TP de logique :
Implémentation d'un SAT-Solver type DPLL

L3 MPCI Semestre 5 - UE de logique

Corentin CAUGANT, Humbert de CHASTELLUX

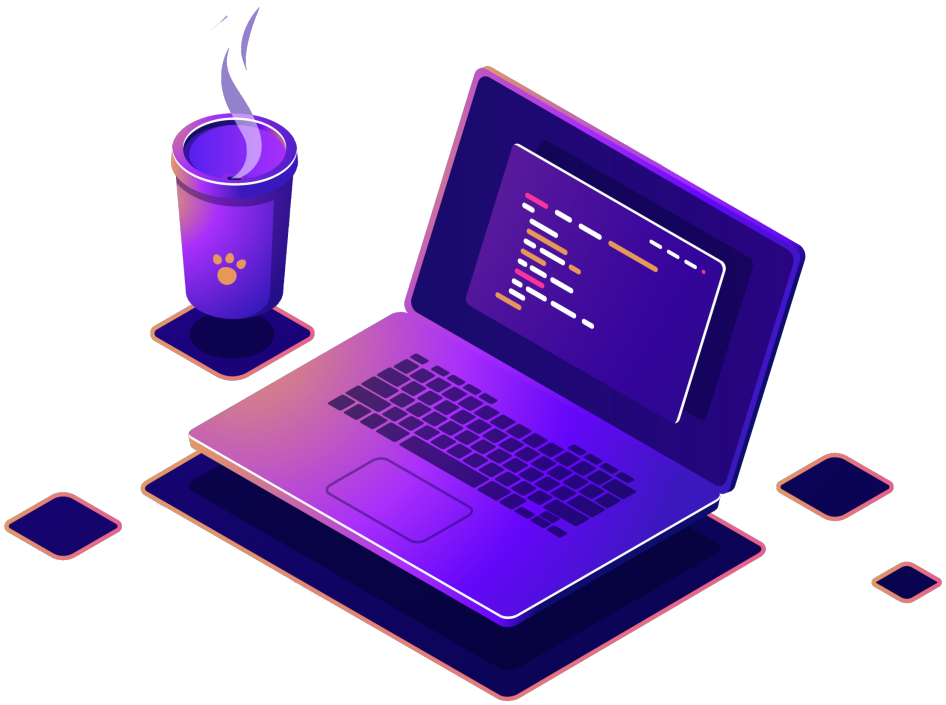


Table des matières

1 Organisation du projet

1.1 Organisation du développement

Étant donné que le développement du projet se faisait à deux, et souvent en distanciel, il a fallu mettre en place une structure de travail pour pouvoir coder à plusieurs sur le même projet.
Pour cela, nous avons opté pour l'utilisation de Git et de Github. Nous avons également mit en place des sessions de programmation en 'peer to peer' en utilisant des modules adaptés sur nos IDE (interfaces de programmations).

