Jouer avec des Cryptarithmes en Programmation par Contraintes

Arnaud Malapert[1], Margaux Schmied[2], Davide Fissore[2], Marie Pelleau[1], Ambre Picard Marchetto[2]

[1] Université Côte d'Azur, CNRS, I3S, France
[2] Université Côte d'Azur, France

[1] firstname.lastname@univ-cotedazur.fr [2] firstname.lastname@etu.univ-cotedazur.fr

Résumé

Un cryptarithme est un casse-tête mathématique et logique dans lequel des mots forment une équation où les lettres représentent des chiffres à déterminer dans une base donnée. Ce problème est populaire en mathématiques récréatives, dans l'enseignement, et en programmation par contraintes. Nous proposons une approche générale, efficace et simple d'utilisation pour résoudre ce problème NP-Complet. La suite naturelle est une approche hiérarchique pour leur génération. Les évaluations expérimentales ont engendré une vaste collection, variée et remarquable.

Mots-clés

Cryptarithme ; puzzle ; résolution ; génération.

1 Introduction

Un cryptarithme est un casse-tête mathématique et logique dans lequel un ensemble de mots est écrit sous la forme d'une équation où les lettres représentent des chiffres à déterminer dans une base donnée. Il faut remplacer les lettres par des chiffres en respectant les règles suivantes :

- Règle i) Chaque lettre est associée à un seul chiffre.
- Règle ii) L'équation est vérifiée dans une base donnée en remplaçant les lettres par les chiffres.
- Règle iii) Les lettres représentent des chiffres distincts.
- Règle iv) Le chiffre de poids fort des mots est non nul.

Les deux dernières règles sont quelquefois relâchées pour obtenir une solution. Idéalement, il existe une solution unique.

Si l'invention des cryptarithmes remontent à la Chine antique, le cryptarithme le plus connu a été publié en juillet 1924 dans le *Strand Magazine* [8] par H.E. Dudeney :

$$\begin{array}{r} \texttt{SEND} \\ +\ \texttt{MORE} \\ \hline \texttt{MONEY} \end{array} \qquad \begin{array}{r} 9567 \\ +\ 1085 \\ \hline 10652 \end{array}$$

$$\texttt{S} = 9; \texttt{E} = 5; \texttt{N} = 6; \texttt{D} = 7; \texttt{M} = 1; \texttt{O} = 0; \texttt{R} = 8; \texttt{Y} = 2$$

Malgré son apparente simplicité, le problème est NP-complet [9] ce qui est prouvé par une réduction de 3-SAT.

Dans sa forme originelle, un cryptarithme est une addition avec un second membre réduit à un seul terme, et les nombres sont écrits en base 10. Mais, il existe d'autres variantes dignes d'intérêt présentées ultérieurement.

Ce puzzle est populaire en mathématiques récréatives comme en attestent les différentes publications et ressources glanées sur internet. Il existe plusieurs livres classiques [3, 10, 13] ou plus récents [7, 14] au contenu similaire avec un catalogue de puzzles classés par niveau avec leurs solutions accompagnés de conseils et méthodes de résolution « à la main ». Les cryptarithmes sont aussi présents dans des magazines, par exemple le journal *Sphinx* publié en Belgique en langue Française dans les années 1930 qui organisait aussi des compétitions. De nos jours, ils sont toujours utilisés dans les compétitions de la fédération Française des jeux mathématiques. Il existe aussi plusieurs sites web [6, 12, 19–21] consacrés à ce puzzle sur lesquels on trouvera des contenus similaires aux livres souvent avec des solveurs ou générateurs de cryptarithmes. Les solveurs disponibles utilisent le plus souvent des algorithmes de recherche exhaustive plus ou moins avancés et efficaces.

Ce puzzle est aussi populaire en enseignement, de l'école primaire à l'université. En primaire et au collège, il permet d'acquérir des compétences en résolution de problème, logique et calcul ce qui a fait l'objet de travaux en sciences de l'éducation [4, 22]. Au lycée et à l'université, c'est un problème simple et structuré qui est donc adapté pour l'apprentissage et la programmation de recherches exhaustives. La difficulté de programmer des recherches exhaustives générales et efficaces permet de présenter les avantages de la modélisation à l'université. C'est particulièrement vrai en programmation par contraintes [18] où $\texttt{SEND} + \texttt{MORE} = \texttt{MONEY}$ apparaît dans la majorité des tutoriels et manuels des solveurs de contraintes. Cependant, le problème n'a jamais fait l'objet d'une publication et n'apparaît pas non plus dans la CSPLib [1].

Les approches actuelles pour résoudre un cryptarithme montrent plusieurs limites. Résoudre un cryptarithme en décimal est un problème difficile pour les humains qui devient extrêmement difficiles dans les autres bases où nos capacités de calcul sont bien moindres.

Programmer une recherche exhaustive générale et efficace est une tâche complexe et la plupart des solveurs disponibles imposent des restrictions sur les instances du problème. Deux restrictions courantes sont de se limiter à une addition ou à la base décimale. Une autre restriction fréquente est de se limiter à l'arithmétique simple précision ce qui limite la taille des mots de l'équation.

Écrire un modèle spécifique est une tâche plus aisée, mais

cela ne résout pas le problème, seulement une instance ! Écrire un modèle général est une tâche plus difficile qui n'a jamais été réalisée à notre connaissance.
Finalement, il est temps de mettre à l'épreuve la réputation de facilité du problème et les performances de la programmation par contraintes. Les deux objectifs principaux sont de participer à la diffusion scientifique de la programmation par contraintes et d'être utilisé pour la création de contenu pédagogique dans l'enseignement.
La première contribution est de proposer un solveur de cryptarithme basé sur la programmation par contraintes qui soit général, efficace, et facile d'utilisation. Les solveurs disponibles satisfont au mieux deux de ces trois critères, abandonnant le plus souvent la généralité.
Assez naturellement, la seconde contribution est de proposer des générateurs de cryptarithmes, et de variantes, à partir d'une liste de mots. En effet, la découverte « à la main » d'un nouveau cryptarithme reste réservée aux experts et les générateurs disponibles sont plus limités que les solveurs. La génération de cryptarithmes constitue un défi plus difficile que leur simple résolution.
Ces contributions prennent la forme d'une bibliothèque Java CRYPTATOR [1] sous licence libre basée sur le solveur Choco [5] proposant deux applications en ligne de commandes.
La suite de ce document est structurée de la manière suivante. La section 2 introduit une forme générale de cryptarithme définie par une grammaire et présente plusieurs variantes du puzzle. La section 3 décrit un modèle pour la résolution de cryptarithme en arithmétique simple précision, et un modèle restreint aux additions en arithmétique multiprécision. La section 4 décrit plusieurs modèles avec un socle commun pour la génération de différentes variantes à partir d'une liste de mots. La section 5 présente l'évaluation expérimentale des performances des solveurs. La section 6 présente les expériences pour créer une vaste collection de cryptarithmes aux propriétés variées et remarquables. La section 7 décrit les choix technologiques et des éléments d'architecture de la bibliothèque, et présente quelques cas d'usage.

