Un projet de LEJEUNE Lucas

IA Pas de Soucis!

Avec une assistance minimale de Evrard Maurice,
Lejeune Grégory,
Mathieu Louca,
Pluvinage Victor
et de Ralet Vincent.

30 décembre 2020

1 Histoire.

1.1 C A VOUS.

2 L'approche logique.

2.1 Les bases de la logique

2.1.1 Introduction à la logique propositionnelle.

L'importance de la logique propositionnelle est immense en mathématiques et en cryptographie, mais également, comme nous allons le voir, en informatique.
Voici une fameuse lapalissade, exemple typique d'utilisation d'une phrase ne découlant de rien d'autre que de cette logique :

"15 minutes avant sa mort, il était encore en vie." [1]

Évidemment, grâce à notre capacité déductionnelle, nous pouvons tous définir cette phrase comme vraie, c'est ici, une vérité dite "de langage".

2.1.2 Un peu de vocabulaire !

La logique propositionnelle possède son propre vocabulaire, il est presque indispensable de connaître son vocabulaire et sa syntaxe afin même de pouvoir en comprendre les concepts.
Tout d'abord, un **langage formel** est un ensemble de mots que l'on peut obtenir en utilisant un alphabet [2], un langage possède plusieurs **lois**, cet ensemble de lois sera appelé la **syntaxe**, qui définissent les différentes manières grâce auxquelles les éléments de l'alphabet (aussi appelés les **symboles**) peuvent se placer pour former quelque chose de cohérent au langage.
Par exemple, en français, vous n'écririez pas "Jème lai vwaturres", mais plutôt, "j'aime les voitures", aussi bête que ce petit exemple puisse paraître, il s'agit là de l'une des nombreuses fois où l'on se plie aux règles d'une grammaire logique.
Un mot respectant toutes les règles de syntaxe sera alors appelé un **mot bien-formé**.
La logique propositionnelle se compose donc d'un langage formel, et de sémantiques donnant du sens aux mots bien formés, répondant au nom de **propositions**.
Les propositions logiques sont désignées par des lettres, comme $A, B, C...$,

1. https ://fr.wikipedia.org/wiki/Lapalissade
2. Cela peut être un alphabet comme abcd...yz tout comme un alphabet composé uniquement de 1 et de 0.

nom	symbole	autre nom
négation	$\neg$	NOT
conjonction	$\wedge$	AND
disjonction (I)	$\vee$	OR
disjonction (E)	$\oplus$	XOR
implication	$\Rightarrow$	IF..THEN
équivalence	$\Leftrightarrow$	IFF

ou par des lettres indicées comme A_2, B_4... Pour relier ces propositions, on utilise des connecteurs, répertoriés dans le tableau ci-dessous.
En plus de ces connecteurs viennent s'ajouter les deux valeurs logiques à la base de tout, Vrai et Faux.

2.1.3 Pour quelques exemples de plus...

Voici quelques exemples d'énoncés de logique propositionnelle dans un cadre assez éloigné des mathématiques, en espérant que cela fasse sens au lecteur.
Supposons que A et B soient deux propositions logiques.
A : Je suis boulanger.
B : Je sais faire des gâteaux.
Nous pouvons ainsi relier ces deux propositions avec les connecteurs vus dans le tableau ci-dessus.
$\neg A$: Je ne suis pas boulanger.
$\neg B$: Je ne sais pas faire des gâteaux.
$A \wedge B$: Je suis boulanger et je sais faire des gâteaux.
$A \vee B$: Je suis boulanger ou je sais faire des gâteaux (Les deux propositions peuvent être vraies, tout comme une seule des deux).
$A \oplus B$: Je suis boulanger ou alors je sais faire des gâteaux. (Une seule de ces deux propositions doit être vraie)
$A \Rightarrow B$: Je suis boulanger, donc je sais faire des gâteaux.
$A \Leftrightarrow B$: Je suis boulanger si et seulement si je sais faire des gâteaux.
...

2.1.4 Quelques tables de vérité.

Après tant d'exemples "instructifs", il serait temps de passer aux fameuses **tables de vérité**, ces tables seront d'une importance capitale lors de résolutions de problèmes, les voici donc :

Table 1 – Négation.

A	$\neg A$
F	V
V	F

Table 2 – Conjonction.

A	B	$A \wedge B$
F	F	F
F	V	F
V	F	F
V	V	V

Table 3 – Disjonction (OR).

A	B	$A \vee B$
F	F	F
F	V	V
V	F	V
V	V	V

Table 4 – Disjonction (XOR).

A	B	$A \oplus B$
F	F	F
F	V	V
V	F	V
V	V	F

Table 5 – Implication.

A	B	$A \Rightarrow B$
F	F	V
F	V	V
V	F	F
V	V	V

Table 6 – Équivalence

A	B	$A \Leftrightarrow B$
F	F	V
F	V	F
V	F	F
V	V	V

Les tables 1 à 4 doivent sans doute paraître logique, je m'attarderai toutefois sur la table 5, en effet les deux phrases $F \Rightarrow V$ et $F \Rightarrow F$ sont toutes deux vraies. Cela est du au **principe d'explosion** [3] : Du faux, on peut déduire absolument n'importe quoi ! (On verra plus tard que l'on peut noter ceci $A \wedge \neg A \models B$).
Pour ce qui est de la table 6, il faut savoir que certains notent $\Leftrightarrow$ comme étant $\equiv$, cette notation a l'avantage d'accentuer le fait que cette relation n'est autre que la relation d'équivalence.

2.1.5 Résolution de problèmes en logique propositionnelle.

Maintenant que nous avons acquis les bases de la logique propositionnelle, attaquons nous à quelques problèmes.
En voici un premier,

$$\boxed{\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B}$$

Pour résoudre ceci, nous allons utiliser une grande table de vérité :

3. https ://fr.wikipedia.org/wiki/Principe_d%27explosion

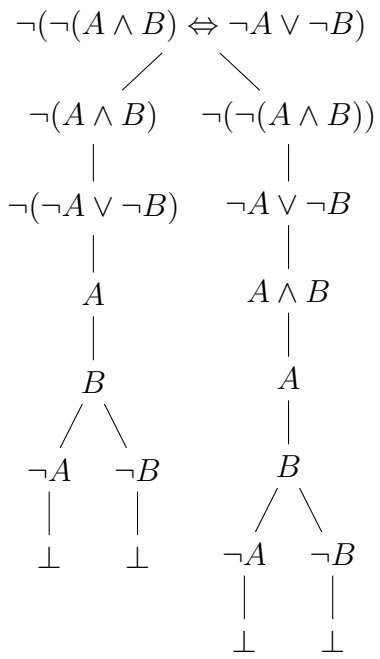
A	B	$A \wedge B$	$\neg(A \wedge B)$	$\neg A$	$\neg B$	$\neg A \vee \neg B$	$\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$
F	F	F	V	V	V	V	V
F	V	F	V	V	F	V	V
V	F	F	V	F	V	V	V
V	V	V	F	F	F	F	V

Ainsi, comme nous pouvons le constater, la dernière ligne est remplie de "V", cela veut donc dire que nous venons de prouver la relation $\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$, aussi connue sous le nom de "la loi de DeMorgan", nous avons prouvé par la même occasion que $\neg(A \wedge B) \Leftrightarrow \neg A \vee B$ est **valide**, cela veut dire qu'elle sera toujours vraie, peu-importe les A, et les B.

