DM3
Conception et utilisation d'un SAT-solver

MP2I Lycée Pierre de Fermat

Présentation

L'objectif de ce DM est d'implémenter un SAT-solver utilisant l'algorithme de Quine, et de l'utiliser pour résoudre quelques problèmes modélisés sous la forme de formules logiques.

Le DM est divisé en deux parties:

- La conception du SAT-solver, en OCaml. Dans cette partie, vous écrirez un programme compilable permettant de lire un fichier texte contenant une formule, sous un format spécifié plus bas, déterminant si cette formule est satisfiable, et affichant une valuation la satisfiant le cas échéant.
- L'utilisation du SAT-solver, en C. Dans cette partie, vous modéliserez deux problèmes, et vous écrirez pour chacun un programme C permettant de générer la formule logique le modélisant, au format attendu par le SAT-solver.

Modalités Ce DM est à déposer sur Cahier de Prépa au plus tard le **lundi 19 mai** avant le début du cours. **Il doit être fait en binôme ou en trinôme.** Il est **impératif** que chaque membre comprenne l'intégralité du code rendu. En particulier, vous vous assurerez que chaque membre travaille sur les deux parties.

Bien que ce DM prenne la forme d'une suite de questions guidées, votre rendu ne sera pas noté directement sur ces questions, et il ne vous est pas demandé de rédiger un document réponse question par question. A la place, vous rendrez un court rapport résumant les points clés de votre projet. Certaines questions vous demanderont explicitement de noter certains points dans votre rapport afin de vous guider, mais libre à vous d'y ajouter tout ce qui vous semble pertinent. Le rendu sera donc évalué dans sa globalité: qualité du code, des commentaires et des tests, clarté du rapport, etc... Plusieurs questions bonus sont éparpillées dans le sujet: il est conseillé de les sauter au premier abord et d'y revenir une fois que vous avez bien avancé. Ces questions sont généralement plus ouvertes et ont pour but de rendre votre programme plus efficace. Si vous les traitez, vous expliquerez votre démarche dans votre rapport.
Pour résumer, vous devez rendre:

- une archive contenant le code source, les tests, ainsi qu'un document README.txt expliquant comment compiler et utiliser vos programmes;
- un court rapport au format PDF / Word / Markdown résumant les étapes principales de votre projet, et répondant aux éventuelles questions bonus que vous avez souhaité traiter.

Format des formules

Les formules lues par le SAT-solver devront suivre le format suivant:

- Le ET s'écrit &, le OU s'écrit "|", le NON s'écrit "∼";
- L'implication s'écrit ">" et l'équivalence "=";
- La formule vraie s'écrit "T", la formule fausse s'écrit "F";
- Les parenthèses et les espaces sont autorisés, pas les retours à la ligne;
- Tout le reste sera interprété comme une variable (attention, les variables ne peuvent pas s'appeler "T" ou "F").

Par exemple, la formule $A \vee (B \wedge \neg C)$ s'écrira "`a | (b & ∼c)`".

La priorité des opérateurs est comme suit: $\wedge$ puis $\vee$ puis $\leftrightarrow$ puis $\rightarrow$. Par exemple, si l'on écrit "`a & b | c`", cela représente la formule $(A \wedge B) \vee C$. Dans le doute, utilisez des parenthèses pour supprimer les ambiguïtés.

Les noms de variable peuvent avoir plusieurs lettres et même contenir des chiffres ou autres symboles (hors ceux des opérateurs), par exemple:

```
(X_1_2 | ∼X_3_3) & (X_2_3 | ∼X_1_3)
```

Partie I: Conception du SAT-solver

L'objectif de cette partie est de programmer en OCaml un logiciel `./satsolver` tel que si l'on dispose d'un fichier `formule.txt` contenant une formule propositionnelle sous le format présenté précédemment, la commande `./satsolver formule.txt` tente de satisfaire cette formule, avec un affichage de la forme suivante:

```
fredfrigo@ordi:~/DM4/satsolver$ ./satsolver bla.txt
La formule est satisfiable en assignant 1 aux variables suivantes et 0 aux autres:
X_1
X_2
Y_5
T_45
```

On pourra aussi lancer une batterie de tests en lançant l'exécutable avec `test` en argument:

```
fredfrigo@ordi:~/DM4/satsolver$ ./satsolver test
Vérification des tests...
Tous les tests ont réussi !
```

Compilation et arguments

Créez un dossier `/satsolver`, dans lequel vous mettrez tout le code de cette partie.