2 Définition d'un cryptarithme

Un cryptarithme est défini par une équation mathématique et une base arithmétique. Une équation est définie par une grammaire présentée en section 2.1 pour la facilité d'utilisation. Par défaut, on suppose que les règles i, ii, iii, et iv de la section 1 doivent être satisfaites par une solution. Cependant, nous verrons en section 3 que les règles iii et iv peuvent être relâchées.
À partir de ces entrées, on peut poser plusieurs questions : existence d'une solution ; existence d'une solution unique ; énumération des solutions. Historiquement, par un souci d'élégance, l'existence d'une solution unique est la question fondamentale.
La section 2.2 introduit plusieurs variantes de cryptarithmes considérées dans ce travail.

1. https://github.com/arnaud-m/cryptator

2.1 Grammaire d'une équation

La grammaire sert à simplifier la définition d'une équation. Elle est exprimée en *Extended Backus-Naur Form* (EBNF), afin de ne pas allourdir cet article nous ne donnons pas ici la grammaire.
La grammaire permet la reconnaissance d'une forme infixe, la forme classique que les humains manipulent, mais elle n'est pas nécessaire pour la reconnaissance des formes préfixe ou suffixe. Elle permet aussi de détecter et d'expliquer des erreurs lors de la saisie d'une équation.
La représentation d'un cryptarithme se base sur la construction d'un arbre syntaxique réalisé à partir d'un parseur. Ce parseur est capable de reconnaître les cryptarithmes classiques, mais il est plus général pour capturer ou imaginer des variantes. La grammaire est une grammaire hors contexte dont les règles de dérivation suivent la structure suivante $X \rightarrow \alpha$ où X est un symbole non terminal et α est une suite de symboles terminaux ou non terminaux.
La vérification de la structure d'une grammaire est subdivisée en deux parties essentielles : un *lexeur*, ayant le rôle de renvoyer sous forme de *tokens* les mots reconnus à partir du texte entré et un *parseur* qui vérifie la bonne structure syntactico-sémantique de la suite des *tokens* reçus par le *lexeur*.
En particulier, le lexeur distingue les *termes* (des suites alphanumériques composées des symboles *UTF-8*), les *comparateurs* tels que l'égalité et l'inégalité stricte ou large, et les *opérateurs* tels que l'addition, la multiplication, la soustraction, la division, l'élévation à la puissance et le modulo. De plus, l'opérateur de conjonction de cryptarithmes && ou ; permet de séparer une équation en plusieurs cryptarithmes. Tous ces opérateurs suivent les règles de priorités usuelles. Le parseur accepte les parenthèses pour permettre la manipulation des priorités d'opération. Tous les espaces, tabulations ou sauts de ligne sont autorisés et ignorés pour procurer un confort d'utilisation.
Le lexeur distingue deux types de terme. Un *mot* est une séquence de lettres à remplacer par des chiffres. Un *nombre* est une séquence de chiffres décimaux dont la valeur décimale est utilisée pour vérifier l'équation. Dans la grammaire, un nombre est délimité par des guillemets simples ou doubles. Ainsi, le cryptarithmes `"11" + 89 = "40"` contient deux constantes, 11 et 40, et un mot `89`. Dans la solution, les lettres (`8` et `9`) peuvent prendre des valeurs différentes de leur propre sémantique (2 et 9).
L'opérateur de conjonction et les nombres ont été introduits pour capturer certaines variantes introduites ci-dessous.

2.2 Variantes

Au cours de son histoire, de nombreuses variantes de cryptarithmes ont été introduites. Ici, nous nous limiterons à celles présentes dans la collection de la section 6.

2.3 Doublement vrai

Un cryptarithme est doublement vrai si chaque mot est un nombre écrit en toute lettre et que l'équation textuelle est elle aussi vérifiée. Ci-dessous, l'équation textuelle est à gauche, le cryptarithme au milieu, et sa solution droite.

```
   0        CERO      8027
 + 6      + SEIS    + 3013
 + 7      + SIETE   + 31040
------    -------   -------
  13        TRECE     42080
```

2.4 Mots croisés

Un mots croisés est une variante dans laquelle plusieurs équations sont écrites dans une grille et une seule affectation des chiffres aux lettres doit permettre de vérifier toutes les équations.

```
 AN + TA | DOL      87 + 38 | 125
  +      +  +       +      +  +
ODE + TEL | LAD    216 + 365 | 581
-----------------  -----------------
TUT + SUT | NUE    303 + 403 | 706
```

Donc, chaque ligne et chaque colonne représente une équation et il faut résoudre la conjonction de ces cryptarithmes donnée dans la grammaire ci-dessous.

$$\texttt{AN} + \texttt{TA} = \texttt{DOL}; \texttt{ODE} + \texttt{TEL} = \texttt{LAD}; \texttt{SUT} + \texttt{TUT} = \texttt{NUE};$$
$$\texttt{AN} + \texttt{ODE} = \texttt{TUT}; \texttt{TA} + \texttt{TEL} = \texttt{SUT}; \texttt{DOL} + \texttt{LAD} = \texttt{NUE}$$

2.5 Multiplications

On va distinguer plusieurs types de multiplications. D'abord, la forme classique de la multiplication courte a un seul terme dans le second membre.

