École Centrale de Lille

Algorithmique avancée et programmation

Résolution du jeu "Le compte est Bon"

Compte-rendu

Nom	**Prénom**
AQUILO	Axel
GOSMANN	Gabriel
DOS SANTOS SILVA	Vitor
STEFFENS WONTROBA	Vinícius

Villeneuve-d'Ascq
2022

École Centrale de Lille

Résolution du jeu "Le compte est Bon"

Compte-rendu

Compte-rendu du projet présenté aux évaluateurs du sujet Algorithmique avancée et programmation de l'École Centrale de Lille.

Villeneuve-d'Ascq
2022

Table des matières

Table des figures

Dreams feel real while we're in them. It's only when we wake up that we realize something was actually strange.

Inception

1. Introduction

1.1 Compréhension du problème

Le problème proposé porte, de manière générale, sur la recherche des solutions pour le jeu télévisé "le compte est bon". Le jeu fonctionne de la manière suivante : étant donné six nombres extraits aléatoirement d'une liste prédéfinie de 14 nombres et un nombre - également obtenu aléatoirement - défini entre 100 et 999, en utilisant les opérations usuelles (addition, soustraction, multiplication et division) entre toutes les combinaisons possibles, de toutes les tailles, des six nombres, pour obtenir le septième nombre généré.

L'idée est d'utiliser une recherche exhaustive de solutions, afin de trouver le plus rapidement possible l'ensemble qui utilise le moins d'opérations. S'il n'y a pas de solution, l'ensemble des opérations qui se rapproche le plus du nombre recherché doit être présenté.

Pour faciliter le codage et le développement du problème, il est proposé d'utiliser la notation RPN (Reverse Polish Notation). En ce qui concerne la fabrication des piles et des listes chaînées, cette notation est particulièrement pertinente pour l'association des opérations entre les expressions et la réduction des itérations inutiles. Cela renvoie la solution la plus optimisée.

de plus, il est suggéré de penser le problème comme un arbre d'expressions (valides et invalides), qui doit être parcouru à la recherche de la solution souhaitée. Concrètement, il s'agit d'une recherche "à la volée en profondeur d'abord", ce qui implique que l'arbre doit nécessairement être construit de manière à ce que chaque branche soit parcourue "de haut en bas", c'est-à-dire de la partie la moins profonde à la partie la plus profonde. Une nouvelle jambe ne sera couverte qu'après que la jambe précédente ai atteint sa partie la plus profonde.

1.2 Compréhension de la solution

Comme mentionné précédemment, l'idée est d'utiliser un algorithme qui recherche de manière exhaustive la solution proposée. De façon générale, cela signifie qu'il faut déterminer parmi toutes les possibilités existantes, celle qui utilise le moins d'opérations.

Cependant, en effectuant une recherche exhaustive, on s'est rendu compte qu'il existe

un nombre important d'opérations différentes qui sont en fait équivalentes. Ainsi, pour rendre le processus encore plus optimisé, on a réalisé ce qu'on a appelé une "recherche exhaustive intelligente", effectuant des opérations uniques, sans répétitions, avec le moins d'itérations possible. En ce sens, l'algorithme fonctionne, brièvement, comme suit :

1. Une fonction récursive génère les expressions possibles à opérer ;
2. Une seconde fonction analyse si les opérations sont effectivement exécutables et si elles renvoient ou non un résultat entier ;
3. Si oui, on compare le résultat obtenu avec la valeur recherchée. S'ils sont égaux, la valeur et le "chemin" utilisé pour l'atteindre sont stockés. De plus, il stocke la différence entre la valeur obtenue et la valeur recherchée. Sinon, le processus récursif est répété.
4. Une fois toutes les expressions possibles opérées, seul le résultat le plus optimisé est conservé, puis renvoyé à l'écran. S'il n'y a pas de solution, le chemin qui a présenté la plus petite différence entre la valeur obtenue et la valeur recherchée sont renvoyés, c'est-à-dire l'expression la plus proche de la solution recherchée. Dans le cas ou plusieurs chemins serait à la même distance de la solution, l'algorithme choisit le premier.

