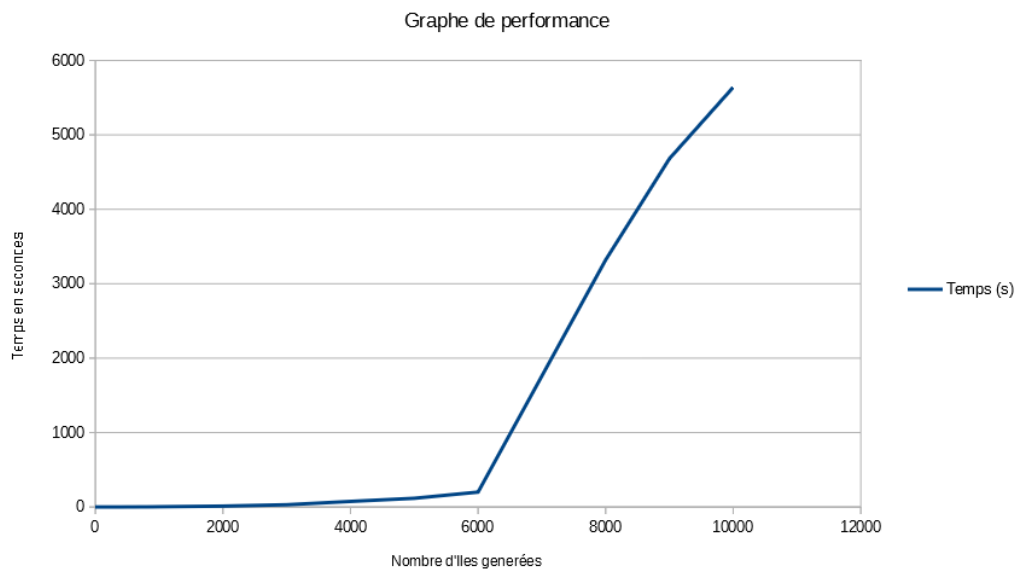
Quatre type de tests sont disponibles :

1. *testsolve* : Ce type de test va lancer une dizaine de tests de résolutions de grilles de taille et de difficultés variables à la chaîne. Les tests ne laissent aucune trace dans un fichier. Les résultats sont cependant directement lisibles dans le terminal.
2. *testgen* : Ce type de test va lancer quelques tests de génération de grilles de taille et de difficultés variables à la chaîne. Les tests génèrent des fichiers de sortie, que l’on peut retrouver dans le répertoire `/Hashiwokakero/src/test/generated/`.
3. *testboth* : Ce type de test va lancer un test de génération et de résolution de la même grille au cours de la même exécution du programme. Le fichier de sortie, appelé `test_gensolve.txt`, se retrouve lui aussi dans le répertoire mentionné précédemment. Le fichier de sortie résolu lui se nomme `test_gensolve_solved.txt`.
4. *testgengraph* : Ce type de test va lancer un time pour chacune des 18 générations de grille de taille maximale (255) mais de nombre d’îles variant de 1 à 10000. Des informations sont stockées dans un fichier test temporaire. Attention, ce test est très long (plusieurs heures). Il

doit être utilisé typiquement en cas de profilage de performances. Nous en donnons un exemple maintenant.

Ce profilage a été réalisé sur un ordinateur portable doté d'un processeur i7-3630QM, à la fréquence de base de 2.4Ghz. On remarque que la complexité de notre générateur semble exponentielle, sans grande surprise : le temps passé à générer une île explose littéralement à partir de 6000 îles. À noter que le programme étant codé purement séquentiellement, le passage à un processeur plus récent d'une vitesse proche des 4ghz a apporté des gains très significatifs, allant du simple au double pour les plus gros cas.

