De la construction de grilles de mots croisés parfaites

Étienne Dupuis

Février 2003

Abstract

A perfect crossword grid is defined as a grid without black squares. We explain two algorithms built to speed up the construction of such a grid, given a word list and a size. We also give experimental results with French and English word lists.

1 Introduction

1.1 Énoncé du problème

Les amateurs de jeux de lettres et plus particulièrement les cruciverbistes tentent tous un jour ou l'autre de construire leurs propres grilles. Un des objectifs naturels lors de cet exercice est de construire une grille dont le nombre de cases noires soit raisonnable, idéalement le plus petit possible. Ce nombre peut-il être nul, autrement dit est-il possible de construire une grille sans case noire ? La réponse dépend évidemment de la taille de la grille et des mots reconnus comme valides. Le lecteur trouvera dans ce document une réponse partielle à cette question, sous forme de deux algorithmes de recherche de telles grilles.

1.2 Définitions

Soit $k > 0$ un entier et A un alphabet c'est-à-dire un ensemble fini de symboles appelés *lettres*. Un mot m de k lettres est défini par un k-tuple

$$m = \langle m_1 m_2 \cdots m_k \rangle ,$$

où les $m_i \in A$ sont des lettres de l'alphabet.

Une grille G de taille k est décrit par une matrice $G = (g_{ij})$ de $k \times k$ composée de lettres de A, c'est-à-dire que $G \in \mathbb{M}_k[A]$. Soit M une liste[1] de $|M|$ mots de k lettres de A. Une grille G de taille k sera dite parfaite si chaque rangée et chaque colonne de G est un mot de M. Plus précisément, $G \in \mathbb{M}_k[A]$ est parfaite si

$$\begin{aligned} \langle g_{i1} g_{i2} \cdots g_{ik} \rangle \in M \quad & \text{pour tout} \quad i = 1, 2, \ldots, k \\ \langle g_{1j} g_{2j} \cdots g_{kj} \rangle \in M \quad & \text{pout tout} \quad j = 1, 2, \ldots, k. \end{aligned}$$

Le problème peut maintenant être formalisé comme suit : Étant donné un alphabet A et un ensemble de mots M de k lettres de A, trouver toutes les grilles G parfaites.

2 Un premier algorithme

2.1 Introduction

Mathématiquement, le problème est résolu. Il suffit de considérer toutes les grilles G possibles et de vérifier si elles sont parfaites ou non. Cette approche est évidemment trop coûteuse en temps de calcul. Nous allons donc étudier la structure de l'ensemble de mots M afin d'obtenir un algorithme plus efficace.

2.2 Distribution des lettres

Commençons par étudier la distribution et la fréquence des diverses lettres composant les mots de M. Soit $a \in A$ une lettre et $1 \leq i \leq k$ un entier. On calcule

$$\rho_i(a) = \frac{1}{|M|} \sum_{m \in M} \mathcal{X}_{\{m_i\}}(a),$$

la probabilité que la $i^{\text{ème}}$ lettre d'un mot $m \in M$ soit un a. $\mathcal{X}$ est la fonction caractéristique, définie sur un ensemble X par

$$\mathcal{X}_E(x) = \begin{cases} 1 & \text{si } x \in E \\ 0 & \text{si } x \notin E, \end{cases}$$

où $E \subseteq X$. Ainsi, $\rho_i(a)$ est le nombre de fois que la lettre a apparaît en $i^{\text{ème}}$ position dans un mot de M, divisé par le nombre de mots de M. Notez que pour un i donné

$$\sum_{a \in A} \rho_i(a) = 1.$$

Grâce à ces probabilités, nous allons associer un score à chacun des mots de M de façon à ce que plus un mot est composé de lettres fréquentes, plus son

[1] Par liste il faut comprendre un ensemble fini.

score soit élevé. Ce score doit également tenir compte de la position des lettres. Par exemple, beaucoup plus de mots français débutent par un **B** qu'il y en a qui se terminent par cette même lettre. On devra donc éviter de placer un mot contenant un **B** dans la dernière rangée (ou colonne) de la grille. Le score donné au mot $m \in M$ s'il est placé dans la $j^{\text{ème}}$ rangée (ou colonne) de la grille sera donc

$$\phi_j(m) = \frac{1}{k} \sum_{i=1}^{k} \rho_j(m_i).$$

Ce score est tout simplement la moyenne des probabilités ρ_j pour les lettres du mot m. Notez que par définition, pour tout $m \in M$ on a $0 < \phi_j(m) \leq 1$.

Nous verrons dans la section 2.4 comment ces scores peuvent accélérer la découverte d'une première grille parfaite pour une liste de mots M de k lettres de A.

2.3 Permutation des lettres

Avant de poursuivre avec les probabilités, arrêtons-nous un instant pour réfléchir. Lorsqu'on tente de construire une grille parfaite à la main, il est plus aisé de placer d'abord les mots des dernières rangées et colonnes que le contraire. En effet, en raison des règles d'accord de la langue française, un très grand nombre de mots se terminent par **A**, **E**, **R**, **S**, **T**. Une fois les derniers mots placés il est plus facile de choisir les premières lettres que le contraire :

?	**?**	**?**	**R**	**E**	**R**		**P**	**A**	**T**	**U**	**R**	**E**
?	**?**	**?**	**A**	**P**	**A**		**A**	**N**	**I**	**M**	**A**	**L**
?	**?**	**?**	**S**	**A**	**T**		**T**	**A**	**R**	**I**	**F**	**S**
R	**E**	**P**	**A**	**R**	**E**		**A**	**L**	**O**	**?**	**?**	**?**
E	**P**	**U**	**I**	**S**	**E**		**T**	**E**	**N**	**?**	**?**	**?**
S	**A**	**S**	**S**	**E**	**S**		**E**	**S**	**S**	**?**	**?**	**?**