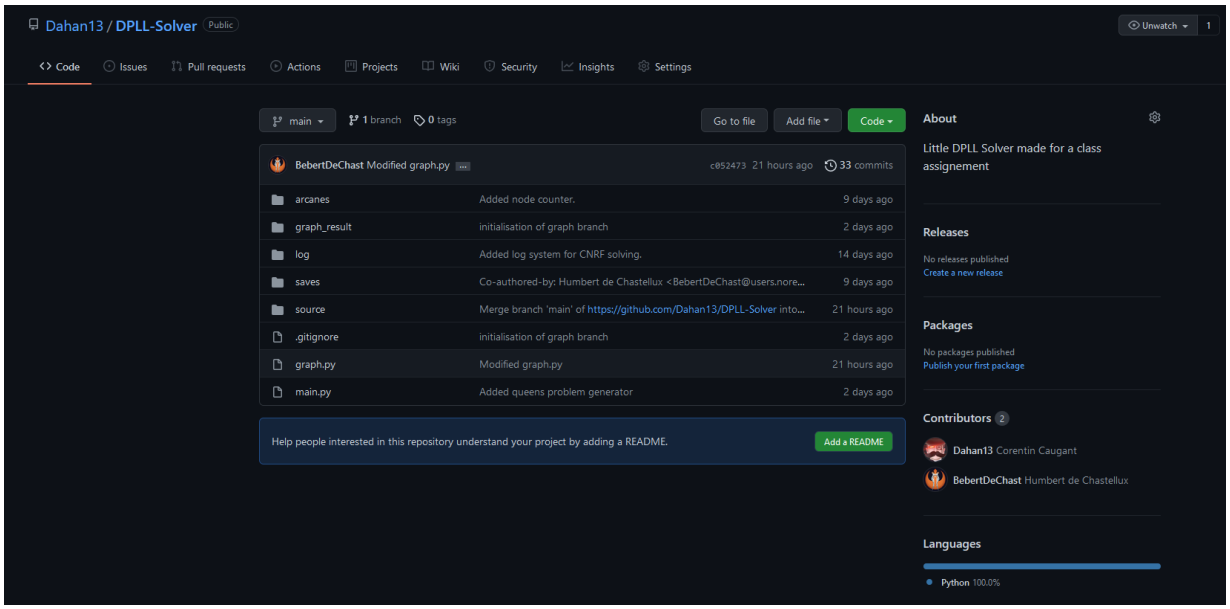


FIG. 1.1 – *Github du projet*

Le projet Github a également été passé en public. En effet, ce projet étant un projet à but éducatif, on ne peut que en tirer des bénéfices en le partageant au monde entier.
Le projet a entièrement été réalisé sous python, en effet, c'est l'unique langage de programmation connu par les deux membres du projet.

1.2 Arborescence des fichiers

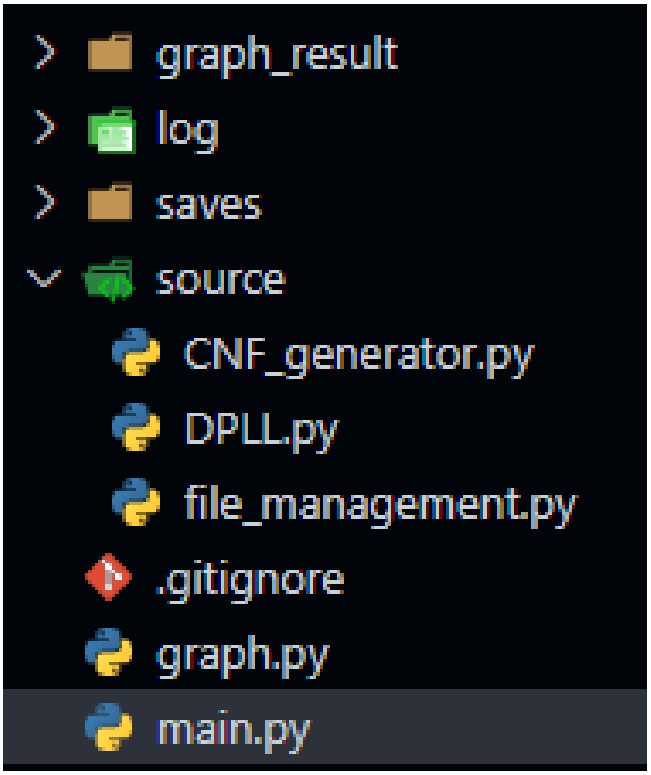


FIG. 1.2 – *Arborescence des fichiers*

L'arborescence des fichiers du projet s'organise comme suit :

- `main.py` : Fichier principal permettant d'utiliser le solver sur les conjonctives de notre choix. Ce fichier utilise les programmes présent dans `source/`.
- `source/` : Contient les programmes suivants :
  - `DPLL.py` : Solver type DPLL ainsi que toutes les fonctions servant son fonctionnement. Gère aussi les heuristiques.
  - `file_management.py` : Toutes les fonctions de sauvegardes et de chargement des conjonctives, ainsi que l'écriture des logs.
  - `CNF_generator.py` : Toutes les fonctions de générations des conjonctives aléatoires, ainsi que des problèmes des pigeons et des dames.
- `log/` : Contient les logs générés par `file_management.py`
- `saves/` : Contient les sauvegardes des conjonctives générés par `CNF_generator.py`
- `graph.py` : Génère des graphes de performances selon les heuristiques et la taille de la conjonctive.
- `graph_result/` : Contient l'intégralité des graphes générés par `graph.py`
- `.gitignore` : Paramètres du dépôt Git