FIGURE 4.5 – Courbe de performance du générateur



Un soin tout particulier a été employé pour éliminer toute erreur ou fuite de mémoire, la charge de la gestion de la mémoire revenant au programmeur en C. C'est ainsi que nous avons testé chaque option et configuration possible à l'aide de valgrind, pour garantir la fiabilité technique de notre code. Dans chaque cas, nous n'avons aucun défaut de libération de mémoire ni erreurs. Ci dessous, un exemple d'exécution de valgrind sur une la résolution de la grille de test test_25x25_hard.txt :


```

==4941== HEAP SUMMARY:
==4941==    in use at exit: 0 bytes in 0 blocks
==4941==   total heap usage: 46 allocs, 46 frees, 145,304 bytes allocated
==4941==
==4941== All heap blocks were freed -- no leaks are possible
==4941==
==4941== For counts of detected and suppressed errors, rerun with: -v
==4941== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

4.2.4 Défauts et bugs du logiciel

Le générateur génère des îles toujours séparés par au moins une case horizontalement et verticalement. Le solveur lui accepte les îles côte à côte horizontalement ou verticalement. On considère que deux îles côte à côte sont tout de même voisines et on peut tracer un pont entre les deux îles. Cependant, on ne peut pas afficher les ponts entre deux îles voisines avec l'option 'o' (output). Si on veut absolument éviter le cas de deux îles côte à côte, alors il faut ajouter une condition dans la lecture du fichier de la grille et si deux îles sont côte à côte horizontalement ou verticalement alors on affiche un message d'erreur.

Le générateur n'arrive pas toujours à trouver la place pour générer le nombre d'îles demandé. La principale raison pour ce défaut est qu'on ne connaît pas en théorie le nombre maximum d'îles qu'on peut tracer sur une grille en fonction de sa taille. On sait qu'on ne peut pas tracer un nombre d'îles supérieur à la moitié du nombre de cases de la grille. Cependant, le nombre maximum semble être plus petit que ça et dépend sûrement de la parité de la taille de la grille. Ainsi, il faudrait changer ce nombre maximum par une vraie valeur théorique si c'est calculable. La deuxième raison est que l'aléatoire bien que limité à chaque essai empêche sûrement de trouver le nombre d'îles. Dans tout les cas, si le nombre d'îles n'est pas atteint, on retourne quand même les îles qu'on a trouvé jusqu'ici afin de fournir tout de même au client une grille. Si le client n'a pas réellement d'exigences sur la taille de la grille, une solution à ce défaut serait d'augmenter la taille de la grille (et relancer le générateur avec cette taille) pour permettre au générateur de trouver plus d'îles étant donné qu'il aura plus de places.

Cependant, le problème le plus important reste que le backtracking ne remonte pas assez haut dans la pile dans certains cas, ce qui l'empêche de tester tout l'arbre des possibilités. Le générateur ne nous a pas aidé à en trou-

ver la cause. Un bon exemple de ce problème sont les grilles de tests de taille 21 incluses. Le solveur arrive à générer des ponts pour une énorme majorité d'îles même lorsqu'il ne finit pas (de l'ordre de 98-99%). La piste d'un problème algorithmique est privilégiée. Nous savons que lorsque ce problème arrive, l'algorithme ne finit jamais (même après plusieurs heures). C'est donc un problème dans le code du backtracking qui empêche de remonter suffisamment dans la pile et empêche donc de balayer l'arbre des possibilités. Il est possible bien que surprenant que le backtracking, après avoir remonté jusqu'à un point de la pile, retrace le même pont entraînant donc ensuite les mêmes choix et une boucle. Cette boucle serait assez dure à détecter étant donné sa taille conséquente. Il faut souligner que le problème n'a été détecté que sur des grilles de taille plutôt grandes. Jusqu'à la taille de 10x10, la résolution a toujours marché sur les grilles testées (même celles du générateur). La taille relativement grande dans laquelle le problème arrive rend la tâche de trouver une solution assez difficile.

C'est évidemment un énorme échec et une déception pour nous, et nous aurions aimé pouvoir passer plus de temps à la résolution de ce problème, mais comme répété en cours, la qualité du travail primant sur la quantité, nous avons préféré travailler à rendre robuste toutes les autres fonctionnalités plutôt que de passer tout notre temps sur ce bug, le résoudre mais livrer un programme bancal sur d'autres aspects, que ce soit le générateur ou d'ordre plus technique. C'est dans ce souci de propreté que nous avons implémenté l'option timeout (-t) qui permet de quitter le programme proprement au bout de X secondes.

Chapitre 5

Conclusion

5.1 Évolutions futures

5.1.1 Extensions possibles pour Inshi no Heya

Les possibilités d'évolution pour le programme de Inshi no Heya ont déjà été mentionnées lors de la description de ses défauts. En effet, les prochaines évolutions dont le code devrait bénéficier consistant évidemment à la correction des défauts actuellement présents, nous rapellons donc ici ces corrections.