$$\texttt{GREY} \times \texttt{BLUE} = \texttt{DARKBLUE}$$
$$8601 \times 3450 = 29673450$$

Il y a aussi les multiplications courtes doublement vrai.

$$\texttt{CINQ} \times \texttt{SIX} = \texttt{TRENTE}$$
$$5409 \times 142 = 768078$$

Et, il y a des multiplications courtes avec plusieurs termes dans le second membre comme dans $\texttt{ORC} \times \texttt{FREAK} = \texttt{ELF} \times \texttt{FAIRY}$.
Une dernière variante amusante est la multiplication longue, une méthode de calcul classique enseignée à l'école.

```
   MU        16
 × MU      × 16
-----     -----
   NU        96
 + MU•     + 160
-----     -----
  TAU       256
```

Une multiplication longue décompose le produit d'une multiplicande et d'un multiplicateur en une somme des produits du multiplicande avec la valeur de chaque chiffre du multiplicateur. Une multiplication longue est divisée en trois parties : le produit du multiplicande et du multiplicateur ; la somme des produits partiels, et le résultat du produit.
Pour définir un tel cryptarithme, la grammaire a besoin de l'opérateur de conjonction comme pour les mots croisés, mais aussi de la multiplication par un nombre, ici une puissance de la base.

$$\texttt{MU} * \texttt{MU} = \texttt{TAU}; \texttt{MU} * \texttt{M} = \texttt{MU}; \texttt{MU} * \texttt{U} = \texttt{NU};$$
$$\texttt{NU} * \texttt{'1'} + \texttt{MU} * \texttt{'10'} = \texttt{TAU};$$

3 Résolution d'un cryptarithme

Nous expliquons ici la compilation d'un cryptarithme vers un modèle en programmation par contraintes. Cette explication est divisée en trois parties. La section 3.1 modélise l'affectation des chiffres aux lettres, soit l'application des règles i, iii, et iv. La section 3.2 modélise l'équation en arithmétique simple précision, soit l'application de la règle ii. Et, la section 3.3 modélise l'équation en arithmétique multiprécision, mais est restreint aux additions. En base décimale, la valeur d'un mot de dix caractères n'est plus représentable en simple précision.

3.1 Affectation des chiffres aux lettres

Un mot w de longueur $|w|$ est représenté par une suite finie de lettres $w = l_{|w|-1} l_{|w|-2} \dots l_1 l_0$. L'entrée d'un cryptarithme définit un ensemble W de mots et un alphabet A, l'union des lettres des mots.

$$A = \bigcup_{w \in W} \bigcup_{i=0}^{|w|-1} \{l_i\}$$

Soit b la base arithmétique du cryptarithme, la variable entière $x_l \in [0, b-1]$ représente le chiffre associé à la lettre $l \in A$ imposant ainsi la règle i.
Le nombre d'occurrences de chaque chiffre parmi les lettres est imposé par la contrainte *global cardinality* [17]. On généralise la règle iii pour traiter le cas où il y a plus de lettres que de chiffres dans la base.

$$\left\lfloor \frac{|A|}{b} \right\rfloor \leq |\{l \in A | x_l = v\}| \leq \left\lceil \frac{|A|}{b} \right\rceil \tag{1}$$

La règle iv est imposée par une contrainte arithmétique.

$$x_{l_{|w|-1}} > 0 \qquad \forall w \in W \tag{2}$$

3.2 Arithmétique simple précision

L'équation est construite en créant des variables auxiliaires associées aux valeurs des mots et en utilisant l'arithmétique du solveur pour vérifier l'équation (règle ii).
La variable auxiliaire V_w représente la valeur du mot $w \in W$ dans la base b. Elle peut être définie par la méthode d'exponentiation.

$$V_w = \sum_{i=0}^{|w|-1} b^i \times x_{l_i} \qquad \forall w \in W \tag{3}$$

Une alternative est la méthode de Ruffini-Horner.

$$V_w = ((\dots((b x_{l_{|w|-1}} + x_{l_{|w|-2}})b + x_{l_{|w|-3}})b + \dots)b + x_{l_1})b + x_{l_0} \qquad \forall w \in W \tag{4}$$

3.3 Arithmétique multiprécision

En multiprécision, il n'est plus possible d'utiliser une variable auxiliaire représentant la valeur d'un mot pour éviter un dépassement de capacité. Il est quelquefois possible de changer l'arithmétique d'un programme, mais c'est plus

compliqué pour un solveur. Il n'existe pas non plus de solveurs multiprécision à notre connaissance. Donc, la seule solution est de proposer un modèle pour le calcul multiprécision. La difficulté est que le résultat d'une opération devient le résultat d'un algorithme et non plus d'une opération gérée par le processeur. Pour le moment, le modèle est restreint à l'addition qui est l'opération la plus facile, mais aussi la plus importante pour un cryptarithme.
Le modèle calcule la somme des chiffres des opérandes (mots ou nombres) à chaque position, puis propage les retenues pour calculer le résultat final de l'addition. Soit W^1 (resp. W^2) la suite des termes du membre gauche (resp. droit) de l'addition. Chaque mot ou nombre peut être répétés dans plusieurs termes. Soit $m = \max_W |w| - 1$ la position maximale dans un mot, la variable $S_i^j \in \mathbb{N}^+$ représente la somme des chiffres en position i des opérandes du membre j.

$$\sum_{\substack{w \in W^j \\ |w| > i}} x_{l_i} = S_i^j \qquad \forall i \in [0, m], \forall j \in [1, 2] \tag{5}$$

Ensuite, soit la variable auxiliaire $D_i^j \in [0, b-1]$ représentant le chiffre à la position i du résultat de l'addition du membre j, et la variable auxiliaire $C_i^j \in \mathbb{N}^+$ représentant la retenue à la position i. Les chiffres du résultat sont calculés en propageant les retenues de l'addition sauf la dernière.

$$b \times C_i^j + D_i^j = S_i^j + C_{i-1}^j \quad \forall i \in [1, m], \forall j \in [1, 2] \tag{6}$$
$$b \times C_0^j + D_0^j = S_0^j \qquad \forall j \in [1, 2] \tag{7}$$

