combine : une bibliothèque OCaml pour la combinatoire

Rémy El Sibaïe, Jean-Christophe Filliâtre

`combine` : une bibliothèque OCaml pour la combinatoire

Rémy El Sibaïe[1] & Jean-Christophe Filliâtre[2,1]

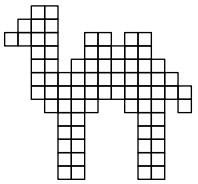
1: LRI, Université Paris Sud,
91405 Orsay CEDEX, France
2: CNRS,
`remy.el-sibaie@u-psud.fr`
`jean-christophe.filliatre@lri.fr`

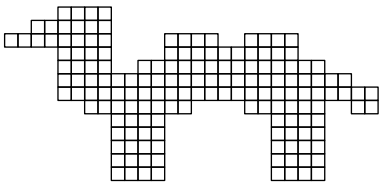
Résumé

Cet article présente `combine`[1], une bibliothèque OCaml pour la combinatoire[2]. Elle fournit deux solutions algorithmiquement très différentes au problème de la couverture exacte d'une matrice (EMC) : les liens dansants de Knuth et une variante des diagrammes de décision binaire appelée ZDD. De très nombreux problèmes de combinatoire peuvent être ramenés au problème EMC. La bibliothèque `combine` l'illustre notamment sur l'exemple du pavage rectangulaire en dimension 2.

1. Introduction

Rien n'est plus excitant en informatique que d'écrire un programme qui calcule en quelques secondes un résultat qu'on ne pourrait obtenir sans l'aide d'un ordinateur. La combinatoire est un domaine particulièrement fertile en la matière. Si certains problèmes se prêtent volontiers à l'analyse mathématique, comme le nombre total de livres dans la bibliothèque de Borgès ou encore le nombre de labyrinthes parfaits rectangulaires, beaucoup d'autres exigent de recourir à la programmation. Considérons par exemple le nombre de façons différentes de paver la figure ci-contre avec des dominos 2×1. Il est relativement aisé d'écrire un programme qui va explorer toutes les solutions possibles. En une fraction de seconde, on déterminera alors qu'elles sont au nombre de 46 976.





Mais considérons maintenant un problème de pavage légèrement plus complexe, avec la figure de gauche qui demande deux fois plus de dominos. Le problème est nettement plus difficile. Une approche brutale, qui trouve les solutions une à une, n'aboutira pas. Il existe en effet 32 420 116 341 024 288 solutions distinctes. Même à raison d'un milliard de solutions trouvées par seconde, il faudrait encore plus d'un an pour parvenir à ce chiffre. Et pourtant le chiffre ci-dessus a été obtenu par les auteurs en quelques dizaines de secondes seulement, à l'aide d'une bibliothèque OCaml, `combine`, qui permet notamment de résoudre des problèmes de pavage rectangulaires en dimension 2. Cet article a pour but de présenter cette bibliothèque.

La bibliothèque `combine` est centrée autour du problème de la *couverture exacte de matrice*, désigné par EMC dans ce qui suit. Étant donnée une matrice contenant uniquement des 0 et des 1, il s'agit

[1] Le lecteur perspicace notera que `combine` est à la fois un mot anglais, français et un anagramme de « combien ».

[2] Il s'agit d'un travail réalisé par le premier auteur pendant un stage de L3 au LRI, d'avril à juin 2012, sous la direction du second auteur.

de déterminer un sous-ensemble de ses lignes contenant un 1 et un seul par colonne. Étant donné un problème EMC, on peut s'intéresser uniquement à la question de savoir s'il possède une solution, au problème d'en construire une le cas échéant, au problème de dénombrer les solutions, ou encore au problème de les construire toutes. Ainsi le problème EMC suivant

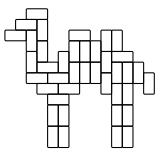
$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

possède exactement deux solutions, à savoir les sous-ensembles de lignes $\{0, 4\}$ et $\{1, 3\}$. L'intérêt d'EMC est qu'il est très facile d'y encoder de nombreux problèmes de combinatoire. Ainsi un problème de pavage, tel que celui présenté en introduction, peut être encodé de la façon suivante : chaque colonne représente une case à recouvrir et chaque ligne une façon possible de poser une pièce. Ainsi le petit chameau donne lieu à une matrice de 80 colonnes et 122 lignes, et le gros chameau à une matrice de 160 colonnes et 268 lignes.