Il existe toutefois une autre manière de faire, il s'agit d'utiliser un arbre, cette méthode requiert moins d'étapes et nous permettra de résoudre certains problèmes de manière plus simple. Toutefois, il y a quelques règles à respecter lors de l'utilisation d'un arbre, ces règles sont répertoriées dans le tableau ci-dessous.

$\neg(\neg\phi)$ $\vert$ ϕ	$\phi \wedge \psi$ $\vert$ ϕ $\vert$ ψ	$\neg(\psi \wedge \phi)$ $/ \quad \backslash$ $\neg\psi \quad \neg\phi$
$\phi \vee \psi$ $/ \quad \backslash$ $\phi \quad \psi$	$\neg(\phi \vee \psi)$ $\vert$ $\neg\phi$ $\vert$ $\neg\phi$	$\phi \Rightarrow \psi$ $/ \quad \backslash$ $\neg\phi \quad \psi$
$\neg(\phi \Rightarrow \psi)$ $\vert$ ϕ $\vert$ $\neg\psi$	$\phi \Leftrightarrow \psi$ $/ \quad \backslash$ $\phi \quad \neg\phi$ $\vert \quad \vert$ $\psi \quad \neg\psi$	$\neg(\phi \Rightarrow \psi)$ $/ \quad \backslash$ $\phi \quad \neg\phi$ $\vert \quad \vert$ $\neg\psi \quad \psi$

Le but du jeu avec un arbre logique, c'est de terminer chacune des branches de l'arbre par $\perp$ (C'est le symbole utilisé pour les contradictions). Ainsi, avec un arbre, nous commençons par utiliser l'opposé de notre hypothèse de base (ici, l'opposé de $\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$, c'est $\neg(\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B)$), (en général, une contradiction arrive quand nous nous retrouvons avec A et $\neg A$ sur la même branche.)
Passons maintenant à la preuve



Chaque branche de l'arbre fini bien par $\perp$, nous venons donc de prouver la loi de DeMorgan, avec l'aide de notre arbre de démonstration.

Ci dessous, le lecteur pourra s'essayer à la démonstrations de certaines lois logiques célèbres, avec l'aide d'un tableau ou avec un arbre.

$$\neg\neg A \Leftrightarrow A \tag{2.1.1}$$

$$A \wedge \neg A \Leftrightarrow F \tag{2.1.2}$$

$$A \vee \neg A \Leftrightarrow T \tag{2.1.3}$$

$$A \wedge F \Leftrightarrow F \tag{2.1.4}$$

$$A \vee V \Leftrightarrow V \tag{2.1.5}$$

$$\begin{cases} A \vee B \Leftrightarrow B \vee A \\ A \wedge B \Leftrightarrow B \wedge A \end{cases} \tag{2.1.6}$$

$$\begin{cases} A \wedge (B \wedge C) \Leftrightarrow (A \wedge B) \wedge C \\ A \vee (B \vee C) \Leftrightarrow (A \vee B) \vee C \end{cases} \tag{2.1.7}$$

$$A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C) \tag{2.1.8}$$

2.2 Les ensembles de propositions.

Les ensembles de propositions, comme leur nom l'indique, sont des ensembles mathématiques, composés de **formules logiques**. Ces formules sont dites soit :

- **consistantes**, signifiant qu'il est possible d'en tirer du Vrai, par exemple $A \wedge B$ ou encore $\neg\neg A \Rightarrow A$.
- **inconsistances**, signifiant que l'on ne peut en tirer que du Faux, par exemple $A \wedge \neg A$, ces formules peuvent être notées $\bot$.
- **valides**, signifiant qu'elles ne peuvent être que Vraies, comme $A \vee \neg A$ (principe du tiers-exclus), une formule **valide** est par définition toujours **consistante**, on appelle bien souvent ces formules valides des **tautologies**, ces formules pourront être notées $\top$.

- **contingentes**, impliquant que l'on peut tirer de la formule du faux, tout comme du vrai, une **tautologie** ne peut pas être formule contingente, un exemple de formule contingente serait $A \vee B$.

Les ensembles aussi ont leur propre terminologie, ainsi, si l'on prend l'ensemble noté S, il pourra être qualifié également de **consistant**, si il n'y a pas de contradictions au sein de S et qu'il n'y a aucune formule inconsistante contenue dans S, autrement S sera défini comme étant **inconsistant**.

Il y a plusieurs manières d'**inférer** quelque chose d'un ensemble logique.
Une première manière est de prendre les **prémisses** qui nous intéressent, et d'écrire

1. **Premisse_A**
2. **Premisse_B**
⋮
n. **Premisse_X**

$\therefore$ **Conclusion**

Prenons un ensemble **consistant** S composé des formules A et $A \Rightarrow B$.
On pourrait alors noter S comme étant $S = \{A, A \Rightarrow B\}$ (ce qui est parfaitement équivalent à écrire $S = A \wedge (A \Rightarrow B)$),
De ceci, nous allons utiliser l'opérateur de la déduction, $\vDash$, ceci nous permettra ainsi écrire $S \vDash A$, littéralement "De S, nous déduisons A".
Ce principe est encore plus flagrant quand nous utilisons la notation suivante : $A, A \Rightarrow B \models A$, où $A, A \Rightarrow B$ n'est autre que l'ensemble S, avec une notation légèrement différente.

Je n'ai pas choisi cet ensemble de manière anodine, car, grâce à celui-ci, nous allons pouvoir utiliser une **règle d'inférence** connue sous le nom du MP (Modus Ponens), le lecteur ne devrait toutefois pas s'inquiéter, un tableau recensant d'autres règles d'inférence sera présenté à la page suivante.
Une première manière de noter cette règle d'inférence serait de faire usage de la notation que nous avons vue plus haut.

1. A (De cet ensemble, nous savons A)
2. $A \Rightarrow B$ (De cet ensemble, nous savons que $A \Rightarrow B$)

$\therefore$ B

Une autre manière serait d'utiliser l'opérateur $\vDash$ (celui de la **déduction**), de la manière suivante : $A, A \implies B \vDash B$, nous pourrions même être tentés d'utiliser la **règle d'addition**, disant que si l'on a $S \vDash B$, alors, on peut rajouter la formule B à S. Et ainsi, notre ensemble S de base pourra être réécrit en $S = A, A \Rightarrow B, B$ Et maintenant, comme promis, voici un tableau comprenant toutes les règles d'inférence qui seront bien pratiques pour travailler avec les ensembles logiques.

Règle d'inférence	Tautologie	Nom de la règle d'inférence
$A, B \vDash A \wedge B$	$A \wedge B \Rightarrow A \wedge B$	Loi de combinaison
$A, B \vDash A$	$(A \wedge B) \Rightarrow A$	Loi de la simplification
$A, A \Rightarrow B \vDash B$	$A \wedge (A \Rightarrow B) \Rightarrow B$	Modus Ponens
$\neg B, A \Rightarrow B \vDash \neg A$	$\neg B \wedge (A \Rightarrow B) \Rightarrow \neg A$	Modus Tollens
$A \Rightarrow B, B \Rightarrow C \vDash A \Rightarrow C$	$(A \Rightarrow B) \wedge (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$	Syllogisme hypothétique
$A \vee B, \neg A \vDash A$	$(A \vee B) \wedge \neg A \Rightarrow B$	Syllogisme disjonctif
$A \Rightarrow B \vDash A \Rightarrow (A \wedge B)$	$(A \Rightarrow B) \Rightarrow (A \Rightarrow (A \wedge B)$	Règle d'absorption
$A \Rightarrow B, C \Rightarrow B, A \vee C \vDash B$	$(A \Rightarrow B) \wedge (C \Rightarrow B) \wedge (A \vee C) \Rightarrow B$	Elimination disjonctive
$A, \neg A \vDash B$	$A \wedge \neg A \Rightarrow B$	Principe d'explosion

Le lecteur pourra s'exercer à démontrer la validité des tautologies présentées dans le tableau-ci dessus en tant qu'exercice.