Mathématiquement, ces considérations linguistiques se traduisent par le fait que la distribution des lettres en début de mot est plus uniforme qu'en fin de mot. En calculant la mesure des écarts par rapport à la distribution uniforme théorique pour chaque position, on obtient une mesure de distance entre cette dernière distribution et celle observée pour chaque position :

$$\Delta_i = \sum_{a \in A} \left(\rho_i(a) - \frac{1}{|A|} \right)^2.$$

On souhaite donc d'abord placer les mots sur les rangées (colonnes) dont la distribution des lettres est la moins uniforme, c'est-à-dire selon une permutation σ de $\{1, 2, \ldots, k\}$ telle que

$$\Delta_{\sigma(1)} \geq \Delta_{\sigma(2)} \geq \cdots \geq \Delta_{\sigma(k)}.$$

Afin de ne pas s'embêter avec cette permutation lors du déroulement de l'algorithme, nous allons l'appliquer à (la position des lettres de) tous les mots de M pour obtenir une nouvelle liste M', sur laquelle nous allons travailler. Ainsi, $m \in M$ si et seulement si

$$m' = \langle m_{\sigma(1)} m_{\sigma(2)} \cdots m_{\sigma(k)} \rangle \in M'.$$

De plus, si $G = (g_{ij})$ est une grille parfaite de mots de M, alors $G' = (g_{\sigma(i)\sigma(j)})$ est une grille parfaite de mots de M'.

Par exemple, si $\Delta_2 \geq \Delta_1 \geq \Delta_3 \geq \Delta_4$ et que

$$M = \{\mathbf{RUSE}, \mathbf{ATRE}, \mathbf{NUIT}, \mathbf{LUNE}, \ldots\}$$

alors

$$M' = \{\mathbf{URSE}, \mathbf{TARE}, \mathbf{UNIT}, \mathbf{ULNE}, \ldots\}.$$

Remarquez qu'il n'est pas nécessaire de recalculer les ϕ_j puisque

$$\phi_{\sigma(j)}(m') = \phi_j(m).$$

L'utilisation de la liste M' en place de M ne nous permettra pas de trouver plus de grilles parfaites, elle ne nous permettra que de les trouver plus rapidement. Il faudra bien sûr appliquer la permutation inverse sur une éventuelle grille parfaite afin de retrouver la grille formée des mots originaux. Comme la transformation de M à M' n'a pas d'influence sur la cohérence de l'algorithme, nous pouvons supposer pour la suite de notre argumentation que la liste M satisfait à la condition

$$\Delta_1 \geq \Delta_2 \geq \cdots \geq \Delta_k.$$

Sinon, on la remplace simplement par M'.

2.4 Tri des mots

L'efficacité de notre algorithme dépend essentiellement de la façon dont sont ordonnés les mots de M. On suppose d'abord que l'alphabet A possède un ordre alphabétique, ce qui signifie en langage mathématique qu'il existe une bijection $\Lambda : A \to \{1, 2, \ldots, |A|\}$ qui associe un rang à chaque lettre de l'alphabet.

La fonction Λ permet de classer les mots de M en ordre alphabétique. Soit $m \in M$ un mot. Définissons $\psi(m) \in \mathbb{N}$ par

$$\psi(m) = \sum_{i=1}^{k} (\Lambda(m_i) - 1) \, |A|^{k-i},$$

une représentation en base $|A|$ du mot m. Par exemple, si A est l'alphabet

français de 26 lettres, alors

$$\begin{aligned}
&\psi(\mathbf{DAME}) \\
&\quad = (\Lambda(\mathbf{D}) - 1)\,|A|^3 + (\Lambda(\mathbf{A}) - 1)\,|A|^2 + (\Lambda(\mathbf{M}) - 1)\,|A|^1 + (\Lambda(\mathbf{E}) - 1)\,|A|^0 \\
&\quad = 3 \times 26^3 + 0 \times 26^2 + 11 \times 26 + 4 \\
&\quad = 53018.
\end{aligned}$$

L'intérêt de cette représentation est que si $m, w \in M$ sont deux mots tels que $\psi(m) < \psi(w)$, alors m précède w dans l'ordre alphabétique. La réciproque est également vrai.

En utilisant la fonction caractéristique $\mathcal{X}$, il est possible de considérer des ordres alphabétiques qui ne tiennent pas compte de toutes les lettres des mots. Soit $E \subseteq \{1, 2, \ldots, k\}$ un sous-ensemble d'entiers. On définit la fonction $\psi_E : M \to \mathbb{N}$ par

$$\psi_E(m) = \sum_{i=1}^{k} \mathcal{X}_E(i)(\Lambda(m_i) - 1)\,|A|^{k-i}\,.$$

Par exemple, si $E = \{1, 4\}$ alors $\psi_E(\mathbf{DAME}) = \psi_E(\mathbf{DUNE})$ puisque ces deux mots possèdent les mêmes premières et dernières lettres. Notez que si $m, w \in M$ sont deux mots tels que $\psi_E(m) = \psi_E(w)$, alors $m_i = w_i\ \forall i \in E$.

Finalement, les scores ϕ_j peuvent également être utilisés pour classer les mots. Définissons les fonctions $\Psi_{E,j} : M \to \mathbb{R}$ par

$$\Psi_{E,j}(m) = \psi_E(m) + (1 - \phi_j(m)).$$

Comme $0 < \phi_j(m) \leq 1$, la partie entière de $\Psi_{E,j}(m)$ représente la composante alphabétique tandis que la partie fractionnaire représente le score associé au mot, dont le signe est inversé afin que le classement des mots à l'aide de $\Psi_{E,j}$ favorise les mots ayant un score plus élevé tout en conservant l'ordre alphabétique.

