Université de Bordeaux

Programme-joueur de Mastermind paramétrique

Stage de première année de Master Informatique

Timothée Jourde

Maîtres de stage

Hugo Gimbert
Nathanaël Fijalkow

Laboratoire Bordelais de
Recherche en Informatique

Enseignant référent

Grégoire Passault

12 septembre 2018

Table des matières

1 Introduction

Le Mastermind[1] est un jeu de logique où l'un des deux joueurs doit deviner un code secret à partir des indications données par l'autre joueur. La version originale du Mastermind se joue avec un code de quatre pions colorés choisis parmi un ensemble de six couleurs.

L'objectif de ce stage est de réaliser un programme-joueur de Mastermind, avec une taille de code et un nombre de couleurs paramétrables à l'exécution.

2 Objectif

2.1 Formalisation du jeu

On formalise une partie de Mastermind par :

- une taille de combinaison n
- un nombre de couleurs m
- une combinaison secrète $c_s : [1\,..\,n] \to [1\,..\,m]$
- une suite de tentatives $(c_i, h(c_s, c_i))$, où :
  - $i \in [1\,..\,]$ est la tentative

[1] https://en.wikipedia.org/wiki/Mastermind_(board_game)

— $c_i : [1 \,..\, n] \to [1 \,..\, m]$ est la combinaison jouée

— $h(s, c) = (g, b)$ est la fonction associant une indication (g, b) à une combinaison secrète s et à une combinaison jouée c, où :

  — g est le nombre de couleurs de c correspondant à celles de s et à la même position (couleurs “bien placés”).

$$g = \sum_{j=0}^{n} \begin{cases} 1 & \text{si } c(j) = s(j) \\ 0 & \text{sinon} \end{cases}$$

  — b est le nombre de couleurs de c correspondant à celles de s mais pas à la même position (couleurs “mal placés”).

$$b = \sum_{j=0}^{n} \begin{cases} 1 & \text{si } c(j) \in s[[1 \,..\, n]] \wedge c(j) \neq s(j) \\ 0 & \text{sinon} \end{cases}$$

L’un des deux joueurs décide de c_s et donne les indications à chaque tentative de l’autre joueur. Le joueur donnant les indications est donc passif et peut être remplacé par un programme trivial. Le but du jeu étant, pour l’autre joueur, de deviner c_s en un minimum de tentatives.

2.2 Présentation de l’algorithme

On souhaite réaliser un programme-joueur de Mastermind (avec n et m paramétrables) capable de deviner c_s en un nombre raisonable de tentatives. L’algorithme présenté ci-après est inspiré d’une méthode[1] mise au point par Miroslav Klimoš et Antonín Kučera.

Pour un n et m fixé, on pose :

— C l’ensemble des combinaisons.

— C_i l’ensemble des combinaisons privé des combinaisons déjà jouées lors des tentatives précédentes à la tentative i (combinaisons candidates).

$$C_i = C \setminus \{c_j \mid j \in [1 \,..\, i-1]\}$$

— S_i l’ensemble des combinaisons secrètes possibles à la tentative i.

$$S_i = \{s \mid s \in C \wedge j \in [1 \,..\, i-1] \wedge h(s, c_j) = h(c_s, c_j)\} \qquad \textbf{[1]}$$

— K_i : pour chaque combinaison c de C, on génère un ensemble d’indications $(s, h(s, c))$ avec chaque combinaison secrète s de S_i.

$$K_i = \{\ (c,\ \{(s, h(s, c)) \mid s \in S_i\})\ \mid\ c \in C_i\ \} \qquad \textbf{[2]}$$

On associe un score $k(K_c)$ à chaque combinaison, et on déduit la prochaine combinaison jouée c_i du score maximal.

$$c_i = c \mid (c, K_c) \in K_i \wedge (c', K'_c) \in K_i \wedge k(K_c) \geq k(K'_c)$$

Le choix de la fonction k associant un score à un ensemble d'indications K_c déterminera la façon dont l'algorithme convergera. On peut simplement choisir le cardinal de l'ensemble sans tenir compte des combinaisons secrètes s associés. Cela aura pour effet de maximiser le fractionnement de S_i dans le cas moyen.

$$k(K_c) = |\{h' \mid (s, h') \in K_c\}|$$

3 Réalisation

Nous allons maintenant nous focaliser sur les étapes clefs de l'algorithme décrit précédement et en expliquer l'implémentation. Les sources de ce programme écrit en Haskell [2] sont disponibles en ligne : `https://github.com/timjrd/mastermind`.

Tel que nous l'avons décrit, l'algorithme nécessite de calculer un produit cartésien[2] K_i entre C_i et S_i, or cette opération est très coûteuse et nécessite d'énumérer complétement les deux ensembles, ce qui devient hors de portée lorsque n et m sont trop grands. On approxime donc K_i par une méthode de type monte-carlo en énumérant seulement un sous-ensemble randomisé de C_i et de S_i.

3.1 Énumération des combinaisons candidates

Voir `src/Mastermind/Candidate.hs`.