La dernière retenue n'est pas propagée et la valeur de C_m^j peut donc être supérieure à la base b.
Pour vérifier l'égalité des résultats, tous les chiffres des résultats et la dernière retenue des membres droit et gauche doivent être égaux.

$$D_i^1 = D_i^2 \qquad \forall i \in [1, m] \tag{8}$$
$$C_m^1 = C_m^2 \tag{9}$$

4 Génération de cryptarithmes

La génération de cryptarithmes est une tâche bien plus ardue que leur résolution, car l'explosion combinatoire du choix des mots et de l'équation est considérable. Une méthode naturelle pour la contrôler est de restreindre le dictionnaire et la forme de l'équation.
Les méthodes de génération prennent en entrée une liste de mots et énumèrent sous contraintes des cryptarithmes candidats. Chaque candidat est résolu et la décision est prise de le conserver ou non.
Certaines restrictions sont assez naturelles. Un cryptarithme n'est réellement amusant qu'avec des mots d'une langue naturelle. De plus, la difficulté pour le construire est bien moindre avec des mots quelconques, car le calcul devient facile. Résoudre un cryptarithme dans une base non décimale est plus difficile et moins amusant. L'élégance demande à ce que la solution soit unique. Sauf mention du contraire, les restrictions suivantes sont appliquées :

1. Les mots sont en langue naturelle.
2. Il n'y a pas de répétition de mots.
3. Les lettres représentent des chiffres distincts.
4. La base est décimale.
5. La solution est unique.

Chaque modèle est spécifique au type de cryptarithme, mais ils partagent un socle commun pour la sélection des mots. Chaque modèle spécifique impose les restrictions communes et des contraintes supplémentaires sur les longueurs des mots du cryptarithme. En effet, chaque modèle intègre des raisonnements sur la longueur du résultat d'une opération en fonction de la longueur de ses opérandes. Ces raisonnements ne seront pas présentés en détail, car les explications seraient trop longues et ils manquent de généralité. Remarquez qu'aucun modèle de génération ne considère l'affectation, partielle ou totale, des chiffres aux lettres du candidat qui est laissée entièrement à la charge de la résolution. En d'autres termes, lors de la génération rien ne certifie qu'il puisse exister une solution et que celle-ci est unique.
Prouver qu'un cryptarithme ne provoque pas de dépassement d'entier n'est pas trivial. À titre exemple, le cryptarithme suivant admet une solution unique.

$$\mathtt{T}^{\mathtt{E}} \times \mathtt{S}^{\mathtt{T}} = \mathtt{TEST}$$
$$2^5 \times 9^2 = 2592$$

Le membre de droite étant sur quatre caractères cela nous assure de ne pas avoir de dépassement d'entier. Cependant si on ne considère que le membre de gauche, celui-ci peut provoquer un dépassement d'entier. L'ordre dans lequel les contraintes associées à cette équation vont être construites dépend du solveur sous-jacent. C'est pourquoi nous n'intégrons pas de vérification au moment de la génération et nous nous reposons uniquement sur le solveur.

4.1 Sélection d'un ensemble de mots

Le modèle de sélection des mots joue un rôle central. Il définit des variables booléennes de décision y_w qui indiquent la présence d'un mot w de l'ensemble W. Les variables auxiliaires N et L représentent le nombre d'opérandes et la longueur du plus long mot.

$$N = \sum_W y_w \tag{10}$$
$$L = \max_W |w| \times y_w \tag{11}$$

Le modèle déclare des contraintes pour que les lettres représentent des chiffres distincts. Il est nécessaire que le nombre de lettres distinctes soit inférieur ou égal à la base b. La variable booléenne X_l indique la présence de la lettre $l \in A$. Soit $W_l \subseteq W$ l'ensemble des mots avec la lettre l.

$$W_l = \{w \in W | \exists i \in [0, |w| - 1], l_i = l\} \qquad \forall l \in A$$

La contrainte (12) impose que la variable X_l soit vraie si et seulement si un mot de W_l est présent. Soit b la base arithmétique du cryptarithme, la contrainte (13) impose que

les lettres représentent des chiffres distincts.

$$X_l = \bigvee_{W_l} y_w \qquad \forall l \in A \tag{12}$$

$$\sum_{A} X_l \leq b \tag{13}$$

4.2 Addition et multiplication

La variable de décision y_w^1 (resp. y_w^2) indique la présence d'un mot $w \in W$ dans le membre gauche (resp. droit). Les variables auxiliaires associées sont aussi définies par les contraintes (10) et (11). Par contre, les contraintes (12) et (13) pour les lettres distinctes ne sont pas définies, car elles seraient redondantes.
La contrainte (14) interdit la répétition des mots à gauche et à droite. La contrainte (15) brise partiellement les symétries.

$$y_w = y_w^1 + y_w^2 \tag{14}$$

$$L^1 \leq L^2 \tag{15}$$

Des contraintes supplémentaires sur la longueur du résultat en fonction de la longueur des opérandes sont aussi déclarées. Elles ne sont pas présentées ici, car les explications seraient trop longues et elles manquent encore de généralité.

4.3 Doublement vrai

On va simplement étendre les modèles pour l'addition et la multiplication. Soit v_w la valeur du nombre écrit dans le mot $w \in W$, on ajoute la contrainte (16) pour l'addition.

$$\sum_{W} v_w \times y_w^1 = \sum_{W} v_w \times y_w^2 \tag{16}$$

Pour la multiplication, poser la contrainte de produit équivalente est difficile en programmation par contraintes à cause de l'arithmétique simple précision. Passer au logarithme nécessite la gestion des domaines réels ou flottants ce qui n'est pas fréquent. En pratique, on calcule une approximation naïve de la contrainte (17) en normalisant et arrondissant au plus près le logarithme.

$$\left| \sum_{W} \log(v_w) \times (y_w^1 - y_w^2) \right| \leq \epsilon \tag{17}$$

4.4 Mots croisés et multiplication longue

Nous ne détaillerons pas les modèles de mots croisés et de multiplication longue. Le modèle de mots croisés inclut un modèle d'affectation des mots dans la grille couplé à un modèle d'addition pour chaque ligne et chaque colonne. Le modèle de multiplication longue est complètement spécifique, car les contraintes sur la longueur des opérandes sont très fortes.