Pour résoudre le problème EMC, plusieurs techniques sont à notre disposition. Une première consiste à découvrir les solutions une par une, en utilisant du *backtracking*. Knuth a proposé pour cela un algorithme appelé les *liens dansants* ou encore DLX [6]. Il consiste à relier ensemble tous les 1 de la matrice EMC dans un réseau de listes circulaires doublement chaînées horizontales et verticales, puis à ôter successivement les éléments de ces listes, au fur et à mesure des choix qui sont faits, et à les réinsérer ensuite. L'efficacité de cet algorithme est due notamment à la possibilité de réinsérer un élément qui vient d'être supprimé d'une liste doublement chaînée sans utiliser d'autre information que celle qui se trouve déjà dans les deux pointeurs vers ses anciens voisins. L'algorithme DLX ne trouve les solutions du problème EMC qu'une à une, mais avec une très grande efficacité. Pour le petit chameau, les 46 976 solutions sont trouvées en un dixième de seconde.

Une seconde approche du problème EMC consiste à utiliser une variante des BDD appelée ZDD, pour *Zero-suppressed binary Decision Diagram*, et due à Minato [5]. Comme les BDD, les ZDD permettent de représenter une fonction booléenne de n variables sous la forme d'un graphe orienté acyclique. Leur particularité est d'offrir une représentation plus compacte que les BDD dans certains cas, et notamment ceux qui nous intéressent ici. On peut effectuer des opérations entre ZDD, comme par exemple la conjonction ou la disjonction, ou encore calculer très simplement le nombre d'assignations distinctes des n variables qui donne la valeur de vérité 1. Les ZDD sont décrits en détail dans le volume 4A de *The Art of Computer Programming* [7]. Le problème EMC peut être réduit à un ZDD de la façon suivante. Les n variables sont les n lignes de la matrice. On commence par convertir chaque colonne de la matrice EMC en un ZDD qui correspond aux sous-ensembles de lignes qui contiennent exactement un 1 dans cette colonne. Puis on fait l'intersection de tous ces ZDD[3]. Cette technique a les avantages et les inconvénients usuels des BDD. Si on parvient à construire le ZDD sans épuiser la mémoire vive, alors le problème est résolu. En effet, on peut alors facilement déterminer s'il existe une solution, et une construire une le cas échéant, mais surtout calculer le nombre total de solutions (en exploitant le partage, comme avec un BDD). C'est ainsi qu'on a calculé le nombre gigantesque de solutions au problème de pavage du grand chameau, avec un ZDD de 784 132 nœuds.

La bibliothèque `combine` contient, outre ces deux solutions au problème EMC, trois exemples d'application. Les deux premiers sont des exemples relativement simples. Il s'agit du problème des N reines et du Sudoku. Le troisième exemple est celui du pavage rectangulaire en dimension 2. Il se présente sous la forme d'une bibliothèque OCaml pour décrire un problème de pavage, le réduire à un problème EMC et interpréter les solutions. C'est ainsi qu'a été obtenue la solution ci-contre[4] au problème de pavage



[3] Les ZDD étaient au concours X/ENS en 2012 mais ce n'est, bien sûr, que pure coïncidence.

[4] Ce résultat de pavage a été dessiné automatiquement avec la bibliothèque OCaml `mlpost` [1], à partir de la solution produite par `combine`.

```
pattern caml = {
....****...................
..******...................
********....****..****......
....****....**********.....
....****..**************....
....**********************..
....************************
......********....******..**
........****........****....
........****........****....
........****........****....
........****........****....
........****........****....}
```

```
pattern domino = {**}

problem tiling_a_caml = caml [domino ~sym]

timing on

solve dlx tiling_a_caml svg "output.svg"

count zdd tiling_a_caml
```

Figure 1: Exemple d'entrée de l'outil `combine` (sur deux colonnes).

donné en introduction. Enfin, la distribution de `combine` contient un interprète pour un petit langage permettant de décrire un problème de pavage dans une syntaxe concrète et d'utiliser les différents outils de la bibliothèque sans recourir à un programme. La figure 1 contient un exemple d'entrée pour cet interprète. On y définit deux motifs `caml` et `domino` en les dessinant en ASCII. Puis on définit le problème de pavage `tiling_a_caml` en donnant le motif à paver (`caml`) et la liste des pièces pour le paver. Ici cette liste est réduite à `domino`. L'option `~sym` indique que la pièce peut être tournée et retournée à volonté, ce qui ici se réduit à deux façons distinctes de la poser (horizontalement et verticalement). On demande la résolution du problème avec l'algorithme DLX (`solve dlx`), la solution étant exportée au format SVG dans un fichier. Enfin, on dénombre les solutions avec un ZDD (`count zdd`).

