1 Préambule

En cours de complexité, vous avez vu que le problème SAT était dans la classe de complexité **NP-Complet** ce qui veut dire qu'aujourd'hui, nous n'avons pas d'algorithme efficace sur le plan théorique pour résoudre ce problème. Cependant, en pratique, il est possible d'inventer des algorithmes qui résolvent des instances de ce problèmes extrêmement rapidement. Le but de ce mini-projet s'articule en trois temps. Dans un premier temps, l'objectif est d'implémenter un algorithme simple, DPLL, afin de résoudre des instances SAT qui ne seraient pas résolubles par l'algorithme naïf qui teste toutes les combinaisons possibles. On va ensuite dans un second temps créer nos propres instances SAT afin d'évaluer les performances de cet algorithme via l'encodage de problèmes connus tel le *sudoku*. Je vous invite à cette occasion de comparer les temps d'exécution entre vos implémentations afin de comparer et voir comment vous pouvez gagner du temps. Dans un troisième temps, je vous propose plusieurs façons de continuer ce projet et ce, notamment afin de gagner en performance.

La conquête de performance peut-être une activité chronophage, et cela peut complexifier énormément votre code. Afin que je puisse évaluer correctement vos programmes, je vous invite donc à respecter les consignes suivantes :

- Vos programmes doivent être corrects ! Cela veut dire qu'avant de rendre votre projet, vérifier bien que la commande `make test` ne renvoie pas une erreure.
- Ne pas modifier les interfaces (fichiers `.mli`) que je vous donne, le fichier `GNUmakefile` ainsi que les options sur du programme lorsqu'il est invoqué via la ligne de commande. Il est cependant possible, pour vos besoins personnels de rajouter des fonctions dans ces fichiers, ou bien de rajouter des options sur la ligne de commande. Mais ne modifiez pas ce que je vous ai donné !
- Lisez-bien les consignes pour les modalités de rendu, sous peine d'être pénalisé si vous ne les respectez pas

Un dernier conseil avant de rentrer dans les détails. Si vous ne cherchez pas à optimiser votre algorithme, le projet que je vous donne est simple mais un peu long. Je vous invite donc à soigner votre rendu. Ceci passe par deux choses :

- code compréhensible (commentaires appropriés, des fonctions courtes et bien nommées, etc... ,
- rapport clair et pas écrit au dernier moment, etc... .

Ceci sera pris en compte dans la notation. La date de rendu pour ce projet n'est pas encore fixée et sera établie durant le TD du lundi 15 avril 2019 et vous serez tenus au courant par mail et sur ma page web.

À titre indicatif, j'ai mis en parenthèse le nombre de lignes de ma solution.

2 DPLL

- le dossier `src` contient le code que vous allez modifier,
- le dossier `sujet` contient le code LaTeX utilisé pour écrire ce document,
- le dossier `rapport` devra contenir votre fichier `rapport.pdf`, idéalement produit à partir du fichier `rapport.tex`,
- le dossier `tests` contient des exemples de formule SAT qui seront utilisés pour tester la correction et l'efficacité de votre programme. Le sous dossier `OK` contient des tests satisfiables, et le sous dossier `KO` contient des tests non satisfiables. Il contient aussi un fichier `run.sh` utilisé par le Makefile,
- le dossier `sudoku` contient un fichier `sudoku.csv` content environ 10000 sudokus ainsi qu'un script `run.sh` utilisé par le Makefile.

Pour cette partie, le seul fichier que vous aurez besoin de modifier est le fichier `src/dpll.ml` où vous devrez écrire votre algorithme. Le parseur (dans `src/parserDimacs.mly`) vous renverra un objet de type `Ast.t`. Si ce type ne vous plait pas, libre à vous d'en changer, cependant cela doit se faire impérativement dans le fichier `src/dpll.ml`.

Les options de la ligne de commande sont spécifiées dans le fichier `src/main.ml`. Pour tester votre programme sur l'ensemble des fichiers situés dans le répertoire `tests`, je vous invite à utiliser la commande `make tests`. Pour votre convenance personnelle, vous êtes autorisées à modifier les fichiers `run.sh` si vous le souhaitez, ils ne seront pas exportés dans l'archive finale.