5 Évaluation expérimentale

Nous présentons ici les expériences menées pour évaluer nos approches de résolution et de génération de cryptarithmes et les comparer avec le solveur specialisé CRYPT [21]. Le solveur CRYPT écrit en C utilise un algorithme de retour arrière. Il se restreint aux cryptarithmes en simple précision, avec additions décimales et des lettres minuscules et majuscules de l'alphabet latin. Seules les additions seront évaluées, car c'est la forme la plus classique et pour comparaison avec CRYPT qui n'accepte que des additions.
La section 5.1 compare les performances de résolution d'additions de nos solveurs PPC et du solveur CRYPT. La section 5.2 étudie le passage à l'échelle de notre solveur lorsque la base arithmétique augmente. La section 5.3 évalue et compare les performances pour la génération d'additions de notre approche et du solveur CRYPT.
Les expériences ont eu lieu sur une machine Dell avec 256 GB de RAM et 4 Intel E7-4870 2.40 GHz processeurs sous CentOS Linux 7.9 (chaque processeur a 10 cœurs).

5.1 Performances de la résolution

Nous analysons les performances de la résolution sur les additions de la collection de la section 6. Toutes les instances ont donc une solution unique. Notre jeu d'instances est composé de 35800 additions dont les mots ont 8 lettres ou moins. Pour résoudre un cryptarithme, un solveur trouve une solution, puis prouve qu'aucune autre existe. Ce choix est guidé par la difficulté et le moindre intérêt d'une collection de cryptarithmes sans ou avec plusieurs solutions.
Nous évaluons quatre solveurs :

SCALAR le modèle simple précision de la section 3.2 avec la contrainte (3) ;
HORNER le même modèle avec la contrainte (4) ;
BIGNUM le modèle multiprécision de la section 3.3 ;
CRYPT un solveur spécialisé [21] écrit en C.

Les solveurs PPC résolvent les 35800 instances. Le solveur CRYPT résout seulement les 35104 en alphabet latin.
La figure 1 donne le nombre d'instances résolues en fonction du temps (en secondes) pour chaque solveur. Le premier constat est sévère puisque le solveur spécialisé CRYPT (à gauche) est extrêmement rapide en résolvant chaque instance en moins d'une demi-seconde. De manière symétrique, la contrainte HORNER (à droite) est de loin la plus inefficace et ne permet même pas de résoudre toutes les instances dans le temps imparti (la ligne verticale à l'extrémité droite). Au milieu, le modèle simple précision SCALAR et multiprécision BIGNUM sont presque 100 fois moins rapides que CRYPT. Après un démarrage plus lent, le modèle SCALAR prend un léger avantage sur BIGNUM et résout plus de 90% des instances en moins d'une seconde. Le temps de résolution moyen de SCALAR est trois fois celui de BIGNUM, mais les temps médians sont quasiment égaux.
Finalement, la complexité d'un solveur de contraintes induit un surcoût non négligeable par rapport à un algorithme très spécialisé sur des additions décimales. Cependant, les performances sont acceptables pour une utilisation interactive du solveur et sont compensées par la généralité de l'approche PPC.

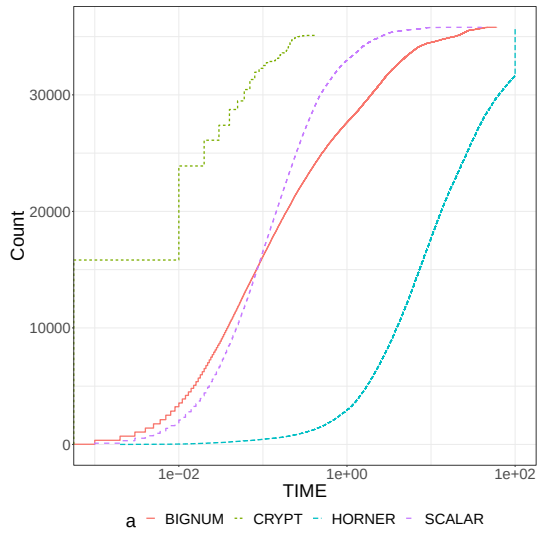

FIGURE 1 – Instances résolues en fonction du temps.

5.2 Passage à l'échelle de la résolution

Nous n'avons trouvé que trois instances non décimales dans notre bibliographie, deux en base 15 et une en base 16. Seul le solveur BIGNUM est capable de les résoudre.
Le tableau 1 donne le nombre d'opérandes, le nombre total de lettres, et la longueur maximale des mots de chaque cryptarithme accompagnés du temps de résolution et du nombre de nœud du solveur. Les métriques du solveur indiquent que la difficulté des instances a considérablement augmenté. Chacune de ces additions en base 15 ou 16 est plus difficile que n'importe quel cryptarithme décimal de la collection.

b	$\|W\|$	$\sum \|w\|$	$\max \|w\|$	Time (s)	Nodes
15	47	215	9	2185.619	549106
15	54	249	9	11716.522	1561266
16	33	154	10	2968.624	1573488

TABLE 1 – Résolution d'additions en base 15 et 16.

Les investigations sur le passage à l'échelle n'ont pas été poussées plus loin. D'abord, générer une collection en base non décimale est une tâche difficile. Puis, l'intérêt d'une telle collection est faible dans les cas d'utilisation pédagogiques, car le calcul devient compliqué ce qui n'est pas très amusant.
En conclusion, la réputation de facilité du problème reflète l'attention quasi-exclusive à la base 10.

5.3 Performances de la génération

Une méthode de génération examine des candidats générés à partir d'une liste de mots, et détermine s'ils ont une solution unique ou non. Une méthode doit filtrer les candidats pour être efficace, car la combinatoire des additions même sans répétition est considérable. Remarquez que la résolution des candidats est séquentielle, et non parallèle. Ici, nous comparons les performances de notre approche CRYPTATOR et du solveur CRYPT pour générer toutes les additions avec un nombre d'opérandes fixé et un terme unique à droite. CRYPTATOR utilise le modèle SCALAR si aucun mot n'a plus de 8 lettres et BIGNUM sinon.
Pour ces expérimentations, nous utilisons deux listes de mots. La première est composée de 24 mots correspondant aux noms des lettres de l'alphabet grec (alpha, beta, ...). La seconde liste est composée de 143 mots correspondant à des couleurs. La table 2 donne quelques caractéristiques des listes de mots utilisées pour la génération de cryptarithmes.