2.2.1 Le besoin d'algorithme, présentation de l'algorithme de Quine.

Un ensemble de formules, tout comme une formule peut-être inconsistant, cela signifie que notre ensemble est équivalent à F,et comme nous l'avons vu dans le tableau ci-dessus, on peut déduire absolument n'importe quoi d'un ensemble inconsistant.

Le problème des tableaux de vérités, c'est qu'ils prennent de la place, et même, beaucoup de place. C'est pour cela qu'est venu la nécessité de créer des algorithmes de résolution d'ensembles de formules, afin de faciliter le travail des logiciens, et ainsi, de réduire le temps de calcul nécessaire à un ordinateur.

L'algorithme de Quine se déroule en plusieurs étapes, tout d'abord, nous allons simplifier, si possibles, les formules logiques contenues dans notre ensemble de formules, par exemple, au lieu d'écrire $A \vee V$, nous pourrons écrire simplement V, et ainsi, supprimer ce V de notre ensemble, comme $B, V \models B$, il y a bien d'autres simplifications possibles, nous laisserons au lecteur le soin d'en trouver.

Une fois cette première étape passée, il suffit d'utiliser un arbre logique (ou éventuellement une table de vérité) et enfin, nous avons pu prouver des formules logiques avec l'aide de l'agorithme de Quine.

2.3 Une logique ? Des logiques !

Les logiciens, non-contents de la seule logique propositionnelle, ont créée un très grand nombre de logique, j'en liste quelques une ci-dessous.

- La logique linéaire, créée par un français, elle est basée sur la gestion de ressources, c'est une des nombreuses logiques n'excluant pas le tiers exclus, en effet, en logique linéaire, $A \vee \neg A \not\models V$, cela est simplement du au fait que nous "utilisons" la ressource A une fois, elle n'existe donc plus réellement, et donc $\neg A$ n'existe pas comme A est devenu indéterminé.
- La logique floue, dans laquelle une proposition est vraie selon un certain degré de probabilité.
- La logique de premier ordre, ou logique des prédicats, cette logique sera abordée dans le chapître sur Prolog.
- La logique booléenne, elle est basée sur les portes logiques, les circuits logiques, et les ensembles.
- La logique combinatoire, logique inventée pour formaliser la notion de fonction, et pour limiter le nombre d'opérateurs nécessaires pour définir le calcul des prédicats.
- La logique modale, ayant recours à des opérateurs comme "il est nécessaire que" ou "il est possible que".
- Et bien d'autres...

2.4 Introduction à Prolog et à la logique des Prédicats.

Dans cette nouvelle sous-section, nous allons nous intéresser à la logique des prédicats, connue également sous le nom de logique de premier ordre.
Tout d'abord, en logique des prédicats, nous aurons besoin de deux nouveaux **quantificateurs**.
Ceux-ci sont le quantificateur **universel**, noté $\forall$ [lisez "pour tout"], et le quantificateur **existentiel**, noté $\exists$ [lisez "il existe"].
A ces deux quantificateurs viennent s'ajouter :

- Des **connecteurs logiques**, qui ont été discutés dans la section précédente.
- Des **constantes**, celles-ci représentent un événement, une personne ou un objet en particulier, nous noterons ces constantes avec une majuscule comme première lettre et un nombre à la fin, par exemple "Turing_1" ou encore "Chaise_2".
- Des **variables**, celles-ci représentent un concept général ou un ensemble, par exemple, l'ensemble des mathématiciens, ou encore l'ensemble des chaises dans le monde. Nous noterons ces variables en minuscules, par exemple "mathématiciens" ou encore "chaise".
- Des **prédicats**, ceux-ci nous permettent d'établir des liens entre nos différentes variables et constantes, nous noterons nos prédicats avec une majuscule en première lettre, par exemple "**Mortel**(x)" ou encore "**Humain**(x)".
- Des **fonctions**, qui ont pour but de retourner une valeur, pouvant-être autre chose que vrai ou faux. Nous noterons celles-ci en toutes minuscules.

2.4.1 Exemples.

Pour se faire une bonne idée, voici quelques phrases françaises "traduites" en logique des prédicats.

1. Tout les mathématiciens sont cools.
$\Rightarrow \forall x$ (**Mathématicien**(x) $\Rightarrow$ **Cool**(x)).

2. Alan Turing et Alonzo Church sont des mathématiciens.
$\Rightarrow$ **Mathématicien**(Turing_1) $\wedge$ **Mathématicien**(Church_1).

3. Il y a des chats qui ne sont pas noirs.
$\Rightarrow \exists x$ (**Chat**(x) $\wedge \neg$**Noir**(x)).

2.4.2 Qu'est-ce que PROLOG ?

Cette petite introduction passée, concentrons-nous maintenant sur le coeur du sujet : PROLOG !

Prolog a été inventé en 1972 par les informaticiens français Alain Colmerauer et Philippe Roussel.
C'est un langage de programmation **logique** et son nom est un acronyme pour PROgrammation LOGique.
Prolog a été très utilisé en Europe et au Japon dans le domaine de l'Intelligence Artificielle, tout en étant basé sur la logique propositionnelle dont nous avons posé les bases juste au dessus.
Il existe de nombreuses distributions de PROLOG [4], nous utiliserons ici SWI-PROLOG, avant tout pour son côté open-source et gratuit.

2.4.3 Introduction à la syntaxe de Prolog

En Prolog, contrairement aux règles que nous avions établies en logique des prédicats, les constantes (ici appelés Atomes) doivent commencer par une minuscule. Les variables commencent par une majucule. A cela viennent s'ajouter les listes, dénotées par des [].

Faits et Règles.

En PROLOG, un fait s'écrit simplement

Un fait n'a pas de "corps", et tiendra toujours. Dans ce cas-ci, cela veut dire que Turing est un mathématicien, quoiqu'il arrive ce **fait** ne changera pas.

Maintenant, si nous essayons de reformuler l'exemple n°1 de notre dernière section, "tous les mathématiciens sont cools" en Prolog, cela donne ceci :

On remarque tout de suite que cela est plutôt facile à lire, de plus, si nous ouvrons notre interprète Prolog, voilà ce que nous obtenons :

Tout cela est très bien, mais, si j'essaye de demander à prolog si alonzo_church est cool, que se passe-t-il ?

Pas grand chose comme nous le constatons, PROLOG préfère éviter de faire

4. https ://en.wikipedia.org/wiki/Comparison_of_Prolog_implementations

la moindre assomption, et répondra “Faux” dès qu'il ne sait pas.

