ERGO – RAPPORT

Delay Emmanuel– Desforêts Nicolas

Table des matières

I. Présentation du projet

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.

2) Le projet

Le but est de réaliser une implémentation en Python de ce jeu. Pour cela, plusieurs points sont 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.

II. Organisation du travail

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

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.

III. Solutions techniques

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.

2) Les modules concernant les cartes

a) 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

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

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__`.

b) 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.

c) 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.

d) 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) 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.

4) L'interface graphique

Nous utilisons tkinter pour l'interface graphique. Avec en complément messagebox.
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. Les constantes correspondant à la taille des différentes cartes, à l'épaisseur des traits, au nom ou au nombre de chaque cartes ont été

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.
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.
Deux boutons permettent d'accéder au jeu dans le mode choisi.

5) La classe Main

La classe `Main` gère le jeu en lui même et la gestion du `Canvas` est déléguée à la classe `ErgoCanvas`.
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.
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.
Les règles du jeu sont dans un fichier texte. Lors de l'appel de la méthode nous faisons une lecture du fichier puis un affichage dans une fenêtre messagebox.

IV. Algorithmes utilisés

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.

2) Évaluation de la preuve

a) 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.

D'après le cours qu'on a eu pour l'instant sur la logique avec Line Jakubie-Jamet, cette méthode constitue une preuve sémantique, mais c'est équivalent à une preuve syntaxique.

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

b) 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 ($\neg\neg A \equiv A$) ;

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

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

- développement $(A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C))$.

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)

3) Jeu de l'ordinateur

Le gestion des coups possibles se fait dans la classe `Ordi`. Cette classe contient une méthode abstraite `joue` en vue de tester différentes idées concernant les deux cartes à jouer.
Une première étape consiste, si on a au moins deux parenthèses en main, à se débrouiller pour en avoir au moins une ouvrante et une fermante. Pour simplifier la gestion des cartes Fallacy et Justification, une deuxième étape consiste à mettre en premier la carte `Justification` s'il y en a une. Enfin, on 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). 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`.

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.
Enfin, on détermine l'ensemble des coups possibles en essayant de jouer chacune des cartes de la main à l'aide de deux boucles imbriquées. Le gestion des cartes spéciales, en particulier celles qui ne se jouent pas dans les prémisses, nécessite l'utilisation d'une variable booléenne `special1`.
Ces calculs sont réalisés dans la méthode `coups_possibles`. Cette méthode est une horreur du point de vue Deep Nesting, mais j'ai eu beau tourner le problème de différentes façons, je n'ai pas réussi à faire plus propre. [2]

Ensuite, la classe `OrdiRandom` concrétise la méthode `joue` en se chargeant de choisir un coup au hasard. Elle joue effectivement les cartes choisies dans les prémisses, et elle renvoie un message à afficher concernant ce jeu. Le gestion des cartes particulières (`Fallacy`, `Justification` et `Ergo`) qui ne concernent pas directement les prémisses est déléguée à la classe appelante.

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

V. Évolutions à venir

- Proposer une version chronométrée où le joueur a un temps limité pour jouer.
- Ajouter un bouton `Cheat` qui indique ce qui est prouvé pour l'instant.
- Revoir l'esthétique des différents messages, et réécrire la règle du jeu un peu plus précisément.
- Tester d'autres méthodes pour le jeu de l'ordinateur. Il faudrait en particulier attribuer un score à chaque coup en fonction de ce qui est prouvé. Ensuite, 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. Peut-être qu'en faisant quelques parties, une stratégie se présentera... Sinon, il restera la solution de faire appel à un ami ?

A Diagramme des classes