	$\|W\|$	$\min \|w\|$	$\text{med}\|w\|$	$\text{mean} \|w\|$	$\max \|w\|$
alphabet	24	2	4	4.167	7
couleurs	143	3	9	9.168	20

TABLE 2 – Caractéristiques des listes de mots.

Le tableau 3 récapitule les performances des deux méthodes avec les indicateurs suivants. Le nombre n indique le nombre de termes à gauche de l'addition. Le nombre s donne le nombre de cryptarithmes avec une solution unique. La partie haute donne les résultats pour l'alphabet grec, et la basse pour les couleurs. Pour chaque méthode, le tableau donne le temps t de génération avec résolution des candidats en secondes, et le nombre c de candidats. Pour CRYPTATOR, le tableau indique en plus le temps t_c de génération sans résolution. À droite, le nombre $\overline{c}$ d'additions sans répétition est une borne supérieure sur le nombre de candidats.

		CRYPTATOR			CRYPT		
n	s	t_c	t	c	t	c	$\overline{c}$
2	4	0.8	13.0	1.4K	**0.1**	2.9K	6.1K
3	38	1.5	165.7	5.9K	**1.9**	17.0K	42.6K
4	128	1.7	631.1	11.4K	**10.9**	75.2K	215.2K
5	207	1.5	652.4	10.6K	**26.1**	261.8K	807.6K
6	184	1.0	540.4	4.7K	**28.6**	735.8K	2.4M
7	30	0.7	120.5	817	**21.5**	1.7M	5.9M
8	2	0.6	**4.1**	13	11.4	3.3M	11.8M
9	0	0.4	**0.4**	0	13.7	5.4M	19.6M
2	66	5.6	67.7	7.7K	**1.8**	558.9K	1.4M
3	315	6.7	307.0	17.2K	**47.0**	20.2M	66.8M
4	357	7.3	**390.4**	24.3K	1232.5	574.8M	2.3G
5	163	6.8	**447.2**	25.8K	31148.9	557.8M	64.1G
6	46	6.2	**537.5**	20.4K	–	–	1.5T

TABLE 3 – Génération avec l'alphabet grec (24 mots) en haut ou des couleurs (143 mots) en bas.

Premièrement, la croissance du nombre $\overline{c}$ d'additions sans répétition est telle qu'une méthode de génération doit absolument en éliminer une large majorité pour passer à l'échelle. On observe aussi que le nombre s de cryptarithmes avec une solution unique est très petit en comparaison. Deuxièmement, la génération des candidats avec CRYPTATOR prend peu de temps t_c pour l'alphabet grec et un temps court par rapport au temps t de résolution des

candidats pour les couleurs. L'intérêt est évident puisque le nombre c de candidats de CRYPTATOR est inférieur d'un ou plusieurs de grandeurs à celui de CRYPT pour les couleurs et dès que n atteint 5 pour l'alphabet. Troisièmement, la comparaison des temps t de génération avec résolution est plus contrastée. Les meilleurs temps sont indiqués en gras. Pour l'alphabet, la réduction du nombre c de candidats par CRYPTATOR par rapport à CRYPT n'est pas suffisante pour compenser sa moindre rapidité de résolution pour les plus petites valeurs de n. Par contre, la comparaison s'inverse pour les plus grandes valeurs de n pour lesquelles le nombre de candidats devient très faible. Pour les couleurs, la liste est plus grande et l'explosion combinatoire bien plus rapide. La plus grande rapidité de CRYPT ne compense plus du tout le plus grand nombre c de candidats. Le temps de CRYPT à l'avant dernière ligne dépasse largement le temps cumulé de CRYPTATOR. Pour la dernière ligne, la résolution a été interrompue après 4 jours de calcul en ayant découvert seulement 2 cryptarithmes sur 46.

Ainsi, si la programmation par contraintes induit un surcoût pour la résolution par rapport à un solveur spécialisé comme CRYPT, elle reprend largement l'avantage pour la génération. La collection de la section 6 n'aurait pas pu être créés avec CRYPT.

Une perspective intéressante est d'hybrider la méthode de génération par la programmation par contraintes avec le solveur CRYPT pour la résolution.

6 Une collection remarquable

Nous présentons les résultats des expériences menées pour créer une collection de cryptarithmes complexes et remarquables. Le temps consacré à cette tâche est important, mais ne fait pas l'objet d'une analyse. Les résultats sont plutôt analysés à travers les caractéristiques et nouveautés de la collection.

6.1 Additions

Les additions sont la forme la plus fréquente de cryptarithmes. Par conséquent, elles constituent la majorité de la collection et se répartissent en trois groupes :

— des collections thématiques les plus complètes possibles ;
— des collections obtenues par échantillonage du dictionnaire français présentant des caractéristiques variées et remarquables en termes de nombres d'opérandes ou de longueurs des mots ;
— des collections de cryptarithmes doublement vrai dans une quinzaine de langues différentes pour les sommes de 1 à 500.

Le tableau 4 présente quelques statistiques de ces additions, de gauche à droite, leur nombre d'opérandes, leur nombre total de lettres, et les longueurs moyenne, minimale, et maximale des mots. On remarque d'abord que la collection est assez variée en termes de nombre d'opérandes et de nombre total de lettres. Une majorité d'additions ont peu d'opérandes, mais un petit quart a plus de vingt opérandes. Même si le plus long cryptarithme connu est un ovni de 200 mots (avec répétitions), la plus longue addition atteint quand même 49 opérandes. Les trois quarts des cryptarithmes sont composés de mots courts entre trois et six lettres.