2.1 Compiler le projet

Après avoir extrait l'archive, vous devriez pouvoir exécuter la commande

```
make
```

pour construire le binaire. La première fois, pour éviter d'avoir à chercher le binaire dans un sous-dossier, vous pouvez exécuter

```
make dpll
```

ce qui va créer un lien symbolique à la racine du projet. Normalement, le projet a besoin de seulement deux dépendances que vous pouvez récupérer via `opam`:

```
opam install dune menhir
```

Si vous n'arrivez toujours pas à compiler le projet, vérifiez bien qu'`opam` est configuré correctement :

```
echo $PATH # vous devriez voir un dossier opam
```

Sinon, il faut lancer la command

```
eval $(opam env)
```

à mettre dans le fichier `~/.bashrc` pour que ce soit permanent.

2.2 Descrption de l'algorithme

Étant donné un ensemble de clauses, on souhaite savoir s'il existe un assignement des variables qui permet de satisfaire l'ensemble de ces clauses. L'algorithme naïf consisterait à tester les 2^n cas possibles où n est le nombre de variables. Cependant, en pratique, le nombre de variables peut exploser et être de lors du million. DPLL propose une critère simple pour repousser au maximum cette explosion combinatoire. L'idée est d'observer les clauses unitaires (de taille 1), c'est à dire celles qui ne contiennent q'un seul litéral. Quand une telle clause existe, la valeure de ce litéral est forcée et peut-être propagée aux autres clauses. Cette étape est appelée en anglais *unit propagation*. À l'issue de cette phase, ou bien on a résolu notre problème, ou bien on se retrouve dans une situation où il faut faire un choix arbitraire (a priori) : choisir un litéral et tester la satisfiabilité de l'ensemble des clauses si elle est vraie ou bien si elle est fausse. À l'issue de ce choix, de nouvelles clauses unitaires peuvent apparaître et donc recommence une phase de *unit propgation* et ainsi de suite. C'est précisément cet algorithme, simple, mais terriblement efficace que vous devez implémenter pour cette première étape. À noter, que pour l'étape d'après, il vous sera nécessaire de retourner un modèle de votre formule, c'est à dire, une liste de littéraux qui satisfont la formule.

Question (80 lignes): Modifier le fichier `src/dpll.ml` afin d'implémenter l'algorithme ci-dessus. Je vous invite dans un premier temps à choisir une variable au hasard lorsque la phase de propagation est terminée.

Vous pouvez vérifier la correction de votre algorithme via la commande `make test`. Si votre algorithme renvoie une réponse incorrect, vous le saurez immédiatement. Il se peut aussi que votre algorithme soit tout bonnement trop lent ou bien boucle. Par défault, un timeout à 30 secondes a été fixé. Vous pouvez le modifier via la commande suivante : `make tests TIMEOUT=20`.

Question : Essayer d'implémenter d'autres heuristiques pour choisir une variable. Vous pourrez trouver des exemples ici : `https://www.cs.ubc.ca/~hutter/EARG.shtml/earg/papers07/lagoudakis01learning.pdf`, mais libre à vous d'utilisez vos propres heuristiques. N'hésitez-pas à commenter dans le rapport les heuristiques qui semblent meilleures.

À titre indicatif, les résultats que j'obtiens sur mon implémentation naïve de DPLLpour une timout à 30 secondes sont présentés en figur 1. À vous faire mieux !

2.3 Encoder des problèmes dans SAT

Comme SAT est **NP-Complet**, il existe de nombreux problèmes qui se réduisent à sat. L'objectif de cette partie est de tester votre algorithme sur des petits problèmes tel que le sudoku. Le sudoku à l'avantage de contenir déjà de nombreux jeux de tests, et c'est donc celui qui va nous occuper pour cette partie.

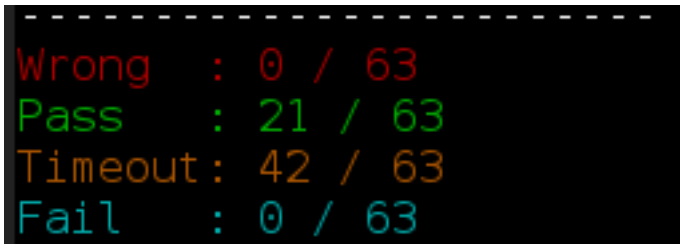

Figure 1: Résultats

Pour les plus curieux, je vous invite à regarder la dernière section si vous êtes intéressé par d'autres problèmes.

