ERGO – RAPPORT

Delay Emmanuel– Desforêts Nicolas

29 mai 2019

Table des matières

1 Présentation du projet

1.1 Présentation du jeu

Le point de départ est le jeu Ergo, « The Game of Proving You Exist ! ». Les règles détaillées sont ici.

Le jeu est composé de 55 cartes : 4 de chaque variable (A, B, C ou D), 4 de chaque opérateur (ET, OU, $\Longrightarrow$), 6 cartes NON, 8 parenthèses, 3 cartes Ergo et 10 cartes particulières.

Chaque joueur (4 maximum) se voit assigné une variable (A, B, C ou D) au début du jeu. À chaque manche, les joueurs essaient collectivement de créer une preuve de leur existence tout en réfutant l'existence des autres joueurs. Au début de la manche, chaque joueur reçoit 5 cartes puis, à chaque tour, un joueur pioche deux cartes et doit jouer deux cartes (éventuellement les défausser). Lorsque une carte Ergo est jouée ou qu'il n'y a plus de carte dans la pioche la preuve est terminée. À condition qu'il n'y ait pas de paradoxe, chaque joueur dont l'existence est prouvée reçoit un nombre de points égal au nombre de cartes dans la preuve. Toutes les cartes sont ensuite mélangées et une nouvelle manche est lancée. Le premier joueur ayant 50 points gagne.

Concernant la construction de la preuve, un certain nombre de règles doivent être respectées :

- la preuve doit avoir au maximum 4 lignes. Dès qu'elle atteint 4 lignes, toutes les cartes supplémentaires doivent être jouées sur une de ces lignes ;
- chaque ligne doit être syntaxiquement correcte (deux opérateurs ou deux variables ne peuvent pas se suivre, chaque parenthèse ouvrante doit correspondre à une parenthèse fermante, ...) ;
- une carte peut être insérée entre deux cartes déjà posées à condition que le résultat reste syntaxiquement correct.

1.2 Le projet

Le but initial était de réaliser une implémentation en Python de ce jeu. Pour cela, plusieurs points semblaient à traiter, plus ou moins par ordre de difficulté croissante :

- analyser une ligne de la preuve pour vérifier qu'elle est syntaxiquement correcte ;
- coder une ligne syntaxiquement correcte sous une forme exploitable (arbre, forme conjonctive normale, forme disjonctive normale, ...?) ;
- déterminer à partir du codage des 4 lignes quelles variables sont prouvées ou s'il y a une contradiction ;
- réaliser une interface graphique avec tkinter ;
- implémenter une fonction pour pouvoir jouer contre l'ordinateur.

Finalement, tous les points précédents ont pu être traités.

2 Organisation du travail

2.1 Outils utilisés

Nous avons configuré un Rasberry Pi comme serveur pour installer redmine dessus, en nous aidant beaucoup du wiki de Frédéric Muller (merci à lui) et des article de Linux Pratique mis à notre disposition par The Big Boss (loué soit-il).

Nous avons aussi créé un dépôt sur github : `https://github.com/isnpaulconstans/Ergo`

La documentation technique est générée avec Sphinx, encore grâce aux articles de GNU/Linux Magasine que notre Big Boss a eu la bonté de nous fournir (Il n'en sera jamais assez remercié [1]).

1. Le cirage de pompe peut-il augmenter significativement la note de ce module ?

Les résultats sont disponibles sur http://paulconstans.ddns.info/redmine/projects/ergo et sur http://paulconstans.ddns.info/documentation/.

2.2 Répartition du travail

Après quelques discutions, le travail s'est assez naturellement réparti. Emmanuel Delay s'est chargé de la partie algorithmique tandis que Nicolas Desforêts s'est occupé de l'interface graphique et de la réalisation d'une page web pour les règles du jeu. Comme nous travaillons tous les deux dans le même lycée, nous avons pu nous voir régulièrement pour faire la jointure entre nos deux parties et nous mettre d'accord sur les étapes suivantes.

3 Solutions techniques

3.1 Les constantes

Les différentes constantes pouvant être utiles dans plusieurs classes, comme le nombre de cartes de chaque type, la taille des images correspondante ou les couleurs du canvas, sont rassemblées dans une classe dédiée.