Le reste de cet article est organisé de la façon suivante. La section 2 présente l'interface de la bibliothèque `combine` et la section 3 illustre son utilisation sur le problème des N reines. La section 4 donne ensuite des détails techniques sur la réalisation des algorithmes DLX et ZDD. La section 5 décrit l'application au pavage. L'article s'achève sur quelques extensions possibles. La bibliothèque `combine` est un logiciel libre, distribué sous licence LGPL à l'adresse `http://www.lri.fr/~filliatr/combine/`.

2. Interface

Cette section décrit l'interface des trois modules principaux de `combine`, à savoir `Dlx`, `Zdd` et `Emc`.

`Dlx`. L'algorithme DLX résout le problème EMC. Un tel problème est ici matérialisé par un type abstrait `Dlx.t`, que l'on construit à partir d'une matrice de booléens.

```
val create: ?primary:int -> bool array array -> t
```

On généralise légèrement le problème présenté dans l'introduction, en distinguant des colonnes primaires qui doivent être couvertes (un 1 et un seul dans ces colonnes-là) et des colonnes secondaires qui peuvent ne pas être couvertes (au plus un 1). L'argument optionnel `primary` permet de spécifier que les `primary` premières colonnes sont des colonnes primaires, les suivantes étant des colonnes secondaires. Si cet argument est omis, toutes les colonnes sont primaires. Une fois le problème construit, le module `Dlx` fournit une fonction pour parcourir toutes ses solutions

```
val iter_solution: (solution -> unit) -> t -> unit
```

et en dérive immédiatement deux fonctions pour obtenir une seule solution (s'il en existe une) et dénombrer les solutions :

```
val get_first_solution: t -> solution
val count_solutions: t -> int
```

Le type `solution` est un type abstrait. Une fonction est fournie pour le décoder sous la forme d'une liste d'entiers, représentant les lignes de la matrice qui ont été sélectionnées. L'idée est ici de ne pas construire cette liste lorsque cela n'est pas nécessaire.

`Zdd`. Comme expliqué dans l'introduction, un ZDD est une structure de donnée représentant une fonction booléenne sur n variables. De manière équivalente, on peut l'interpréter comme un ensemble de parties de $\{0, 1, \ldots, n-1\}$, à savoir l'ensemble des entrées pour lesquelles la fonction booléenne renvoie 1. Le module `Zdd` fournit donc en premier lieu toutes les opérations de la signature `Set.S` pour des éléments étant eux-mêmes des ensembles d'entiers, c'est-à-dire

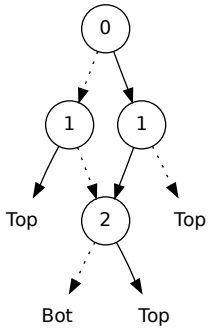
```
module S: Set.S with type elt = int
include Set.S with type elt = S.t
```

(Contrairement aux BDD, il n'est pas nécessaire de fixer à l'avance le nombre de variables.) Parmi les fonctions ensemblistes ci-dessus, on trouve bien évidemment une fonction `cardinal: t -> int`. Cependant, le cardinal d'un ZDD peut atteindre de très grandes valeurs, comme l'a montré notre exemple introductif, et il faut donc être également capable de calculer le cardinal avec un type autre que `int`. Pour cela, on propose en supplément un foncteur

```
module Cardinal(A: ARITH) : sig val cardinal: t -> A.t end
```

où `A` est un module d'arithmétique minimal fournissant un zéro, une unité et une addition. Une autre différence avec le module `Set` d'OCaml vient de l'unicité de la représentation d'un ZDD. Elle permet notamment de fournir des opérations `compare` et `equal` de complexité $O(1)$.

Par ailleurs, le module `Zdd` fournit une interface de plus bas niveau, qui expose la représentation interne. Un ZDD est un arbre binaire dont les feuilles sont `Bottom` ou `Top` et dont les nœuds internes sont étiquetés par des entiers. Plutôt que de parler de sous-arbres gauche et droit, on parle ici de sous-arbres négatif et positif. L'interprétation ensembliste est alors la suivante : `Bottom` est l'ensemble vide $\emptyset$; `Top` est le singleton $\{\emptyset\}$; et un arbre de racine i, de sous-arbre négatif représentant S^- et de sous-arbre positif représentant S^+, est l'ensemble $S^- \cup \{\{i\} \cup s \mid s \in S^+\}$. Ainsi le ZDD ci-contre représente l'ensemble $\{\{0\}, \{1\}, \{2\}, \{0, 1, 2\}\}$, les sous-arbres négatifs étant représentés en pointillés. Dit autrement, il y a autant d'éléments que de chemins menant de la racine à `Top`, et chaque descente vers un sous-arbre positif correspond à un élément sélectionné. L'unicité de représentation est par ailleurs assurée par les invariants suivants : d'une part un sous-arbre positif n'est jamais `Bottom` et d'autre part les étiquettes vont croissantes lorsqu'on descend dans l'arbre. Des fonctions d'affichage sont fournies, aux formats ASCII pour la forme ensembliste et au format DOT pour la représentation sous forme d'arbre. Le dessin ci-dessus illustre le résultat de la sortie DOT. Enfin, pour permettre notamment l'écriture de fonctions opérant directement sur les ZDD, le module `Zdd` expose le type `t` des ZDD :




```
type t = private Bottom | Top | Node of unique * int * t * t
```

