KaboSolve - Rapport

Alexis BONNIN - Jocelin CAILLOUX

Mars 2017

Table des matières

1 Introduction

Le projet KaboSolve s'inscrit dans le module de "Constraint Programming". Il consiste en la réalisation d'un solveur générique de CSP de la forme (X,D,C) où $X = \{x_1, \ldots, x_v\}$ est un ensemble de valeurs, $D = \{D_1, \ldots, D_n\}$ est un ensemble de domaines et $C = \{c_1, \ldots, c_m\}$ est un ensemble de contraintes. Celui-ci est basé sur un algorithme de type "branch-and-prune". Nous nous sommes fixé l'objectif de réaliser un solveur le plus générique possible, afin que toutes sortes de problèmes soient facilement modélisable en C++, tout en tentant de garder une bonne efficacité de résolution. Tout d'abord, nous présenterons la structure du programme ainsi que les différents choix algorithmiques, puis nous décrirons la complexité des différentes opérations (pruning, backtrack, ...), leur fréquence et la complexité des différents problèmes. Nous étudierons les complexités des problèmes des n-reines, du carré magique et du sudoku et sa variante x-sudoku. Le projet propose également de résoudre le problème de Send More Money mais aucune étude n'est réalisée à ce sujet.

2 Installation et exécution du solveur

2.1 Installation

Tout d'abord, le projet est disponible sur GitHub à l'adresse suivante : https://github.com/Asteip/KaboSolve. Pour installer le solveur, exécuter les commandes suivantes dans un terminal :

```
# cloner le depot
git clone https://github.com/Asteip/KaboSolve

# installer le solveur
cd KaboSolve
make
```

2.2 Exécution

Une fois le solveur installé, exécuter la commande suivante pour lancer le solveur :

```
./KaboSolve <problem> <number of solutions> [<options>]
```

L'argument <problem> prend les valeurs suivantes :

- n-queens : problème des N-Queens, il faut ajouter l'option "number of queens" correspondant au nombre de reines (N). Exemple d'utilisation : *./KaboSolve n-queens all 10.*
- more-money : problème du Send More Money. Exemple d'utilisation : *./KaboSolve more-money all.*
- magic-square : problème du carré magique, il faut ajouter l'option "size of square" correspondant à la taille du carré. Exemple d'utilisation : *./KaboSolve magic-square all 5.*

➢ sudoku / x-sodoku : problème du sudoku, il est possible de choisir entre "sudoku" qui ne prend pas en compte les diagonales de la grille et "x-sudoku" qui prend en compte les diagonales. Il faut ajouter l'option "size of the latin square" qui correspond à la taille N d'une "région" du sudoku, la taille de la grille sera donc de $N^2 \times N^2$. Exemple d'utilisation : *./KaboSolve sudoku all 3*.

L'argument <number of solutions> prend les valeurs suivantes :

➢ one : exécution du solveur jusqu'à ce qu'il trouve une solution du problème.

➢ all : exécution du solveur jusqu'à ce qu'il trouve toutes les solutions du problème.

3 Présentation de la structure du solveur

Afin de rendre l'application extensible, celle-ci est découpée en plusieurs classes. Le pattern *Strategy* est utilisé pour permettre d'implémenter différents type de contraintes suivant le problème donné. Le projet est structuré de la manière suivante :

➢ **Problem** (classe abstraite) : Cette classe définit un problème, elle est constituée d'un ensemble de domaines et de contraintes associées aux domaines. Pour créer un problème spécifique, il faut étendre cette classe, par exemple le problème des n-queens a été défini dans une classe héritée **CNqueen**. Cette classe contient les domaines et les contraintes. C'est la classe problème qui choisit le prochain domaine à fixer (celui possédant le plus petit domaine par défaut). C'est aussi le problème qui définit la manière dont sont appliquées les contraintes (par défaut, elles sont toutes appliquées une fois et sont toutes appliquées à nouveau tant qu'une modification à été apportée).