Indicateur	$\lvert W\rvert$	$\sum \lvert w\rvert$	mean $\lvert w\rvert$	min $\lvert w\rvert$	max $\lvert w\rvert$
Minimum	3.0	12.0	3.0	2.0	4.0
1er Quartile	5.0	27.0	4.5	3.0	6.0
Médiane	7.0	38.0	4.9	3.0	6.0
Moyenne	13.5	64.1	5.0	3.3	6.3
3ème Quartile	18.0	88.0	5.2	4.0	6.0
Maximum	49.0	216.0	12.8	12.0	20.0

TABLE 4 – Statistiques sur les 36821 additions.

La figure 2 présente une carte thermique dont l'abscisse représente le nombre d'opérandes et l'ordonnée la longueur du mot le plus court. Le gradient de couleur indique le logarithme décimal du nombre de cryptarithmes dans la collection. Le nombre de cryptarithmes par catégorie ($\lvert W\rvert$, $\min \lvert w\rvert$, $\max \lvert w\rvert$) est limité à mille pour la méthode par échantillonage du dictionnaire. La zone claire, donc dense, en bas à gauche correspond aux collections thématiques. La zone foncé, donc peu dense, en haut à gauche correspond aux cryptarithmes avec de longs mots. La zone moins foncé, en bas à droite correspond aux cryptarithmes avec beaucoup d'opérandes.

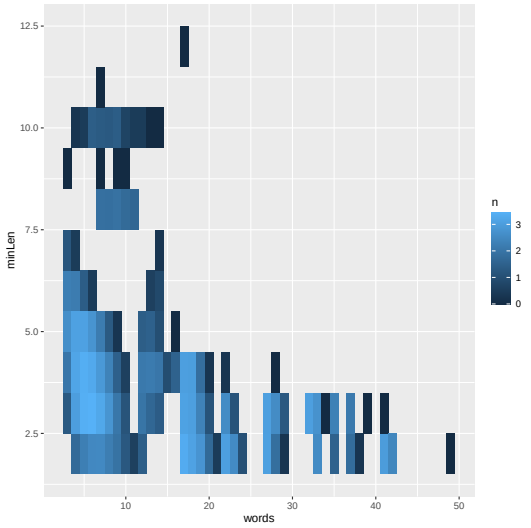

FIGURE 2 – Carte thermique des additions.

Certains des plus longs cryptarithmes, et le plus long en particulier, ne sont composés que de palindromes !

AA + ALLA + ANA + ANONA + ARA + ASA + AXA + ELLE + ERE + ERRE
+ESSE + ETE + ETETE + EUE + NANAN + NON + OXO + REER + ROTOR
+SALAS + SANAS + SAS + SASSAS + SELLES + SENES + SENNES+
SERES + SERRES + SES + SEXES + SOLOS + SONOS + SOS + STATS+
STOTS + STUUTS + SUS + TALAT + TALLAT + TANNAT + TARAT+
TASSAT + TATAT + TAXAT + TET + TNT + TOT + TUT = NAURUAN

Finalement, la découverte de cryptarithmes doublement vrai s'est avérée très fructueuse avec presque mille cryptarithmes dans dix langues différentes, dont plus de la moitié en Hindi !

SEIS + SETENTA + TRESCIENTOSCATORCE = TRESCIENTOSNOVENTA
ZERO + TRES + SEIS + CENTOEDOIS = CENTOEONZE
BES + ON + ONIKI + ELLIUC = SEKSEN

6.2 Multiplications

Cette fonctionnalité est plus récente et limitée par l'arithmétique simple précision, mais elle est également plus rare. Il a été assez rapide de générer quelques milliers de multiplications thématiques, ainsi que des doublement vrai. l'échantillonnage du dictionnaire est inutile, car l'arithmétique simple précision limite le nombre d'opérandes et la longueur des mots.

Nous n'avons réussi à générer pour le moment qu'une dizaine de multiplications longues dont les résultats ont au plus cinq lettres. Cependant, nous retrouvons rapidement celles-ci lorsque la liste de mots en contient.

6.3 Mots croisés

La collection contient une centaine de mots croisés composés de mots de trois lettres ou moins. À notre connaissance, c'est la première fois qu'ils sont créés dans une langue naturelle.

7 Diffusion scientifique

La bibliothèque CRYPTATOR est développée en Java et utilise Maven pour la compilation et le déploiement automatique du projet. Ses dépendances principales sont le solveur Choco [5], le générateur de parseur antl4 [16], le parseur de ligne de commandes args4j [11], et le framework de tests unitaire junit 5 [2].

Plus précisément, Choco est utilisé pour la modélisation, la résolution, et la génération de cryptarithmes sous la forme de modèles en programmation par contraintes. Antlr4 est utilisé pour définir la grammaire, lire l'équation du problème, et la génération de l'arbre syntaxique correspondant à cette équation. Args4j est utilisé pour définir les options et lire les arguments de la ligne de commandes. Ces options permettent de changer les niveaux de verbosité dans la console, de configurer les modèles, de choisir les variantes, de relâcher les contraintes. La libraire junit 5 sert à tester profondément CRYPTATOR. Avec une couverture du code dépassant les 85%, les tests vérifient le bon fonctionnement de la résolution et de la génération. Entre outre, de nombreux résultats ont été doublement validés avec le solveur CRYPT ou les collections de cryptarithmes en ligne.

Une application mobile [2] est à un stade précoce de développement. Ses fonctionnalités basées sur la bibliothèque Java intégrerons la résolution et la génération de cryptarithmes, mais aussi un jeu pédagogique autour de leur résolution. Le support pour la création de l'application est le framework *React Native* [15]. Ce framework permet la création d'applications utilisables autant sur les systèmes Android que sur les systèmes iOS ou sur le web. Un objectif principal est de faire connaître au grand public la programmation par contraintes par le vecteur des cryptarithmes. L'application pourra s'utiliser en autonomie ou dans un cadre pédagogique.