Ensuite, Prolog nous permet de faire de l'arithmétique, regardons un peu cela, avec une fonction dont le seul rôle est d'additionner deux nombres :

Ouvrons maintenant l'interprète et regardons la magie opérer :

Mais, si pour une obscure raison, je décidais de vouloir avoir B, juste en entrant A et C, comment faire ? Regardons d'abord comment notre première version réagirait à cela :

Heureusement pour nous, Prolog possède un module basé sur la logique par contraintes, pour l'utiliser, il suffit d'ajouter [5]

au dessus de votre fichier Prolog, maintenant, modifions légèrement notre prédicat

Et voilà ! Essayons là maintenant avec notre interprète :

2.4.4 Mon arbre familial, avec Prolog !

```
% Ecrivons quelques faits.
fils(gaia, ouranos, coeos).
fils(gaia, ouranos, cronos).
fils(gaia, ouranos, oceaonos).
fils(gaia, ouranos, japet).
fils(rhea, cronos, hera).
fils(rhea, cronos, hades).
fils(rhea, cronos, poseidon).
fils(rhea, cronos, zeus).
fils(hera, zeus, ares).
fils(hera, zeus, hephaistos).
fils(aphrodite, ares, phobos).
fils(aphrodite, ares, deimos).
fils(clymene, japet, atlas).
fils(maia, zeus, hermes).
fils(leto, zeus, appolon).

fille(gaia, ouranos, rhea).
fille(gaia, ouranos, phebee).
fille(phebee, coeos, leto).
```

5. https ://www.swi-prolog.org/man/clpfd.html

```
fille(phebee, coeos, asteria).
fille(tethys, oceanos, metis).
fille(tehtys, oceanos, dione).
fille(asteria, perses, hecate).
fille(rhea, cronos, hestia).
fille(rhea, cronos, demeter).
fille(rhea, cronos, hera).
fille(hera, zeus, hebe).
fille(hera, zeus, ilithye).
fille(metis, zeus, athena).
fille(demeter, zeus, persephone).
fille(aphrodite, ares, harmonie).
fille(pleionee, atlas, maia).
fille(leto, zeus, artemis).

% Maintenant, réflechissons à quelques lois.
parents(X, Y, Z) :-
    fille(X, Y, Z) ;
    fils(X, Y, Z).

grands_parents(X, Y, Z) :-
    parents(X1, Y1, Z),
    (   parents(X, Y, X1)
    ;   parents(X, Y, Y1)
    ).

frere(A, B) :-
    dif(A, B),
    fils(X, Y, A),
    parents(X, Y, B).

soeur(A, B) :-
    dif(A, B),
    fille(X, Y, A),
    parents(X, Y, B).

oncle(A, B) :-
    parents(X, Y, A),
    (   frere(B, X)
    ;   frere(B, Y)
    ).


tante(A, B) :-
    parents(X, Y, A),
    (   soeur(B, X)
    ;   soeur(B, Y)
    ).
```

```
fos(A, B) :-
    frere(A, B) ;
    soeur(A, B).


cousin_m(A, B) :-
    parents(X, _, A),
    fos(X, X1),
    (   parents(X1, _, B)
    ;   parents(_, X1, B)
    ).

cousin_p(A, B) :-
    parents(_, Y, A),
    fos(Y, Y1),
    (   parents(Y1, _, B)
    ;   parents(_, Y1, B)
    ).

cousin(A, B) :-
    cousin_m(A, B) ;
    cousin_p(A, B).
```

3 L'approche algorithmique.

3.1 Qu'est-ce qu'un algorithme ?

Un algorithme est une suite d'instructions permettant de résoudre un problème.

Il faut savoir que nous utilisons des algorithmes bien plus souvent que ce que l'on pourrait croire. Par exemple, quand vous préparez un gâteau pour célébrer une quelconque occasion, vous aurez besoin d'une recette. Cette recette n'est autre que l'algorithme aidant à la préparation du gâteau, chaque étape de la recette n'étant qu'une instruction faisant partie de l'algorithme.

Un des tout premiers exemples d'algorithmes est "l'agorithme d'Euclide", celui-ci permettait de trouver le PGCD de deux nombres.
Voici comment celui-ci fonctionne :

- Tout d'abord, nous prenons deux nombres a et b.
- Si b = 0 alors, nous retournons a comme étant le pgcd.
- Sinon, nous écrivons que **pgcd**(a, b) = **pgcd**(b, a **mod** b)

Et en voici sa traduction en Common Lisp, le langage de programmation que nous utiliserons dans cette partie du dossier.

```
(defun pgcd (a b)
  (if (zerop b)
      a
      (pgcd b (mod a b))))
```

3.1.1 La complexité des algorithmes

Evidemment, tout les algorithmes ne se valent pas, certains sont bien plus lents que d'autres, et ce, pour une seule et même tâche.
Pour juger de la **complexité** d'un algorithme, nous utilisons la notation de Landau (dite du "Big-O" en anglais).
Cette notation a pour but d'estimer l'évolution du nombre d'opérations qui seront effectuées par l'algorithme au cours du temps. Car au plus l'algorithme effectue d'opérations, au plus il sera couteux en temps, ce qui n'est pas pour nous arranger.

Pour montrer l'importance d'avoir des algorithmes performants, nous avons implémentés en Lisp deux algorithmes de tris différents :

— Le **tri à bulles** de complexité $O(n^2)$
— Le **tri par fusion** de complexité $O(n \log n)$

Le **tri à bulles** fonctionne de la manière suivante :
— Prendre les deux premiers éléments de la liste.
— Si le premier est plus grand que le second, les échanger, sinon, les laisser en place.
— Faire de même jusqu'à la fin de la liste.
— Une fois arrivé à la fin de la liste, reprendre à partir du début de la liste.
— Continuer ce procédé jusqu'à ce que la liste soit parfaitement rangée.

Il est implémenté de la manière suivante en Common Lisp :

```
(defun bubble-sort (lat)
  "Implémentation non-récursive, avec copie,
   de l'algorithme du tri à bulles."
  (let ((arr (make-array (length lat) :initial-contents lat
   )))
    (loop for i from (1- (length arr)) downto 1
            do (loop for j from 0 to (1- i)
                     when (< (aref arr (1+ j)) (aref arr j))
                         do (rotatef (aref arr (1+ j))
                                     (aref arr j))))
    arr))
```

Le **tri par fusion** marche de manière assez différente :
— Si le tableau n'a qu'un seul élément, il est considéré comme déjà trié.
— Si il a plus d'un élément, séparer le tableau en deux parties à peu près égales.
— Trier les deux parties ainsi séparées.
— Fusionner les deux tableaux triés en un seul tableau trié.

La fonction merge étant prédéfinie en common Lisp, nous implémenterons notre tri par fusion ainsi.

```
(defun merge-sort (arr)
  "Implémentation du tri par fusion,
en utilisant la fonction merge, prédéfinie."
  (if (<= (length arr) 1)
      arr
      (let* ((middle (floor (length arr) 2))
```

```
            (left (subseq arr 0 middle))
            (right (subseq arr middle (length arr))))
        (merge 'list (merge-sort left)
               (merge-sort right) #'<))))
```

Passons maintenant aux **tests de performances**.
Afin de mesurer le temps pris par chacun de mes deux algorithmes, j'ai utilisé "time", une macro bien pratique pour ce genre de tests. J'utilise le compilateur SBCL pour faire mes tests. En plus de cela, dans un soucis de facilité, j'utilise la macro random-sample, créée par mes soins, me permettant de créer une liste contenant des nombres aléatoires, de taille fixée.

```
;;; Ma macro random-sample bien pratique.
(defmacro random-sample (x)
   (loop for _ below ,x collect (random 1000)))
```

Sans plus attendre, voici le tableau montrant les résultats de notre petite expérience.

Nombre d'éléments à triers.	Temps Tri à Bulles.	Temps Tri par fusion.
10	0.0000008 secondes	0.000016 secondes
100	0.000089 secondes	0.000140 secondes
1000	0.007863 secondes	0.001188 secondes
10,000	0.318904 secondes	0.012558 secondes
100,000	31.164620 secondes	0.090092 secondes

Pour des raisons pratiques, je n'ai pas pu obtenir les résultats du Tri à Bulles pour le million d'éléments.
Comme nous pouvons le constater sur le tableau précédent, la complexité d'un algorithme (estimé par le Big-O) ne mesure pas le temps exact que l'algorithme prendra afin de trier une certaine liste, il s'agit simplement d'un indicateur, montrant comment évoluera le nombre de comparaisons au cours du temps, cet indicateur devenant extrêmement siginificatif quand n devient très grand.