Des étapes intermédiaires avec des fonctions supplémentaires ont évidemment été développées à chaque étape de l'algorithme élaboré. Celles-ci seront présentées plus en détail dans les sections suivantes de ce rapport.

Il est également mentionné qu'il existe des différences entre l'utilisation de piles statiques, de piles dynamiques et de listes chaînées. Cette différence se trouve dans le temps d'exécution, c'est-à-dire dans la vitesse du code à trouver la solution. Cependant, la solution trouvée ne diffère pas. Le choix de la méthode à implémenter doit être fait par l'utilisateur lui-même, dans une ligne de code, à l'intérieur de la bibliothèque *rpn.h*, en supprimant les barres de commentaires du format choisi.

1.3 L'organisation du programme et du groupe

1.3.1 Le programme

Le programme a été divisé en :

1. Un fichier main.c qui reçoit les données fournies sur la ligne de commande et appelle les autres fonctions ;
2. Un fichier rpn.c qui s'occupe de :
   (a) La fonctions de transition de rpn à liste régulière et vice versa ;
   (b) Évaluer la validité des opérations ;
   (c) Déterminer si oui ou non la solution a été obtenue (si oui, quel est le chemin qui y mène ; sinon, quelle est la solution la plus proche) ;
3. Un fichier search.c qui recherche, de manière récursive, toutes les possibilités d'expressions à opérer ;
4. D'autres fichiers de définition de pile et listes chaînées produits en classe mais adaptés au problème proposé.

1.3.2 Le groupe

L'équipe a été divisée pour que la mise en œuvre se fasse de manière optimale.

— AQUILO Axel : Mise à jour et adaptation des fonctions de pile et de liste chaînée, ainsi que relecture du texte.

— GOSMANN Gabriel : Développement de fonctions récursives pour la recherche d'expressions et d'opérations, ainsi que leurs fonctions auxiliaires respectives dans le fichier search.c ;

— DOS SANTOS SILVA Vitor : Production du travail de Parsing et évaluation de la validité des expressions dans le fichier rpn.c ;

— STEFFENS WONTROBA Vinícius : Vérification de l'existence de solution et comparaison de la valeur des expressions avec la valeur de la solution recherchée dans le fichier rpn.c, production du rapport de travail.

2. Développement

Afin de comprendre la notion de "recherche exhaustive intelligente", il faut d'abord comparer le fonctionnement d'une expression RPN avec une expression "traditionnelle". Supposons qu'une calculatrice effectue la relation R entre deux nombres entiers quelconques a et b. Pour utiliser la notation traditionnelle, il faut, d'une manière ou d'une autre, entrer les données puis appuyer sur le signe égal pour que l'opération soit effectuée. Dans le cas de la RPN, la relation R elle-même remplit la fonction du bouton égal. Dans ce cas, on a que :

$$aRb = abR \tag{2.1}$$

Imaginons maintenant cette même logique, mais appliquée à trois valeurs entières a, b et c, avec le même opérateur R. En supposant que les opérations sont effectuées de sorte que la première valeur entrée soit a, la seconde b et la troisième c, l'opération écrite naturellement serait $abRcR$. C'est-à-dire que, logiquement, une façon de mettre en œuvre la recherche exhaustive serait de permuter l'opérateur R entre les données a, b et c. Il est donc évident que :

$$abRcR = abcRR \tag{2.2}$$

D'autre part, en restreignant la façon dont les expressions sont traitées à la façon dont elles sont écrites dans la partie droite de l'égalité, on postule qu'il n'y aura pas d'expressions répétées ou équivalentes, et donc les expressions égales ne seront jamais évaluées de manière répétée. Par conséquent, nous savons que la meilleure façon de présenter - et d'exploiter - toutes les expressions valides est de les recevoir comme :

$$[a_1, a_2, ..., a_n, R_1, R_2, ..., R_{n-1}] \tag{2.3}$$

Ainsi, toutes les valeurs possibles de toutes les expressions valides ne seront obtenues qu'une seule fois, puisque cette situation générique englobe des cas particuliers comme celui présenté dans l'équation (2.2).

