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, $\implies$), 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 (à priori avec tkinter) ;
- implémenter une fonction pour pouvoir jouer contre l'ordinateur.

Si on finit tout ça et qu'on a peur de s'ennuyer, on pourra toujours creuser pour améliorer la façon dont l'ordinateur joue. Au pire, on demandera à Frédéric Muller de nous prêter ses TetrisBot pour qu'ils apprennent à jouer à Ergo ;-)

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. 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

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. Il a juste fallu redimensionner nos images, car nous avions prévus des constantes (CARD_WIDTH et CARD_HEIGHT) en cas de changement.
Nous avons choisi de créer nos images au format gif. Une carte particulière, "back" associée à l'image carteDos.gif permet d'afficher les mains des autres joueurs face cachée.
Les cartes sont gérées à l'aide du dictionnaire IMAGE.Nous avons eu des problèmes au début avec nos images, Tkinter ne reconnaissant pas celles-ci.
En cas de sortie prématurée du programme, il arrive (principalement avec Pyscripter) que nous ayons encore des soucis : tkinter ne retrouve plus les images à afficher. Relancer Pyscripter permet de palier au problème. Nous aimerions comprendre exactement les raisons de ce bug.

Notre classe ErgoGui gère toute la partie graphique du jeu.
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.

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

Il y a eu un problème de définition du deck : quand nous avons réalisé la méthode permettant de retourner les parenthèses, la modification s'appliquait à toutes les parenthèses du même type. Une liste en compréhension a résolu le problème.
Les règles du jeux 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.

```
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

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 je le mémorise. Ensuite, je 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).

```
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

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

3) Jeu de l'ordinateur

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.
Ensuite, on détermine l'ensemble des coups possibles en essayant de poser chacune des cartes de la main à un endroit de la preuve.
Pour l'instant, l'ordinateur choisit alors un coup au hasard parmi l'ensemble des coups possibles.

V. Évolutions à venir

- Gérer les scores, et la fin de partie.
- Proposer au(x) joueur(s) de rentrer leurs noms.
- Pas mal de factorisations possibles dans ErgoGui (séparer le canvas dans une classe ErgoCanvas dédiée, fonctions de calcul pour passer des coordonnées dans le canvas à un numéro de prémisse et un numéro de colonne, …)
- Gérer les cartes spéciales (Ergo, Tabula Rasa, …)

```
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)

- 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