3.2 Les modules concernant les cartes

3.2.1 Card

La classe `Card` définit les différentes cartes. Chaque carte a un nom, un niveau de priorité. De nombreuses méthodes permettent de déterminer de quelle carte, ou de qelle famille de carte (lettre, opérateur, joker, ...) il s'agit. Une méthode permet aussi de « retourner » les parenthèses pour transformer un parenthèse ouvrante en parenthèse fermante et réciproquement. Comme il est parfois nécessaire de comparer deux cartes, la méthode `__eq__` permet de tester l'égalité de deux cartes (en comparant les noms), ou d'une carte et d'une chaîne de caractères. Les cartes pouvant aussi servir de clé dans un dictionnaire, une fonction de hachage a été implémentée par la méthode `__hash__`.

3.2.2 CardList

La classe `CardList` hérite de la classe `list` et gère les listes de cartes. Elle permet d'ajouter, modifier ou supprimer une carte de la liste. Cette classe permet de déterminer si la liste de carte est syntaxiquement correcte, et dans ce cas d'y associer son écriture en notation polonaise inversée (NPI). Un attribut `modif` permet de savoir si la liste a été modifiée depuis le dernier calcul de la NPI pour éviter de le refaire inutilement.

3.2.3 Proof

La classe `Proof` gère les quatre prémisses. Elle répercute les modifications (ajout, suppression, changement) aux différentes prémisses et gère leur conjonction pour déterminer la NPI associée à la preuve. Elle permet également de savoir si toutes les lettres ont été jouées (pour pouvoir éventuellement jouer une carte Ergo), et le nombre de cartes jouées (pour calculer le score correspondant). Les méthodes `insert` et `pop` ont un paramètre supplémentaire qui permet de gérer deux cas différents pour chacune :

- l'ajout peut provenir soit d'une nouvelle carte jouée par le joueur soit de l'annulation d'une carte Tabula Rasa qui doit être gérée différemment.

- De même , la suppression peut soit correspondre à une carte qui vient d'être jouée, soit au jeu d'une carte Tabula Rasa.

Pour pouvoir gérer ces différents cas, on maintient une liste `currently_added` des numéro de prémisse et index des cartes qui peuvent être modifiées, ainsi qu'un booléen indiquant si la carte correspond à un jeu « classique » (une carte qui vient d'être jouée et qui peut être retirée) ou au jeu d'une carte Tabula Rasa (carte qui a été supprimée par un Tabula Rasa et qui peut être remise). Les indices de cette liste doivent être actualisés à chaque ajout ou suppression de cartes dans les prémisses pour suivre les modifications. Par exemple, si une carte est ajoutée avant une carte mémorisée, l'indice de cette dernière doit être augmenté de 1 pour refléter sa nouvelle position dans la prémisse.

3.2.4 Deck

La classe `Deck` hérite de `list` et gère le paquet de cartes. Elle permet de tirer un certain nombre de cartes du paquet (cinq en début de partie, et deux au début de chaque tour). Les méthodes `append` et `pop` permettent de gérer les cartes supprimées par un Tabula Rasa qui doivent être remises à la fin du paquet. La méthode `is_finished` permet de savoir si le paquet est terminé, pour le cas échéant mettre fin au tour.

3.3 Les modules concernant les démonstrations

L'analyse de la preuve se fait par la classe abstraite `Demonstration`. Cette classe est concrétisée par les deux classes `ForceBrute`, qui cherche à déterminer les « variables prouvées » par force brute, et `DPLL` qui utilise l'algorithme de Davis-Putnam-Logemann-Loveland. Cette dernière commence par faire appel à la classe `FCN` pour obtenir l'écriture en forme conjonctive normale de la preuve. Les algorithmes utilisés dans ces classes seront décrits dans la section suivante.

3.4 Jeu de l'ordinateur

3.4.1 Ordi

Le gestion des coups possibles se fait dans la classe `Ordi`. Cette classe contient une méthode abstraite `choix_coups` en vue de tester différentes idées concernant les deux cartes à jouer. La méthode `joue` joue effectivement les cartes choisies par `choix_coups` dans les prémisses, et elle renvoie un message à afficher concernant ce jeu. La liste des cartes particulières (`Fallacy`, `Justification` et `Ergo`) jouées qui ne concernent pas directement les prémisses est renvoyée afin que la classe `Main` puisse les gérer.

À l'initialisation, un appel à `__parentheses` permet, si la main comporte au moins deux parenthèses, d'en avoir au moins une ouvrante et une fermante. Pour simplifier la gestion des cartes `Fallacy` et `Justification`, la méthode `__justification` met en première position dans la main la carte `Justification` s'il y en a une, puisque c'est le seul cas où l'ordre est important : il faut d'abord la jouer avant de pouvoir poser une autre carte dans la preuve.