Il est important de mentionner que la mise en œuvre de ce type d'algorithme fait l'hypothèse qu'il n'est pas possible d'effectuer des opérations intermédiaires. A titre

d'exemple, la relation *abR(cdR)R* n'est pas incluse dans la portée du problème, car la description du jeu et le problème lui-même ne précisent pas s'il s'agit ou non d'une expression valide.

Partant du principe que le jeu ne recevra que des expressions telles que *2.3* et que les règles précitées seront respectées, pour un nombre n de nombres reçus, le nombre d'opérations possibles (valides et invalides) sera donné par :

$$\sum_{n=1}^{n_n} \frac{n_n!}{(n_n - n)!} \times n_R^{(n-1)} \tag{2.4}$$

Où n_n représente le nombre de nombres utilisés dans le jeu et n_R le nombre d'opérateurs. Pour $n_n = 6$ et $n_R = 4$, il s'ensuit que :

$$\sum_{n=1}^{6} \frac{6!}{(6 - n)!} \times 4^{(n-1)} = 946.686 \tag{2.5}$$

2.1 Le code

Désormais, chaque section présentera un fichier qui constitue le code dans son intégralité. Ainsi, nous commencerons par discuter du fichier main.c et de ses fonctions respectives, suivis de fichiers tels que rpn.c, search.c, etc.

2.1.1 main.c

La fonction commence à donner les définitions d'opérateur de base en termes de T_elt, ainsi qu'à lire les valeurs saisies via la ligne de commande. Une fois cela fait, il appelle les fonctions *search* et *show_operators*, toutes deux présentes dans le fichier *search.c*.

2.1.2 search.c

void put_botton(T_stack *list1, T_elt element)

La fonction est chargée de réordonner les éléments des listes traitées au sein des fonctions *compute* et *search* (dont il sera question dans les sections suivantes), afin de ne jamais perdre aucun terme utilisé, mais aussi de s'assurer qu'il ne soit utilisé qu'une seule fois. L'algorithme fonctionne de manière similaire à une tour de Hanoï. Encore une fois, ce

processus simple évite des centaines de milliers de répétitions, et finit donc par optimiser le programme.

`int is_valid(T_stack * exp)`

La fonction compte les opérateurs et les nombres dans une expression. A noter qu'elle renvoie une valeur entière (différence entre les numéros d'opérateur ou simplement zéro). Cette action constitue la base de la validation d'une expression et garantit que les expressions sont lues et exploitées comme proposé par l'équation (2.3).

`int has_op(T_stack*exp)`

La fonction vérifie si un opérateur a déjà été inséré dans l'expression. Ceci est important car, plus tard, s'il y a déjà un opérateur dans l'expression, rien ne sera ajouté à l'exception des autres opérateurs. Cette action fonctionne comme un limiteur d'expressions possibles et garantit que, pour chaque suite de nombres dans une expression, toutes les possibilités ne sont analysées qu'une seule fois. De plus, il fournit l'idéalisation de l'équation (2.3).

`T_elt compute(T_stack exp_)`

Fonction chargée de parcourir les expressions et d'exécuter les opérations présentées. Il convient de mentionner que cette partie du code vérifie déjà à l'avance si les opérations effectuées sont en fait valides ou non et les classe comme telles. Cela est dû entre autres au fait que la fonction est de type *T_elt*, il faut donc retourner non seulement sa valeur, mais aussi son statut.