Q1. Créez un fichier `satsolver.ml`, et écrivez-y le code suivant:

```
let test () =
  assert (1 = 1);
  print_string "Tous les tests ont réussi\n"

let main () =
  test ();
  print_int (Array.length Sys.argv); print_string "\n\n";
  print_string Sys.argv.(0); print_string "\n\n";
  print_string Sys.argv.(1); print_string "\n\n"

let _ = main () (* exécution de la fonction main *)
```

Compilez avec `ocamlc satsolver.ml -o satsolver` puis lancez la commande `./satsolver coucou`. Que remarquez-vous ? Et pour `./satsolver` sans argument ?

`Sys.argv: string array` est donc un tableau contenant les différents arguments utilisés au lancement du programme, exactement comme en C.

Q2. Modifier le code de `satsolver.ml` pour que le programme:

- affiche un message d'erreur et s'arrête s'il est lancé sans arguments;
- lance une fonction `test: unit -> unit` vide (pour l'instant) si le premier argument du programme est "`test`";
- affiche son premier argument sinon.

Q3. En utilisant la fonction suivante, modifier le programme pour qu'il affiche le **contenu** du fichier donné en argument, du moment que ce dernier ne fait qu'une seule ligne (i.e. qu'il ne contient pas de `\n`):

```
1  (* Renvoie le contenu du fichier fn sous forme de string.
2     Le fichier ne doit contenir qu'une seule ligne *)
3  let read_file (fn: string) : string =
4    let ic = open_in fn in
5    let res = input_line ic in
6    close_in ic; res
```

Vous disposez maintenant du squelette basique du programme. Dans la suite, pour chaque nouvelle fonction implémentée, vous viendrez écrire des tests dans la fonction `test`.

Manipulation de formules logiques

Pour manipuler les formules, on utilise le type suivant:

```
1  type formule =
2    | Var of string | Top | Bot
3    | And of formule * formule | Or of formule * formule | Not of formule
```

L'archive du DM contient un fichier de départ `base_satsolver.ml` dont le code est à copier/-coller au début de votre fichier `satsolver.ml`. Il contient la définition du type ci-dessus ainsi que deux fonctions permettant de lire des formules:

```
1  (* Renvoie une formule construite à partir de la chaîne s.
2     Lève une exception Erreur_syntaxe si la chaîne ne représente pas une formule valide. *)
3  let parse (s: string) : formule = ...
4
5  (* Renvoie une formule construire à partir du contenu du fichier fn.
6     Lève une exception Erreur_syntaxe si le contenu du fichier n'est pas une formule valide.
7     Lève une exception Sys_error si le nom du fichier n'est pas valide. *)
8  let from_file (fn: string) : formule = ...
9
10 let test_parse () = ... (* fonction de tests pour parse *)
```

Vous n'avez pas besoin de comprendre en détail comment `parse` et `from_file` fonctionnent, vous devez seulement comprendre comment les utiliser.

Q4. Compléter la fonction `test_parse` et ajoutez la à la fonction `test`. N'oubliez pas de tester aussi les cas d'erreurs.

Q5. Créer un sous-dossier `/tests`, et quelques fichiers dedans pour tester la fonction `from_file`.