2 Fonctionnement du solver DPLL

2.1 Variables et structures de données utilisés

La structure utilisé pour représenter nos littéraux et nos conjonctives est très simple :

- Les littéraux sont représentés par un dictionnaire, qui associe à chaque littéral sa valeur. Les valeurs possibles d'un littéral sont les suivantes : 'True' ou 'False' pour 'Vrai' ou 'Faux' et la valeur 'None' pour les littéraux n'ayant aucune valeur attribuée.
- Les clauses sont représentés par une liste contenant les littéraux (ou négation de littéraux) contenus dans la clause.
- Les conjonctives sont représentés par une liste de clauses.

Nous utilisons également d'autres structures de données dans notre solver:

- Pile : On empile dans cette dernière au fur et à mesure les littéraux auxquels on a attribué une valeur, cela permet de remonter facilement tous les littéraux auxquels nous avons attribués des valeurs si jamais la conjonctive s'avère insatisfaisable avec les littéraux donnés.
- Vecteur de longueur : C'est une liste construit de la même façon que la conjonctive sauf que lieu de contenir une clause par terme de la liste. Elle va contenir la longueur de la clause correspondante. En effet, au lieu de vérifier si la conjonctive est satisfaisable en simplifiant les clauses dedans selon les littéraux (suppression des clauses contenant des littéraux vrais et suppression des termes faux dans les clauses), nous allons changer la longueur de la clause (on y soustrait 1 pour chaque terme faux, et on lui attribue la valeur -1 si elle contient au moins un terme vrai. Ainsi, on sait que notre conjonctive est juste dès lors que le vecteur longueur ne contient que des -1 et on sait aussi qu'elle est fausse si il y a au moins un 0 dans le vecteur longueur. Cela permet de booster les performances du programmes étant donné que la complexité de la suppression d'un élément d'une liste est en O(n), et celle de la modification d'un entier O(1).

2.2 Fonctionnement de l'algorithme

Nous avons articulé le fonctionnement de notre solver DPLL autour de nombreuses sous-fonctions, qui accompliront toutes des tâches précises afin de faciliter l'entretiens et la lisibilité du code.

Tout le principe de fonctionnement du solver va s'articuler autour d'un arbre des possibilités : à chaque itération du programme, ce dernier va en fonction de conditions spécifiques faire appel à certaines sous-fonctions exécutant des actions précises dans l'arbre des possibilités.

Nous pouvons distinguer 3 sous-fonctions en particulier :

- '**proceed**' : Cette fonction sera appelé lorsque le solver verra que l'attribution actuelle des valeurs de littéraux n'amène aucune insatisfaisabilité ou tautologie dans la conjonctive simplifié. La fonction va alors se charger d'assigner une valeur arbitraire à un nouveau littéraux en suivant le protocole suivant :
  - Si au moins un mono-littéral est présent, **proceed** va lui assigner une valeur telle que la clause de longueur une correspondante soit vraie. Il va également indiquer la valeur de ce littéral comme étant définitive.
  - Sinon, si au moins un littéral pur est présent, **proceed** va lui assigner une valeur telle que les clauses le contenant soient juste
  - Sinon, **proceed** va prendre le premier littéral non attribué qu'elle va trouver, soit de façon naïve, soit en appliquant une heuristique si demandé par l'utilisateur et lui assigner par défaut la valeur 'True'.
- '**fail**' : Se lance dans le cas où la conjonctive est devenue insatisfaisable et que le dernier littéral traité dans la pile n'a été assigné à une valeur que une seule fois. **fail** va alors changer la valeur du

littéral par la valeur qui ne lui a pas été attribué, tout en précisant que ce dernier a été modifié pour la deuxième fois.