Lors de l'exécution de ce processus, une série d'allocations et de réallocations de mémoire sont effectuées, ce qui peut être assez dangereux du point de vue de la consommation effective de mémoire RAM pour le traitement du programme. Par conséquent, dans la dernière partie de la fonction, l'action de *FreeStack* est exécutée, ce qui garantit qu'il n'y aura pas d'accumulation et d'utilisation inutile de la mémoire.

`T_stack search(T_stack * exp, T_stack * num, int target, T_stack * best)`

C'est la fonction centrale du code. En gros, il construit l'arbre entier à parcourir à la recherche de la meilleure expression de solution. C'est-à-dire qu'il renvoie la meilleure

expression au format pile. Ceci est accompli dans un processus de récursivité, où la fonction s'appelle elle-même un nombre fini de fois. Détaillant l'exécution, la fonction prend l'élément supérieur de la pile analysée et l'alloue au sommet de la pile d'expressions. Comme il a été prédéfini comment les expressions seraient construites et donc ce qu'est une expression valide, ce processus se produit jusqu'à ce qu'une expression invalide soit obtenue. De cette façon, la récursivité de la fonction se produit au maximum 11 fois, puisqu'il y a 6 valeurs et 5 opérations. À la douzième récursivité, la fonction signale une expression invalide et s'arrête donc.

void show_operations(T_stack * list1, int res)

La fonction rétablit essentiellement la traduction effectuée en RPN. Le but du programme étant de présenter les opérations "traditionnelles" qui conduisent à la valeur recherchée, il est nécessaire, après avoir trouvé la solution la plus adéquate, de la présenter également en mode "traditionnel". Par conséquent, la fonction se charge de traduire les expressions RPN en notation commune.

void show1list(T_stack * list1, int res) et void show2lists(T_stack * list1, T_stack * list2, int res)

Les deux fonctions effectuent des débogages de programme. Au cours du processus de développement, ils ont été beaucoup utilisés pour s'assurer que les résultats obtenus étaient bien ceux attendus et mis en œuvre.

2.1.3 rpn.c

`T_elt createElt(char * current)`

La fonction reçoit un pointeur char qui représente l'élément de l'expression à modifier en type T_elt. Comme cette structure est composée de deux caractéristiques (une valeur entière et un caractère de statut)

T_list s2list(char * exp)

La fonction prend une expression *string* composée de *chars* et la transforme en une liste chaînée. Il est important de noter que la fonction susmentionnée (createElt) est

appelée, ce qui garantit que la T_list est bien composée de T_elt.

T_elt rpn_calculate(T_stack * pilha)

La fonction reçoit un *T_stack* et vérifie simultanément la validité des expressions et détermine leurs valeurs respectives. Ce qu'il fait est, en fait, quelque chose de très similaire à ce qui se fait dans *compute*, mais implémenté dans RPN. En règle générale, les deux fonctions sont équivalentes et renvoient exactement les mêmes résultats. Ainsi, il y a peu de - voir aucune - différence si la fonction à appeler par *search* est celle-ci ou celle mentionnée ci-dessus.

T_elt rpn_eval_stack(T_stack * exp)

C'est la fonction équivalente de *int est valide*, mais retournant une valeur *T_elt*. C'est précisément cette valeur qui est envoyée à *rpn_calculate* puis exploitée. Il convient de mentionner que cette fonction effectue son travail sur une pile.

T_elt rpn_eval(char * exp

L'équivalent de la fonction précédente, mais qui travaille maintenant nécessairement avec des listes chaînées, et non avec des piles.

Mention à la bibliothèque rpn.h

C'est précisément dans cette bibliothèque que s'effectue la définition du type d'implémentation réalisée. Elle permettre de sélectionner le type d'implémentation choisie pour les piles d'évaluation, afin de pouvoir étudier l'impact de ce choix sur la vitesse de votre programme. La différence entre les résultats sera discutée dans les chapitres suivants.