Dans la suite, pour vos tests, vous pourrez utiliser les fonctions `parse` et `from_file` afin de lancer des tests sur des formules sous forme de texte:

```
(* compte_ops f renvoie le nombre d'opérateurs utilisés dans f *)
let compte_ops (f: formule) : int = ...

let test_compte_ops () =
  assert (compte_ops (parse "x | (y & ~z)") = 3);
  assert (compte_ops (parse "~(x | (x & ~z) | y)") = 5);
  ...
```

Q6. Écrire une fonction comptant le nombre de $\wedge, \vee$ et $\neg$ dans une formule.

Une première étape clé est de pouvoir récupérer la liste des variables d'une formule, sans doublons. Pour cela, on implémente une structure d'ensemble en utilisant des listes. Un ensemble d'éléments de type `'a` sera représenté par une `'a list` **croissante**, ce qui permettra de faire des unions simplement.

Q7. Écrivez une fonction qui vérifie si une liste est triée et sans doublons (i.e. strictement croissante).

Q8. Écrivez une fonction `union: 'a list -> 'a list -> 'a list` telle que si `l1` et `l2` sont des listes triées sans doublons, alors `union l1 l2` est une liste triée sans doublons qui contient les éléments de `l1` et ceux de `l2`

Q9. Écrire une fonction calculant la liste des variables d'une formule, sans doublons.

Q10. (Question pour le rapport) quelle est la complexité asymptotique pire cas de la fonction précédente ? Quelles autres structures concrète implémentant les ensembles pourrait-on utiliser pour améliorer sa complexité ?

Q11. (Bonus) mettre en place une des solutions proposées à la question précédente.

Valuations et SAT-solver naïf

On représentera les valuations par le type suivant:

```
1 type valuation = (string*bool) list
```

Une valuation est donc une liste de couples (x, b) avec x un nom de variable et $b \in \mathbb{B}$. Un tel couple signifiera que la valuation assigne b à x. On rappelle l'existence de la fonction `List.assoc`: allez lire sa documentation pour voir son utilité dans ce contexte.

Q12. Écrivez une fonction permettant de calculer l'interprétation d'une formule dans une valuation donnée.

Écrivons maintenant notre premier SAT-solver. Son principe est le suivant: énumérer toutes les valuations possibles et les tester une à une. On énumèrera les valuations en comptant en binaire, comme lorsque l'on écrit une table de vérité. Par exemple, si la formule contient X, Y et Z, on testera toutes les valuations dans l'ordre suivant:

$$\begin{array}{l}
[X \mapsto 0;\quad Y \mapsto 0;\quad Z \mapsto 0] \\
[X \mapsto 1;\quad Y \mapsto 0;\quad Z \mapsto 0] \\
[X \mapsto 0;\quad Y \mapsto 1;\quad Z \mapsto 0] \\
[X \mapsto 1;\quad Y \mapsto 1;\quad Z \mapsto 0] \\
[X \mapsto 0;\quad Y \mapsto 0;\quad Z \mapsto 1] \\
[X \mapsto 1;\quad Y \mapsto 0;\quad Z \mapsto 1] \\
[X \mapsto 0;\quad Y \mapsto 1;\quad Z \mapsto 1] \\
[X \mapsto 1;\quad Y \mapsto 1;\quad Z \mapsto 1]
\end{array}$$

On mettra les bits de poids faibles à gauche afin de faciliter la manipulation en OCaml. Pour passer d'une valuation à la suivante, on fera donc une addition avec 1 en binaire.

Q13. Question préliminaire: écrivez une fonction `add_one: bool list -> bool_list` qui prend en entrée une liste de booléens représentant un nombre en binaire x, et renvoie la liste représentant $x + 1$. Faites des exemples à la main pour voir comment la retenue se propage et pour en extraire un comportement récursif.

Q14. Écrivez une fonction `valuation_next: valuation -> valuation option` qui renvoie la valuation suivante selon l'ordre décrit plus haut, et renvoie None si l'on en est à la valuation maximale (toutes les variables valent true)

Q15. Écrivez une fonction `valuation_init: string list -> valuation` qui renvoie la valuation constante égale à false sur la liste des variables donnée en argument.