On cherche à énumérer un sous-ensemble randomisé de C_i. On génère pour cela une suite de combinaisons aléatoires, et on vérifie simplement que chaque nouvelle combinaison est unique et n'a pas déjà été jouée.

3.2 Énumération efficace des combinaisons secrètes

On cherche à énumérer un sous-ensemble randomisé de S_i. On peut aborder le problème de la même manière que pour C_i : générer une suite de combinaisons aléatoires s et vérifier que chacune satisfasse $h(s, c_j) = h(c_s, c_j)$[1]. Cette première approche a été implémentée mais s'est révélée impraticable au delà d'un certain n et m.

Il a donc fallu chercher une autre solution : déduire directement S_i des tentatives précédentes (contraintes). La méthode ainsi retenue se déroule en deux étapes : l'énumération des "permutations d'indications", puis la génération des combinaisons secrètes à partir de ces permutations. Nous illustrerons cette section par l'exemple suivant :

[2] `https://www.haskell.org`

$n = 3 \quad m = 3$

$c_s(1)$	$c_s(2)$	$c_s(3)$
3	1	3

i	$c_i(1)$	$c_i(2)$	$c_i(3)$	$h(c_s, c_i)$
1	2	3	1	$(0, 2)$
2	3	2	2	$(1, 0)$

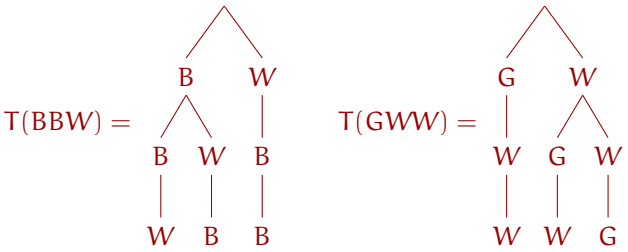
3.2.1 Permutations d'indications

Voir `src/Mastermind/Permutation.hs`.

La première étape consiste à représenter les indications sous la forme de séquences d'étiquettes. G pour une bonne couleur bien placée, B pour une bonne couleur mal placée, *W* pour une mauvaise couleur. Appliquons ceci à notre exemple :

$$h(c_s, c_1) = (0, 2) \equiv P(BBW)$$
$$h(c_s, c_2) = (1, 0) \equiv P(GWW)$$

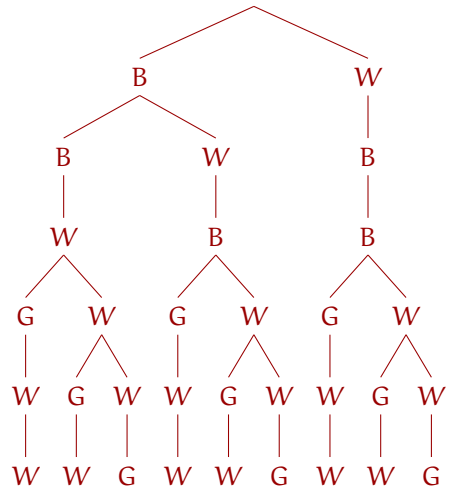
Ici $n = 3$: on obtient donc une séquence de trois étiquettes par combinaison jouée. En associant une séquence à sa combinaison, on associe dans l'ordre une étiquette à chaque membre [3] de la combinaison, et on exprime ainsi le fait qu'un membre en particulier est bien placé, mal placé, ou absent de la combinaison secrète. Une indication n'est cependant pas équivalente à une séquence particulière x, mais à l'ensemble de ses permutations $P(x)$, que l'on construit grâce à un arbre de permutation $T(x)$:



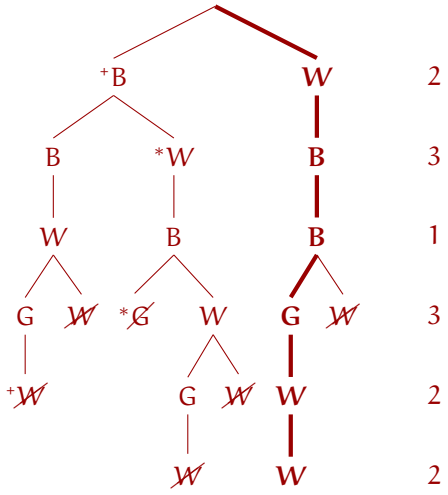
On cherche maintenant à élaguer ces arbres pour ne conserver que les permutations valides au regard de l'ensemble des contraintes (une contrainte correspondant à un arbre). Il faut donc mettre en relation ces différents arbres, on procède ainsi à leur concaténation :

[3] couleur et position

$T(BBW) \cdot T(GWW) =$



Chaque niveau de l'arbre correspond à une position et à une couleur. Le second niveau correspond ainsi à $c_1(2) = 3$ tandis que le dernier niveau correspond à $c_2(3) = 2$. Pour élaguer l'arbre on effectue un parcours en largeur depuis la raçine en supprimant les nœuds incompatibles avec leurs ancêtres. Illustrons cette opération avec notre exemple (les couleurs correspondantes sont représentées à droite de l'arbre) :



On a entre autre supprimé le nœud $^{+}\not{W}$ puisqu'il signifie "2 est une mauvaise couleur" alors que le nœud ascendant ^{+}B signie de manière contradictoire "2 est une bonne couleur mal placée". De même, on a supprimé $^{*}\not{G}$ qui signifie "3 est une bonne couleur bien placée" alors que $^{*}W$ signifie "3 est une mauvaise couleur". Le chemin en gras est entièrement valide, on retiendra donc cette séquence d'étiquettes pour l'étape suivante.