Nous allons maintenant définir $2k$ façon d'ordonner les mots de M. Soient $\Omega_{H,1}, \ldots, \Omega_{H,k}$ et $\Omega_{V,1}, \ldots, \Omega_{V,k}$ des fonctions bijectives de M vers $\{1, 2, \ldots, |M|\}$ qui associent un rang à chaque mot de M telles que

- $\Omega_{H,1}$ représente la liste M triée par ordre décroissant de scores ϕ_1 ;
- $\Omega_{H,2}$ représente la liste M triée selon la première lettre de chaque mot puis, à l'intérieur de chaque groupe de mots débutants par la même lettre, par ordre décroissant de scores ϕ_2 ;
- $\Omega_{H,3}$ représente la liste M triée selon les deux premières lettres de chaque mot puis, à l'intérieur de chaque groupe de mots débutants par les deux mêmes lettres, par ordre décroissant de scores ϕ_3 ;
- ...
- $\Omega_{V,1}$ représente la liste M triée selon la première lettre de chaque mot puis, à l'intérieur de chaque groupe de mots débutants par la même lettre, par ordre décroissant de scores ϕ_1 ;
- $\Omega_{V,2}$ représente la liste M triée selon les deux premières lettres de chaque mot puis, à l'intérieur de chaque group de mots débutants par les deux mêmes lettres, par ordre décroissant de scores ϕ_2 ;

– ...
– $\Omega_{V,k}$ représente la liste M triée par ordre alphabétique.
Mathématiquement, pour $l \in \{1, 2, \ldots, k\}$ et $m, w \in M$,

$$\begin{aligned} \Omega_{H,l}(m) < \Omega_{H,l}(w) \quad &\Longleftrightarrow \quad \Psi_{\{1,2,\ldots,l-1\},l}(m) < \Psi_{\{1,2,\ldots,l-1\},l}(w) \\ \Omega_{V,l}(m) > \Omega_{V,l}(w) \quad &\Longleftrightarrow \quad \Psi_{\{1,2,\ldots,l\},l}(m) < \Psi_{\{1,2,\ldots,l\},l}(w). \end{aligned}$$

Ces tris sont cruciaux pour l'algorithme détaillé dans la prochaine section.

2.5 L'algorithme

L'algorithme se déroule en $2k$ étapes. Il s'agit simplement d'essayer un mot dans la première rangée de la grille, puis un dans la première colonne, puis un dans la deuxième rangée et ainsi de suite. Si on parvient à remplir toute la grille, alors on a construit une grille parfaite. Si on parvient à une étape où il n'y a aucun mot possible pour la rangée (colonne) traitée, alors on retourne à l'étape précédente et on choisit un autre mot pour la colonne (rangée) précédente.

C'est ici que les tris $\Omega_{H,j}$ et $\Omega_{V,j}$ prennent leur importance. En effet, lors du choix du mot de la $j^{\text{ème}}$ colonne, les j premières lettres de ce mot sont connues puisqu'elles font partie des mots des j premières rangées. Le mot doit donc être choisi parmi tous les mots débutant par ces j premières lettres et ce choix se fera par ordre décroissant de score. D'où l'utilisation du tri $\Omega_{V,j}$. Les tris $\Omega_{H,j}$ sont quant à eux utilisés pour les mots horizontaux.

En détail, on commence par fixer $s_1 = 1$ et $t_1 = |M|$, les bornes entre lesquelles peuvent être choisis les mots de la première rangée. Les variables u_j et v_j seront les bornes pour le choix des mots des colonnes. La grille $G = (g_{ij})$ contient initialement des lettres arbitraires.

Pour une rangée i, le choix d'un mot consiste en les opérations suivantes :

1. Si $s_i > t_i$ alors il n'y a plus de mots possibles pour cette rangée. On doit choisir un nouveau mot pour la $(j-1)^{\text{ème}}$ colonne. Si $i = 1$ alors l'algorithme est terminé. Toutes les grilles parfaites ont été identifiées.
2. Sinon, on choisit le mot $m = \Omega_{H,i}^{-1}(s_i)$, c'est-à-dire qu'on attribue les valeurs $g_{il} = m_l$ pour $l = 1, 2, \ldots, k$. En fait on peut commencer à $l = i$ puisque que les premières lettres du mot sont déjà dans la grille.
3. On ajoute 1 à s_i.
4. On calcule u_i la valeur minimale telle que si $m = \Omega_{V,i}^{-1}(u_i)$ alors $m_l = g_{li}$ pour $l = 1, 2, \ldots, i$. On calcule v_i la valeur maximale qui satisfait à la même condition.
5. On passe au choix du mot de la $j^{\text{ème}}$ colonne.

Pour une colonne j, le choix d'un mot consiste en des opérations semblables :

1. Si $u_j > v_j$ alors il n'y a plus de mots possibles pour cette colonne. On doit choisir un nouveau mot pour la $(j)^{\text{ème}}$ rangée ;

2. Sinon, on choisit le mot $m = \Omega_{V,j}^{-1}(u_j)$, c'est-à-dire qu'on attribue les valeurs $g_{lj} = m_l$ pour $l = j+1, j+2, \ldots, k$. Les j premières lettres du mot m sont déjà dans la grille.
3. On ajoute 1 à u_j.
4. On calcule s_{j+1} la valeur minimale telle que si $m = \Omega_{H,j+1}^{-1}(s_{j+1})$ alors $m_l = g_{(j+1)l}$ pour $l = 1, 2, \ldots, j$. On calcule t_{j+1} la valeur maximale qui satisfait à la même condition.
5. On passe au choix du mot de la $(i+1)^{\text{ème}}$ rangée. Si $j = k$, alors la grille G est parfaite. On peut alors soit terminer l'algorithme ou choisir un nouveau mot pour la $i^{\text{ème}}$ rangée afin de trouver d'autres grilles parfaites.