L'objectif de cette partie est d'utiliser votre implémentation de DPLL pour résoudre des sudokus. Des exemples de sudoku sont stockés dans le fichier `sudoku/sudoku.csv`. La première colonne de ce fichier contient une grille de sudoku incomplète, tandis que la second contient sa solution.

Question (200 lignes): Compléter la fonction

```
val to_cnf : t -> Ast.Cnf.t
```

qui encode un sudoku comme un problème SAT

Question (10 lignes): Compléter la fonction

```
val solution_of : Ast.model -> t
```

qui étend donné un modèle renvoyé par votre implémentation de DPLL vous renvoie une grille de Sudoku complétée.

Vous pouvez tester votre programme via la commande `make sudoku`. Par défault, son comportement est déterministe, et teste votre programme sur les 10 premiers sudoku de la liste. Vous pouvez modifier ce nombre via la variable `NSUDOKU` et vous pouvez choisir le sudoku de départ via la variable `SUDOKUSTART` par exemple, pour tester sur le 50ième sudoku, vous pouvez faire

```
make sudoku NSUDOKU=1 SUDOKUSTART=50 TIMEOUT=20
```

ce qui fixera aussi le timeout de votre programme à 20 secondes. A priori, la solution des sudokus est unique pour ceux que j'ai testé. Cependant, il n'est pas impossible que ce ne soit pas le cas et que votre solveur retourne une autre solution. Si c'est le cas, il vous faudra implémenter une fonction `val is_valid : t -> bool` et modifier le fichier `main.ml` en conséquence.

2.4 Aller plus loin

Avoir fait les deux premières parties, c'est bien, mais je vous recommande chaudement d'en faire un peu plus. En particulier, tout ce que vous faites à

partir d'ici pourra vous donner des points bonus qui pourront se repercuter sur la note de votre projet logique final (y compris le projet Coq).

2.4.1 Une extension de DPLL

Dans l'article de départ, l'algorithme proposait aussi de vérifier, en plus de la phase de *unit propagation*, si une variable apparaissait seulement positivement ou négativement, ce qui permet de fixer sa valeur directement. Je vous invite à implémenter ce critère, et regarder s'il améliore vos performances. Attention cependant, ce critère est incompatible avec ce qui est décrit au paragraphe 2.4.4. Dans une autre mesure, pouvez-vous trouver d'autres critères simples à implémenter qui amélioreraient les performances de votre algorithme ?

2.4.2 Générer des grilles de sudoku

La partie précédente supposait déjà que l'on sache générer des grilles de sudoku. Je vous invite ici à compléter votre programme afin de générer vous-mêmes des grilles de sudoku. Pour cette question, l'évaluation prendra en compte critères suivants :

- La génération de grille se fait bien de façon aléatoire,
- Le nombre d'indices de vos sudokus,
- Pouvoir garantir l'unicité de la solution de vos grilles

2.4.3 Encoder d'autres problèmes

Il existe des problèmes un peu plus difficile à encoder. Cependant, cette fois, vous n'aurez peut-être pas une base de tests pour tester votre encodage ce qui rend cette question un peu longue à implémenter, notamment si vous souhaitez vérifier la solution trouvée (excepté peut-être pour les *griddlers/picross*). Il existe de nombreux jeux logiques dans la nature, mais un en particulier le site suivant regorge d'idées : `https://www.chiark.greenend.org.uk/~sgtatham/puzzles/`. Si vous souhaitez un challenge un peu plus théorique que pratique, je vous invite à regarder notamment les *griddlers* et le jeu *loopy*. En effet, ces derniers ne sont pas si simple à réduire vers SAT en temps polynomial. Pour cette question, l'évaluation prendra en compte les critères suivants :

- L'encodage sur le plan théorique,
- L'encodage sur le plan pratique (i.e. est-ce que ça marche ?),
- Vos tests.

2.4.4 Améliorer les performances de votre algorithme

Avant de commencer cette partie, je vous invite tout d'abord à vous assurer que les deux premières parties du projet sont finies et fonctionnent correctement. De plus, je vous invite à versioner votre projet sur Git, car il vous allez modifier votre implémentation de la première partie, et très souvent vous allez introduire des bogues. Utiliser Git vous assurera que vous pouvez revenir à un état stable. Une autre manière de faire, consiste à refaire un développement dans un autre fichier mais cela vous demandera de modifier un peu le fichier `src/main.ml`.