2.1.4 elt.c

Le programme elt.c étant déjà fourni, peu de modifications y ont été apportées, à l'exception de la seule exception : création du cas *#ifdef ELT_RPN*. Dans ce cas, l'élément *e* est composé, comme mentionné précédemment, d'une valeur entière et d'un statut. C'est, dans l'ensemble, la seule augmentation significative apportée au code.

3. Conclusion et analyse des résultats

Tout d'abord, nous mentionnons que les résultats présentés ci-dessous ont tous été obtenus à l'aide de piles dynamiques. Les autres implantations ont également été réalisées. Cependant, comme prévu, il n'y avait pas de différence dans les résultats obtenus. Plus que cela, la différence de temps d'exécution est négligeable, ne pouvant donc pas créer de corrélation entre le mode de mise en œuvre et la vitesse d'exécution. Il convient de mentionner que, dans le cas de l'utilisation de piles statiques, l'un des facteurs qui influencent l'exécution du code est la taille de pile fixe elle-même. Ainsi, lors de l'utilisation d'une taille inhabituellement grande, le temps d'exécution est également grand. Cependant, si la sélection de ce facteur n'implique pas une valeur absurdement grande, c'est-à-dire qu'elle est effectuée avec prudence, le temps d'exécution a tendance à être faible et se rapproche d'une exécution avec des piles dynamiques.

3.1 Nombre d'opérations possibles

Pour commencer, nous allons exécuter le programme afin qu'il stocke toutes les opérations possibles entre les éléments, qu'ils soient valides ou non. La figure ci-dessous montre le nombre d'opérations effectuées.