Lors du choix du mot de la $i^{\text{ème}}$ colonne (deuxième opération ci-haut), on pourra vérifier que le mot n'est pas le même que celui de la $i^{\text{ème}}$ rangée afin d'obtenir des grilles ne présentant pas de mots répétés symétriquement. Une grille parfaite satisfaisant à cette contrainte sera dite hyperparfaite.

2.6 Classement des grilles parfaites

Étant donné plusieurs grilles parfaites, peut-on les classer d'une façon quelconque ? Une solution simple consiste à calculer la moyenne des scores ϕ des mots de la grille. Ainsi on obtient le score $\Phi(G)$ d'une grille parfaite $G = (g_{ij})$ grâce à l'équation

$$\begin{aligned}
\Phi(G) &= \frac{1}{2k}\left(\sum_{i=1}^{k}\phi_i(\langle g_{i1}g_{i2}\cdots g_{ik}\rangle) + \sum_{j=1}^{k}\phi_j(\langle g_{1j}g_{2j}\cdots g_{kj}\rangle)\right) \\
&= \frac{1}{2k}\left(\sum_{i=1}^{k}\frac{1}{k}\sum_{j=1}^{k}\rho_i(g_{ij}) + \sum_{j=1}^{k}\frac{1}{k}\sum_{i=1}^{k}\rho_j(g_{ij})\right) \\
&= \frac{1}{k^2}\sum_{i=1}^{k}\sum_{j=1}^{k}\frac{\rho_i(g_{ij}) + \rho_i(g_{ji})}{2}.
\end{aligned}$$

2.7 Résultats

Publié par les Éditions Larousse, *L'Officiel du Scrabble* (l'ODS) est un recueil de tous les mots de la langue française acceptés au Scrabble par la Fédération internationale de Scrabble francophone. La troisième édition sert de référence officielle depuis le 1^{er} janvier 1999. Toutefois, cette liste de mots est protégée par l'éditeur. Cependant, une liste informatisée se voulant être en accord avec l'ODS circule sur Internet. C'est cette liste non-officielle qui a été utilisée par l'auteur. Le tableau 1 résume les résultats obtenus.

Ce tableau précise le nombre de grilles hyperparfaites ($3^{\text{ème}}$ colonne) parmi le nombre total de grilles parfaites ($4^{\text{ème}}$ colonne) découvertes par l'algorithme,

TAB. 1 – Résultats obtenus par le 1er algorithme, langue française.

k	Mots	Grilles		Temps	
2	75	466	802	0,0	0,0
3	560	143844	230545	0,0	2,0
4	2319	17504964	22289255	0,0	9573
5	7184	390572048	496610547	0,1	–
6	16396	–	–	0,2	–
7	29610	–	–	2574	–
8	44109	–	–	–	–

lorsque ce dernier s'est terminé en un temps raisonnable, donné en secondes dans la sixième colonne. Le temps de la cinquième colonne est le temps requis pour trouver une première grille hyperparfaite. Notez que ces temps dépendent de la configuration matérielle de la machine ainsi que de la réalisation logicielle de l'algorithme. Les temps figurant dans ce document ont été obtenus à partir d'un programme C++ non optimisé compilé sous Microsoft Visual C++ .NET et s'exécutant sur un Pentium 4 à 2,4 GHz muni de 256 Mo de mémoire vive en environnement Windows XP.

2.8 Observations

Bien que le programme ne soit pas suffisamment rapide pour identifier l'ensemble des grilles parfaites, l'objectif de trouver rapidement une grille est atteint, du moins pour les tailles de grilles $k < 8$.

La permutation σ, calculée à partir de l'uniformité de la distribution des lettres, est cruciale pour la vitesse de l'algorithme. Par exemple, pour $k = 6$, seulement deux grilles sont identifiées en 2 secondes en omettant la permutation et plus de 500 en 1 secondes avec la permutation, dont celle-ci parmi les toutes premières :

S	**E**	**A**	**S**	**S**	**S**		**C**	**R**	**A**	**B**	**E**	**S**
E	**E**	**T**	**T**	**E**	**E**		**R**	**E**	**V**	**E**	**T**	**U**
U	**T**	**E**	**E**	**V**	**R**		**E**	**L**	**I**	**R**	**A**	**S**
S	**L**	**A**	**E**	**N**	**P**		**P**	**A**	**N**	**E**	**L**	**S**
S	**A**	**L**	**R**	**I**	**E**		**E**	**T**	**E**	**T**	**E**	**E**
S	**E**	**R**	**B**	**A**	**C**		**S**	**A**	**S**	**S**	**E**	**S**

Par contre, si l'objectif est d'identifier toutes les grilles parfaites, le rôle de la permutation σ n'est pas nécessairement bénéfique.

3 Un meilleur algorithme

3.1 Introduction

Nous allons améliorer notre algorithme afin d'en diminuer le temps d'exécution. Introduisons à l'alphabet A le symbole **?**[2] qui marquera l'absence de lettre. Nous désignerons cet alphabet étendu par $A^+ = A \cup \{\,\mathbf{?}\}$. Soit $G^+ = (g^+_{ij}) \in \mathbb{M}_k[A^+]$ une grille incomplète. Compléter G^+ consiste à trouver une grille parfaite $G = (g_{ij}) \in \mathbb{M}_k[A]$ telle que si $g^+_{ij} \in A$ alors l'égalité $g_{ij} = g^+_{ij}$ est vérifiée. Autrement dit, on remplace tous les **?** de G^+ par des lettres de A.