– '**back**' : Se lance dans le cas où la conjonctive est devenue insatisfaisable et que le dernier littéral traité dans la pile a déjà été assigné aux deux valeurs possibles. **back** va alors retirer ce littéral de la pile et regarder celui se trouvant juste après dans la pile, tant qu'il tombera sur des littéraux ayant déjà été modifié deux fois, il continuera de remonter en dépilant la pile. Une fois qu'il tombe sur un littéral modifié qu'une seule fois, il va activer **fail** sur ce dernier. Si il dépile l'intégralité de la pile, la résolution sera alors terminé et on pourra stopper le programme.

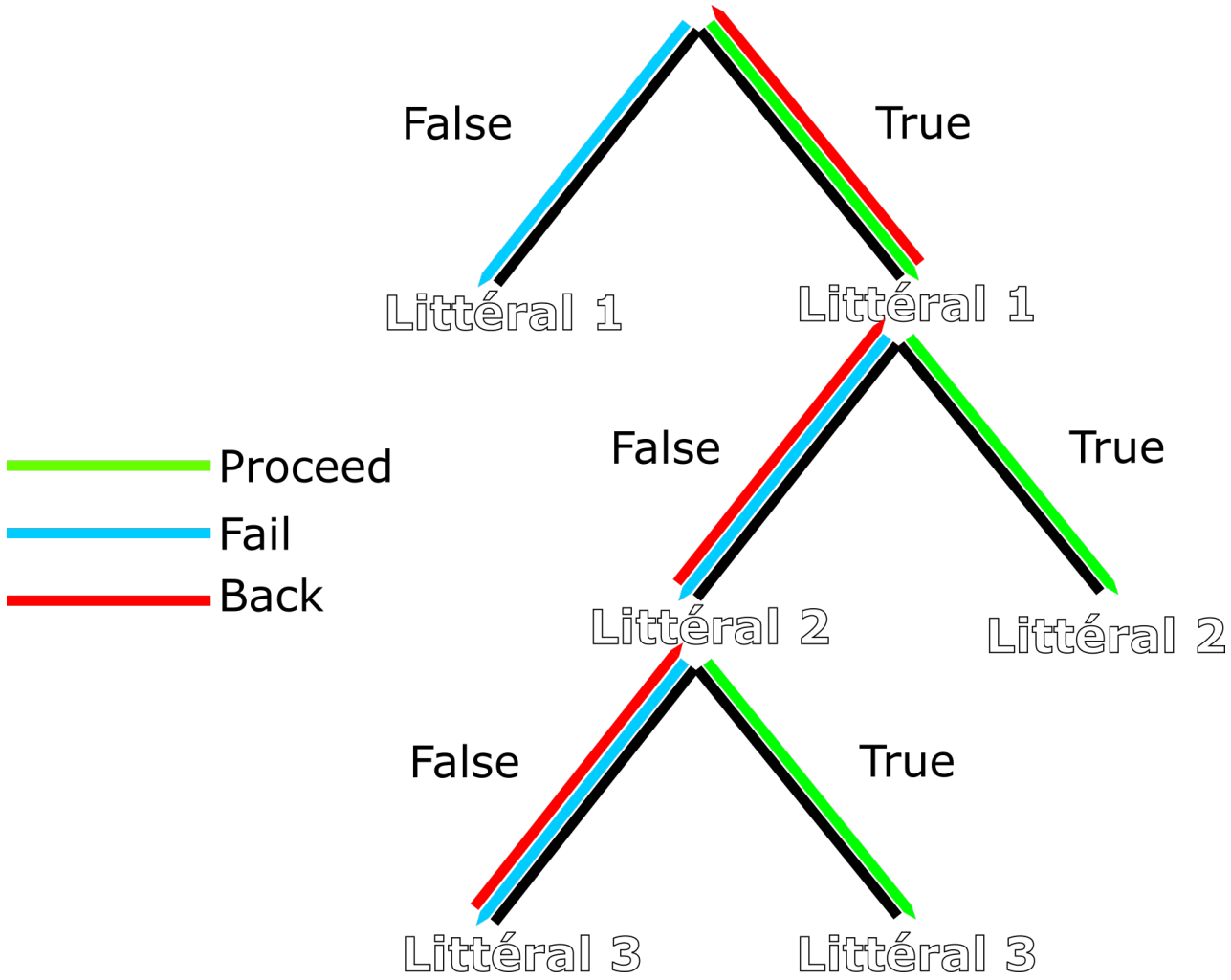


FIG. 2.1 – *Exemple du solver parcourant un arbre des possibilités simple*

2.3 Système de logs

Nous avons également élaboré un système de logs, afin de récupérer et de pouvoir sauvegarder toutes les données relatives à l'exécution du solver sur une conjonctive. La création de logs est géré par file_management.py et se présente comme suit :

- Nombre total des combinaisons possibles de littéraux (2^n).
- Nombre des solutions trouvés par le solver.
- Nombre de noeuds de l'arbre traversé par le solver.
- L'heuristique choisit pour résoudre.
- Temps d'exécution du solver.

Optionnel :
- Temps d'exécution du solver naïf (teste toutes les combinaisons possibles de littéraux sur la conjonctive, complexité en 2^n).
- Résultat de la comparaison entre les résultats du solver et du solver naïf (True ou False selon si les solutions correspondent ou non).

- Conjonctive traité.

- Enumération des solutions trouvés par le solver.