Il serait avantageux de passer l'utilisation des `uint64_t` à une nouvelle structure de données qui permettrait de conserver le système de bit à bit utilisé au sein de notre programme. Cette nouvelle structure de données nous permettrait ainsi de lire des grilles d'une taille supérieure à 64. Il serait également possible de remplacer les `unsigned long long` à une autre structure de données qui permettrait de stocker des valeurs au-delà de 20! et ainsi permettre l'utilisation de salles dont la longueur serait supérieure à 20.

La priorité actuelle d'évolution serait d'améliorer les performances de nos algorithmes de backtracking actuels, notamment en augmentant le nombre d'heuristiques employées. Ceci pourrait être fait avec une meilleure utilisation de nos ensembles de facteurs, avec la technique de markup pour garder trace des valeurs qu'une salles peut prendre en fonction des autres cases de la rangée et de la colonne, ou encore avec la technique des ensembles préemptifs qui est utilisé dans le sudoku.

Une fois le backtracking amélioré, le prochain bottleneck en terme de performance devrait être la factorisation des produits des salles. Il serait donc judicieux de passer de notre algorithme de factorisation par divisions successives à un algorithme plus performant. Ceci relevant du domaine de la théorie des nombres, il faut s'attendre à ce que de nombreuses améliorations puissent être faites avec un certain niveau de difficulté. Il est aussi possible d'améliorer l'algorithme de tri à bulles utilisées pour trier nos listes chaînées en passant à un autre type de tri plus performant.

Il serait également possible de considérer l'étude plus en détail du puzzle Inshi no Heya afin de déterminer si il existe un ou plusieurs motifs qu'une grille peut posséder qui augmenterait ses chances de n'admettre qu'une seule et unique solution. Ceci nous permettrait de réduire le facteur aléatoire de notre génération stricte et donc d'en améliorer ses performances.

5.1.2 Extensions possibles pour Hashiwokakero

Il est possible d'ajouter des heuristiques que l'on peut utiliser au fur et à mesure. La première consisterait à regarder si le nombre de ponts que l'île doit tracer est pair. Si le nombre de voisins de l'île qui n'ont pas finis est égale au nombre de ponts que l'île doit tracer divisé par 2 alors on trace deux ponts entre l'île et chacun de ces voisins.

La deuxième consisterait à regarder si le nombre de ponts que l'île doit tracer est impair. Si le nombre de voisins d'une île qui n'ont pas finis est égale au nombre de ponts que l'île doit tracer $+ 1$ et le tout divisé par 2 alors on trace 1 pont entre l'île et chacun de ses voisins.

Ces deux heuristiques sont très semblables aux deux premières heuristiques décrites dans le fonctionnement du solveur. Leur but et leur différence est de pouvoir être réutilisé en prenant en compte si les voisins ont finis ou pas. Ces heuristiques permettront au solveur de fonctionner plus rapidement et facilement. Le backtracking aura plus de chances de fonctionner étant donné qu'on réduit son arbre de possibilités. Cependant, ces heuristiques ne résoudront pas réellement le problème du backtracking. Ils ne feront qu'améliorer le nombre de grilles qu'on réussit à résoudre.

Le générateur unique n'ayant pas été fait, c'est évidemment une des extensions possible. Il faudrait un nouveau générateur qui utilise des fonctions semblables aux heuristiques mais de manière inverse. Il faudrait partir d'une île puis choisir sa valeur paire ou impaire selon les cas puis tracer le nombre d'îles et de ponts comme dans l'heuristique. La difficulté est que l'heuristique 'n3' ne finit pas les îles mais ne fait que tracer certains ponts donc il faudrait faire un changement si on veut utiliser cette heuristique inversement. Après avoir tracé le nombre d'îles demandé, il faudrait vérifier que le solveur simple (celui qui n'utilise que des heuristiques) arrive à résoudre la grille et donne une solution semblable au générateur. Cependant, il est possible que ça ne suffise pas pour prouver que c'est une solution unique. Il faudrait donc essayer de créer un nouveau solveur qui teste toutes les possibilités. S'il existe un seul chemin de possibilité qui amène à une solution alors notre grille a une solution unique.

