

# ERGO – RAPPORT

Delay Emmanuel– Desforêts Nicolas

## Table des matières

<b>I. Présentation du projet</b>	<b>1</b>
1) Présentation du jeu	1
2) Le projet	1
<b>II. Organisation du travail</b>	<b>2</b>
1) Outils utilisés	2
2) Répartition du travail	2
<b>III. Solutions techniques</b>	<b>2</b>
<b>IV. Algorithmes utilisés</b>	<b>2</b>
1) Passage en notation polonaise inversée : algorithme Shunting-yard	2
2) Évaluation de la preuve	2
<b>A Diagramme des classes</b>	<b>5</b>

## 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 Raspberry Pi comme serveur pour installer redmine dessus, en nous aidant beaucoup du [wiki](#) de Frédéric Muller (merci à lui) et des articles de Linux Pratique mis à notre disposition par The Big Boss (loué soit-il). La documentation technique est générée avec Sphinx, encore grâce aux articles de GNU/Linux Magazine que notre Big Boss a eu la bonté de nous fournir (Il n'en sera jamais assez remercié <sup>1</sup>).

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 discussions, 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

TODO Nico

## IV. Algorithmes utilisés

### 1) Passage en notation polonaise inversée : algorithme Shunting-yard

Une des premiers problèmes algorithmiques 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 là-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

Ici, mon idée a été d'attaquer le problème en force brute : tester tous les modèles possibles (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 le 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, et si j'ai bien compris, cette méthode constitue une preuve sémantique, mais c'est équivalent à une preuve syntaxique. Normalement, on devrait voir dans la suite du cours d'autres algorithmes pour répondre à ce problème (le teaser parle d'arbre

---

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

**Entrée :** Une liste input de cartes (propositions ou connecteurs)

**Sortie :** Une liste np<sub>i</sub> correspondant a la notation polonaise inversée de l'entrée

**Traitement**

Créer une pile vide

np<sub>i</sub> ← []

**pour chaque** *carte de input* **faire**

**si** *carte est une lettre* **alors**

        | ajouter *carte* à np<sub>i</sub>

**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 np<sub>i</sub>

**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 np<sub>i</sub>

        empiler *carte*

**tant que** *pile est non vide* **faire**

    | dépiler une *carte* et l'ajouter à np<sub>i</sub>

**si** *la carte est une parenthèse ouvrante* **alors**

        | quitter // Problème de parenthésage

**Algorithme 1 :** Algorithme de passage en notation polonaise inversée

sémantique, d'algorithme de Quine, d'algorithme de Davis et Putnam). Une évolution possible (intéressante?) serait de programmer ces algorithmes.

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

## A Diagramme des classes