- Énumération des solutions trouvés par le solver naïf ....

Ces logs sont sauvegardés automatiquement au format texte dans le dossier log/

3 Sauvegarde, chargement et génération de conjonctives

3.1 Sauvegarde et chargement de conjonctives

Nous avons créé un algorithme dans `file_management.py` qui nous permet de sauvegarder et de charger des conjonctives. Le format choisis est:

```
Nombre de littéraux conjonctive 1

Clause 1
Clause 2
...
Clause n


Nombre de littéraux conjonctive 2
....
```

Le tout sauvegardé dans un fichier texte dans le dossier `saves/`. Nous disposons donc de fonctions dans `file_management.py` pour sauvegarder une conjonctive donné dans un fichier dont on choisit le nom ou pour charger une conjonctive en fournissant le chemin vers le fichier de sauvegarde.

3.2 Générations de conjonctives

Des fonctions de générations de conjonctives ont été implémentés dans `CNF_generator.py` afin de pouvoir faire des tests de performances et également des tests de débuggages à plus grande échelle.

Nous disposons de 3 fonctions de générations :

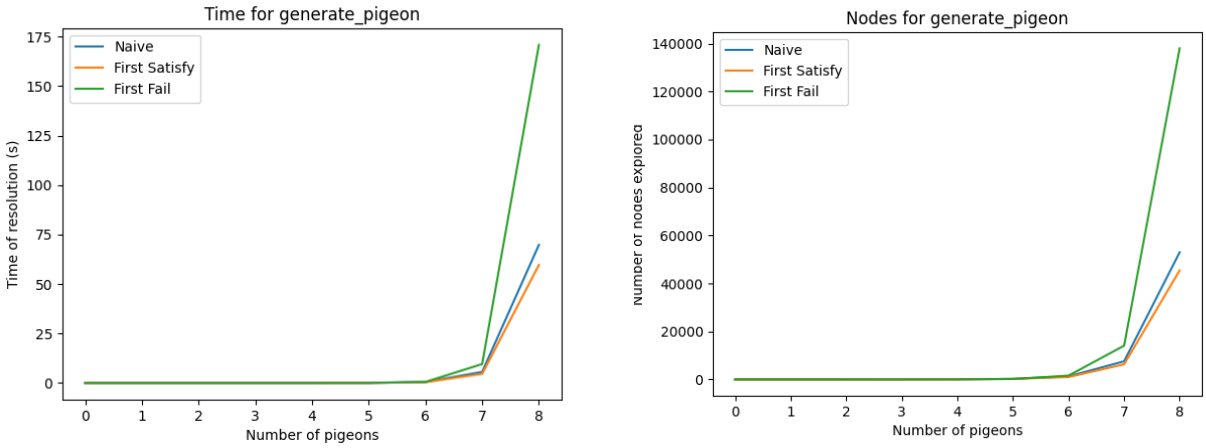
- 'generate_conjonctive' : Génère une conjonctive aléatoire étant donné un nombre de littéraux et un nombre de clauses. Dans chaque clause, chaque littéral a 33% de chances d'y apparaître sous sa forme normale, 33% d'y apparaître sous sa forme négationnée et 33% de chances de ne pas y apparaître.
- 'generate_pigeon' : Génère un problème des pigeons d'une taille N donné en utilisant les relations vu en TD.
- 'generate_queens' : Génère un problème des dames d'une taille N donné en utilisant les relations vu en TD.