➢ **Domain** : Cette classe définit un domaine de valeurs ordonnées pour chacune des variables définies dans le problème. Elle contient un tableau toujours trié des valeurs possibles (non prunées) et une pile (LIFO) des valeurs prunées. Ainsi une recherche dichotomique est possible dans le tableau des valeurs possibles et lors du backtracking, les valeurs à remettre dans le domaine des valeurs possibles sont les dernières des valeurs prunées et la recherche dans le domaine des valeurs possibles est simplifiée en O(log n). Le plus coûteux dans cette classe est le décalage récurrent des valeurs tu tableau des possibles (en O(n)), qui est heureusement fortement accéléré par l'usage automatique de la mémoire cache. Chaque domaine possède un identifiant unique (définit manuellement lors de l'appel du constructeur). Cet identifiant sert à reconnaître les valeurs qui ont été retirées par la fixation du domaine et ainsi effectuer le backtracking simplement. La fixation du domaine se fait en choisissant une valeur aléatoire dans la liste des valeurs possibles.

➢ **Constraint** (classe abstraite) : Cette classe définit une contrainte. Pour définir une contrainte spécifique, il faut étendre cette classe. Par exemple, la classe **CAllDiff** hérite de **Constraint** et représente la contrainte "AllDifferent". Cette classe possède la liste des domaines concernés par la contrainte. Elle peut imposer aux domaines une valeur maximum, minimum ou simplement pruner une valeur unique. Lors de l'appel de la contrainte, elle applique le pruning sur tous les domaines non fixés.

➢ **Solver** : Cette classe prend en entrée un problème. Elle le résout en fixant une à une les variables et en appliquant les contraintes après chaque fixation. Le domaine fixé est alors ajouté à une pile pour le récupérer lors du backtracking. Lorsque le domaine courant est vide ou est déjà fixé, le solveur appelle la méthode de bactracking du problème qui l'applique sur tous les domaines.

Après l'implémentation de cette base pour notre solveur, ajouter des contraintes et des problèmes au solveur est une tâche grandement facilitée. Le plus difficile lors de la création d'un problème est d'associer les bonnes contraintes aux bons domaines, ce qui correspond justement à la difficulté de la modélisation.

Un gros avantage de cette structure est que nous n'effectuons aucune recopie, en effet, nous stockons l'identifiant du domaine qui a imposé le pruning des valeurs des autres domaines. Ainsi, il est simple de réinsérer ces valeurs dans la liste des valeurs possibles.

Ci-dessous l'algorithme de branch-and-prune utilisé par le solveur :

```
Data: Problème prob
Result: Les variables du problème sont fixées et correspondent à une solution, s'il en
        existe une
pile : pile de domaines;
dom : domaine courant;
ind : entier;
ind ← 0;
dom ← meilleurDomaine(prob);
while ((ind != -1) ET (ind != prob.n)) do
    if (estFixé(dom)) then
        bactrack(prob);
        if (size(dom) > 0) then
            fixer(dom);
            appliquerContraintes(prob);
            i ← i+1;
            dom ← meilleurDomaine(prob);
        else
            i ← i-1;
            dom ← top(pile);
        end
    else
        if (size(dom) > 0) then
            fixer(dom);
            push(pile, dom);
            appliquerContraintes(prob);
            i ← i+1;
            dom ← meilleurDomaine(prob);
            pop(pile);
        else
            i ← i-1;
            pop(pile);
            push(pile, dom);
        end
    end
end
```

Algorithm 1: Algorithme de branch-and-prune

4 Étude de la complexité des différents problèmes

4.1 Complexité algorithmique des différentes opérations

Cette partie détaille les différentes complexités concernant les n domaines de v valeurs et les m contraintes.

4.1.1 Domaines

Le tableau 1 donne la complexité algorithmique des opérations d'un domaine.

	Pire cas		**Meilleur cas**	
Opération	**Notre structure**	**Bitset**	**Notre structure**	**Bitset**
Pruning une valeur	$O(v)$	$O(log_2 v) \vert O(1)$	$\Omega(1)$	$\Omega(1)$
Pruning valeurs supérieures	$O(v)$	$O(v)$	$\Omega(1)$	$\Omega(1)$
Pruning valeurs inférieures	$O(v)$	$O(v)$	$\Omega(1)$	$\Omega(1)$
Backtracking	$O(v^2)$	$\Theta(v)$	$\Omega(1)$	$\Theta(v)$
Fixer un domaine	$O(v)$	$O(v)$	$\Omega(1)$	$\Omega(1)$

FIGURE 1 – Complexités algorithmiques des opérations sur un domaine

4.1.2 Contraintes

Le tableau 2 donne la complexité algorithmique du pruning des différentes contraintes.