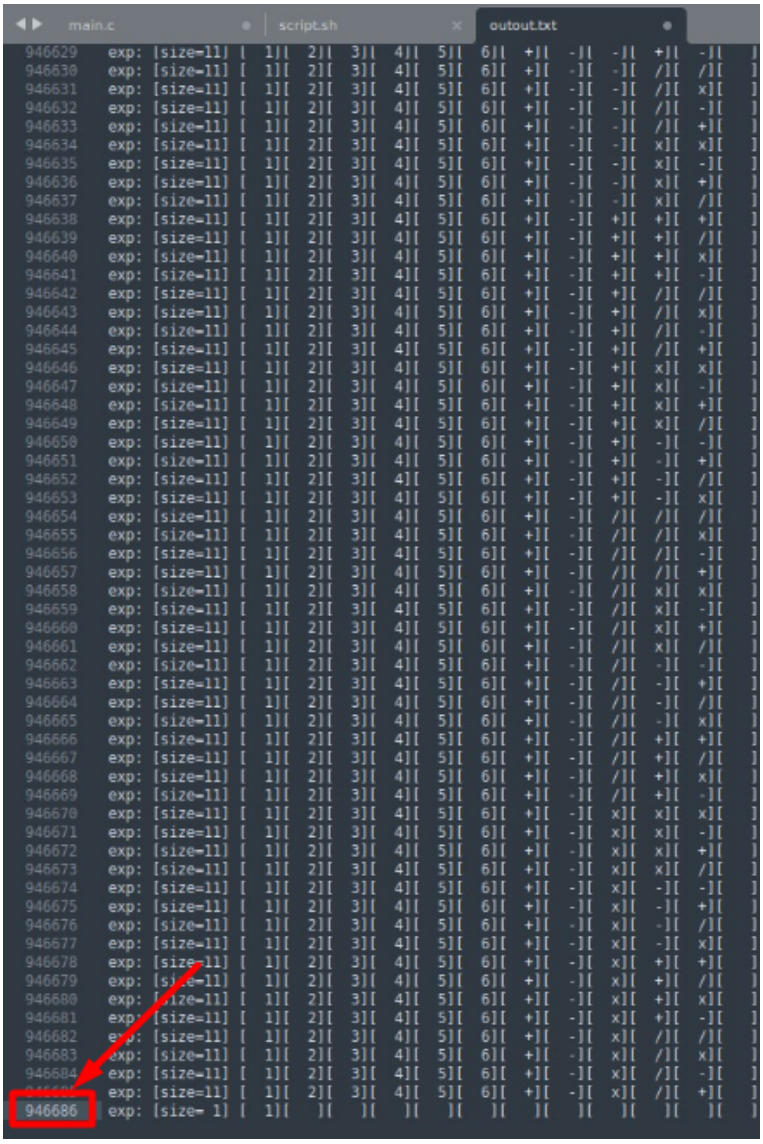

FIGURE 3.1 – Opérations

Notez que cette valeur est exactement égale à celle prédite par l'équation *2.5*

3.2 Résultat #1

Pour le premier exemple qui prouve le fonctionnement du programme, nous utilisons un cas dans lequel le programme trouve effectivement le résultat attendu et présente la solution.

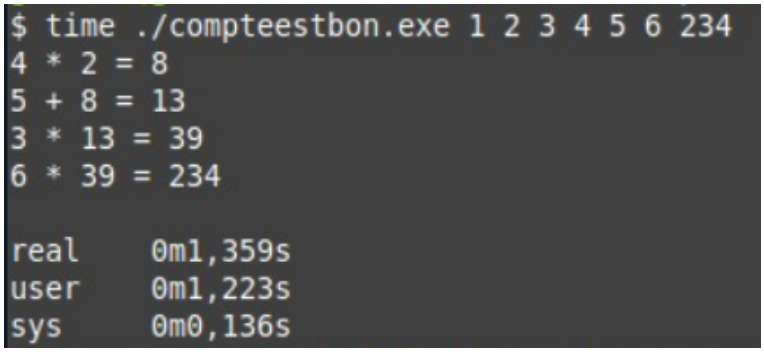

FIGURE 3.2 – Résultat exact

Il est intéressant de noter que la réponse est trouvée en un peu plus de 1 seconde.

3.3 Résultat #2

Le deuxième exemple présente un cas où le programme ne trouve pas de solution exacte. De cette manière, il renvoie la solution la plus proche de la valeur recherchée.

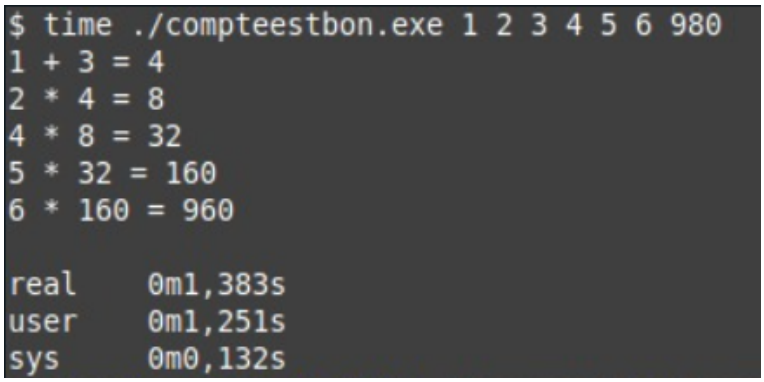

FIGURE 3.3 – La solution la plus proche

3.4 Résultat #3

Dans ce cas, nous présentons un exemple "intermédiaire". Comme on peut le voir sur la figure, le résultat trouvé est approximatif. Cependant, on sait qu'il existe des solutions possibles. Pour que ceux-ci soient trouvés, il est nécessaire d'effectuer des opérations intermédiaires et, comme mentionné, il a été supposé que ce type d'opération ne faisait pas partie du jeu.