(Le type `unique` représente ici un entier unique, propre à chaque ZDD.) Le type étant privé, deux valeurs `bottom` et `top` de type `t` et une fonction `construct: int -> t -> t -> t` sont fournies pour construire des ZDD.

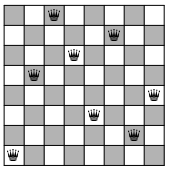
`Emc.` Le module `Emc` est une interface commune aux deux modules `Dlx` et `Zdd`, permettant d'utiliser indifféremment l'un ou l'autre sur un même problème EMC. Pour cela, il fournit deux modules `Emc.D` et `Emc.Z` ayant tous les deux la même interface `S` suivante :

```
module type S = sig
  type t
  val create: ?primary:int -> bool array array -> t
  type solution = int list
  val find_solution: t -> solution
  val iter_solution: (solution -> unit) -> t -> unit
  val count_solutions: t -> int
  module Count(A: ARITH): sig val count_solutions: t -> A.t end
end
```

Pour DLX, il s'agit d'une simple interface, les fonctionnalités étant exactement les mêmes. Pour ZDD, en revanche, il faut effectuer une réduction du problème EMC vers un ZDD, tel que les éléments du ZDD correspondent exactement aux solutions du problème EMC, c'est-à-dire aux sous-ensembles de lignes sélectionnées dans la matrice. On procède comme expliqué dans l'introduction.

3. Exemple d'utilisation : les N reines

L'article sur les liens dansants [6] contient notamment une application au problème des N reines. On reprend ici cet exemple pour illustrer l'utilisation de la bibliothèque `combine`. On rappelle que ce problème consiste à déterminer le nombre de façons de placer N reines sur un échiquier $N \times N$ sans que deux d'entre elles soient en prise. Voici par exemple l'une des 92 solutions pour $N = 8$:



On construit une matrice EMC dont les colonnes primaires représentent les N abscisses et les N ordonnées sur l'échiquier, et les colonnes secondaires les $2N - 1$ diagonales montantes et les $2N - 1$ diagonales descendantes. On a donc au total $6N - 2$ colonnes, que l'on peut numéroter de 0 à $6N - 3$ de la façon suivante :

colonnes primaires						colonnes secondaires					
abscisse			ordonnée			diagonale 1			diagonale 2		
0	…	$N-1$	N	…	$2N-1$	$2N$	…	$4N-2$	$4N-1$	…	$6N-3$

La matrice EMC contient autant de lignes qu'il y a de cases sur l'échiquier, soit N^2. Pour chaque case (i, j), avec $0 \leq i, j < N$, la ligne correspondante de la matrice EMC contient exactement quatre 1, indiquant respectivement l'abscisse, l'ordonnée et les deux diagonales de la case (i, j). Avec la numérotation ci-dessus, il s'agit donc des colonnes i, $N + j$, $2N + i + j$ et $4N - 1 + (N - 1 - i) + j$. On suppose que la variable OCaml `n` contient le nombre N. On commence par écrire une fonction `row` qui construit une ligne de la matrice EMC, comme un tableau de booléens :

N	nombre de solutions	matrice EMC	temps DLX	taille ZDD	temps ZDD	mémoire max.
4	2	16x22	0,00 s	8	0,00 s	3.88 Mo
5	10	25x28	0,00 s	40	0,00 s	5,81 Mo
6	4	36x34	0,00 s	24	0,02 s	7,75 Mo
7	40	49x40	0,00 s	186	0,03 s	13,56 Mo
8	92	64x46	0,00 s	373	0,45 s	45,48 Mo
9	352	81x52	0,00 s	1309	0,76 s	64,85 Mo
10	724	100x58	0,02 s	3120	3,56 s	221,03 Mo
11	2680	121x64	0,07 s	10503	14,57 s	697,39 Mo
12	14200	144x70	0,35 s	—	—	$>$ 2Go

Figure 2: Résoudre les N reines avec `combine`.

```
let row i j =
  let f k = k = i || k = n + j || k = 2*n + i + j || k = 4*n-1 + n-1-i + j in
  Array.init (6 * n - 2) f
```