	Pire cas		**Meilleur cas**	
Opération	**Notre structure**	**Bitset**	**Notre structure**	**Bitset**
attaques diagonales (n-queen)	$O(n * v)$	$O(n * log_2 v) \vert O(n)$	$\Omega(n)$	$\Omega(n)$
all different	$O(n * v)$	$O(n * log_2 v) \vert O(n)$	$\Omega(n)$	$\Omega(n)$
inférieur ou égal	$O(n * v)$	$O(n * v)$	$\Omega(n)$	$\Omega(n)$
supérieur ou égal	$O(n * v)$	$O(n * v)$	$\Omega(n)$	$\Omega(n)$
égalité	$O(n * v)$	$O(n * v)$	$\Omega(n)$	$\Omega(n)$

FIGURE 2 – Complexités algorithmiques de l'application des contraintes

Vous nous avez soumis l'idée d'utiliser un bitset pour différencier les valeurs possibles ou retirées d'un domaine. Nous avons décidé de ne pas l'implémenter car notre algorithme est déjà suffisamment efficace et nous devions aussi passer du temps sur l'implémentation des contraintes et le débogage. Cela dit, nous avons estimé qu'un bitset permettrait d'atteindre une complexité globalement meilleure sans être significatif. Deux versions du bitset sont possibles, celle où nous stockons les valeurs non présentes dans le domaine entre le min et le max (ces valeurs sont associées à une valeur fausse systématiquement dans le bitset), dans ce cas retirer une valeur se fait en $O(1)$ mais l'algorithme est d'autant ralenti que le domaine est creux. Ou bien la variante que je préfère pour sa stabilité où nous stockons pour chaque valeur son indice dans le bitset, ce qui permet de stocker juste le nombre de bits nécessaires et où retirer une valeur se fait en $O(log_2 v)$.

4.2 Temps de résolution des problèmes

Cette partie présente les temps de résolution (en secondes) des problèmes suivants : n-reines, carré magique et sudoku. Le problème Send More Money n'est pas étudié puisqu'il possède une complexité constante et sa résolution est rapide. Une valeur X signifie qu'une

solution est trouvée en temps raisonnable dans au moins 5% des essais (inférieur à 3 minutes). Dans les autres cas, ∞ est indiqué.

4.2.1 N-reines

Le tableau 3 présente les temps d'exécution du solveur sur le problème des N-Queen. Les temps pour trouver une solution sur ce problème sont assez stables. Nous n'avons pas effectué de moyenne, les valeurs indiquées sont des ordres de grandeur. Nous avons ajouté une contrainte afin de casser une symétrie, ce qui nous permet d'après nos tests de trouver plus vite toutes les solutions et aussi de trouver plus vite une solution unique. Nous avons réussi à casser une seule symétrie, le nombre de solutions est au mieux divisé par deux.

N	Une solution	Toutes les solutions	Nombre de solutions
8	$1.7e^{-4}$	$4.8e^{-4}$	46
10	$2.2e^{-4}$	$7.5e^{-3}$	362
11	$4.9e^{-4}$	$2.7e^{-2}$	1340
12	$2.1e^{-4}$	$1.2e^{-1}$	7100
13	$2.1e^{-4}$	$5.9e^{-1}$	36856
14	$2.2e^{-4}$	3.3	182798
15	$3.6e^{-4}$	20	1139592
16	$3.2e^{-4}$	130	7386256
20	$3.8e^{-4}$	∞	∞
100	$6.8e^{-4}$	∞	∞
1000	$2.1e^{-1}$	∞	∞
2000	2.0	∞	∞
3000	7.5	∞	∞
4000	22	∞	∞
5000	38	∞	∞
6000	62	∞	∞
7000	96	∞	∞
8000	140	∞	∞
9000	180	∞	∞
10000	230	∞	∞

FIGURE 3 – Temps de calcul pour le problème des N-Queen en secondes

4.2.2 Carré magique

Le tableau 4 présente les temps d'exécution du solveur sur le problème du carré magique. Les temps pour trouver une solution sur ce problème sont variés et nous n'avons pas réalisé de moyenne. Pour le carré magique, nous nous arrêtons à n=20 de manière arbitraire. Nous avons ajouté trois contraintes de comparaison sur les coins du carré afin de casser toutes les symétries possibles. Ainsi, nous pouvons ainsi trouver toutes les solutions plus rapidement (bien que cela soit trop long à partir de n=5). Nous n'avons pas déterminé l'influence de cette symétrie sur la recherche d'une seule solution, ces contraintes peuvent accélérer, ralentir ou

bien n'avoir aucun impact sur le temps résolution et cela semble aussi dépendre de la taille du carré d'après nos essais.

