Partie 1 — Encodage de problèmes vers SAT

Nicolas Margulies nicolas.margulies@lmf.cnrs.fr
Théo Vignon theo.vignon@lmf.cnrs.fr

Dans cette première partie du projet, nous allons automatiser la résolution de plusieurs problèmes logiques. Pour ce faire, nous allons tous les réduire à un même type de problème, la satisfaisabilité d'une formule booléenne (ou juste SAT). Ensuite, nous résoudrons ces instances de SAT toutes de la même façon, c'est-à-dire en appelant `minisat` un solveur déjà existant. Plus précisément, chaque instance `I` d'un problème devra être encodée (en temps polynomial) en une instance `S` de SAT, telle que `S` est satisfiable ssi `I` a une solution. De plus, vous devrez extraire (encore une fois en temps polynomial) une solution de `I` à partir d'une affection des variables satisfaisant `S`. Toutes les solutions de `I` n'ont pas à être obtenues de cette façon.

Vous trouverez la base de code ici[1]. Le module OCaml fourni `Dimacs` vous aidera pour la gestion des formules booléennes, tands que le module `Hex` vous aidera pour le problème des pingouins. La documentation de ces modules peut être générée dans le dossier `html/` à l'aide de la commande `make doc`. Pour compiler et tester le projet, vous aurez besoin d'ocaml, de minisat, et des paquets opam `cairo2` (pour l'affichage des pavages uniquement) et `camlp-streams`.

Exercice 1 — Carré latin

Le fichier `src/latin.ml` montre comment encoder le carré latin en SAT (l'entrée `N` est la taille du carré). On peut le tester avec par exemple la commande `make N=10 test_latin`. Cependant, cet encodage peut être amélioré: essayez de modifier le fichier, en ajoutant des clauses afin d'aider le solveur.

1	2	3	4	5
2	3	4	5	1
3	4	5	1	2
4	5	1	2	3
5	1	2	3	4

[1] L'archive contient aussi les fichers pour le projet SAT (Partie 4 du cours), pour le moment vous pouvez les ignorer.

Exercice 2 — Carré gréco-latin

Adaptez `src/latin.ml` en `src/greek.ml` pour encoder le problème du carré gréco-latin.

Aα	Bγ	Cδ	Dβ
Bβ	Aδ	Dγ	Cα
Cγ	Dα	Aβ	Bδ
Dδ	Cβ	Bα	Aγ

Exercice 3 — Sudoku

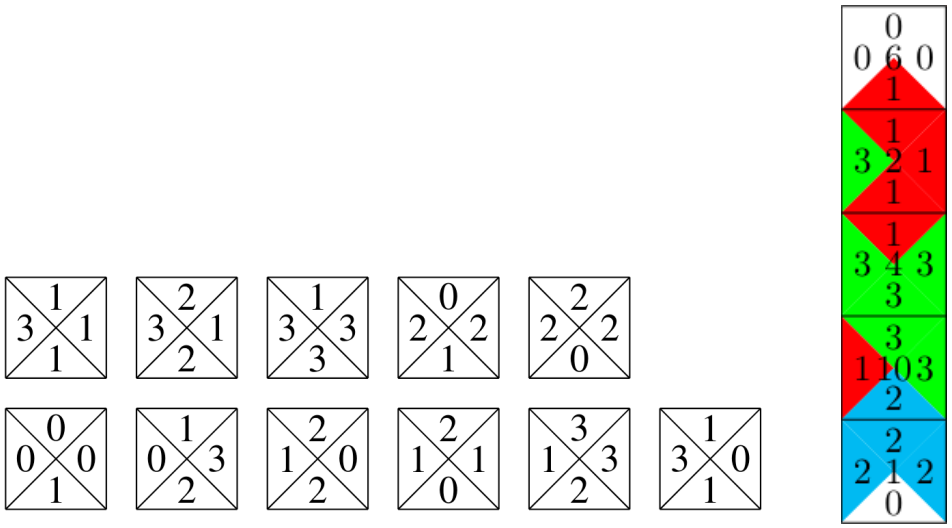
Dans le fichier `src/sudoku.ml`, encodez un solveur de sudoku. Une grille sera représentée par une suite de 81 chiffres entre 0 et 9 (lus de gauche à droite et de bas en haut), avec le 0 signifiant un emplacement vide. Par exemple, l'encodage de la grille ci-dessous est : `530070000600195000098000060800060003400803001700020006060000280000419005000080079`. Le fichier `src/sudoku.ml` gèrera l'encodage entre ce format et SAT, tandis que `src/pp_sudoku.ml` transformera les suites de chiffres en leur représentation habituelle.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Pour ce qui est du test, la commande `make GRID=... test_sudoku` appellera votre programme sur la grille donnée et affichera joliment le résultat (par défaut c'est encore celle ci-dessus), tandis que `make tests_sudoku` lira le fichier `problems/sudoku.csv` et testera votre programme sur tous les sudoku contenus dans le csv (sans affichage joli, au vu du nombre de tests).