Puis on construit la matrice EMC en accumulant toutes les lignes renvoyées par `row` dans une liste, puis en transformant au final la liste en tableau :

```
let emc =
  let lr = ref [] in
  for i = 0 to n - 1 do for j = 0 to n - 1 do lr := row i j :: !lr done done;
  Array.of_list !lr
```

Enfin on définit le nombre de colonnes primaires :

```
let primary = 2 * n
```

Pour dénombrer les solutions avec l'algorithme DLX, on utilise le module `Emc.D`. On crée la structure interne de DLX avec `create` puis on appelle la fonction `count_solutions` :

```
  let p = Emc.D.create ~primary emc in
  Emc.D.count_solutions p
```

Si on préfère utiliser les ZDD, on procède exactement de la même façon, en utilisant le module `Emc.Z` au lieu du module `Emc.D` :

```
  let p = Emc.Z.create ~primary emc in
  Emc.Z.count_solutions p
```

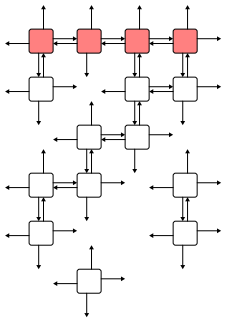
Les résultats sont données figure 2. On indique notamment le temps passé dans chaque algorithme et la quantité de mémoire utilisée. Cette dernière est exclusivement due aux ZDD, l'algorithme DLX n'utilisant quasiment pas de mémoire sur cet exemple (pour $N = 12$, l'algorithme DLX n'utilise pas plus de 40 ko sur une machine 64 bits). L'article de Knuth va un peu plus loin, en considérant notamment plusieurs heuristiques pour l'ordre dans lequel les colonnes sont sélectionnées par l'algorithme DLX. On peut alors aller assez loin avec l'algorithme DLX (jusqu'à $N = 18$). En revanche, comme on le voit sur la figure, le coût en mémoire des ZDD est rapidement prohibitif.

4. Réalisation

Cette section détaille la réalisation technique des trois modules `Dlx`, `Zdd` et `Emc`.

`Dlx`. L'algorithme DLX opère sur un ensemble de listes doublement chaînées circulaires, qui relient tous les 1 de la matrice EMC horizontalement et verticalement. Par ailleurs, chaque colonne de la matrice contient un nœud supplémentaire, appelé entête. Les entêtes sont également liés entre eux dans une liste chaînée circulaire. Voici un exemple de matrice et la structure DLX correspondante :

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$



Les pointeurs s'étendent jusqu'aux nœuds suivants, circulairement. Les entêtes apparaissent en haut. Chaque nœud est représenté par le type OCaml suivant :

```
type node = {
  mutable up:   node; mutable down:  node;
  mutable left: node; mutable right: node;
  mutable c:    node; mutable s:     int;  mutable name: string; }
```

Les quatre champs `up`, `down`, `left` et `right` matérialisent les deux listes doublement chaînées dont fait partie chaque nœud. Le champ `c` contient un pointeur vers l'entête de la colonne (ces pointeurs ne sont pas dessinés dans la figure ci-dessus, par souci de clarté). Le champ `name` contient le nom de chaque colonne, à des fins d'affichage. Enfin, l'entier contenu dans le champ `s` est interprété ainsi : pour un entête, il contient le nombre de nœuds dans la colonne correspondante et est utilisé pour choisir une colonne minimisant cette valeur ; pour tout autre nœud, il contient le numéro de la ligne de la matrice et est utilisé pour décoder la solution, le cas échéant.

Il est important de préciser une fois encore que l'algorithme DLX travaille en espace constant : une fois les nœuds créés à partir de la matrice, toute la recherche se fait en place, en modifiant seulement les champs `s`, `up`, `down`, `left` et `right`. C'est la danse des liens. Le pseudo-code de l'algorithme DLX est donné à la fin de cet article (annexe A). Pour plus de détails, on consultera l'article de Knuth introduisant cet algorithme [6].

`Zdd`. Le module `Zdd` est en tous points analogue à un module de BDD. Le partage maximal est obtenu par la technique du *hash-consing*, dont la mise en œuvre dans un langage comme OCaml est établie depuis longtemps [3]. En particulier, les ZDD déjà construits sont stockés dans une table de hachage utilisant des pointeurs faibles, ce qui permet au système de détruire les ZDD qui ne sont plus utilisés nulle part. Le type OCaml des ZDD est celui qui a déjà été donné plus haut, à savoir

```
type t = Bottom | Top | Node of unique * int * t * t
```

En particulier, chaque nœud de ZDD occupe exactement 5 mots mémoire, auxquels s'ajoute son entrée dans la table de *hash-consing*, c'est-à-dire environ deux mots.