Remarquez que si G^+ ne contient que des points d'interrogation, toute grille parfaite G permet de compléter G^+. Nous dirons que les lettres de G^+, s'il y en a, sont des lettres imposées. L'algorithme décrit dans cette section est plus souple que le précédent car il permet de rechercher des grilles parfaites permettant de compléter n'importe quelle grille G^+.

3.2 De nouveaux tris

A partir des fonctions $\Psi_{E,j}$ définies dans la section 2.4, nous pouvons définir tout un ensemble de tris $\Omega_{E,j} : M \to \{1, 2, \ldots, |M|\}$. Ces tris sont tels que pour deux mots $m, w \in M$ et un sous-ensemble $E \subseteq \{1, 2, \ldots, k\}$,

$$\Omega_{E,j}(m) > \Omega_{E,j}(w) \iff \Psi_{E,j}(m) < \Psi_{E,j}(w).$$

On doit donc trier la liste de mots selon $2^k k$ fonctions de tris, puisque le nombre de sous-ensembles de $\{1, 2, \ldots, k\}$ est 2^k. L'espace mémoire requis est donc de $2^k k\,|M|$ pointeurs. Pour $k = 8$, $|M| \approx 2^{16}$ et 4 octets par pointeur, la taille mémoire requise est de 2^{29} octets, soit plus d'un demi gigaoctet.

Si cette consommation s'avérait trop coûteuse, une des solutions consiste à utiliser la fonction lissée

$$\overline{\Psi}_E(m) = \psi_E(m) + 1 - \frac{1}{k}\sum_{j=1}^{k} \phi_j(m).$$

On peut également remplacer les pointeurs par des indices sur 2 octets si $k < 2^{16}$. Le tableau 2 résume le nombre de pointeurs ou indices requis par ces tris pour les mots de l'ODS.

3.3 L'algorithme

Le principe de cet algorithme est simple : placer d'abord les mots dans les rangées (ou colonnes) pour lesquelles les contraintes sont plus fortes.

[2] Sous réserve qu'il n'en fasse pas déjà partie !

TAB. 2 – Pointeurs requis par les tris $\Omega_{E,j}$.

k	Mots	$\overline{\Psi}$	Ψ
2	75	300	600
3	560	4480	14k
4	2319	38k	149k
5	7184	225k	1123k
6	16396	1025k	6149k
7	29610	3702k	30M
8	44109	11M	87M
9	54620	27M	241M
10	57412	57M	562M
11	52357	103M	1124M
12	41696	163M	1955M

Considérons un exemple simple :

N	**A**	**Z**	**E**	**N**	**A**	**Z**	**E**	**N**	**A**	**Z**	**E**	**N**	**A**	**Z**	**E**
?	**?**	**?**	**?**	**?**	**?**	**I**	**?**	**?**	**?**	**I**	**?**	**O**	**?**	**I**	**?**
?	**?**	**?**	**?**	**?**	**?**	**N**	**?**	**?**	**?**	**N**	**?**	**R**	**?**	**N**	**?**
?	**?**	**?**	**?**	**?**	**?**	**C**	**?**	**D**	**E**	**C**	**U**	**D**	**E**	**C**	**U**

Une fois le premier mot placé, **NAZE**, la rangée ou la colonne pour laquelle il y a le moins de possibilité est la troisième colonne. On passe donc on traitement de cette dernière, dans laquelle on ajoute le mot **ZINC**. Il faut ensuite choisir parmi les $2^{\text{ème}}$, $3^{\text{ème}}$ et $4^{\text{ème}}$ rangées et les $1^{\text{ère}}$, $2^{\text{ème}}$ et $4^{\text{ème}}$ colonnes celle dont le nombre demots possible est le plus faible et ainsi de suite. L'avantage d'une telle méthode est que dans la situation où deux mots très probables mais conflictuels sont insérés, cet algorithme pourra corriger le tir immédiatement, contrairement au premier. Dans l'exemple suivant, la présence des deux **S** consécutifs dans la dernière colonne n'empêchera pas le premier algorithme d'essayer tous les mots possibles dans les autres rangées et colonnes :

M	**A**	**R**	**R**	**A**	**N**	**T**	**S**
E	**P**	**U**	**I**	**S**	**E**	**E**	**S**
L	**?**	**?**	**?**	**?**	**?**	**?**	**?**
A	**?**	**?**	**?**	**?**	**?**	**?**	**?**
N	**?**	**?**	**?**	**?**	**?**	**?**	**?**
G	**?**	**?**	**?**	**?**	**?**	**?**	**?**
E	**?**	**?**	**?**	**?**	**?**	**?**	**?**
R	**?**	**?**	**?**	**?**	**?**	**?**	**?**

Le nouvel algorithme se déroule donc de façon récursive, en $2k$ étapes. On commence par initialiser les variables globales $I = 1, 2, \ldots, k$, $J = 1, 2, \ldots, k$,

$G = G^+ = (g^+_{ij}) =$ **?** et les variables locales $s_i = 1$ et $t_i = |M|$ pour $i \in I$, $u_j = 1$ et $v_j = |M|$ pour $j \in J$.