Pour générer nos problèmes des pigeons et des dames nous avons utilisé une astuce. En effet, nous avons vu en TD des formules sous la forme de couples (par exemple pour pigeon on utilisait les couples (i, j) avec i le numéro du pigeon et j le numéro du pigeonnier). Or, nos littéraux ici sont 1, 2, 3, 4, 5, ..., N. Nous avons donc dû générer toutes nos clauses en nous basant sur les couples mais nous avons en plus de tout ça fait appel à une bijection reliant chaque couple à un littéral (par exemple (1, 1) avec 1, (1, 2) avec 2 etc...). Cela nous a permis de générer facilement les clauses tout en ne déviant pas trop de ce que nous avions déjà vu en TD.

4 Tests de performances

Nous avons procédé à des tests de performances chronométré en mesurant le temps que mets l'algorithme de DPLL pour résoudre le problème ainsi que le nombre de noeud parcouru dans l'arbre de possibilité. Ces tests ont été faits sur le problème des pigeons et des dames.

4.1 Problème des pigeons



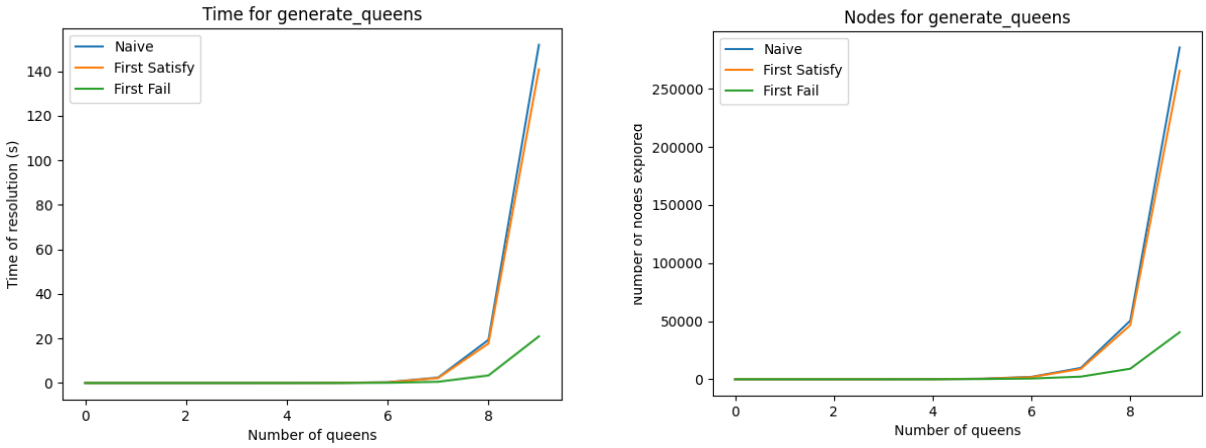
(a) *Mesure du temps de calcul*

(b) *Mesure du nombre de noeuds parcourus*

FIG. 4.1 – *Problème des pigeons*

On observe que l'heuristique First Satisfy est le plus efficace/rapide. En utilisant une échelle logarithmique on peut dire que notre algorithme a une complexité exponentielle.

4.2 Problème des dames



(a) *Mesure du temps de calcul*

(b) *Mesure du nombre de noeuds parcourus*

FIG. 4.2 – *Problème des dames*

On observe que l'heuristique First Fail est le plus efficace/rapide. En utilisant une échelle logarithmique on peut dire que notre algorithme a une complexité exponentielle.