3.2 Présentation de structures de données de base

Pour produire une intelligence artificielle efficace, il nous faut de bons algorithmes, mais il nous faut également les structures de données adéquates.
Tout d'abord, qu'est-ce qu'une structure de données ? D'après Wikipedia [6],

6. https ://fr.wikipedia.org/wiki/Structure_de_donn%C3%A9es

une structure de données est une manière d'organiser les données, pour les traiter plus facilement.
On conclut donc que le choix judicieux d'une structure de données est indispensable à l'optimisation de la performance de notre IA.

Je présente ici trois structures de données de base.

a. Les listes simplement chaînées.

b. Les tableaux (ou arrays en anglais.)

c. Les tables de hachages (ou les dictionnaires.)

3.2.1 Les listes simplement chaînées.

Une **liste simplement chaînée** est une structure de données pouvant contenir plusieurs éléments. Chaque élément appartenant à la liste chaînée contient deux choses :

a. La valeur de l'élément.

b. Un pointeur pointant vers l'élément suivant.

Ainsi, nous pouvons représenter une liste simplement chaînée de la manière suivante

En Lisp, une liste simplement chaînée peut être crée de la manière suivante.

```
(defparameter *my-linked-list* '(1 2 3 4 5))
```

Ainsi, si nous reprenions le dessin que nous avions utilisé ci-dessus, *my-linked-list* ressemblera donc à ceci :

D'ailleurs, pour pouvoir travailler avec des listes chaînées, deux opérateurs bien pratiques s'offrent à nous : "car" et "cdr".

```
CL-USER> (car *my-linked-list*)
1
CL-USER> (cdr *my-linked-list*)
(2 3 4 5)
```

Nous pouvons également chaîner nos opérateurs “car” et “cdr”, afin de nous assurer un contrôle total sur notre liste simplement chaînée.

```
CL-USER> (car (cdr *my-linked-list*))
2
CL-USER> (cadr *my-linked-list*) ; Il existe aussi une mani
    ère abrégée.
2
```

Il existe également la fonction “nth”, permettant de récupérer le n-ième élément d’une liste chaînée. (Il ne faut pas oublier que l’indexing marche comme pour les tableaux, et commence à 0.)

```
CL-USER> (nth 2 *my-linked-list*)
3
```

3.2.2 Les tableaux.

Le tableau est une structure extrêmement importante en programmation, celle-ci nous permet de stocker un nombre fixe de données, et nous permet d’accéder à ces données de manière rapide. Chaque élément du tableau se retrouve collé en mémoire aux éléments qui lui sont adjacent, ainsi, pas besoin de traverser chaque élément du tableau lorsque l’on désire accéder au n-ième élément du tableau.

Voici comment créer un tableau en Lisp :

```
CL-USER> (defparameter *my-array* #(1 2 3 4 5))
```

Il existe également une manière alternative :

```
CL-USER> (defparameter *my-array* (make-array 5 :
  initial-contents '(1 2 3 4 5)))
```

Pour récupérer le n-ième élément d’un tableau, il nous suffit de faire :

```
CL-USER> (aref *my-array* 2)
3
```

3.2.3 Tableaux vs Listes simplement chaînées.

Pour en conclure avec ces deux structures de données, je me propose de faire un petit tableau comparatif final, afin que le lecteur puisse mieux comprendre les différennces entre Tableau, et Liste simplement chaînée, (que je noterai SLL pour Single-linked list).

	Tableaux	SLL
Accession	O(1)	O(n)
Insertion	O(n)	O(1)
Délétion	O(n)	O(1)

Comme on le constate sur ce tableau, les arrays ont un réel avantage lorsque nous désirons uniquement lire un élément de notre structure de données, toutefois, si nous désirons modifier la structure de données en elle-même, alors, tout devient plus problématique, et cela, car notre tableau devra retrouver un autre emplacement libre pour s'y mettre en mémoire.
La liste chaînée quant à elle, n'a pas besoin de se repositionner entièrement lorsqu'on lui ajoute ou qu'on lui retire un élément, il lui suffit plutôt de rajouter un pointeur vers le nouvel élément, ce qui se fait en temps constant (O(1)), peu importe la taille de la liste. Toutefois, la liste chaînée a pour inconvénient d'être en croissance linéaire lors de la recherche d'un élément dans la liste, en effet, lors de la recherche d'un élément dans une liste chaînée, le programme devra d'abord passer par tout les éléments se trouvant avant celui que l'on cherche.
Pour démontrer l'importance d'utiliser la bonne structure au bon endroit, nous allons reprendre notre bon tri à bulles.
Pour ceux qui en avaient oublié l'implémentation sur les tableaux, la voici :

```
(defun bubble-sort (lat)
  "Implémentation non-récursive, avec copie,
   de l'algorithme du tri à bulles."
  (let ((arr (make-array (length lat) :initial-contents lat
   )))
    (loop for i from (1- (length arr)) downto 1
            do (loop for j from 0 to (1- i)
                     when (< (aref arr (1+ j)) (aref arr j))
                       do (rotatef (aref arr (1+ j))
                                   (aref arr j))))
    arr))
```

Le lecteur attentif remarquera l'utilisation dans ce code de "make-array" et de "aref", tout deux synonymes de l'utilisation d'un tableau. Maintenant, transformons cette implémentation en une implémentation utilisant des listes simplement chaînées.
Et maintenant, voici le même algorithme, mais, sur les listes simplement chaînées cette fois-ci.

```
(defun sll-bubble-sort (lat)
  "Implémentation non-récursive, avec copie,
   de l'algorithme du tri à bulles,
  sur des listes simplement chaînées"
```

```
(let ((lat-copy (copy-list lat)))
  (loop for i from (1- (length lat-copy)) downto 1
        do (loop for j from 0 to (1- i)
                 when (< (nth (1+ j) lat-copy) (nth j
 lat-copy))
                   do (rotatef (nth (1+ j) lat-copy)
                               (nth j lat-copy))))
 lat-copy))
```

Et maintenant, comparons leurs temps respectifs sur des listes de tailles différentes avec l'aide d'un nouveau tableau, (mon mode opératoire reste le même, j'utilise SBCL, et la macro time, pour créer une liste contenant des nombres aléatoires, j'utilise toujours ma macro random-sample.)

	Tri à bulles (Tableaux)	Tri à bulles (SLL)
10	0.0000008 secondes	0.000009 secondes
100	0.000089 secondes	0.002111 secondes
1000	0.007863 secondes	0.750918 secondes
10000	0.318904 secondes	888.190640 secondes
100000	31.164620 secondes	>1000 secondes

La différence de performance entre nos deux implémentations du même algorithme est énorme, et pourtant, nous n'avons pas changé l'algorithme en lui-même ! J'espère que notre petite expérience montre bien au lecteur l'importance d'un choix de structure de données adéquat.