Soit $K = \{1, 2, \ldots, k\}$. La fonction récursive comprends les opérations suivantes :

1. Si $I = \emptyset$ et $J = \emptyset$ alors G est une grille parfaite. On peut terminer l'algorithme ou backtracker pour poursuivre la recherche.
2. Si $I \neq \emptyset$, on trouve $i \in I$ la valeur telle que $t_i - s_i$ est minimal. En cas d'égalité entre deux ou plusieurs différences, on utilisera les Δ_i pour les départager et choisir celle dont la valeur correspondante est la plus grande. On effectue le même travail pour J s'il est non-vide. Finalement, on ne retiendra que le i ou le j en utilisant les mêmes critères de comparaison. On a donc choisi la rangée (ou colonne) dont le nombre de mots possibles est le plus bas. Pour la suite de l'algorithme, nous allons supposer qu'une valeur $i \in I$ a été choisie. Le déroulement est similaire pour une valeur $j \in J$.
3. Si $s_i > t_i$, il n'y a pas de mot pouvant se trouver dans la $i^{\text{ème}}$ rangée. On doit backtracker.
4. On retire i de I de façon à marquer cette rangée comme traitée.
5. Pour chaque mot $m \in M$ tel que $s_i \leq \Omega_{K\setminus J,i}(m) \leq t_i$, les trois étapes suivantes doivent être effectuées :
   (a) Pour tout $j \in J$, on place la lettre m_j dans la grille, en position g_{ij}. Il s'agit simplement d'insérer le mot choisi dans la grille.
   (b) Pour tout $j \in J$, on affine les bornes u_j et v_j, grâce aux tris $\Omega_{K\setminus I,j}$. Le fait d'ajouter un mot sur la $i^{\text{ème}}$ rangée de la grille a évidemment une répercussion sur le nombre de mots possibles dans les colonnes de la grille.
   (c) On effectue l'appel récursif.
6. On replace i dans I.

Dans le cas où la grille initiale G^+ contient des lettres imposées, on calculera les ensembles

$$\begin{aligned} I'_i &= \left\{ j \in \{1, 2, \ldots, k\} \mid g^+_{ij} \neq \textbf{?} \right\} \\ J'_j &= \left\{ i \in \{1, 2, \ldots, k\} \mid g^+_{ij} \neq \textbf{?} \right\} \end{aligned}$$

afin d'initialiser les bornes s_i, t_i à l'aide des tris $\Omega_{I'_i,i}$ et les bornes u_j, v_j à l'aide des tris $\Omega_{J'_j,j}$. Dans le corps de l'algorithme, on remplace $\Omega_{K\setminus J,i}$ par $\Omega_{(K\setminus J)\cup I'_i,i}$ et $\Omega_{K\setminus I,j}$ par $\Omega_{(K\setminus I)\cup J'_j,j}$.

Notez qu'en pratique il n'est pas nécessaire de calculer tous les tris $\Omega_{E,i}$. Il est possible de les calculer au besoin, ce qui réduira la quantité de mémoire requise telle que calculée au tableau 2.

TAB. 3 – Résultats obtenus par le $2^{\text{ème}}$ algorithme, langue française.

k	Mots	Grilles		Temps		Gain	
2	75	466	802	0,0	0,0	–	–
3	560	143844	230545	0,0	0,6	–	3×
4	2319	17504964	22289255	0,1	166,4	–	57×
5	7184	390572048	496610547	62,1	38385	0×	–
6	16396	–	–	0,8	–	$\frac{1}{4}$×	–
7	29610	–	–	50,7	–	51×	–
8	44109	–	–	–	–	–	–

3.4 Résultats

Toujours avec l'ODS, le tableau 3 résume les résultats obtenus. Les deux dernières colonnes contiennent le gain de vitesse par rapport au premier algorithme, dont les résultats figurent en page 8. Par exemple, on y apprend que cet algorithme est près de 60 fois plus rapide que le premier pour identifier toutes les grilles de 4×4. Le temps d'une minute requis pour trouver la première grille hyperparfaite de 5×5 s'explique par le fait que l'algorithme a identifié plus de 3,4 millions de grilles parfaites durant ce laps de temps.

De plus, la complétion de grilles avec lettres imposées fonctionne tout aussi rapidement. En imposant les lettres **DUPUIS** sur la diagonale d'une grille G^+ de 6×6, il ne faut que 6,3 secondes à l'algorithme pour identifier les 36 grilles parfaites (dont aucune n'est hyperparfaite) pouvant compléter G^+. Voici deux de ces grilles :

D	**R**	**A**	**P**	**A**	**S**
R	**U**	**M**	**I**	**N**	**A**
A	**M**	**P**	**L**	**E**	**S**
P	**I**	**L**	**U**	**M**	**S**
A	**N**	**E**	**M**	**I**	**E**
S	**A**	**S**	**S**	**E**	**S**

D	**O**	**R**	**A**	**N**	**T**
O	**U**	**A**	**T**	**A**	**I**
R	**A**	**P**	**O**	**N**	**S**
A	**T**	**O**	**U**	**T**	**S**
N	**A**	**N**	**T**	**I**	**E**
T	**I**	**S**	**S**	**E**	**S**

4 Conclusion

Le deuxième algorithme s'avère supérieur au premier en terme de rapidité et de souplesse. Cependant, sa consommation excessive de mémoire nous obligera à faire des compromis pour les grilles de taille k supérieure à 8.

Nous terminons par deux comparaisons fort instructives. La première s'intéresse au nombre de grilles parfaites en anglais tandis que la deuxième s'intéresse à la performance des algorithmes sous divers compilateurs C++.

Tab. 4 – Résultats comparatifs entre l'anglais et le français

	Français :			Anglais :		
k	Mots	Grilles	Temps	Mots	Grilles	Temps
2	75	802	0,0	96	2035	0,0
3	560	230545	0,6	978	1966600	3,8
4	2319	22289255	166,4	3919	139981287	868,5
5	7184	496610547	38385	8672	84665864	34350
6	16396	–	–	15290	–	–
7	29610	–	–	23208	–	–
8	44109	–	–	28558	–	–

4.1 Grilles parfaites en anglais

L'*Official Scrabble Player's Dictionary*, publié par les éditions Merriam-Webster, est aux joueurs anglophones l'équivalent de *L'Officiel du Scrabble*. Cependant, tout comme son homologue francophone, la reproduction de cette liste est interdite par les droits d'auteurs. Les internautes anglophones amateurs de jeux de lettres utilisent donc une liste de mots alternative, l'*Enhanced North American Benchmark Lexicon*, dont la dernière version est intitulée *Enable2000*.