De plus, la méthode `coups_possibles` travaille avec une copie de la main dans laquelle les joker (`WildVar` et `WildOp`) sont remplacés par une lettre et un opérateur (ça ne change rien à la syntaxe). C'est ce que fait la méthode `__wild`. Pour retrouver les cartes qui étaient initialement des jokers, j'ai ajouté un attribut `wild` à la classe `Card` qui mémorise l'état initial de la carte.

Pour la gestion de la carte `Revolution` (qui permet d'échanger deux cartes), j'ai mis bout à bout les quatre prémisses pour pouvoir plus facilement les parcourir avec seulement deux boucles imbriquées. Pour retrouver les coordonnées initiales (sous forme d'un numéro de prémisse et d'un index), j'ai fait une petite fonction `index_flat2premise_index`.

Dans la méthode `coups_possibles`, on détermine l'ensemble des coups possibles. Si le joueur est sous le coup d'une falsification, il peut soit commencer par jouer une carte `Justification`, soit jouer une (ou éventuellement deux) carte(s) `Fallacy` sur un (deux) joueurs. Cette partie étant assez différente du reste, on la traite dans une boucle séparée. Sinon, on essaye de jouer chacune des cartes de la main à l'aide de deux boucles imbriquées. Cette méthode est une horreur du point de vue Deep Nesting [2], mais j'ai eu beau tourner le problème de différentes façons, je n'ai pas réussi à faire plus propre. Le gestion des cartes spéciales, en particulier celles qui ne se jouent pas dans les prémisses, nécessite l'utilisation d'une variable booléenne `special1`.

3.4.2 OrdiRandom

La classe `OrdiRandom` concrétise la méthode `choix_coups` en se chargeant de choisir un coup au hasard dans la liste des coups possibles. En cas de défausse, les cartes à défausser sont elles aussi choisies au hasard dans la main. De même, la « victime » d'une carte `Fallacy`, les cartes à échanger avec `Revolution` ou les joker sont choisis aléatoirement.

3.4.3 OrdiScore

La classe `OrdiScore` concrétise elle aussi la méthode `choix_coups`, mais en attribuant un score à chaque coup et en choisissant un coup parmi ceux ayant le meilleur score.

Pour cela, une valeur est affectée à chaque carte, puis la méthode `sort_hand` trie la main par valeurs décroissantes. Ainsi, les éventuelles cartes à défausser seront les dernières de la main. Ensuite, la liste des coups possibles est calculée par la méthode `coups_possibles`. Mais cette liste doit être complétée en détaillant tous les échanges possibles pour les cartes `Revolution`, et toutes les possibilités pour les cartes joker. C'est la méthode `extend_coups` qui se charge de ce travail. Pour indiquer par quelle carte un joker doit être remplacé, l'index du coup à jouer est transformé en un couple (index, name) où name est le nom à affecter au joker.

Pour le jeu d'une carte `Fallacy`, la méthode `choice_fallacy` se charge de désigner la « victime ». Pour cela, on privilégie les joueurs ayant le plus gros score, et n'étant pas (ou peu) sous le coup d'une falsification.

Ensuite, tous les coups possibles sont joués dans la preuve, avec à chaque fois une évaluation par la méthode `calc_score` du score obtenu en utilisant un certain nombre de coefficients (pour l'instant choisis au « doigt mouillé »). Le coup est ensuite annulé pour restaurer la preuve dans son état initial. Le coup ayant obtenu le meilleur score est choisi. En cas d'ex æquo, un coup au hasard est choisi parmi les meilleurs.

Avant de renvoyer le coup choisi, les joker éventuellement à jouer sont remplacés dans le main par la valeur choisie.

3.5 L'interface graphique

3.5.1 Généralités

Toute la partie graphique est déléguée à la classe `ErgoCanvas` qui hérite de la classe `Canvas` de tkinter.

La gestion du jeu se fait à la souris. On peut attraper (bouton gauche), déplacer (bouton gauche maintenu) et déposer (bouton gauche relâché) la carte à l'aide des méthodes `select`, `move` et `drop`.

Le bouton droit permet, associé à la méthode `switch`, si la carte est une parenthèse, de la retourner.

