
# Résolution de problèmes, génération de mots-croisés

Androide M1 – RP

Thibault Gigant
Laura Greige

*Enseignants :*

Patrice Perny
Morgan Chopin

2015 – 2016

# Table des matières

**Introduction** **1**

**1 Modélisation par un CSP et résolution** **1**
1.1 Modélisation . . . . . 1
1.2 Expérimentation . . . . . 1

**2 Extension au cas pondéré** **3**
2.1 Modélisation . . . . . 3
2.2 Expérimentation . . . . . 4
2.3 Expérimentation . . . . . 5
2.4 Bonus . . . . . 5

**Conclusion** **6**

# Introduction

Dans ce projet, on s'intéresse à compléter une grille de taille $M \times N$. Dans un premier temps, on modélisera ce problème comme un problème de satisfaction de contraintes et on développera plusieurs méthodes différentes de recherche d'une solution au problème CSP : arc consistance du graphes des contraintes ($AC3$), forward checking ($FC$), conflict back-jumping ($CBJ$). La grille peut être déjà partiellement remplie par l'utilisateur.

Dans un second temps, on modélisera ce problème comme un CSP valué. Chaque mot du dictionnaire sera muni d'un poids positif ou nul appartenant à $[0, 1]$ et on développera un algorithme de type Branch and Bound qui permette de rechercher la solution optimale qui serait la solution de poids maximal dans la grille.

# 1 Modélisation par un CSP et résolution

## 1.1 Modélisation

Pour résoudre ce problème, on peut le modéliser comme un problème de satisfaction de contraintes en associant une variable à chaque mot de la grille. Supposons qu'il y ait $m$ mots dans la grille.

**Variables :** $x_i \ , \ \forall i \in \{1, ..., m\}$

**Domaines :** Soit *dict* un dictionnaire de mots admissibles :

$$D(x_i) = \{X \in dict\}$$

**Contraintes :**

— Soit $l_i$ la taille du mot en $i$ :

$$len(x_i) = l_i \tag{1}$$

— Pour tous mots $x_i$ et $x_j$ qui se croisent à la $q$-ième lettre de $x_i$ et à la $p$-ième lettre de $x_j$, on a :

$$x_i[q] = x_j[p] \tag{2}$$

— Si l'on ajoute la contrainte supplémentaire qu'un même mot ne peut apparaître plus d'une fois dans la grille, il suffit d'ajouter la contrainte *AllDiff* :

$$AllDiff\,(x_1, x_2, ...x_m) \tag{3}$$

## 1.2 Expérimentation

### Application

Dans cette section, nous allons tester les performances de nos algorithmes, d'abord en utilisant RAC avec Forward Checking sans AC3 préalable, ensuite RAC avec Forward Checking et AC3 préalable.

| | Grille A | Grille B | Grille C |
|---|---|---|---|
| AC3 | t1 | t2 | t3 |
| FC sans AC3 préalable | t1 | t2 | t3 |
| FC avec AC3 préalable | t1 | t2 | t3 |

### Conflict BackJumping

### Applications sur des grilles de tailles plus grandes

La méthode ( ) étant la plus efficace, on l'applique sur des instances de grilles de tailles plus grandes.

| | Temps d'éxecution | % de solution trouvée |
|---|---|---|
| Taille = 10 | t1 | % |
| Taille = 100 | t2 | % |

# 2 Extension au cas pondéré

## 2.1 Modélisation

Dans cette partie, on suppose que chaque mot du dictionnaire est muni d'un poids positif ou nul $\in [0, 1]$ représentant l'importance qu'un utilisateur attache à ce mot. Pour trouver la solution optimale du problème, c-à-d la solution de poids maximum dans la grille, il faut utiliser une méthode exacte, comme le « Branch & Bound » qui sera représenté par un arbre de recherche binaire.

Chaque sous-problème créé au cours de l'exploration est désigné par un nœud qui représente les mots choisis pour chaque variable $x_i$ de la grille. Les branches de l'arbre symbolisent le processus de séparation, elles représentent la relation entre les nœuds (ajout du mot $m_i \in D(x_i)$. Cette méthode arborescente nous permettra donc d'énumérer toutes les solutions possibles.

En chacune des feuilles de l'arbre, on a une solution possible qui correspond ou non à une solution admissible et on retient la meilleure solution obtenue, qui dans ce cas, sera la solution vérifiant toutes de poids maximum.

Pour améliorer la complexité du « Branch & Bound », seules les solutions qui vérifient les contraintes de consistances potentiellement de bonne qualité seront énumérées, les solutions ne pouvant pas conduire à améliorer la solution courante ne sont pas explorées.

Le « Branch & Bound » est basé sur trois principes :

### Principe de séparation

Le principe de séparation consiste à diviser le problème en un certain nombre de sous-problèmes qui ont chacun leur ensemble de solutions réalisables. En résolvant tous les sous-problèmes et en prenant la meilleure solution trouvée, on est assuré d'avoir résolu le problème initial. Ce principe de séparation est appliqué de manière récursive à chacun des sous-ensembles tant que celui-ci contient plusieurs solutions.
Remarque : La procédure de séparation d'un ensemble s'arrête lorsqu'une des conditions suivantes est vérifiée :
– on sait que l'ensemble ne contient aucune solution admissible (cas où les mots ne vérifient pas les conditions de consistance) ;
– on connaît une solution meilleure que toutes celles de l'ensemble ;

### Principe d'évaluation

Le principe d'évaluation a pour objectif de connaître la qualité des nœuds à traiter. La méthode de « Branch and Bound » utilise deux types de bornes : une borne inférieure de la fonction d'utilité du problème initial et une borne supérieure de la fonction d'utilité. La connaissance d'une borne inférieure du problème et d'une borne supérieure de la fonction d'utilité de chaque sous-problème permet d'arrêter l'exploration d'un sous-ensemble de solutions qui ne sont pas candidats à l'optimalité.
– borne supérieure :
– borne inférieure :

### Parcours de l'arbre

Le type de parcours de l'arbre permet de choisir le prochain nœud à séparer parmi l'ensemble des nœuds de l'arborescence. L'exploration en profondeur privilégie les sous-problèmes obtenus par le plus

grand nombre de séparations appliquées au problème de départ, c'est-à-dire aux sommets les plus éloignés de la racine (= de profondeur la plus élevée). L'obtention rapide d'une solution admissible en est l'avantage.

## 2.2 Expérimentation

**Variables :** $x_i$ , $\forall i \in \{1, ..., m\}$

**Domaines :** Soit *dict* un dictionnaire de mots admissibles :

$$D(x_i) = \{X \in dict\}$$

**Contraintes :** Soit $l_i$ la taille du mot en $i$ :

$$len(x_i) = l_i \tag{4}$$

Pour tous mots $x_i$ et $x_j$ qui se croisent à la $q$-ième lettre de $x_i$ et à la $p$-ième lettre de $x_j$, on a :

$$x_i[q] = x_j[p] \tag{5}$$

Si l'on ajoute la contrainte supplémentaire qu'un même mot ne peut apparaître plus d'une fois dans la grille, il suffit d'ajouter la contrainte *AllDiff* :

$$AllDiff\,(x_1, x_2, ...x_m) \tag{6}$$

## 2.3 Expérimentation

## 2.4 Bonus

# Conclusion