Les deux SAT-solvers que nous allons implémenter renvoient une valuation satisfaisant leur formule d'entrée, et doivent renvoyer une valeur spéciale si aucune valuation ne satisfait la formule. Les types option semblent alors parfaitement convenir:

```
1 type sat_result = valuation option
```

Ainsi, les SAT-solvers renverront `None` pour une formule non-satisfiable, et `Some sigma` si `sigma` est une valuation satisfaisant la formule.

Q16. Écrivez une fonction `satsolver_naif: formule -> sat_result` implémentant l'algorithme naïf de résolution de SAT.

Avant de passer à la suite du TP, vérifiez que votre algorithme fonctionne: testez le en long et en large, sur des formules diverses, aussi bien satisfiables qu'insatisfiables.

Algorithme de Quine

On rappelle le principe de l'algorithme de Quine: pour déterminer si une formule φ est satisfiable, on remplace une variable X quelconque de φ par $\top$, on simplifie la formule obtenue, et on teste récursivement si elle est satisfiable. Si c'est le cas, alors φ est aussi satisfiable, en mettant X à 1, et sinon, on réessaie en remplaçant X par $\bot$. Le pseudo-code de l'algorithme de Quine est donc:

Algorithme 1 : Algorithme de Quine

Entrée(s) : φ une formule propositionnelle
Sortie(s) : Une valuation σ satisfaisant φ, ou IMPOSSIBLE si ce n'est pas possible

```
si φ = ⊤ alors
    retourner ∅ la valuation vide
si φ = ⊥ alors
    retourner IMPOSSIBLE
X ← une variable quelconque de φ ;
φ_⊤ ← φ[⊤/X];
Simplifier φ_⊤;
Tester récursivement si φ_⊤ est satisfiable.
si φ_⊤ est satisfiable par une valuation σ alors
    retourner σ' = σ[X ↦ 1]
sinon
    φ_⊥ ← φ[⊥/X];
    Simplifier φ_⊥;
    Tester récursivement si φ_⊥ est satisfiable.
    si φ_⊥ est satisfiable par une valuation σ alors
        retourner σ' = σ[X ↦ 0]
    sinon
        retourner IMPOSSIBLE
```

Les simplifications utilisent la fonction **simpl** définie inductivement par les 9 règles suivantes:

1. Pour φ une formule, $\mathbf{simpl}(\top \wedge \varphi) = \mathbf{simpl}(\varphi \wedge \top) = \varphi$
2. Pour φ une formule, $\mathbf{simpl}(\bot \wedge \varphi) = \mathbf{simpl}(\varphi \wedge \bot) = \bot$
3. Pour φ une formule, $\mathbf{simpl}(\bot \vee \varphi) = \mathbf{simpl}(\varphi \vee \bot) = \varphi$
4. Pour φ une formule, $\mathbf{simpl}(\top \vee \varphi) = \mathbf{simpl}(\varphi \vee \top) = \top$
5. Pour φ une formule, $\mathbf{simpl}(\neg\neg\varphi) = \varphi$
6. $\mathbf{simpl}(\neg\top) = \bot$
7. $\mathbf{simpl}(\neg\bot) = \top$
8. Pour φ et ψ deux formules ne rentrant pas dans les cas précédents:
   - $\mathbf{simpl}(\varphi \wedge \psi) = (\mathbf{simpl}(\varphi)) \wedge (\mathbf{simpl}(\psi))$
   - $\mathbf{simpl}(\varphi \vee \psi) = (\mathbf{simpl}(\varphi)) \vee (\mathbf{simpl}(\psi))$
   - $\mathbf{simpl}(\neg\varphi) = \neg(\mathbf{simpl}(\varphi))$
9. $\mathbf{simpl}(\varphi) = \varphi$ dans tous les autres cas.

Pour simplifier une formule ϕ, on lui applique **simpl** jusqu'à atteindre un point fixe, i.e. jusqu'à ne plus pouvoir simplifier.