Les différentes opérations sur les ZDD sont mémoïzées. Comme pour les BDD, c'est là la clé de l'efficacité. Chaque ZDD contenant un entier unique, il est extrêmement simple de mettre en œuvre cette mémoïzation et elle est de fait très efficace. Le code effectuant la mémoïzation est factorisé dans des opérateurs de point fixes. Ainsi le cardinal d'un ZDD s'écrit aussi simplement que

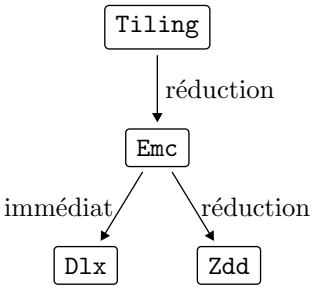
```
let cardinal = memo_rec1 (
  fun cardinal -> function
    | Top -> 1
    | Bottom -> 0
    | Node(_, i, z1, z2) -> cardinal z1 + cardinal z2)
```

et son coût est proportionnel au nombre de nœuds *distincts* dans le ZDD.

`Emc`. La seule difficulté du module `Emc` est la réduction du problème EMC en un ZDD. En effet, s'il est facile de construire le ZDD correspondant à une colonne de la matrice (en temps linéaire, en procédant de bas en haut), il reste de multiples possibilités pour effectuer l'intersection de tous ces ZDD et elles ne sont pas toutes de même coût. Nous avons testé plusieurs solutions et finalement retenu la solution consistant à trier les colonnes de la matrice selon la ligne du premier 1 qu'elles contiennent puis à les fusionner selon la méthode dite *balanced* proposée par Knuth [7, exercice 212 page 274].

5. Application au pavage

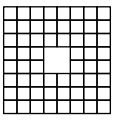
La bibliothèque `combine` fournit par ailleurs un module `Tiling` pour résoudre des problèmes de pavage rectangulaires en dimension 2, en les réduisant vers le problème EMC. On a donc l'agencement suivant des différents modules de `combine` :



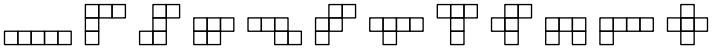
Interface. Le module `Tiling` fournit tout d'abord un sous-module `Tiling.Pattern` pour décrire des motifs rectangulaires. Il ne s'agit ni plus ni moins que de matrices de booléens, munies d'opérations de transformation (redimensionnement, fusion, isométrie, etc.). Vient ensuite un module `Tiling.Tile` pour décrire les pièces destinées au pavage. Une pièce est la donnée d'un motif, de sa multiplicité (zéro ou un exemplaire, exactement un exemplaire, ou une infinité d'exemplaires) et des isométries que l'on peut appliquer à cette pièce (rotations et/ou réflexions). Enfin le module `Tiling` définit un problème de pavage comme un motif à paver et une liste de pièces. La fonction `Tiling.emc` réduit un tel problème vers un problème EMC. En particulier, son résultat permet de décoder toute solution du problème EMC comme une liste de pièces, avec leurs positions et leurs orientations.

Le module `Tiling` utilise un module auxiliaire `D4` décrivant le groupe diédral D_4 des 8 isométries du plan qui laissent un carré invariant. On y trouve la structure de groupe de D_4 mais aussi tous ses sous-groupes. Ce module est d'intérêt général.

Langage d'entrée de l'outil `combine`. La syntaxe du petit interprète qui vient avec la bibliothèque `combine` est résumée figure 3. Un exemple a été donné plus haut, dans la figure 1. Donnons un autre exemple avec le problème de Scott [8]. Il s'agit de dénombrer le nombre de façons de paver la figure suivante



avec les douze pentaminos, c'est-à-dire les douze pièces suivantes :



Chaque pièce doit être utilisée une et une seule fois, mais peut être tournée ou retournée à volonté. On commence par décrire les douze pièces.

```
pattern I = {*****}

pattern V = {
***
*
*
}
```

etc. On définit ensuite la liste des douze pentaminos avec, pour chacun, l'option `~one` pour indiquer qu'il doit être utilisé une et une seule fois et l'option `~sym` pour indiquer qu'il peut être orienté arbitrairement :

```
tiles pentaminos =
  [ L ~one ~sym, T ~one ~sym, V ~one ~sym, N ~one ~sym,
    Z ~one ~sym, F ~one ~sym, X ~one ~sym, W ~one ~sym,
    P ~one ~sym, I ~one ~sym, Y ~one ~sym, U ~one ~sym ]
```