2. J'espère que Régis Barbanchon, qui nous a fait le cours de PCOO l'an dernier, ne verra pas ça ;-)

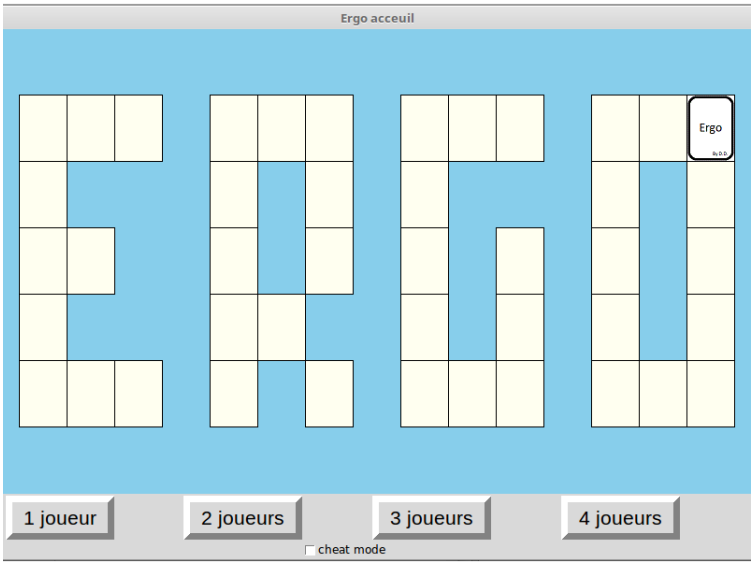

FIGURE 1 – Fenêtre d'introduction

De plus, il s'est avéré qu'il fallait autoriser l'annulation d'une carte effacée avec `Tabula Rasa` sous peine de bloquer le joueur. C'est possible avec la touche ESC.

Il a fallu créer les images des cartes, en choisissant une dimension pratique pour la gestion du placement de cartes. La première dimension était trop importante et ne permettait pas de faire des lignes de preuve suffisamment longues.

Nous avions initialement choisi de créer nos images au format gif, mais à l'usage le format png s'est révélé plus facile à manipuler. L'image carteBack permet d'afficher les mains des autres joueurs face cachée.

Avec les nouvelles fonctionnalités qui permettent de choisir soit le mode multijoueur ou de jouer contre l'ordinateur une nouvelle classe `ErgoIntro` a été créée (voir figure 1).

Elle permet de créer une fenêtre d'accueil avec une animation au démarrage. La méthode animate_letter permet de simuler une distribution des cartes suivant la forme des lettres du nom du jeu ERGO.

Quatre boutons permettent d'accéder au jeu dans le mode choisi, et une case à cocher permet de choisir le « cheat mode ». Ce dernier ajoute un bouton `cheat` à l'interface de jeu qui permet de savoir quels sont les joueurs actuellement démontrés.

Le résultat est visible Figure 2.

3.5.2 Outils

Comme les cartes doivent être placées sur une « grille », deux méthodes `row_col2x_y` et `x_y2row_col` ont été créées pour permettre de passer des coordonnées écran aux coordonnées dans la grille et réciproquement.

Les méthodes `init_bind` et `reset_bind` permettent de désactiver ou modifier les événements souris à certaines étapes du jeu (pendant l'ouverture de messages ou la sélection des cartes à échanger par `Revolution`). Dans le cas d'une carte `Revolution`, la méthode `select_revolution` permet de choisir les deux cartes à échanger.