N	Une solution	Toutes les solutions	Nombre de solutions
1	$1e^{-5}$	$1e^{-5}$	1
2	$5e^{-5}$	$5e^{-5}$	0
3	$2e^{-4}$	$2e^{-4}$	1
4	$1e^{-3}$	$3.1e^{-1}$	880
5	$mostly < 1$	∞	∞
6	$mostly < 1$	∞	∞
7	$mostly < 1$	∞	∞
8	$mostly < 1$	∞	∞
9	$mostly < 1$	∞	∞
10	$mostly < 1$	∞	∞
11	$mostly < 1$	∞	∞
12	$mostly < 1$	∞	∞
13	X	∞	∞
14	X	∞	∞
15	X	∞	∞
16	X	∞	∞
17	X	∞	∞
18	X	∞	∞
19	X	∞	∞
20	X	∞	∞

FIGURE 4 – Temps de calcul pour le problème des N-Queen en secondes

4.2.3 Sudoku

Le tableau 5 présente les temps d'exécution du solveur sur le problème du Sudoku. Les temps pour trouver une solution sur ce problème sont variés et nous n'avons pas réalisé de moyenne. Nous n'avons trouvé aucun moyen de casser les symétries des solutions du sudoku. En revanche, les contraintes all different sur les diagonales du x-sudoku nous permettent de casser deux symétries. Nous pensions que le x-sudoku serait plus simple à résoudre. En effet, nous pensions que ses deux contraintes supplémentaires nous permettraient de plonger plus vite vers de bonnes solutions et d'éviter plus facilement les mauvaises. Pourtant, et ceux même en cassant les deux symétries, nous remarquons que le problème du sudoku classique est plus simple à résoudre.

	Une solution		Toutes les solutions	
N	**sudoku**	**x-sudoku**	**sudoku**	**x-sudoku**
1	$1e^{-4}$	$1e^{-4}$	$1e^{-5}$	$1e^{-5}$
2	$1e^{-4}$	$1e^{-4}$	$2e^{-3}$	$1e^{-4}$
3	$2e^{-4}$	$5e^{-4}$	∞	∞
4	$2e^{-3}$	$5e^{-3}$	∞	∞
5	$mostly < 1$	$mostly < 1$	∞	∞
6	X	∞	∞	∞
7	∞	∞	∞	∞

FIGURE 5 – Temps de calcul pour le problème du sudoku en secondes

5 Conclusion

Même si nous n'avons pas implémenté plusieurs méthodes de pruning, nous pensons avoir atteint notre objectif : la réalisation d'un solveur générique relativement efficace. En effet, pour résoudre un problème sur notre solveur, il suffit au minimum de définir ses domaines et ses contraintes comme on le ferait pour utiliser un solveur comme GLPK (mais en C++). Nos plus grandes difficultés ont été le débogage. En effet, notre solveur utilise beaucoup les indices de tableau et certaines erreurs nous ont donné des bugs difficiles à remarquer, à comprendre et à identifier. Après avoir atteint une base solide pour notre solveur, nous en avons profité pour effectuer beaucoup d'essais et mieux réaliser l'importance des symétries ou l'influence de l'ordre des contraintes sur le temps de résolution selon le problème. Nous avons choisi des problèmes compliqués (sudoku et carré magique) afin de tester le comportement de notre solveur et le temps de résolution en fonction de la taille des instances. Pour faire évoluer notre projet, nous pourrions tenter l'implémentation d'un bitset et la comparer, effectuer plus de tests sur des stratégies de résolution comme d'autres méthodes de pruning (bien que celle que nous utilisons semble être la plus efficace et la plus généralisable). De plus, nous pourrions implémenter un problème de cryptarithme plus général que le Send More Money en laissant choisir les différents mots voire les opérateurs. Nous pourrions implémenter un parseur qui permettrait de lire une modélisation sous un format bien défini et ainsi résoudre tous types de problèmes de manière encore plus générale (à la manière de MiniZinc).