Nommer cette liste permettra notamment de réutiliser facilement les pentaminos dans d'autres problèmes de pavage. (On notera la présence d'une commande `include` dans la grammaire.) Le motif à paver peut être dessiné en ASCII ou encore défini en effaçant les quatre points centraux d'un carré 8×8 :

```
pattern scott_board =
  set set set set (constant 8x8 true) 3x3 false 3x4 false 4x3 false 4x4 false
```

Il ne reste plus qu'à définir le problème de Scott et à dénombrer ses solutions, par exemple avec l'algorithme DLX :

```
problem scott_problem = scott_board pentaminos
count dlx scott_problem
```

On trouve 520 solutions en 5,2 secondes, avec une matrice de 1400 lignes et 72 colonnes, toutes primaires. Plus subtilement, on peut éliminer les symétries du problème en ne considérant qu'une orientation possible pour l'une des pièces n'ayant aucune symétrie (par exemple la pièce  dite F). Il suffit pour cela de lui retirer l'option `~sym` dans la définition de la liste `pentaminos` ci-dessus. On obtient alors 65 solutions uniques, en 2,6 secondes.

⟨*decl*⟩ ::= `pattern` ⟨*ident*⟩ `=` ⟨*expr*⟩
| `problem` ⟨*ident*⟩ `=` ⟨*tiles*⟩
| `tiles` ⟨*ident*⟩ `=` `[` tile$^{\star}_{;}$ `]`
| `print` ⟨*ident*⟩
| `solve` ⟨*algo*⟩ ⟨*ident*⟩ (`ascii` | `svg` ⟨*string*⟩)
| `count` ⟨*algo*⟩ ⟨*ident*⟩
| `debug` (`on` | `off`)
| `timing` (`on` | `off`)
| `include` ⟨*string*⟩

⟨*algo*⟩ ::= `dlx` | `zdd`

⟨*tiles*⟩ ::= ⟨*ident*⟩ | `[` tile$^{\star}_{;}$ `]`

⟨*tile*⟩ ::= ⟨*expr*⟩ ⟨*option*⟩*

⟨*expr*⟩ ::= ⟨*ident*⟩
| `{` ⟨*ascii-art*⟩ `}`
| `constant` ⟨*xy*⟩ ⟨*bool*⟩
| ⟨*expr*⟩ (`&&` | `||` | `^` | `-`) ⟨*expr*⟩
| `set` ⟨*expr*⟩ ⟨*xy*⟩ ⟨*bool*⟩
| `crop` ⟨*xy*⟩ ⟨*xy*⟩ ⟨*expr*⟩
| `shift` ⟨*expr*⟩ ⟨*xy*⟩
| `resize` ⟨*expr*⟩ ⟨*xy*⟩
| `apply` ⟨*isometry*⟩ ⟨*expr*⟩
| `(` ⟨*expr*⟩ `)`

⟨*xy*⟩ ::= ⟨*integer*⟩ `x` ⟨*integer*⟩

⟨*option*⟩ ::= `~one` | `~maybe` | `~sym` | `~rot`

⟨*isometry*⟩ ::= `Id` | `Rot90` | `Rot180` | `Rot270`
| `VertRefl` | `HorizRefl` | `Diag1Refl` | `Diag2Refl`

Figure 3: Grammaire des problèmes de pavage.

6. Conclusion et perspectives

Nous avons présenté `combine`, une bibliothèque OCaml pour la combinatoire. Elle combine — c'est le cas de le dire — l'algorithme DLX et la structure de ZDD autour du problème unique de la couverture exacte, et réduit le problème du pavage rectangulaire en deux dimensions vers ce dernier.

Ce n'est qu'un début. Il est évident que cette bibliothèque pourrait fournir de nombreuses autres réductions vers le problème EMC, à commencer par d'autres types de pavage, tels que des pavages en trois dimensions ou non rectangulaires par exemple.

Concernant le code déjà existant, une amélioration substantielle du module de pavage consisterait à prendre en compte les symétries du problème pour réduire l'espace de recherche. Dans le problème de Scott décrit plus haut page 9, il était facile de prendre en compte les symétries du problème car d'une part il existait une pièce ne possédant aucune symétrie et d'autre part le motif à paver possédait, lui, toutes les symétries. L'espace de recherche était donc très simplement divisé par huit. De manière générale, le problème est plus complexe : on ne dispose pas nécessairement d'une pièce sans aucune symétrie, elle n'est pas nécessairement utilisée une fois et une seule, le motif à paver peut posséder certaines symétries mais pas toutes, etc.