On peut également étendre notre solveur en vérifiant que la solution forme un seul groupe. Pour rappel, le fait de former un seul groupe est une condition de la validation d'une solution du jeu Hashiwokakero. Actuellement, notre générateur crée forcément une grille qui a une solution qui forme un seul groupe étant donné qu'il trace toujours une nouvelle île en partant d'une île qui existe déjà. Notre solveur lorsqu'il écrit les voisins de chaque île décide de ne pas mettre comme voisins deux îles de valeur 1. Deux îles de valeur 1 qui sont reliés signifient forcément que la solution ne forme pas un seul groupe. La seule exception est lorsque la grille ne contient que deux îles de valeur 1 alors dans ce cas on les met comme voisin et les heuristiques traceront le pont entre les deux. Pour le moment, toutes les grilles que l'on a testé et qui ont été résolues ont toujours eu une solution qui contient un seul groupe. Néanmoins, il est sûrement possible de créer une grille qui amènera à une solution qui ne forme pas un seul groupe.

Le problème est qu'une fonction vérifiant que la grille a des îles reliées qui forment un seul groupe ne pourrait pas être utilisée dans la résolution. Le backtracking comme les heuristiques ne tracent pas forcément des ponts au même endroit de la grille ce qui fait que pendant la résolution, il existera de nombreux cas où la grille n'aura pas des îles reliées qui forment un seul groupe. Le seul intérêt de cette fonction serait de vérifier que la solution forme bien un seul groupe. Si ce n'est pas le cas, on pourrait soit se contenter d'afficher un message d'erreur indiquant que la solution ne forme pas un seul

groupe ou sinon il faudrait relancer la résolution mais en trouvant un système permettant de ne pas retomber sur la même solution.

5.2 Conclusion générale

Ce projet nous a beaucoup appris, et ce sur de nombreux points. Premièrement, si le choix de deux jeux très différents l'un de l'autre à été intéressant d'un point de vue variété et couverture du sujet, une charge de travail plus conséquente en est ressortie : nous n'avons pour ainsi dire pas de code en commun. La répartition des tâches en fut cependant plus aisée.

On remarque aussi que notre décision de travailler sur les deux jeux en parallèle dès le début nous a permis de diviser les tâches plus facilement en évitant d'avoir plusieurs personnes occupées par une seule tâche ou des personnes en attente qu'une fonction soit finie d'être implémentée.

De même, il est important d'avoir une batterie de tests automatisée prête pour effectuer des tests de regression pour chaque changement effectué au code, autant qu'il est nécessaire d'avoir une base technique solide plutôt qu'un éventail de fonctionnalités peu robustes.

Toujours d'un point de vue organisationnel, la présence de nombreux autres projets dans les autres matières a pu rendre certaines semaines difficiles à gérer, ce qui nous habitue à certaines périodes de "rush" de la vie professionnelle.

Chapitre 6

Bibliographie

- [And05] Sean Eron Anderson. Bit twiddling hacks. <http://graphics.stanford.edu/~seander/bithacks.html>, 2005. [Visité le 2 avril 2018].
- [And09] Daniel Andersson. Hashiwokakero is NP-complete. *Information Processing Letters*, 109(19) :1145–1146, 2009.
- [Cro09] JF Crook. A pencil-and-paper algorithm for solving sudoku puzzles. *Notices of the AMS*, 56(4) :460–468, 2009.
- [Mor09] Timo Morsink. Hashiwokakero. <http://www.liacs.nl/assets/Bachelorscripties/2009-11TimoMorsink.pdf>, 2009. [Visité le 31 janvier 2018].
- [Nik14] Co. Nikoli. Présentation de Inshi no heya. https://web.archive.org/web/20140306224423/http://www.nikoli.co.jp/ja/puzzles/inshi_no_heyas.html, 2014. [Archivé le 6 mars 2014. Visité le 2 février 2018].
- [Nik18] Co. Nikoli. Présentation de Hashiwokakero. <http://www.nikoli.co.jp/en/puzzles/index.html>, 2018. [Visité le 24 janvier 2018].
- [PU06] Patrick Prosser and Chris Unsworth. Rooted tree and spanning tree constraints. In *17th ECAI workshop on modelling and solving problems with constraints*, 2006.