Q17. Écrivez une fonction `simpl_step : formule -> formule * bool` appliquant une étape de simplification à une formule, et renvoyant un booléen indiquant si une simplification a été effectuée, i.e. si une règle autre que la 8 a été appliquée. Cette fonction doit être récursive pour pouvoir appliquer la règle numéro 8.

Q18. Écrivez une fonction `simpl_full : formule -> formule` appliquant la fonction précédente sur son entrée jusqu'à atteindre un point fixe.

Q19. (Question pour le rapport) Donner une famille de formules pour lesquelles la fonction précédente s'exécute en $\Theta(n^2)$, avec n la taille de la formule.

Q20. (Bonus) Écrire une version de `simpl_full` de complexité linéaire en la taille de la formule.

Q21. Écrire une fonction `subst: formule -> string -> formule -> formule` telle que `subst f x g` renvoie la formule f dans laquelle toutes les occurrences de x ont été remplacées par g.

Q22. Implémentez l'algorithme de Quine en une fonction `quine: formule -> sat_result`. Vous pourrez utiliser une fonction auxiliaire prenant également en entrée la liste des variables restantes à tester.

L'algorithme de Quine implémenté, il ne reste qu'à compléter le programme pour qu'il réponde au cahier des charges (voir début page 3).

Q23. Écrire une fonction `print_true: valuation -> unit` prenant en entrée une valuation et affichant chaque variable mise à vrai, sur une ligne chacune.

Q24. Compléter votre programme pour qu'il réponde au cahier des charges.

Q25. (Question pour le rapport) Proposer une stratégie pour choisir la variable à remplacer à chaque étape de l'algorithme, et pour choisir s'il vaut mieux essayer $\top$ ou $\bot$ en premier. Implémenter cette stratégie, et discuter de son efficacité.

Q26. (Bonus++) En vous aidant de la dernière partie du document de cours, implémenter une version de l'algorithme de Quine spécialisée aux formules sous FNC, et permettre à votre programme de l'utiliser s'il détecte que la formule d'entrée est sous cette forme. Discuter de l'efficacité expérimentale de cet algorithme.

Partie II: Résolution de problèmes

Dans la suite de ce document, vous trouverez différents problèmes, de difficulté variable, que vous devez résoudre en utilisant le SAT-solver. Pour chaque problème, vous devez donc procéder aux étapes suivantes:

1. Modéliser une instance du problème par des variables propositionnelles et une formule exprimant les diverses contraintes, de telle sorte que la formule est satisfiable si et seulement l'instance du problème initial admet une solution;
2. Écrire un programme C qui écrit dans un fichier la formule correspondant à une instance donnée;
3. Lancer votre SAT-solver sur le fichier généré, et retransformer la solution en une solution du problème initial.

Vous devez documenter la première étape dans votre rapport de manière claire et précise, en expliquant les variables propositionnelles choisies, et les formules utilisées pour exprimer les différentes contraintes du problème.

Le premier problème est le même pour toute le monde, il vous fera écrire quelques fonctions utilitaires. Une fois que vous l'avez résolu à l'aide de votre super SAT-solver, choisissez un des problèmes proposés en dessous, et résolvez-le aussi. Vous pouvez aussi proposer **votre propre problème**. Dans ce cas, envoyez-moi un mail pour que je le valide !

Q27. Créez un dossier `/problemes/`. Vous y mettrez tout votre code pour cette partie.

Prise en main et problème des N dames

Q28. Créer deux fichiers `utils.h` et `utils.c`, dans lequel vous mettrez des fonctions utilitaires assez générales, réutilisables dans le reste du code.