3.2.4 Les tables de hachages.

Une table de hachage (ou un dictionnaire en python) est une des structures de données les plus utiles en ce qui concerne l'optimisation d'algorithmes.
Son utilisation est simple ; avec une table de hachage, nous relions des clés avec des valeurs.
Ainsi, si dans une table de hachage, je relie le mot “bonjour” avec le mot “hello”, mon mot “bonjour” sera considéré comme étant la clé, et le mot “hello” comme étant la valeur.
En Lisp, pour créer une telle table de hachage, il faut faire :

```
CL-USER> (defparameter *my-dict* (make-hash-table))
*MY-DICT*
CL-USER> (setf (gethash 'bonjour *my-dict*) 'hello)
HELLO
CL-USER> (gethash 'bonjour *my-dict*)
HELLO
```

Ce procédé étant toutefois un peu fastidieux, nous utiliserons ici la bibliothèque lisp connue sous le nom de serapeum.

```
CL-USER> (ql:quickload :serapeum) ;; Ici, nous déclarons la
    bibliothèque.
To load "serapeum":
  Load 1 ASDF system:
    serapeum
; Loading "serapeum"
.
Switching to the BALLAND2006 optimizer

(:SERAPEUM)
CL-USER> (defparameter *my-dict* (serapeum:dict 'bonjour '
   hello))
*MY-DICT*
CL-USER> (gethash 'bonjour *my-dict*)
HELLO
```

Pour montrer un cas pratique d'utilisation de tables de hachages, nous allons créer un mini-programme qui permet de convertir des phrases en morse.
Ce programme se fait en deux parties, tout d'abord, nous définissons une table de hachage qui nous remet la traduction morse de chaque lettre et de chaque chiffre.

```
(ql:quickload :serapeum)

(defparameter *latin->morse*
  (serapeum:dict #\a "._" #\b "_..." #\c "_._." #\d "_.."
                 #\e "." #\f ".._." #\g "__." #\h "...."
                 #\i ".." #\j ".___" #\k "_._" #\l "._.."
                 #\m "__" #\n "_." #\o "___" #\p ".__."
                 #\q "__._" #\r "._." #\s "..." #\t "_"
                 #\u ".._" #\v "..._" #\w ".__" #\x "_.._"
                 #\y "_.__" #\z "__.." #\1 ".____" #\2 "..
   ---"
                 #\3 "...__" #\4 "...._" #\5 "....." #\6 "_
   ...."
                 #\7 "__..." #\8 "___.." #\9 "____." #\0 "
   -----"
                 #\SPACE " "))
```

Voici notre dictionnaire ! Ne soyez pas surpris par les "#', c'est ainsi que nous définissons des charactères en common lisp.
Maintenant, la deuxième partie consistera simplement à créer une fonction chargée de transformer nos phrases vers du morse.

```
(defun convert-to-morse (sentence)
  "Cette fonction s'occupe de convertir
nos phrases vers du code Morse, grâce au dictionnaire
créé ci-dessus!"
  (let ((converted-list
          (loop for letter across sentence
                collect (gethash letter *latin->morse*))))
  (format nil "~{~A~}" converted-list)))
```

Il ne nous reste plus qu'à tester.

```
CL-USER> (convert-to-morse "ceci est une phrase")
"-.-.-..-.... ..... ..--.. .--.....-..-...."
```

Et voilà comment notre programme marche !
Petite parenthèse au niveau de la complexité de notre fonction "convert to morse". Tout d'abord, il faut savoir que pour chercher une valeur avec une clé dans un dictionnaire, la complexité temporelle est de O(1) ! Peu importe la taille du dictionnaire, cela prendra toujours aussi peu de temps de chercher une clef dedans, c'est la raison pour laquelle ceux-ci sont très utilisé de nos jours. Pour les lecteurs intéressés, nous conseillons fortement de vous renseigner sur la fonction SHA, c'est grâce à cette fonction de hachage que les dictionnaires marchent aussi bien.
Ainsi, pour en revenir à notre programme, nous appliquons une opération de complexité O(1) sur chaque élément de notre liste, soit, n fois. On peut donc en déduire que notre fonction "convert-to-morse" est donc de complexité O(n), car nous appliquons n fois une opération de complexité O(1).

3.2.5 Promenons-nous dans les bois.

Pour comprendre l'intérêt des arbres dichotomiques en programmation, intéressons nous à un petit problème assez simple.
Imaginez que je pense à un nombre, compris entre 1 et 1000, et que vous deviez deviner ce nombre avec le moins d'essais possibles, à la même manière que dans le jeu du "Juste Prix", je vous dirai si votre essai est trop grand, ou trop petit.
La première stratégie évidente, serait de commencer par 1, puis ensuite, si mon nombre n'est pas 1, essayer avec 2, puis ensuite, essayer avec 3, et ceci, jusqu'à ce que vous trouviez le nombre auquel je pense.
Cette première stratégie pourrait faire penser à la manière à laquelle il faut chercher un élément dans une liste simplement chaînée, en effet, si je pense au nombre 629, il va falloir passer par les 628-ièmes éléments se trouvant avant. Dans ce cas-ci, comme pour lire un élément dans une liste simplement

chaînée, nous dirons que la complexité de cet algorithme sera de O(n).
Une deuxième manière de faire serait de faire ce que l'on appelle une recherche dite "par dichotomie". Tout d'abord, prenons un nombre au milieu entre 1 et 1000, ici, ce sera 500, si je dis "au dessus", alors, il suffira de prendre le milieu entre 500 et 1000, ici, ce sera 750, désormais, si je dis "en dessous", il faudra alors faire $\frac{500+750}{2} = 625$, ainsi, en appliquant cet algorithme jusqu'au bout, vous trouverez le nombre auquel je pense en utilisant un algorithme de complexité O(log n) !
Ainsi, vous trouverez grâce à cette technique le nombre auquel je pense en maximum $\log_2 n$ essais (où n est le nombre maximum, qui est ici 1000).
J'espère que cette petite explication aura pu expliquer au lecteur le fonctionnement d'une recherche par dichotomie. Maintenant, pour ce qui est de l'implémentation d'une telle recherche en Lisp :

```
(defun binary-search (l-bound u-bound &optional (tries 0))
  (let ((guess (floor (+ l-bound u-bound) 2)))
    (format t "~&~D?~&" guess)
    (case (read)
      (plus-haut (binary-search guess u-bound (1+ tries)))
      (plus-bas  (binary-search l-bound guess (1+ tries)))
      (t         (format t "Ton nombre a été trouvé en ~D
   essais" tries)))))
```

Plus qu'à la tester !

```
[1] CL-USER> (binary-search 1 1000)
500?
plus-haut
750?
plus-bas
625?
plus-haut
687?
plus-bas
656?
plus-bas
640?
plus-bas
632?
plus-bas
628?
plus-haut
630?
plus-bas
629?
oui
Ton nombre a été trouvé en 9 essais
```

Cette petite introduction à la recherche par dichtomie servait surtout à donner une intuition au lecteur de l'importance des arbres binaires.
Ceux-ci fonctionnent d'une manière similaire à notre recherche dichotomique. Sur un arbre binaire, chaque noeud est soit relié à rien (ou "nil"), soit relié à sa "gauche" à un noeud contenant une valeur plus petite que celle de son noeud parent, soit relié à sa "droite" à un noeud contenant une valeur plus grande que celle de son noeud parent. Nous pouvons donc représenter un arbre binaire ainsi :

3.3 Qu'est-ce que Lisp ?

Lisp est une famille de langages de programmation fonctionnels, inventés en 1958 par John McCarthy (l'homme ayant inventé le terme "intelligence artificielle").
Ceux-ci sont reconnaissables facilement grâce au très grand nombre de parenthèses présentes dans chacun des dialectes de Lisp.
Aujourd'hui, le dialecte Lisp le plus utilisé reste **Clojure**, toutefois, il en reste d'autres gardant toujours leur cote de popularité, pour n'en citer que quelques-un, nous avons

- **Scheme**, qui est un dialecte très minimaliste de Lisp, très utilisé au niveau académique.
- **Emacs Lisp**, un dialecte très pratique pour tout ceux désirant configurer l'éditeur de texte Emacs.
- **Racket**, un super-set de Scheme.
- **Common Lisp** est le dialecte que nous utiliserons ici, ce dialecte a toujours eu la réputation d'être plus orienté vers les applications pratiques, et n'a jamais reçu de grandes faveurs académiques. Common Lisp a toutefois le mérite d'avoir été standardisé, de bénéficier d'un système orienté-objet connu sous le nom de CLOS.