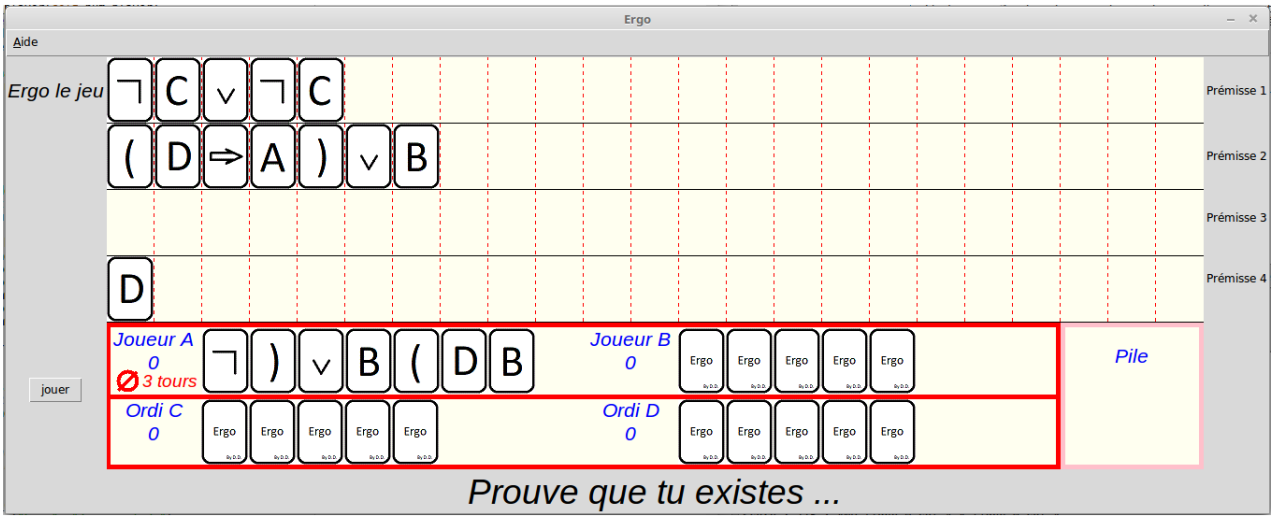

FIGURE 2 – Rendu final

3.5.3 La méthode drop

Elle permet de placer la carte sélectionnée par `select` au bon endroit. Dans le cas d'une carte « simple », elle doit gérer les cas où on lâche la carte :

- dans les prémisses. La carte est alors insérée à la positon choisie dans la preuve ;
- dans la pile. La carte est alors ajoutée à l'attribut `pile` qui permet éventuellement de la récupérer ;
- dans la zone des mains. La carte est alors remise dans la main du joueur.

Différents cas particuliers sont à gérer pour les cartes spéciales :

- les cartes `fallacy` et `Justification` jouées dans la zone des mains. L'attribut `fallacy` de la classe `Main` est mis à jour en fonction ;
- la carte `Tabula Rasa`. Un message informe de l'effacement qui peut être annulé en appuyant sur la touche Echap ;
- la carte `revolution`. Un message invite à sélectionner les cartes à échanger ;
- les cartes joker. Une liste à puce permet de sélectionner la carte voulue ;
- la carte `Ergo`. Si toutes les lettres ont été jouées, met fin à la manche en appelant la méthode `fin_manche` de la classe `Main`.

3.6 La classe Main

La classe `Main` gère le jeu en lui même. C'est elle qui gère l'initialisation du jeu, le passage d'un tour au suivant et la fin de partie. Elle utilise la classe `ErgoCanvas` pour la gestion du canvas, la classe `Deck` pour le jeu de carte, les classes `Proof` et `Demonstration` (implémentée par `ForceBrute` ou `DPLL`) pour la gestion de la preuve et la classe `Ordi` (implémentée par `OrdiRandom` ou `OrdiScore`) pour le jeu de l'ordinateur.

Les règles du jeu sont dans un fichier html associé à un fichier css . Elles sont affichées dans le navigateur par défaut grâce au module `webbrowser`. On y trouve les détails d'utilisation des cartes spéciales, les règles de logique et quelques captures d'écran.

4 Algorithmes utilisés

4.1 Passage en notation polonaise inversée : algorithme Shunting-yard

Une des premier problème algorithmique a été de transformer l'écriture algébrique des preuves en une notation plus utilisable. Ayant pas mal travaillé avec mes élèves sur l'évaluation d'une expression en notation polonaise inversée (NPI), je me disais que je devrais arriver à quelque chose si je pouvais transformer l'écriture algébrique en NPI. J'ai fait quelques recherches la dessus, et je suis tombé sur l'algorithme de Shunting-yard.

Je l'ai légèrement adapté au contexte (proposition logique au lieu de d'expression mathématique) pour obtenir l'algorithme 1.