La comparaison entre ces langues apparaît au tableau 4, où on retrouve pour chacune d'entre elles le nombre de mots, le nombre de grilles parfaites identifiées ainsi que le temps requis (en secondes) pour la recherche de ces grilles, effectuée à l'aide du deuxième algorithme.

La différence entre le nombre de grilles pour les deux langues est considérable. Bien qu'il y ait six fois plus de grilles de 4×4 en anglais qu'en français, on observe exactement le contraire pour les grilles de 5×5. Dans le premier cas, le plus grand nombre de mots anglais (3919 contre 2319) est responsable de la différence tandis que dans le deuxième, ce sont les nombreux accords de verbe en français qui permettent de démultiplier le nombre de grilles, qui atteint près d'un demi milliard. Il est probable que plus les grilles seront de tailles importantes, plus l'écart entre le nombre de grilles françaises et anglaises se creusera, hypothèse qui reste à vérifier bien sûr.

4.2 Performance de divers compilateurs

Difficile de résister, lorsqu'on écrit un tel algorithme, de comparer les divers compilateurs qui s'offrent sur le marché. Sous Windows, les compilateurs des compagnies Borland, Intel, Metrowerks et Microsoft sont les principaux disponibles, auxquels on ajoute les projets Digital Mars, GNU et Watcom. Le traducteur de C++ vers C de Comeau Computing est également une option à

considérer.

4.2.1 Borland C++ Compiler 5.5.1

Depuis un certain temps déjà, Borland distribue gratuitement le compilateur de son environnement de développement C++ Builder. La compagnie loue la puissance de son compilateur en terme de rapidité et de respect de la norme ANSI mais ne fait aucune allusion à la vitesse d'exécution du code. Notez que Borland est le seul compilateur ne permettant pas d'utiliser les fonctions C standard hors du namespace std, ce qui correspond exactement aux consignes du standard sur ce point.

Les principales options de compilation retenues furent :

1. **-A** : Use ANSI keywords and extensions.
2. **-O2** : Generate fastest possible code.
3. **-5** : Generate Pentium instructions.

4.2.2 Comeau C/C++ 4.3.0

Disponible à un faible coût, ce pseudo-compilateur transforme le code C++ en C, qui doit être compilé à son tour, avec un compilateur tel que gcc. Le site web[3] vante l'excellence du compilateur en dénigrant ceux des concurrents :

> Sick and tired of running into problems with your "major brand" (take your pick) compiler ? [...] Comeau C/C++ is stable, reliable and solid on all fundamental language features, and firmly supports all up-to-date language features. Can your "major brand" vendor say that ? Or, are they so poorly supported and bug-ridden that they make you pay for tech support ?

Une version d'évaluation n'étant pas disponible au public, ce compilateur n'a pas été testé.

4.2.3 Digital Mars C++ Compiler 8.32

Toujours en développement par Walter Bright, ce compilateur est disponible gratuitement. Projet personnel, de grande envergure certes, ce compilateur semble encore incomplet. Des problèmes de compilation ont eu lieu à propos du namespace std. Bien que l'auteur affirme que son compilateur est compatible avec la STL de SGI, je n'ai pas réussi à la compiler.

[3] `http://www.comeaucomputing.com/`

4.2.4 Intel C++ Compiler 7.0 Build 20021018Z

Disponible pour une période d'évaluation de 30 jours, la force de ce compilateur résiderait en sa capacité à d'optimisation du code. Selon Intel[4] :

> The Intel C++ Compiler 7.0 delivers optimal performance on Intel 32 bit processors [...]. It offers improvements to its advanced optimization features, such as vectorization and Streaming SIMD Extensions 2 (SSE2) for IA 32, software pipelining for the Itanium/Itanium 2 architecture, and Interprocedural Optimization and Profile Guided Optimization. [...] The Intel Compiler has added extended language directives for software pipelining, loop unrolling and data prefetching, that provide information to the compiler to enhance optimization of application code. In addition, it supports industry standards such as ANSI C/C++ and ISO C/C++.

La documentation qui accompagne le compilateur est précise et bien fournie. De plus, ce dernier s'intègre entièrement dans l'environnement de développement de Microsoft.

Les principales options de compilation retenues furent :

1. **-Za** : Enforces strict conformance to the ANSI standard for C.
2. **-O3 -Qansi_alias** : Optimizes for speed assuming no type aliasing and enables high-level optimization.
3. **-QaxW** : Generate specialized code for Pentium 4, generating also generic code for other processors.
4. **-Qipo** : Enables interprocedural optimizations across files.

Aucune directive **#pragma** particulière n'a été insérée manuellement dans le code par l'auteur.

4.2.5 Metrowerks CodeWarrior 8

Une version d'évaluation gratuite de cet environnement de développement complet est disponible par courrier. Les frais de transports et de manutention sont cependant prohibitifs, cinquante dollars américains (!) au moment de la rédaction de ce document. D'après l'éditeur, ce logiciel est le meilleur de l'industrie[5] :

> Best-in-class ANSI/ISO C/C++ Compiler – The CodeWarrior optimizing compiler produces compact and efficient code so that you can create smaller and faster applications.

Ce compilateur n'a pas été testé.