3.3.1 Les bases de Lisp.

Tout d'abord, pour commencer notre aventure avec Lisp, ouvrons donc notre REPL (cela peut-être SBCL, Clisp, ou encore CCL), et additionnons deux nombres.

```
CL-USER> (+ 3 5)
8
```

Cela peut sembler bien étrange pour le non-inité, et pourtant, tout cela est parfaitement logique.
En fait, lorsque l'on désire appeler une fonction (comme par exemple ici "+"), la syntaxe sera toujours "(**fonction arguments**)". Prenons quelques autres exemples avec quelques opérations arithmétiques.

```
CL-USER> (- 2 1)
1
CL-USER> (* 3 4 2)
24
CL-USER> (/ 5 6)
5/6
CL-USER> (/ 5 6.0)
0.8333333
```

Comme on peut le voir, la logique reste la même pour tout les opérateurs, aussi, cette syntaxe nous permet également de mettre à ces opérateurs autant d'arguments que l'on le souhaite, (l'exemple est donné ci-dessus avec l'opérateur de multiplication.).
Egalement, l'opération de division ne remet pas directement un nombre à virgule, mais plutôt un Ratio, si l'on ne lui donne pour argument uniquement des nombres entiers.
Pour déclarer une variable en Lisp, plusieurs choix s'offrent à nous, je présenterai uniquement ici "defparameter".

```
CL-USER> (defparameter *test* 'bonjour)
*TEST*
CL-USER> *test*
BONJOUR
```

J'en profite pour attirer l'attention sur l'utilisation de "*" autour du nom de la variable. Ceux-ci sont appelés les "cache-oreilles", il ne s'agit ici que d'une convention, il est tout à fait acceptable de ne pas utiliser de caches-oreilles lors ed la définition d'une variable globale.

Maintenant, j'utiliserai la fin de cette petite introduction comme prétexte pour présenter "la récursion".
Pour montrer un exemple de fonction récursive, je présenterai ici la fonction factorielle, fonction récursive par excellence.
Tout d'abord, pour déclarer une fonction, nous utiliserons le mot-clé **defun**.

```
CL-USER> (defun factorial (x) ...)
```

Il faut ensuite réfléchir à ce qu'il faudra mettre au sein de notre fonction.
La récursion est un processus en 3 étapes.

a. Trouver la valeur de f(0) (ou de n'importe quel cas de base.)

b. Supposer que cette fonction remettra un nombre correct pour f(n-1).

c. Trouver la valeur de f(n) en fonction de f(n-1).

L'exemple ici est assez simple, et est souvent donné en introduction à la récursion. Voici donc une définition formelle de notre fonction factorielle.

$$\begin{cases} 0! = 1 \\ n! = n \cdot (n-1)! \end{cases}$$

Ainsi, uniquement grâce à cette simple définition, nous allons pouvoir coder cette fonction en Common Lisp

```
CL-USER> (defun factorial (x)
                    (if (= x 0)
                        1
                       (* x (factorial (- x 1)))))
FACTORIAL
```

On constate directement la simplicité et l'élégance avec laquelle cette fonction peut-être ainsi programmée. D'autres fonctions peuvent-être codée de manière récursive, l'exemple de l'algorithme de tri par fusion, ou encore l'agorithme d'Euclide pour calculer le pgcd sont deux exemples de fonctions récursives dont j'ai pu montrer l'implémentation ci-dessus.
Un autre exemple d'algorithme utilisant la récursion serait l'algorithme "quick-sort" dit du "tri-rapide", en voici son implémentation

```
(defun quick-sort (lat)
  "Implémentation récursive de l'algorithme de tri-rapide."
  (if lat
      (let ((partition (car lat))
            (rest (cdr lat)))
      (nconc
       (quick-sort (remove-if (lambda (a)
                                (>= a partition))
                              rest))
       (remove-if (lambda (a) (/= a partition)) lat)
       (quick-sort (remove-if (lambda (a)
                                (<= a partition))
                              rest))))
    nil))
```

Si le lecteur en sent l'envie, nous l'encourageons à essayer de coder la fonction récursive d'Ackermann, ou encore la McCarthy 91.

3.3.2 Programme Lisp pour résoudre parfaitement le jeu de Nim.

Afin que le lecteur se fasse une idée de comment utiliser dans des cas plus concrets, nous allons vous présenter une intelligence artificielle, assez basique, gagnant au jeu de Nim à plusieurs tas. (Aussi connu sous le nom de jeu de Marienbad.), dans ce jeu, plusieurs alumettes sont disposées sur plusieurs piles, le but du jeu est de récolter la dernière alumette.
L'intelligence gagnant à ce genre de jeux est assez basique, en effet, le jeu de Nim à plusieurs tas est un jeu résolu parfaitement.
La stratégie gagnante pour ce jeu est simple, nous définissons une position gagnante une position où la"nim-somme" (qui est en réalité l'opérateur "xor" ou $\oplus$ appliqué à la représentation binaire de notre tas) est différent de 0, et à l'inverse, une position perdante est une position où la "nim-somme" appliquée aux tas est égale à 0.
Un exemple de position perdante serait la position où les alumettes seraient réparties sur 4 tas avec 1 alumette sur le premier tas, 3 sur le deuxième, 5 sur le 3ième, et 7 sur le dernier.

$$1 \oplus 3 \oplus 5 \oplus 7 = 0$$

Pour se le prouver, on peut utiliser notre interprète Lisp et faire

```
CL-USER> (logxor 1 3 5 7)
0
```

Le but de l'ordinateur sera donc de toujours se trouver en position de sécurité, et de toujours mettre le joueur en position perdante.
Le twist que nous avons apporté est le fait que le joueur peut décider des tas avec lesquels ils va jouer, et ainsi, mettre directement l'ordinateur en position de défaite, notre IA devra donc se montrer capable de jouer des coups, même en perdant.

```
(defparameter *heaps* nil
  "*heaps* représente les différents tas avec lesquels nous
    allons jouer.")
```

Ici, je ne fais que définir une variable globale *heaps*, celle-ci représentera les tas disponibles au travers de mon programme.

```

(defun heap-print (heaps)
  (format t "~&~{TAS ~D: ~D alumettes.~%~}"
          (loop for number from 1 to (length *heaps*)
                for heap in heaps
                collect number
```

```
                collect heap)))
```