Entrée : Une liste `input` de cartes (propositions ou connecteurs)
Sortie : Une liste npi correspondant a la notation polonaise inversée de l'entrée
Traitement

```
Créer une pile vide
npi ← []
pour chaque carte de input faire
    si carte est une lettre alors
        ajouter carte à npi
    sinon si carte est un parenthèse ouvrante alors
        empiler carte
    sinon si carte est une parenthèse fermante alors
        tant que pile est non vide et que le sommet de la pile n'est pas une parenthèse
          ouvrante faire
            dépiler une carte et l'ajouter à npi
        si pile est vide alors
            quitter // Problème de parenthésage
        sinon
            dépiler la parenthèse ouvrante
    sinon
        tant que pile est non vide et que le sommet de la pile a une priorité supérieure à
          carte faire
            dépiler une carte et l'ajouter à npi
        empiler carte
tant que pile est non vide faire
    dépiler une carte et l'ajouter à npi
    si la carte est une parenthèse ouvrante alors
        quitter // Problème de parenthésage
```

Algorithme 1 : Algorithme de passage en notation polonaise inversée

4.2 Évaluation de la preuve

4.2.1 Force brute

Ici, mon idée a été d'attaquer le problème en force brute : tester tous les modèles possible (comme il y a 4 variables, il y a seulement $2^4 = 16$ possibilités) et pour chacun évaluer la preuve. Si le résultat est Vrai, c'est que le modèle est admissible et on le mémorise. Ensuite, on regarde pour

chaque variable si elle a toujours la même valeur (Vrai ou Faux) dans tous les modèles admissibles. Si c'est la cas, la variable est prouvée ou niée.

Pour l'évaluation d'une formule, j'ai utilisé l'algorithme classique d'évaluation d'une expression en NPI (algorithme 2).

Entrées : Une liste `npi` de carte en NPI et une interprétation
Sortie : L'évaluation de la liste
Traitement
  Créer une pile vide
  pour chaque *carte de npi* **faire**
    si *carte est une lettre* **alors**
      empiler sa valeur dans l'interprétation
    sinon si *carte est un opérateur binaire* **alors**
      dépiler les deux dernières valeurs
      effectuer l'opération entre ces valeurs
      empiler le résultat
    sinon `// c'est un opérateur unaire, le NON`
      dépiler la dernière valeur
      empiler sa négation
  retourner *le sommet de pile* `// qui ne doit avoir qu'un élément`

Algorithme 2 : Algorithme d'évaluation d'une formule

4.2.2 Algorithme de Davis-Putnam-Logemann-Loveland (DPLL)

Même si l'algorithme précédent marche bien dans le contexte qui nous intéresse (4 variables propositionnelles), comme l'algorithme DPLL (algorithme 3) nous a été présenté dans le cours de logique, j'ai voulu l'implémenter. Pour cela, il fallait commencer par transformer la preuve au format NPI en une Forme Conjonctive Normale (FCN) sous forme d'une liste de clauses. Cela se fait en 4 étapes :

- élimination des implications ($A \implies B \equiv \neg A \vee B$) ;
- utilisation des lois de Morgan ($\neg(A \vee B) \equiv \neg A \wedge \neg B$ et $\neg(A \wedge B) \equiv \neg A \vee \neg B$) ;
- élimination des doubles négations ($\neg\neg A \equiv A$) ;
- développement ($A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$).

5 Évolutions possibles

Les calculs du score dans la classe `OrdiScore` dépend de plusieurs paramètres. Je pense que le choix de ces paramètres pourrait se prêter assez bien à un algorithme génétique en faisant jouer l'ordinateur contre lui même et en faisant des croisements entre les coups gagnants.

On pourrait aussi tester d'autres méthodes pour le jeu de l'ordinateur. J'avais éventuellement pensé à un minimax en testant toutes les cartes qui n'ont pas encore été jouées, mais je pense que ça va faire trop de calculs.

Entrées : Une liste de clauses `clause_list` et un modèle partiel `model`
Sortie : Vrai si on peut compléter le modèle partiel en un modèle complet, Faux sinon
Traitement

```
tant que clause_list contient une clause unitaire [lit] faire
    ajouter la valeur de lit au modèle
    supprimer toutes les clauses contenant lit
    supprimer ¬lit de toutes les clauses où il apparaît
si il ne reste plus de clause alors retourner Vrai
si clause_list contient une clause vide alors retourner Faux
pour chaque variable var non testée faire
    model_tmp[var] ← Faux (resp. Vrai)
    si non DPLL(clause_list, model_tmp) alors      // La négation est prouvée
        model[var] ← Vrai (resp. Faux)
        supprimer toutes les clauses contenant ¬var
        supprimer var de toutes les clauses où il apparaît
        retourner DPLL(clause_list, model)
    model[var] ← None                              // var est indécidable
retourner Vrai                          // Toutes les variables on été testées
```

Algorithme 3 : Algorithme de Davis-Putnam-Logemann-Loveland (DPLL)

A Diagramme des classes