3.2.2 Génération des combinaisons secrètes

Voir `src/Mastermind/Secret.hs`.

On retient donc la séquence d'étiquettes $WBBGWW$, soit WBB pour c_1 et GWW pour c_2. Pour chaque position de chaque combinaison, on détermine l'éventuelle couleur requise ainsi que les éventuelles couleurs exclues. On détermine également les couleurs requises sur l'ensemble d'une combinaison. On agrège ensuite les différentes contraintes ainsi obtenues.

c_1	2	3	1
étiquette	W	B	B
requise	$\varnothing$	$\varnothing$	$\varnothing$
exclues	$\{2\}$	$\{3,2\}$	$\{1,2\}$
requises	$\{3,1\}$		

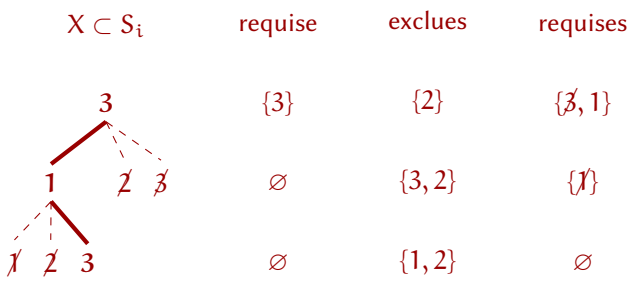
$\cup$

c_2	3	2	2
étiquette	G	W	W
requise	$\{3\}$	$\varnothing$	$\varnothing$
exclues	$\{2\}$	$\{2\}$	$\{2\}$
requises	$\{3\}$		

$=$

requise	$\{3\}$	$\varnothing$	$\varnothing$
exclues	$\{2\}$	$\{3,2\}$	$\{1,2\}$
requises	$\{3,1\}$		

Enfin, on génère le sous-ensemble X de S_i correspondant à notre séquence d'étiquettes grâce à une structure arboressante à partir des contraintes agrégées obtenues précédemment.



On repète l'opération pour chaque séquence d'étiquettes obtenues à l'étape précédente.

4 Limitations et améliorations possibles

4.1 Randomisation

Comme évoqué au début de la section 3, on énumère un sous-ensemble *randomisé* de S_i. Cette propriété est partiellement obtenue en randomisant les deux structures arborresantes utilisées. L'ordre des enfants d'un nœud est ainsi aléatoire ; on espère ainsi, lors du parcours final de l'arbre, énumérer l'ensemble correspondant dans un ordre aléatoire. Or cette technique, bien que peu coûteuse, n'offre pas une randomisation de bonne qualité, ce qui risque d'affecter les résultats de l'algorithme.

4.2 Consommation mémoire

Par soucis de simplification du code, on utilise deux structures de données arboressantes intermédiaires, et on exploite l'évaluation paresseuse de Haskell pour construire et consommer progressivement ces structures, à la demmande.

Au delà d'un certain n et m, on observe une explosion de la consommation mémoire. Cette consommation pourrai très certainement être réduite en se passant de ces structures intermédiares, au prix d'une obfuscation du code.

4.3 Parallélisation

Haskell étant un langage fonctionnel pur, il n'autorise aucun changement d'état ni effet de bord. Notre programme pourrait ansi être parallélisé sans réécriture majeur. Une première approche pourrait être de calculer simultanément la génération des combinaisons secrètes à partir des différentes séquences d'étiquettes.

5 Conslusion

Nous avons décrit dans ce rapport la réalisation de notre programme-joueur de Mastermind paramétrique. L'objectif de ce stage a été atteint avec un programme exécutable offrant des performances bien supérieures — et des résultats au moins équivalents — à une approche naïve par recherche exhaustive.

Sur un plan personnel, j'ai trouvé ce stage très enrichissant puisqu'il m'a poussé à réfléchir au problème posé de manière formelle, et d'une façon approfondie. Je souhaite à ce titre remercier M. Hugo Gimbert ainsi que M. Nathanaël Fijalkow pour m'avoir donné l'opportunité de travailler sur ce sujet de stage, et pour m'avoir guider dans la réalisation du projet.

Références

{1} Miroslav Klimoš et Antonín Kučera. "Strategy Synthesis for General Deductive Games Based on SAT Solving". In : *CoRR* abs/1407.3926 (2014). arXiv : 1407.3926. URL : http://arxiv.org/abs/1407.3926.