Ces deuc fonctions seront dites des “fonction d’aide”, elles auront surtout pour utilité de ne pas devoir surcharger les fonctions principales.
Le lecteur attentif remarquera que je termine le nom de ces fonctions par un “p”, cela est dû au fait que ces fonctions sont des **prédicats**, elles ne peuvent retourner que “t” ou “nil” (“vrai” ou “faux”), ceci n’étant également qu’une convention, si le lecteur désire écrire des prédicats sans laisser de “p” au bout, SBCL ne lui trouvera rien à redire.
Aussi, on peut remarquer dans les deux premières fonctions l’utilisation de “reduce”, il s’agit d’une fonction destinée aux **catamorphismes**, cela veut dire qu’elles permettent de passer d’une structure de données comme une liste, ou un arbre, vers un scalaire, comme un entier ou encore un flottant.
Egalement, j’utilise #’, ce symbole nous permet d’expliciter que nous utilisons une fonction, et pas une variable.

```
(defun play-winning-move (heaps)
  "Si l'adversaire n'est pas en poistion gagnante, nous
   jouons alors le meilleur
coup disponible."
  (let ((unchanged t))
    (loop for heap in heaps
          for i from 0
          while unchanged
          do (loop for num from 0 to heap
                   when (winningp num i heaps)
                     do (progn
                          (format t "L'ordinateur a décidé
   de prendre ~D alumettes du tas ~D~%"
                                  num (1+ i))
                          (decf (nth i heaps) num)
                          (setf unchanged nil)))))
  heaps)
```

Nous voici enfin au cœur du programme. Avec nos fonctions “play-winning-move” et “play-random-legal-move” sont nos deux fonctions où est contenue notre joueur, les commentaires explicitant déjà leurs actions, je me concentrerai surtout sur l’implémentation.
Comme nous le remarquons, j’utilise **let** et **loop**, **let** me permet de déclarer des variables dites “locales”, cela me permet d’avoir un code propre, où chaque chose reste à sa place.
loop est une macro, qui permet d’itérer, ou de répéter un nombre certain de fois une même action, je l’utilise ici en raison de la lisibilité de celle-ci, certains préfèrent la macro “do”, je trouve cette dernière illisible, mais j’encourage le lecteur se renseigner sur cette dernière si le coeur le lui en dit.

```
(defun c-move (heaps)
  "Cette fonction fait office d'aiguilleur,
| Les tas sont vides -> On arrête de jouer.
| Nous sommes dans une poisition perdante -> On joue un
   coup au hasard.
| Nous gagnons -> On joue le meilleur coup disponible."
  (cond
    ((emptiedp heaps)
     (format t "Félicitations, vous avez gagné!~%"))
    ((unsafep heaps)
     (play-random-legal-move heaps))
    (t (play-winning-move heaps))))
```

Je ne ferai remarquer ici que l'utilisation de la macro **cond**, une autre manière d'écrire des conditions en Lisp.

```
(defun heaps-repl ()
  "Cette fonction récursive est la fonction que le joueur
   voit lorsqu'il joue."
  (when (emptiedp *heaps*)
    (format t "Pas de chance, l'ordinateur a gagné.~%A une
   prochaine fois peut-être."))
  (format t "C'est votre tour, de quel tas désirez vous
   retirer des alumettes? ")
  (heap-print *heaps*)
  (princ 'PLAYER> )
  (let ((player-heap-choice (read)))
    (if (or (> player-heap-choice (length *heaps*))
            (>= 0 player-heap-choice))
        (progn (format t "Veuillez choisir un tas compris
   entre 1 et ~D~%" (length *heaps*))
               ;; Remarquez l'appel récursif à heaps-repl
               ;; si le joueur tente une mauvaise entrée.
               (heaps-repl))
      (progn
        (format t "Combien d'alumettes désirez-vous retirer
   de ce tas?~%")
        (princ 'PLAYER>)
        (let ((player-heap-choice (1- player-heap-choice))
              (number-of-matches (read)))
          (if (or (< (nth player-heap-choice *heaps*)
   number-of-matches)
                  (>= 0 number-of-matches))
              (progn (format t "Veuillez choisir un
   nombre correct d'éléments.~%")
                     ;; Encore un appel récursif à
   heaps-repl!
                     (heaps-repl))
```

```
                ;; Si nous parvenons à passer à travers
    toutes les conditions,
                ;; alors, nous modifions la valeur des tas.
                (decf (nth player-heap-choice *heaps*)
    number-of-matches)))))))
```

Cette très longue fonction fait office d'interface utilisateur. Nous avons déjà fait remarqué au lecteur en commentaires l'utilisation de la récursion. Egalement, nous signalons **progn**, une fonction bien pratique pour définir des blocs de code.

3.4 Le pathfinding.

3.4.1 Présentation du problème.

3.4.2 L'algorithme de Dijkstra.

3.4.3 Le besoin d'heuristiques.

3.4.4 Dijkstra avec de l'heuristique : A* !

3.5 Des programmes et des jeux !

3.5.1 L'agorithme Minimax.

3.5.2 $\alpha\beta$-élagage.

3.5.3 Du dynamisme bon sang !

3.6 Un programme de jeu.

4 Le Machine Learning.

4.1 Définition du Machine Learning

Le Machine Learning, ou Apprentissage Automatique, est un type d'intelligence artificielle qui avec les données à analyser et sur lesquelles s'entraîner permet aux ordinateurs d'apprendre par expérience sans avoir été explicitement programmé à cet effet ou par intervention humaine. Cela consiste en algorithmes d'apprentissage qui améliorent leur performance à exécuter des tâches au fil du temps grâce à de l'expérience.

4.2 Les Maths dans le Machine Leanring.

a. De nombreux data scientists (chargés de la gestion, de l'analyse et de l'exploitation des données au sein d'une entreprise) considèrent le machine Learning comme un apprentissage statistique.

b. Matrice et algèbre matricielle, exemple :

   i. Les suggestions d'amis sur Facebook

   ii. Recommandation de vidéo sur Facebook

   iii. $\cdots$

c. Fonction, variable, équation et graphique, ...

4.3 Les résaux de neurones

Une couche de neurones d'entrées, plusieurs couches cachées, une couche de sortie suivie d'une fonction d'activation. Chaque neurone possède une valeur obtenue par une fonction de combinaison étant la somme des valeurs des neurones de la couche précédente, chacune multipliée par un poids spécifique
$z = x_1 \cdot p_1 + x_2 \cdot p_2 + \cdots + x_n \cdot p_n$
Une fois la (ou les) valeur de la couche de sortie obtenue, on applique à celle-ci une fonction d'activation qui transforme la valeur en fonction d'un seuil. Si en dessous du seuil, inactif (0/-1), aux environs du seuil, phase de transition, et au-dessus du seuil, actif (1/>1). Le type de fonction varie d'un cas à l'autre, mais les plus récurrentes sont la fonction sigmoïde $\frac{1}{1+e^{-x}}$ la fonction tangente hyperbolique $\frac{2}{1+e^{-2x}} - 1$ ou encore la fonction ReLU $\begin{cases} 0 & x < 0 \\ x & x >= 0 \end{cases}$

Durant l'apprentissage, les poids sont des valeurs prises au hasard. Il faut donc les ajuster pour fournir une réponse qui se rapproche au mieux de la réalité. Comme on entraîne notre réseau, on connaît la vraie valeur finale. On va donc appliquer une fonction de coût afin de calculer le gradient d'erreur entre la valeur réelle et la valeur prédite $\frac{1}{2}(y_r - y_p)^2$, et ainsi mettre à jour les poids par rétropropagation (! il y a des maths plus compliquées derrière). A chaque nouvelles données injectées lors de l'apprentissage, le réseau est plus performant.

4.4 Intro au langage Python

a. Créé par Guido van Rossum au Stichting Mathematisch Centrum en Hollande.

b. Python est le successeur d'un langage de programmation nommé "ABC"

c. Le nom du langage vient de la série Monty Python's flying Circus dont Guido Van Rossum était fan. Cependant l'image du serpent paraissait plus évidente pour tout le monde, il a donc décidé d'utiliser celle-ci comme symbole du langage.

d. La première version de Python paraissait en 1991

e. C'est un langage de programmation open source, c'est à dire gratuit et libre d'utilisation