Exercice 4 —— Pavage de Wang

Jeandel et Rao ont trouvé un ensemble de 11 tuiles de Wang qui est apériodique et minimal pour cette propriété. Les tuiles en question sont représentées sur l'image de gauche. Sur la droite, on a un exemple d'un schéma vertical de 5 tuiles (ignorez les nombres au centre des tuiles sur l'image).



Créez le binaire `jr`, qui encode le problème de pavage avec cet ensemble de 11 tuiles sur un carré de dimension $N \times N$, avec une contrainte: le motif de cinq tuiles ci-dessus doit se trouver au milieu du pavage, c'est-à-dire sur la colonne $\frac{N}{2}$ et des lignes $\frac{N}{2} - 1$ à $\frac{N}{2} + 2$. Un pavage devrait exister pour $N = 70$, mais pas pour $N = 80$.

Le fichier `src/jr_cairo.ml` permet de convertir vos pavages en SVG afin de les visualiser. Il contient en commentaire des instructions à propos de comment représenter les pavages en ascii en utilisant A, B, C, ..., K pour désigner les tuiles dans l'ordre (de droite à gauche et de haut en bas dans l'image ci-dessus). Le Makefile contient une cible `make N=70 test_jr` pour tester tout cela.

Exercice 5 — Pingouins

Dans le jeu Hey, That's My Fish! plusieurs pingouins se déplacent sur des icebergs hexagonaux afin de manger autant de poissons que possible. Nous considérons un problème simplifié où un pingouin doit manger un certain nombre de poissons, avec exactement un poisson par iceberg. Plusieurs instances de ce problème sont disponibles dans le dossier `problems/`, organisés en sous-dossiers en indiquant combien d'icebergs peuvent être non visités à la fin (ce nombre est optimal, mais on ne demande pas à votre encodage de le vérifier).

Écrivez un encodage SAT du problème dans `src/pingouins.ml` en utilisant le module OCaml `Hex` fourni pour parser les fichiers d'entrée et gérer les déplacements sur la grille hexagonale. Votre binaire `pingouins` utilisera la variable d'environnement `PENALTY` pour savoir combien d'icebergs peuvent être laissé non visités. Ceci est nécessaire pour pouvoir utiliser les cibles make fournies e.g. `make PROBLEM=problems/1/flake1 PENALTY=1 test_pingouins` pour tester un problème donné et `make PENALTY=1 tests_pingouins` pour tester sur tous les problèmes de `problems/1/`.

Rendu 1 — Mardi 11 février 23h59

Envoyer vos fichers sous la forme d'une archive `nom_prénom.tar.gz` contenant l'intégralité de vos fichers ainsi qu'un readme contenant :

- La liste des exercices rendus, il faut obligatoirement rendre les exercices 2,3 et au choix l'un des exercices 4 et 5. Les exercices rendu en plus pourront compté en bonus.
- Si vous avez rencontrer des problemes, n'hesitez pas a les mentionner.