L'interface actuelle du module `Dlx` est relativement simpliste, car elle n'accepte en entrée qu'une matrice de booléens. Or une matrice EMC est le plus souvent creuse. Par exemple, le pavage par des dominos conduit à des lignes ne contenant que deux 1. Il serait donc souhaitable que le module `Dlx` accepte également une description de matrice creuse ou, de façon équivalente, propose une construction incrémentale de la matrice.

Dans le module `Zdd`, un certain nombre de constantes ont été choisies de manière arbitraire, notamment les tailles initiales des différentes tables de hachage, et l'utilisateur n'a pas de contrôle sur ces valeurs. De même, chaque opération sur les ZDD pourrait, au choix, utiliser une table globale, partagée entre tous les appels à cette opération, ou bien au contraire une table locale à chaque appel. Pour l'instant, le choix de tables globales a été fait mais il serait souhaitable que l'utilisateur puisse choisir au cas par cas. Toujours dans ce module, les tables de mémoïzation des différentes opérations n'utilisent pas de pointeurs faibles (au contraire de la table de *hash-consing*). Par conséquent, elles maintiennent en vie des ZDD qui pourraient être récupérés par le GC. Pour y remédier, il faut utiliser des *éphémérons* [4]. Cette technologie est déjà à l'œuvre en OCaml dans l'outil Why3 [2] et pourrait donc être mise en place facilement dans `combine`.

Enfin, une perspective intéressante consiste à réduire le problème EMC vers le problème SAT, pour se servir ensuite d'un solveur SAT existant, à supposer qu'il sache donner une solution ou dénombrer les solutions. Cette idée a été abordée sur la toute fin du stage, en étendant le module `Emc` d'une sortie au format DIMACS. Il reste cependant à mener des expériences avec des solveurs SAT pour comparer les performances avec DLX et ZDD.

Remerciements. Les auteurs remercient chaleureusement Claude Marché pour avoir relu une première version de cet article et pour avoir suggéré de nombreuses améliorations.

Bibliographie

[1] Romain Bardou, Jean-Christophe Filliâtre, Johannes Kanig, and Stéphane Lescuyer. Faire bonne figure avec Mlpost. In *Vingtièmes Journées Francophones des Langages Applicatifs*, Saint-Quentin sur Isère, January 2009. INRIA.

[2] Francois Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 platform, version 0.73*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.73 edition, July 2012.

[3] Sylvain Conchon and Jean-Christophe Filliâtre. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, September 2006.

[4] Barry Hayes. Ephemerons: a new finalization mechanism. *SIGPLAN Not.*, 32(10):176–183, October 1997.

[5] Shin ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *DAC*, pages 272–277, 1993.

[6] Donald E. Knuth. Dancing links, 2000. `http://arxiv.org/abs/cs/0011047`.

[7] Donald E. Knuth. *The Art of Computer Programming, volume 4A: Combinatorial Algorithms, Part 1.* Addison-Wesley Professional, 1st edition, 2011.

[8] Dana S. Scott. Programming a combinatorial puzzle. Technical Report 1, Department of Electrical Engineering, Princeton University, 1958.

A. Pseudo-code de l'algorithme DLX (liens dansants)

L'algorithme DLX est réalisé par une fonction récursive *search*. Elle prend en argument un indice k qui permet de remplir un tableau global S avec la solution. Initialement, on appelle *search*(0). Le nœud h est l'entête de la liste des (entêtes des) colonnes.

search(k) ≡
  si h`.right ==` h on a trouvé une solution
  sinon, choisir une colonne c
  cover(c)
  pour chaque r dans c`.down`, c`.down.down`, ...
    $S_k \leftarrow r$ g
    pour chaque j dans r`.right`, r`.right.right`, ...
      cover(j)
    search($k+1$)
    pour chaque j dans r`.left`, r`.left.left`, ...
      uncover(j)
  uncover(c)

La fonction *cover* exprime le fait que la colonne c est maintenant couverte.

cover(c) ≡
  c`.right.left` $\leftarrow$ c`.left`
  c`.left.right` $\leftarrow$ c`.right`
  pour chaque i dans c`.down`, c`.down.down`, ...
    pour chaque j dans i`.right`, i`.right.right`, ...
      j`.down.up` $\leftarrow$ j`.up`
      j`.up.down` $\leftarrow$ j`.down`

La fonction *uncover* annule l'effet de la fonction *cover*. On note que les opérations y sont effectuées dans l'ordre inverse de la fonction *cover*.

uncover(c) ≡
  pour chaque i dans c`.up`, c`.up.up`, ...
    pour chaque j dans i`.left`, i`.left.left`, ...
      j`.down.up` $\leftarrow$ j
      j`.up.down` $\leftarrow$ j
  c`.right.left` $\leftarrow$ c
  c`.left.right` $\leftarrow$ c