[4] `http://www.intel.com/software/products/compilers/cwin/whatsnew.htm`

[5] `http://www.metrowerks.com/MW/Develop/Desktop/Windows/Professional/Default.htm\#`

4.2.6 Microsoft C/C++ Optimizing Compiler 13.00.9466

Ce compilateur, partie intégrante de l'environnement de développement Visual Studio .NET, peut être obtenu gratuitement via le package Microsoft .NET Framework SDK. Tiré du site web de Microsoft[6], voici un commentaire sur la performance du code produit :

> The Visual C++ .NET compiler offers developers a slew of new and improved code-generation and tuning features, among them Whole Program Optimization. Whole Program Optimization allows the compiler to perform optimizations using information on all modules in the program, instead of traditional optimizations that can see only a single module at a time. With information on all modules, the compiler can optimize the use of registers across function boundaries and inline functions in a module even when the function is defined in another module. Whole Program Optimization results in dramatic increases in application performance for very little cost.

Cependant, ce qui semble un bogue de compilation a dû être contourné. Le compilateur n'arrive pas à associer une déclaration à sa définition correspondante pour une méthode donnée du code. Le message d'erreur précise la définition et la déclaration, qui sont identiques à l'écran ! Plus précisément, le fragment de code suivant provoque l'erreur de compilation :

```
template <class T, int N>
class array {
};

template <int N>
class object {
  void function(const array<int, N>&);
};

template <int N>
void object<N>::function(const array<int, N>&) {
}
```

Les principales options de compilation retenues furent :

1. **-Za** : Disables language extensions, thus improving ANSI compatibility.
2. **-Ox -Oa** : Maximum optimization assuming no type aliasing.
3. **-GL** : Enables whole program optimization.
4. **-G5** : Optimizes code to favor the Pentium.

[6] `http://msdn.microsoft.com/visualc/productinfo/topten/upgrade.asp`

4.2.7 MinGW 20020817-1 / GNU Compiler Collection 3.2

L'ensemble de compilateurs GNU est l'un des projets phare de la Free Software Foundation. Débuté il y a plus de vingt ans, c'est certainement le meilleur ensemble de compilateurs gratuits disponible sur le marché. MinGW est la traduction pour l'environnement Windows de ce jeu de compilateurs. Aujourd'hui un projet open-source à part entière, MinGW semble avoir encore de beaux jours devant lui. Le seul reproche qu'on pourrait lui faire est le nombre délirant d'options de compilation !

Les principales options de compilation retenues furent :

1. **-ansi -pedantic** : Remove all GNU extensions to support only ANSI/ISO C++.
2. **-O3** : Turn on aggressive optimization.
3. **-march=pentium** : Generate code for the Pentium processor.
4. **-fstrict-aliasing -fmove-all-moveables -funroll-loops -frerun-loop-opt -fstrength-reduce** : Enable various advanced optimizations.

4.2.8 Watcom 11.0c

Le compilateur de Watcom est aujourd'hui un projet open-source. Le compilateur a cependant de la difficulté à interpréter (le programme « plante »...) la ligne de commande en raison des noms de fichiers contenants des espaces, des traits-d'unions, etc. De plus la STL n'est pas disponible. En incluant les fichiers entêtes distribués avec un autre compilateur, le programme »plante « à nouveau. Il semble donc que ce soit peine perdue de compiler un programme C++ moderne avec Watcom.

4.2.9 Résultats

Trois tâches furent soumises aux quatre compilateurs retenus :

1. La découverte des 143 844 grilles hyperparfaites de 3×3 à l'aide du premier algorithme.
2. La découverte des 32 grilles parfaites de 6×6 avec les six lettres **DUPUIS** imposées sur la diagonale.
3. La découverte des $22,3$ millions de grilles parfaites de 4×4 à l'aide du deuxième algorithme.

Les résultats figurent au tableau 5. Pour chacune des tâches, la première colonne représente le temps en secondes requis tandis que la deuxième colonne est une note donnée au compilateur par rapport au plus rapide, qui obtient 100%.

L'analyse du deuxième algorithme permet d'expliquer la bonne performance du compilateur d'Intel par rapport à celui de Microsoft pour la deuxième tâche. En

Tab. 5 – Résultats comparatifs entre divers compilateurs

Tâche	$1^{\text{ère}}$		$2^{\text{ème}}$		$3^{\text{ème}}$	
Borland	$4,4$	$40,9\%$	$9,9$	$60,6\%$	$331,5$	$50,0\%$
Gnu	$2,6$	$69,2\%$	$7,0$	$85,7\%$	$217,3$	$76,3\%$
Intel	$2,3$	$78,3\%$	$6,0$	100%	$178,1$	$93,1\%$
Microsoft	$1,8$	100%	$6,4$	$93,8\%$	$165,8$	100%

effet, cet algorithme consiste d'abord en le calculs des tris puis en la construction récursive des grilles. Le tri des 192 listes de mots (tableau 2) pour les grilles de 6×6 est fait de façon plus efficace par le compilateur d'Intel que par celui de Microsoft, qui se reprend sur la construction des grilles. Cette différence est confirmée par la troisième tâche, où le temps requis pour effectuer les tris et négligeable par rapport au temps total d'exécution, d'où la meilleure performance du compilateur de Microsoft.

Au niveau performance brute, les compilateurs d'Intel et de Microsoft se démarquent clairement du lot, tandis qu'au niveau de la facilité d'utilisation des compilateurs et du support du standard, ces derniers sont tous plus ou moins équivalents. Seul Microsoft aura posé quelques soucis de compilation parmi les produits dits « professionnels »tandis qu'au niveau « amateur », le compilateur GNU n'a pas d'équivalent. Ces résultats démontrent que le choix d'un compilateur pour ce type d'algorithme peut avoir une influence considérable sur la vitesse d'exécution du code.