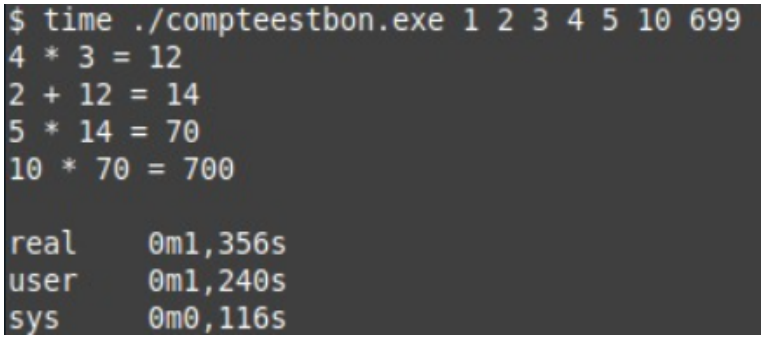

FIGURE 3.4 – Résultat intermédiaire

Une solution possible serait :

$$2 \times 5 = 10 \tag{3.1}$$

$$10 \times 10 = 100 \tag{3.2}$$

$$4 + 3 = 7 \tag{3.3}$$

$$7 \times 100 = 700 \tag{3.4}$$

$$700 - 1 = 699 \tag{3.5}$$

3.5 Exécution du script fourni

L'image ci-dessous montre l'exécution du script fourni qui prouve le bon fonctionnement du programme.

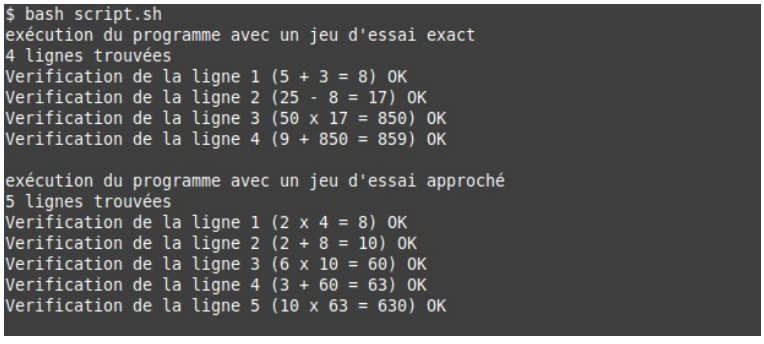

FIGURE 3.5 – Résultat script

4. Perspectives

Au départ, il est essentiel de mentionner que l'hypothèse de la règle qui annule l'utilisation d'opérations intermédiaires a influencé la prise de décision, tant dans le groupe que dans le programme lui-même. Il est clair, comme le montre la figure 3.4, qu'il existe des cas non couverts par le programme. Cependant, il convient de noter que la mise en œuvre de ce type d'opération entraînerait non seulement une utilisation plus élevée de la mémoire, mais également un temps d'exécution plus long. Cela dit, il est jugé important de bien faire comprendre, de la part de la direction des travaux, si ce type d'opération est valable ou non.

De plus, on pense que la méthode de recherche "à la volée en profondeur d'abord" n'est plus optimisée pour le problème. Parcourir l'arbre "de bas en haut" implique NÉCESSAIREMENT de parcourir tous les cas valides possibles. Si ce processus se faisait "de gauche à droite", ou plus généralement, horizontalement, le premier résultat trouvé serait déjà en fait le meilleur résultat possible, puisque tous les résultats trouvés après celui-ci utiliseraient nécessairement plus d'itérations et peut-être même un plus grand nombre d'opérations. Les deux seuls cas où une recherche verticale et horizontale sont équivalentes seraient :

1. Il n'y a qu'une seule solution au problème et elle se trouve à l'extrémité de l'arbre, c'est-à-dire dans la dernière itération effectuée ;
2. Il n'y a pas de solutions exactes au problème et il est nécessaire de trouver la solution la plus proche.

Dans tous les cas, un algorithme ainsi implémenté devrait, en théorie, être plus efficace que celui proposé par les travaux.

5. Références bibliographiques

[1] D.M. Ritchie et B.W. Kernighan. *The C programming language*. Bell Laboratories, 1988.