La littérature fourmille de papiers sur l'implémentation de SATsolveurs efficaces. Avant de la regarder, je vous recommande de vous concentrez d'abord sur un aspect : avez-vous choisi les bonnes structures de données pour votre algorithme DPLL? Peut-être qu'en changeant vos structures, ou bien en mémorisant des informations, vous allez pouvoir gagner du temps.

Attaquons maintenant les choses sérieuses. Peut-être avez vous remarquer que votre implémentation est extrèmement lente pour un sudoku vide, alors que cela peut sembler trivial. le problème provient du fait que DPLL ne se souvient pas de ces erreurs. Cela veut dire qu'à un moment donné, l'algorithme à brancher sur une variable, et peut importe ce qui va se passer ensuite, la formule sera insatisfiable. Cependant, s'il n'y a pas de clause unitaire, il faudra peut-être que l'algorithme fasse un certain nombre de choix avant de s'en rendre compte. Au cours de l'exécution, cette suite de séquence de choix de variables qui mène à une contradiction peut se produire souvent (à séquence fixée). Il faut donc faire comprendre à l'algorithme que cette séquence de choix n'est pas bonne. L'idée devient alors de rajouter une clause à notre problème. Cette clause encodant le fait que cette séquence de choix n'était pas la bonne. Ainsi, l'algorithme ne retombera pas dans le même écueil puisque dès que cette clause sera insatisfiable, l'algorithme reviendra en arrière (et va donc *backtracker*).

Cette idée est primordial pour faire des SAT solveurs qui puisse rivaliser. Je ne vais pas rentrer plus dans les détails ici. Pour ceux que ça intéresse, je vous invite à regarder un papier sur *minisat*, un minisat solveur qui est très petit (environ 600 lignes de C++) et qui pourtant est extrêmement rapide comparé à vos implémentations : `http://minisat.se/downloads/MiniSat.pdf`. Le papier décrit une implémentation pour C++, mais il n'est pas très compliqué de l'adapter à OCaml. Si vous vous en sentez l'âme, je vous invite à tenter une implémentation. Bien sûr, ce dernier parle d'autres optimisations dont je n'ai pas parlé ici. À vous de faire le tri sur ce qui vous semble rapide à implémenter.

2.5 Modalités de rendu

En plus du code, vous devez écrire un fichier `rapport/rapport.pdf` qui contiendra des informations sur ce que vous avez implémentez, et en particulier, j'aimerai y voir quelques statiques, notamment sur le `TIMEOUT` (< 2 mn) qui vous permet de résoudre un maximum de tests situés dans le dossier `tests` ainsi que le `TIMEOUT` minimal qui permet de résoudre les 50 premières grilles

de sudoku. Je vous invite à écrire votre rapport en LaTeX, et je vous ai déjà fourni un modèle dans le fichier `rapport/rapport.tex` que vous pouvez compiler via la commande `make rapport/rapport.pdf`. Il se peut que la commande échoue car il vous manque un programme : `rubber`. Je vais m'assurer qu'il soit installé sur les machines de la salle 411, sinon je vous invite à l'installer sur vos propres machines via votre gestionnaire de paquets. Si vous n'êtes pas encore à l'aise avec LaTeX, je vous invite à regarder les sources du sujet (`sujet/sujet.tex`). Ensuite, je vous invite à écrire votre nom et prénom dans le fichier `GNUmakefile.conf`, il faut remplir le champ `FIRSTNAME` (prénom) et `LASTNAME` (nom). Enfin, vous devriez pouvoir générer votre archive via la commande `make archive`. Ci cette dernière réussit, elle devrait vous produire un fichier `NOM_PRENOM.tar.gz` qui contient tout le nécessaire pour votre évaluation. Attention cependant, si vous avez fait des choses dans la partie trois, veuillez modifier le Makefile en conséquence pour que je puisse avoir accès à tous les fichiers dont j'aurais besoin pour l'évaluation. En particulier, si vous avez implémenté d'autres réductions que le sudoku, je veux avoir des jeux d'essais sur lesquels je peux tester votre programme. Je veux que vous m'envoyez cette archive avant la deadline qu'on aura fixée le lundi 15 avril par mail à l'adresse : `mailto:francois.thire@lsv.fr`.