Q29. Écrire une fonction `char* au_moins_une(char** l, int n)` qui prend en entrée une liste l de n formules supposées atomiques (i.e. n'ayant pas d'opérateur binaire en dehors des parenthèses), et qui fabrique la disjonction de ces formules, i.e. qui fabrique une formule exprimant "au moins une des formules de l est vraie". Par exemple:

```
1 char* forms[3] = {"(x & ~y)", "y", "z"};
2 char* phi = au_moins_une(forms);
3 // phi pointe vers la chaîne "((x & ~y) | y | z)"
```

On fera attention à mettre des parenthèses autour du résultat afin de renvoyer une formule étant aussi atomique.

Q30. Écrire une fonction `char* au_plus_une(char** l, int n)` qui prend en entrée une liste l de n formules atomiques, et qui fabrique une formule atomique exprimant qu'au plus une des formules de l est vraie.

Q31. (Question pour le rapport) la formule générée par la fonction précédente est-elle sous FNC ? Quelle est sa taille par rapport au nombre de formules en entrée ?

Nous allons utiliser ces fonctions pour résoudre le problème des N dames, qui consiste à considérer un damier d'échecs de N lignes et N colonnes sur lequel on veut placer N dames de telle sorte qu'aucune dame ne puisse en attaquer une autre. Selon les règles des échecs, il faut donc que chaque colonne, ligne, et diagonale, contienne au plus une dame.

On modélise la situation avec N^2 variables propositionnelles $(X_{i,j})_{i,j\in[\![0,N-1]\!]}$, de telle sorte que $X_{i,j}$ représente l'affirmation "la case (i,j) contient une dame". On notera ces variables "`X_i_j`" dans les formules générées. On veut construire une grande formule exprimant les contraintes du problème, à savoir:

- Sur chaque ligne, il y a exactement une reine;
- Sur chaque colonne, il y a au plus une reine;
- Sur chaque diagonale, il y a au plus une reine.

Q32. Créer un fichier `n_dames.c`, dans lequel vous écrirez le code concernant le problème des N dames.

Q33. Écrire une fonction `char* variable(int i, int j)` renvoyant la chaîne de caractères "`X_[i]_[j]`". Par exemple, `variable(7, 23))` crée et renvoie la chaîne "`X_7_23`". Vous pourrez vous renseigner sur la fonction `sprintf` permettant d'écrire dans une chaîne de caractère selon le même principe que `printf`.

Q34. Écrire une fonction `char* contrainte_une_ligne(int i, int n)` renvoyant une formule exprimant la contrainte sur la ligne i dans le problème des n dames.

Q35. Écrire une fonction `char* contrainte_toutes_lignes(int n)` renvoyant une formule exprimant la contrainte sur toutes les lignes dans le problème des n dames.

Q36. Écrire des fonctions similaires exprimant les contraintes sur les colonnes et les diagonales.

Q37. Écrire une fonction `void gen_formule_n_dames(int n, char* filename)` qui génère la formule modélisant le problème à n dames, et la stocke dans le fichier `filename`.

Q38. (Question pour le rapport) Donner la taille de la formule générée en fonction de n, sous la forme d'un $\mathcal{O}$.

Q39. Compléter votre programme pour pouvoir le lancer avec un entier n en argument et le faire générer un fichier contenant la formule modélisant le problème des n dames:

```
fredfrigo@ordi:~/DM4/problemes$ ./n_dames 8
fichier 8_dames.txt créé (87932 octets)
```

Q40. Donner une solution pour le problème des 5 dames, puis des 8 dames (cela peut prendre longtemps dépendant de l'efficacité de votre solver). Vérifier qu'il n'existe pas de solution pour le problème des 3 dames.

Les sections suivantes décrivent chacune un problème à modéliser par des formules de logique propositionnelle. Certains sont très simples, et pourront se modéliser assez directement, mais d'autres pourront demander d'introduire des variables intermédiaires, ou de construire des sous-formules complexes. La difficulté du problème choisi n'aura pas d'impact direct sur votre note, choisissez-en un adapté au temps que vous souhaitez investir sur le projet.