2. https://github.com/FissoreD/CryptatorApp

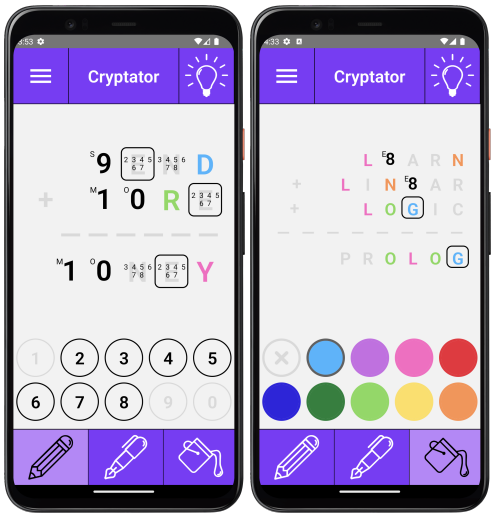

FIGURE 3 – L'application mobile du CRYPTATOR

L'application doit permettre de saisir et visualiser un cryptarithme, une solution, ou une liste de mots. Le jeu doit aussi permettre d'annoter des cryptarithmes, de construire une solution même partielle, ou d'afficher des indices. Le jeu devra aussi proposer différents niveaux de difficulté pour s'adapter à tous les publics.

Le mode de résolution des cryptarithmes permet d'affecter à chaque lettre une valeur numérique. Lorsqu'une lettre est affectée à une valeur, cette valeur est supprimée des domaines des autres lettres. Lorsque toutes les lettres sont affectées à une valeur, le cryptarithme est résolu. Lorsque l'utilisateur affecte une valeur à une occurrence d'une lettre, cette valeur est affectée à toutes les autres occurrences de cette lettre. En sélectionnant une lettre, on peut ainsi voir les valeurs possibles pour cette lettre. Pour aider à la résolution, il est possible d'annoter les lettres par des valeurs encore disponibles dans leurs domaines. Il est donc possible d'attribuer plusieurs valeurs provisoires à une lettre.

La figure 3 présente une capture d'application d'un prototype du jeu. Le concept pédagogique est de représenter visuellement le domaine des lettres et des chiffres. L'utilisateur résout le problème en réduisant les domaines par des décisions prises graphiquement avec l'outil stylo. Il peut aussi annoter ces domaines avec l'outil pinceau et colorier les lettres et chiffres avec l'outil seau de peinture. Le moteur de jeu peut indiquer quand une décision invalide est prise ou afficher des indices avec l'outil ampoule lumineuse.

8 Conclusion

Nous proposons une approche générale, efficace, et facile d'utilisation pour la résolution et la génération de cryptarithmes en programmation par contraintes. Ce travail n'a jamais été réalisé à notre connaissance malgré la popularité du problème. L'approche est très générale en simple précision, c'est-à-dire quand les mots sont courts, mais restreinte aux additions en multiprécision. La résolution est ef-

ficace, mais reste largement dominé par un algorithme spécialisé moins général. La génération est sans conteste un point fort comme en atteste la variété de la collection de cryptarithmes. L'approche est disponible actuellement sous la forme d'une bibliothèque Java et d'une application en ligne de commande.
On peut dégager trois perspectives principales. L'ajout de la multiplication pour généraliser le modèle multiprécision. L'hybridation avec l'algorithme spécialisé améliorerait l'efficacité de la génération. La finalisation de l'application mobile avec les améliorations et les ajouts nécessaires et son adaptation à une page web pour faciliter l'utilisation et la diffusion.

Références

[1] CSPLib : A problem library for constraints. http://www.csplib.org, 1999.

[2] junit 5. https://junit.org/junit5/, 2023.

[3] Maxey Brooke. *150 Puzzles in Crypt-Arithmetic*. Dover Publications, Inc., 1963. URL http://cryptarithms.awardspace.us/150-puzzles-in-crypt-arithmetic.pdf.

[4] V. Chandra Prakash, V. Kantharao, JKR Sastry, and V. Bala Chandrika. Expert system for building cognitive model of a student using crypt arithmetic game and for career assessment. *International Journal of Recent Technology and Engineering*, 7 :684–689, 03 2019.

[5] Choco Team. Choco : an Open-Source Java Constraint Programming Library. www.choco-solver.org, 2023.

[6] Truman Collins. Alphametic puzzles. http://www.tkcs-collins.com/truman/alphamet/index.shtml, 2023.

[7] F. Rea Cyrus. *KLOOTO Games Cryptodigits*. CreateSpace Independent Publishing Platform, 2015. ISBN 151681973X.

[8] Henry Dudeney. The strand magazine vol. 68, juillet 1924, p. 97 et 214. 1924.

[9] D Epstein. On the NP-completeness of cryptarithms. *SIGACT News*, 18(3) :38–40, apr 1987. ISSN 0163-5700. doi : 10.1145/24658.24662. URL https://doi.org/10.1145/24658.24662.

[10] Steven Kahan. *At last!! Encoded totals second addition*. Baywood Publishing Co., 1994.

[11] Kohsuke Kawaguchi. args4j : Java command line arguments parser. https://args4j.kohsuke.org/, 2023.

[12] Mike Keith. An alphametic page. http://www.cadaeic.net/alphas.htm, 2023.

[13] Joseph S. Madachy. *Madachy's Mathematical Recreations*. Dover Pubns, 1979.

[14] Mahmoha. *Alphametic Puzle - Numbers Behind Letters . Special Edition : Names of Countries with Colorful Flags*. 2022. ISBN 979-8360917267.

[15] Meta Open Source. React native. https://reactnative.dev/, 2022.

[16] Terence Parr. antlr4. https://www.antlr.org/, 2023.

[17] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*, AAAI'96, page 209–215. AAAI Press, 1996. ISBN 026251091X.

[18] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, USA, 2006. ISBN 0444527265.

[19] Torsten Sillke. Alphametics. https://www.math.uni-bielefeld.de/~sillke/PUZZLES/ALPHAMETIC/, 2023.

[20] Jorge A. C. B. Soares. Cryptarithms online. http://cryptarithms.awardspace.us/index.html, 2002.

[21] Naoyuki Tamura. Cryptarithmetic puzzle solver. https://tamura70.gitlab.io/web-puzzle/cryptarithm/, 2023.

[22] S Widodo, U Najati, and P Rahayu. A study of cryptarithmetic problem-solving in elementary school. *Journal of Physics : Conference Series*, 1318(1) : 012120, oct 2019. doi : 10.1088/1742-6596/1318/1/012120. URL https://dx.doi.org/10.1088/1742-6596/1318/1/012120.