A (Facile) Coloriage de carte

On considère la carte des régions de France métropolitaine. On souhaite colorier les régions de cette carte, de telle sorte que deux régions adjacentes ne soient jamais de la même couleur. On veut utiliser le moins de couleurs possible. Est-il possible de procéder au coloriage en 4 couleurs seulement ? Et en 3 ?
Bonus: Qu'en est t-il de la carte des départements ? Du monde ?

Figure 1: Régions de France

B (Pas encore testé, probablement assez difficile) Jeu du calendrier

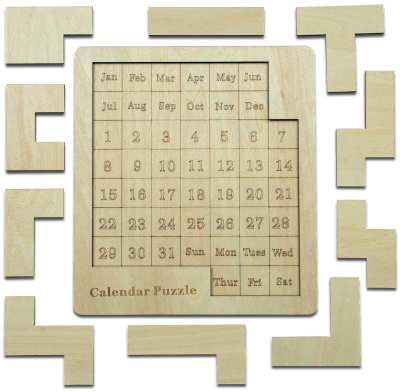

Figure 2: Pièces du jeu

Ce jeu se présente sous la forme d'une grille, ou chaque case est un nom de mois, un numéro de jour, ou un nom de jour de la semaine. 10 pièces en bois peuvent être placées sur la grille afin de recouvrir les différentes cases. Chaque jour, l'objectif du jeu est de placer les pièces de façon à ce que les trois seules cases non recouvertes correspondent à la date du jour. Trouver une solution pour le jour de votre naissance.
Bonus: Vérifier que chaque jour admet une solution. Trouver un jour ayant deux solutions distinctes.

C (Moyen) Problème des cinq maisons

Cinq maisons de couleurs différentes sont alignées le long d'une route. Dans chacune habite une personne de nationalité différente. Chaque personne boit une boisson différente, a un animal domestique différent, et pratique un sport différent:

1. L'Anglais vit dans une maison rouge.
2. Le Suédois a des chiens.
3. Le Danois boit du thé.
4. La maison verte est à gauche de la maison blanche.
5. Le propriétaire de la maison verte boit du café.
6. La personne qui fait du vélo a des oiseaux.
7. Le propriétaire de la maison jaune fait de la danse.
8. La personne qui vit dans la maison du centre boit du lait.
9. Le Norvégien habite la première maison.
10. La personne qui fait de l'escalade vit à côté de celle qui a des chats.
11. La personne qui a un cheval est voisine de celle qui fait de la danse.
12. La personne qui fait du basket boit du Yop.
13. L'Allemand fait du karaté.
14. Le Norvégien vit juste à côté de la maison bleue.
15. Le fan d'escalade a un voisin qui boit de l'eau.

La question est: **à qui appartient le poisson rouge ?**

D (Difficile) Emploi du temps

Les trois professeurs de MPI veulent mettre en place un emploi du temps pour la dernière semaine de cours, afin d'organiser des séances de révisions en demi-groupes. Chaque jour est divisé en trois créneaux: 9h-12h, 12h-15h, 15h-18h. En pratique, les créneaux sont un peu plus court, ce qui permet aux élèves de manger même s'ils ont cours sur les trois créneaux. La classe dispose donc de 5 jours, chaque jour divisé en 3 créneaux, et chaque créneau peut accueillir deux cours simultanés, pour les deux demi-groupes.
En plus des contraintes structurelles évidentes (un prof ne peut pas participer à deux séances sur le même créneau, etc...), l'emploi du temps doit respecter plusieurs règles:

- Aucun élève ne doit finir deux jours d'affilée à 18h;
- Chaque élève a au plus un créneau de maths par jour;
- Le prof de maths doit aller chercher ses enfants à l'école, et doit finir ses cours avant 16h;
- La prof d'informatique ne veut pas commencer avant 10h;
- Il doit y avoir un créneau complètement libre dans la semaine pour le discours annuel du proviseur.

Au total, chaque demi-groupe doit avoir 3 créneaux de maths, 3 créneaux de physique, 3 créneaux d'informatique